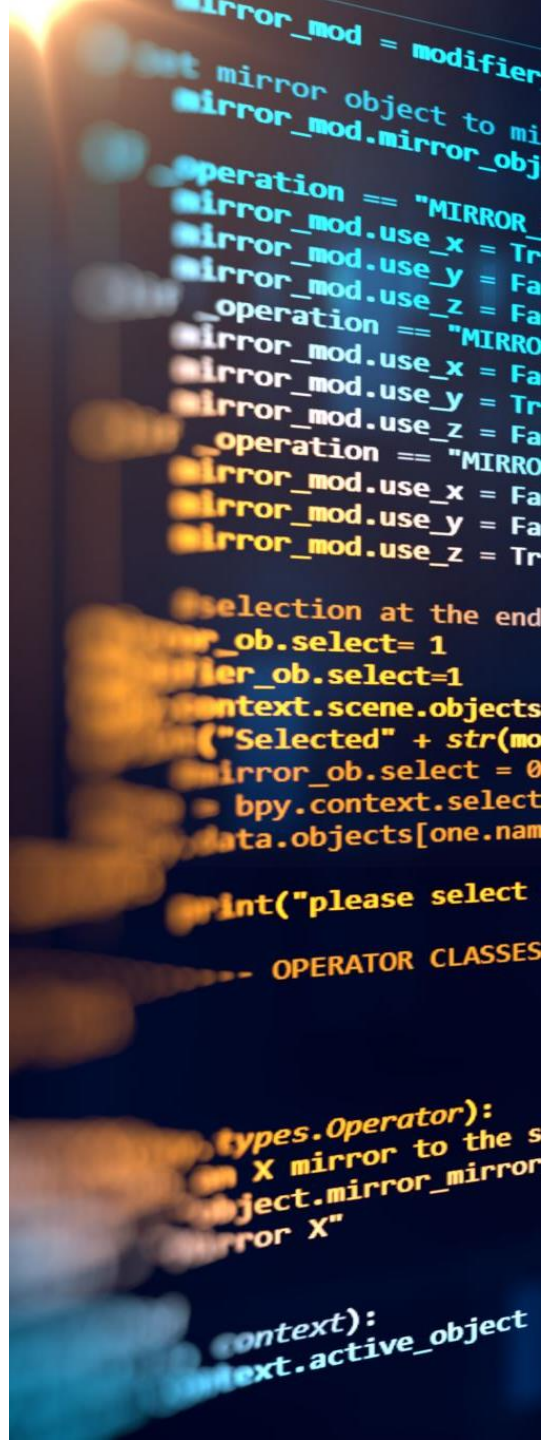
Please use the following QR code to check in and record your attendance.

CS 1027
Fundamentals of Computer Science II

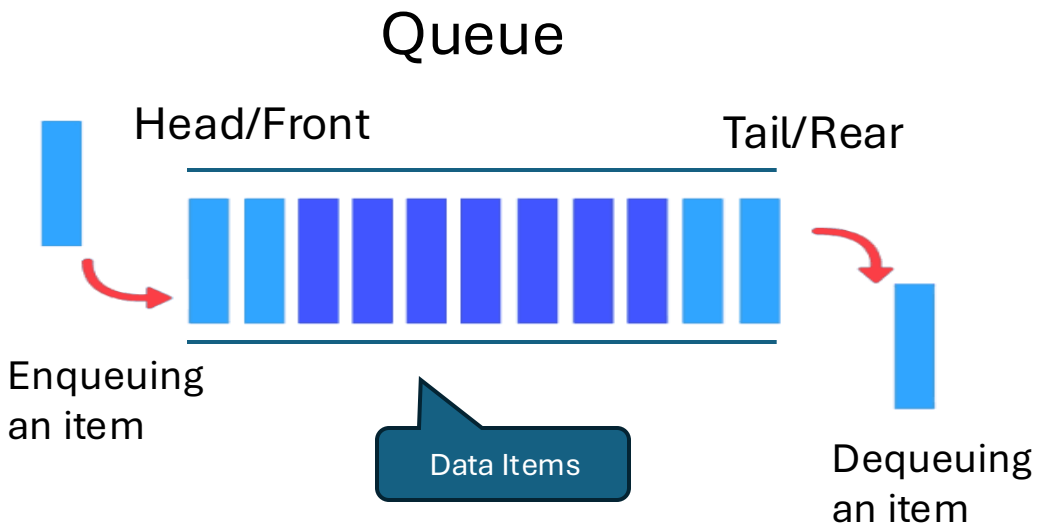
The Queue ADT

Ahmed Ibrahim



The Queue ADT

- The concepts of **queues** consist of the **abstract queue** and **queue data structures** (the implementations of the abstract queue).
- The abstract queue's characteristic is the **First-In-First-Out** (or simply FIFO), which deletes the first element currently in the data structure.



- Example – **CPU Task Scheduling:** In round-robin scheduling, processes waiting to be executed are placed in a queue, and the CPU handles them in the order they arrive.

Real Life Queues

Properties of the Queue ADT

- The properties of a queue are as follows:
  - A queue is a **linear collection** of data elements with the following operations:
    - **Enqueue** adds an element to the back of the queue. The order of elements is based on the time they were added, with the earliest at the front and the latest at the back.
    - **Dequeue** removes the front element from the queue.
    - **First (Peek)** retrieves the front element without removing it.
    - **isEmpty** Determines whether the queue is empty
    - **Size** Determines the number of data items in the queue
    - **toString** Returns a string representation of the queue

Java Interface for the Queue ADT

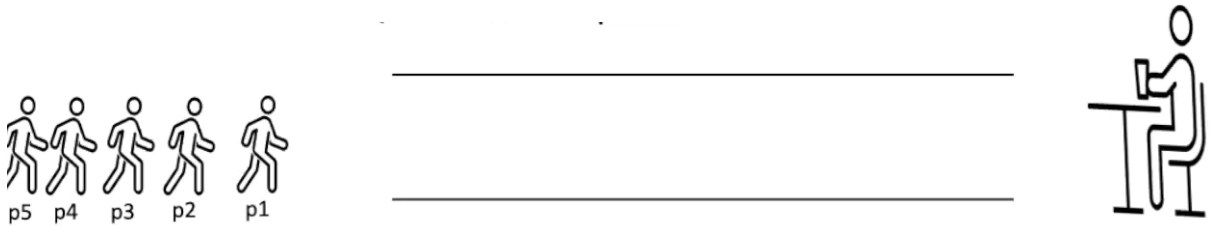
```
public interface QueueADT<T> {
// 1. Adds one data item to the rear of the queue
public void enqueue (T dataItem);
// 2. Removes and returns the data item at the front
of the queue
public T dequeue( ) throws EmptyCollectionException;
// 3. Returns without removing the data item at the
front of the queue
public T first( ) throws EmptyCollectionException;
// 4. Returns true if the queue contains no data items
public boolean isEmpty( );
// 5. Returns the number of data items in the queue
public int size( );
// 6. Returns a string representation of the queue
public String toString();
}
```

Underflow vs. Overflow

- A queue is said to be empty if it does not contain an element. Deletion cannot be done when a queue is empty; such a situation is called **underflow**.
- The length of a queue is the number of elements in the queue. When the length reaches the maximum length that a queue is allowed, insertion can not be done, and such a situation is called **overflow**.

Queue ADT: Real-World Operations

Simulation of Real-World Scenario: Waiting List



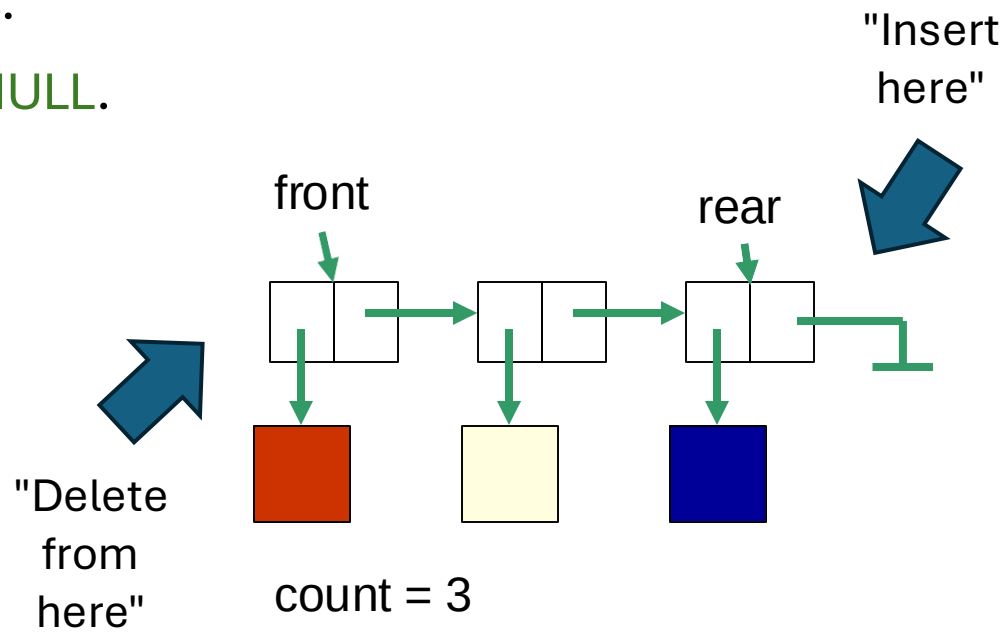
- To implement a queue, we need:
  - A data structure to hold the data items
  - A way to indicate the *front* of the queue
  - A way to indicate the *rear* of the queue
- A queue can be implemented using an **array** or **linked list** representation, with two accessing variables/pointers, front and rear, representing the queue's front and rear (back) positions.

Source:
https://www.youtube.com/watch?app=desktop&v=HcB1P9sJZB4

Linked List Queue

Linked List Queue

- A **linked list queue** stores queue data values in a single linked list and uses **two front and rear pointers** to represent the front and rear positions.
- A linked list queue is empty if both front and rear are NULL.
- The queue operations are defined as follows.
  - The **enqueue** operation first creates a node containing the data value, inserts the node after the rear (back) node, and updates both front and rear.
  - The **dequeue** operation deletes the front node (i.e., the node pointed by the front pointer) and updates the front and rear.



LinearNode Class

```
public class LinearNode<T>
{
private LinearNode<T> next;
private T dataItem;

  public LinearNode() {
    next = null;
    dataItem = null;}

  public LinearNode(T value) {
    next = null;
    dataItem = value;}
```

creates an empty node

creates a node with a specific data item

```
public LinearNode<T> getNext() {return next;}

public void setNext(LinearNode<T> node) {

next = node;}

public T getDataItem() {return dataItem;}

public void setDataItem(T value) {

dataItem = value;}

}
```

Getter and Setter Methods

Recall: Java Interface for the Queue ADT

```
public interface QueueADT<T> {
// 1. Adds one data item to the rear of the queue
public void enqueue (T dataItem);
// 2. Removes and returns the data item at the front
of the queue
public T dequeue( ) throws EmptyCollectionException;
// 3. Returns without removing the data item at the
front of the queue
public T first( ) throws EmptyCollectionException;
// 4. Returns true if the queue contains no data items
public boolean isEmpty( );
// 5. Returns the number of data items in the queue
public int size( );
// 6. Returns a string representation of the queue
public String toString();}
```

Implementing QueueADT

```
public class LinkedQueue<T> implements
QueueADT<T>
{

private int count;
private LinearNode<T> front, rear;

//Creates an empty queue
public LinkedQueue() {
  count = 0;
  front = rear = null;
}
```

The integer **count** is the number of nodes in the queue

References to the head and tail of the queue

```
// Enqueue operation – adds an item to the rear
of the queue
public void enqueue(T dataItem) {//code??}
// Dequeue: removes an item from the front of
the queue
public T dequeue() {//code??; return null;}
// Returns the current size of the queue
public int size() {return count;}
// Checks if the queue is empty
public boolean isEmpty() {return count == 0;} }
```

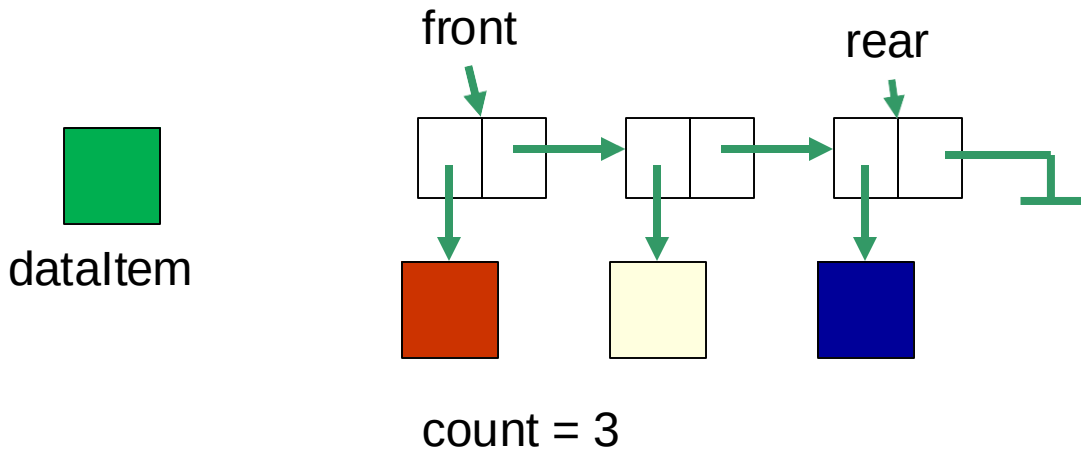
Enqueue Operation

Two cases need to consider

```
public void enqueue(T dataItem) {
  LinearNode<T> newNode = new
  LinearNode<T>(dataItem);
// Case 1: Queue is empty
  if (front == null) {front = newNode;
rear = newNode;}
// Case 2: Queue is not empty
else {rear.setNext(newNode);
       rear = newNode;}
  count++; // increment counter by 1
}
```

1. The queue is empty

2. The queue is not empty



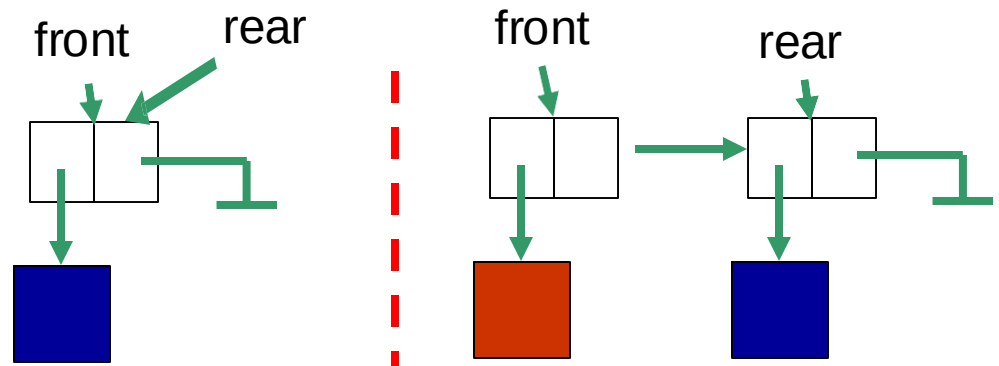
Dequeue Operation

Cases to be consider

```
public T dequeue() {
// Case 1: Queue is empty
  if (front == null) {return null;} // Or throw an
exception
T dataItem = front.getDataItem(); // Retrieve data from
the front node
front = front.getNext(); // Move front to the next node
// If the queue becomes empty, set the rear to null as
well
if (front == null) { rear = null;}
count--; // Decrement the count
return dataItem;} // Return the removed data
```

1. The queue is empty

2. The queue is not empty



One data item

More than one data item

toString Method Override

```java
@Override
public String toString() {
if (front == null) {return "Queue is empty";}
StringBuilder result = new StringBuilder();
LinearNode<T> current = front;
while (current != null) {
result.append(current.getDataItem()).append(" -> ");
current = current.getNext();}
// Remove the last arrow and space for a cleaner
output
result.setLength(result.length() - 4);
return result.toString();
}
```

Why?

`StringBuilder` Class

`StringBuilder` is a Java class used to create and manipulate strings efficiently, especially when frequent modifications are required.

Unlike String, which is immutable (meaning a new object is created every time you modify it), StringBuilder is **mutable**, so it can modify the original object without creating new ones.

This makes StringBuilder particularly useful when you're appending or altering strings repeatedly, as it saves time and memory.

Question!

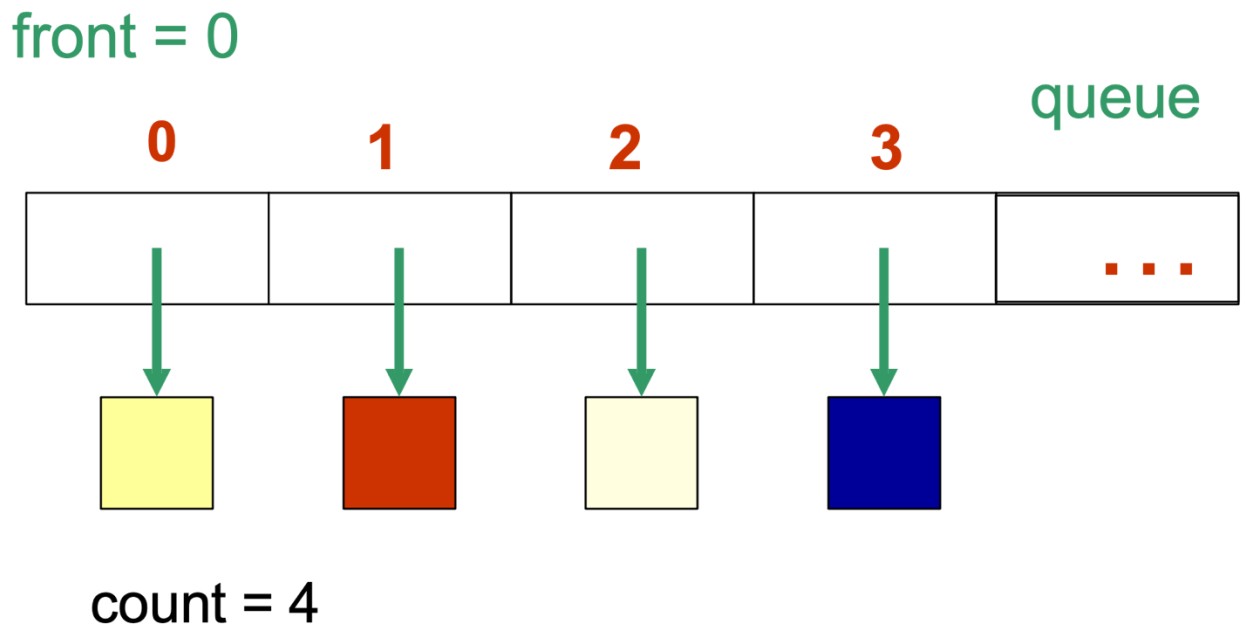
```
public class QueueTest{
 public static void main(String[] args){
  Queue<Integer> queue = new LinkedList<Integer>();
  Stack<Integer> stack = new Stack<Integer>();
  for (int i = 1; i <= 9; i += 2)
    queue.add(i);
    while (!queue.isEmpty()) {stack.add(queue.poll());}
    while (!stack.isEmpty()) {queue.add(stack.pop());}
 }
}
```

- Which values will be contained in the queue after the two while loops? (from first to last, without square braces). Example answer format (spacing between the numbers): 1, 2, 3, 4, 5

Array Implementation of a Queue

- Use an array to store the data items of the queue
- front = 0 (index of the first data item)
- count = number of data items

Note: rear = count – 1



Step-by-Step Operations in an Array-Based Queue

index	0	1	2	3	4	5	6	7	8	9
a[i]	X	X	X	X	X	X	X	X	X	X

Empty queue: front =rear = -1

index	0	1	2	3	4	5	6	7	8	9
a[i]	6	X	X	X	X	X	X	X	X	X

Insert 6: front = rear = 0

index	0	1	2	3	4	5	6	7	8	9
a[i]	6	5	X	X	X	X	X	X	X	X

Insert 5: front=0, rear = 1

index	0	1	2	3	4	5	6	7	8	9
a[i]	6	5	4	X	X	X	X	X	X	X

Insert 4: front=0, rear = 2

index	0	1	2	3	4	5	6	7	8	9
a[i]	X	5	4	X	X	X	X	X	X	X

delete: front=1, rear = 2

Implementation of a Queue using an Array

```
public class ArrayQueue<T> implements QueueADT<T> {
    private final int DEFAULT_CAPACITY = 100;
    private int count; // Current number of elements in the queue
    private T[] queue; // Array that holds the queue elements

 public ArrayQueue() {//Default constructor that initializes the queue with the default capacity.
  count = 0;
  queue = (T[])(new Object[DEFAULT_CAPACITY]);}

// Constructor that initializes the queue with a specified initial capacity.
 public ArrayQueue (int initialCapacity) {count = 0;
  queue = (T[])(new Object[initialCapacity]);
}
```

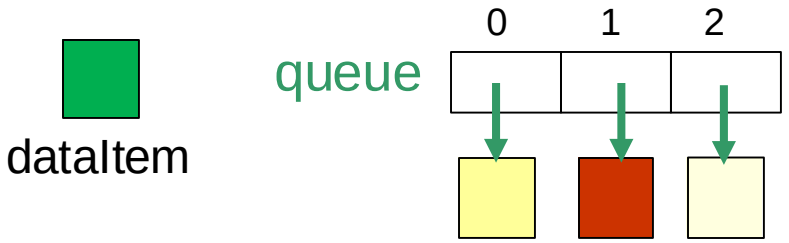
Enqueue Operation

```
public void enqueue(T dataItem) {
if (count == queue.length) { // Check if the array is full
  expandCapacity(); // Expand the array's capacity if needed
}
  queue[count] = dataItem; // Add the new data item at the end
  count++; // Increment the count
}
```

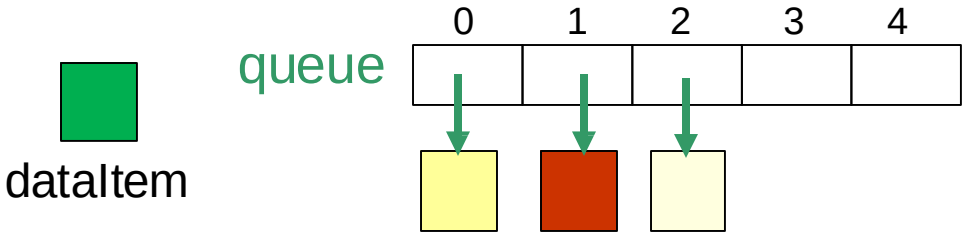
We Need to Consider Two Cases

1. The array is full

count = 3 = size of array



2. The queue is not empty



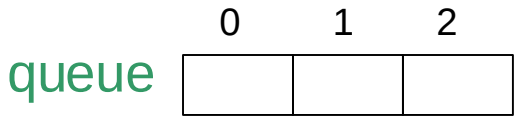
Dequeue Operation

```
public T dequeue() throws EmptyCollectionException {
if (count == 0) { // Check if the queue is empty
   throw new EmptyCollectionException("Empty queue"); }
T result = queue[0]; // Store the front element to return
later
count--; // Decrease count as one item is removed
// Shift elements to the left
for (int i = 0; i < count; i++) {queue[i] = queue[i + 1];}
  queue[count] = null; // Optional: clear the last element
for garbage collection
 return result; }
```

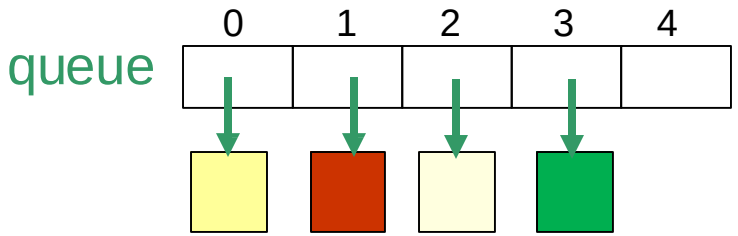
We Need to Consider Two Cases

1. The queue is empty

count = 0



2. The array is not full



count = 4

Remaining Queue Operations

```
// Returns the element at the front of the queue
without removing it
public T first() throws EmptyCollectionException {
if (isEmpty()) {
throw new EmptyCollectionException("Queue is
empty");}
return queue[0];}

// Checks if the queue is empty
public boolean isEmpty() {return count == 0;}

// Returns the number of elements in the queue
public int size() {return count;}
```

```
// Returns a string representation of the queue
elements from front to rear
@Override
public String toString() {
if (isEmpty()) {return "Queue is empty";}
StringBuilder result = new StringBuilder();
for (int i = 0; i < count; i++) {
  result.append(queue[i]);
 if (i < count - 1) {result.append(" -> ");}
}
return result.toString();}
```

Priority Queue

- A priority queue is a collection of elements in which each element is assigned a **priority**. The priority of the elements determines the order in which they will be processed.
- The rule for processing elements of a priority queue is the following:
  1. An element of a **higher priority** is processed before an element with a lower priority.
  2. Two elements of the same priority are processed on a first-come, first-served (FCFS) order.
- **A general queue can be viewed as a special priority queue using insertion time as a priority**.
- Priority queues can be implemented by either linked lists or arrays.
- **Use case**: Priority queues are used in operating systems to manage processes for running. The highest priority process will be processed first.

Class Node for Priority Queue

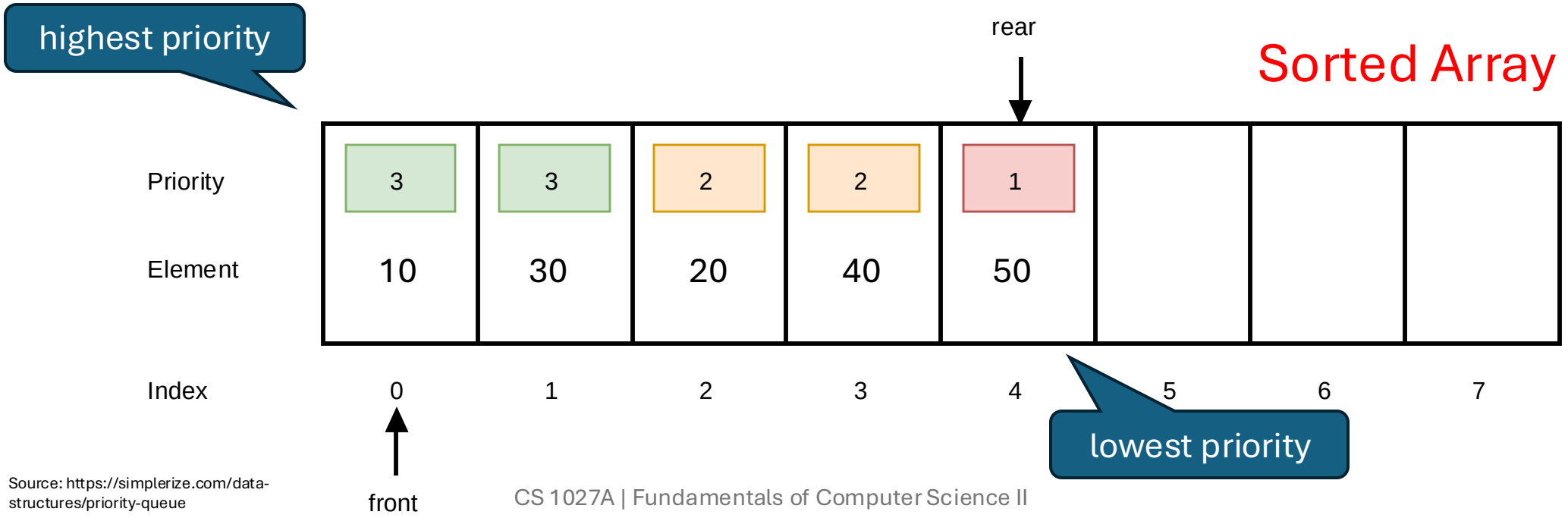
- A linked list priority queue utilizes a **singly linked list** to store data values, with a pointer front that points to the first node.
- Each node in this list has three components:
- The list is sorted based on **priority**.
- When inserting a **new node**, it is placed after a specific node that meets the following conditions:
  - The priority of the **new node** is less than or equal to the priority of the current node.
  - The new node's priority is greater than the next node's priority unless the next node is NULL.

```
public class Node<T> {
private T data;
private int priority;
private Node<T> next;

// Constructor
public Node(T data, int priority)
{this.data = data;
this.priority = priority;
this.next = null;}
```

Array-Based Priority Queue Enqueue

- The elements are enqueued in the order 10, 20, 30, 40, and 50, each **sorted** by **priority** during the enqueue operation. This way, the highest priority element is readily accessible for quick dequeuing from the front.



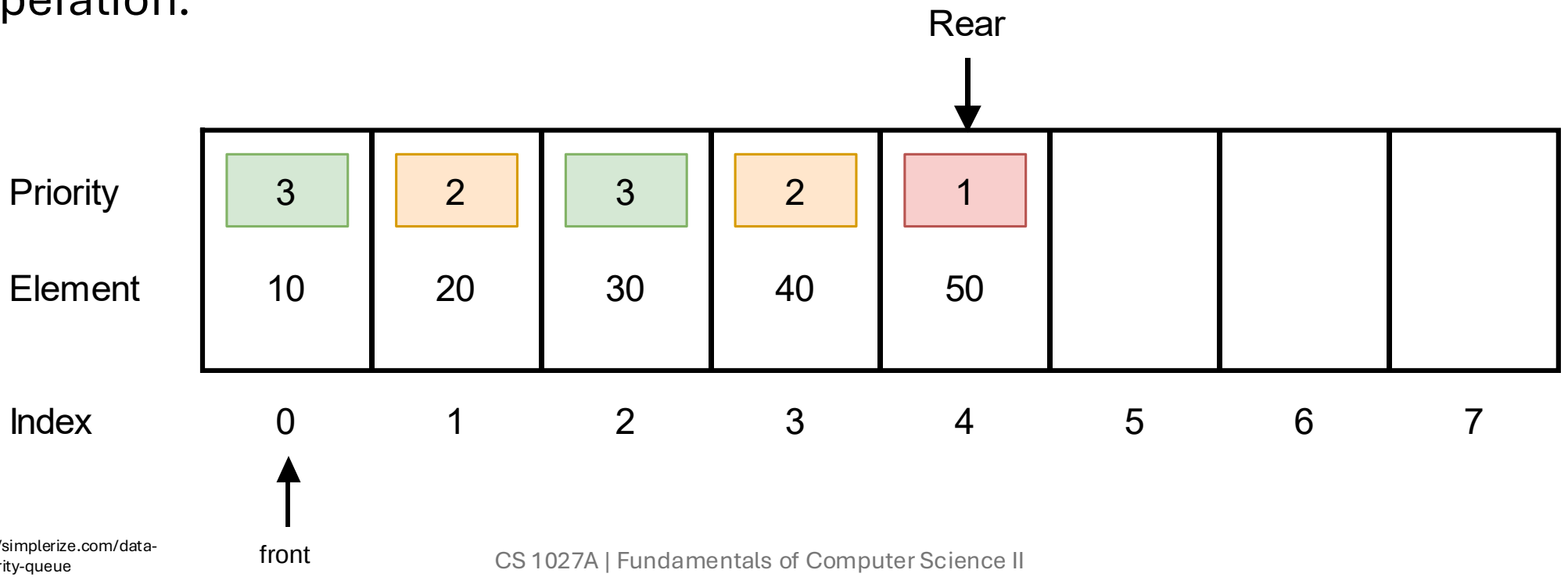
Source: https://simplerize.com/data-structures/priority-queue

Pseudocode for Enqueue in Array-Based Sorted Priority Queue

```
Function Enqueue(priorityQueue, element, priority):
  # Step 1: Create a new item with an element and priority
  newItem = (element, priority)
  # Step 2: Find the correct position to insert the new item
  position = 0
  While position < length(priorityQueue) AND
priorityQueue[position].priority >= priority:
    position = position + 1
  # Step 3: Shift elements to make space for the new item
  For i = length(priorityQueue) - 1 down to position:
    priorityQueue[i + 1] = priorityQueue[i]
  # Step 4: Insert the new item at the correct position
  priorityQueue[position] = newItem
  # Step 5: Update the rear of the queue
  rear = rear + 1
End Function
```

Array-Based Priority Queue Dequeue

- The dequeue operation locates and serves the highest-priority element by searching through the entire queue, making it more costly than the enqueue operation.



Source: https://simplerize.com/data-structures/priority-queue

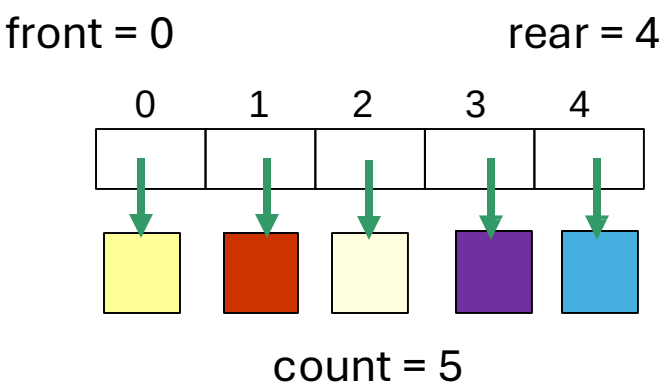
Pseudocode for Dequeue in Array-Based Unsorted Priority Queue

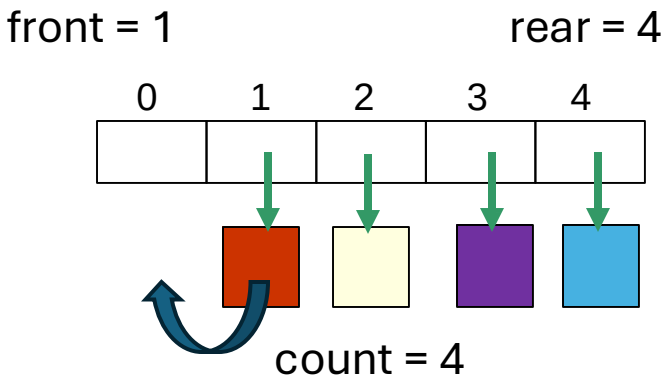
```
Function Dequeue(priorityQueue):
  # Step 1: Check if the queue is empty
  If length(priorityQueue) == 0:
    Print "Queue is empty"
    Return Null
  # Step 2: Find the highest-priority element
  highestPriorityIndex = 0
  For i = 1 to length(priorityQueue) - 1:
    If priorityQueue[i].priority > priorityQueue[highestPriorityIndex].priority:
      highestPriorityIndex = i
  # Step 3: Remove the highest-priority element
  highestPriorityElement = priorityQueue[highestPriorityIndex]
  # Step 4: Shift elements to fill the gap
  For j = highestPriorityIndex to length(priorityQueue) - 2:
    priorityQueue[j] = priorityQueue[j + 1]
  # Step 5: Remove the last element (duplicate after shifting)
  Remove the last element from priorityQueue (or decrease its length by 1)
  # Step 6: Return the dequeued element
  Return highestPriorityElement
End Function
```

Circular Array

Managing Queue Shifts

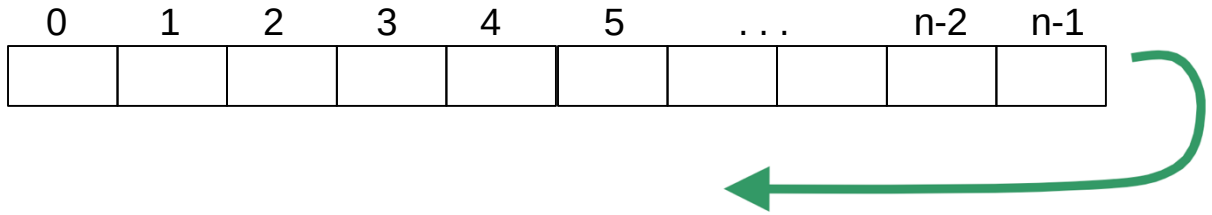
Dequeue()

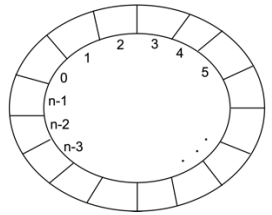




Wrap-around Logic

- In a regular array-based queue, once the rear pointer reaches the end of the array, there is no more room to add elements, even if there is free space at the beginning due to dequeued elements.



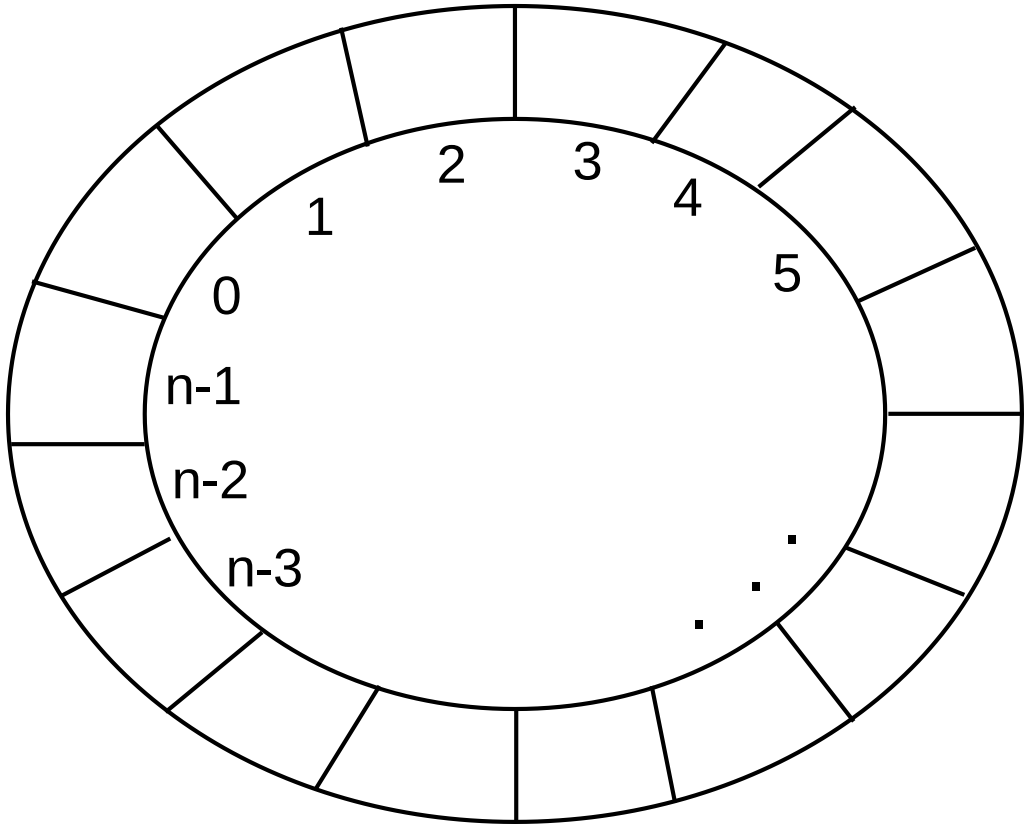


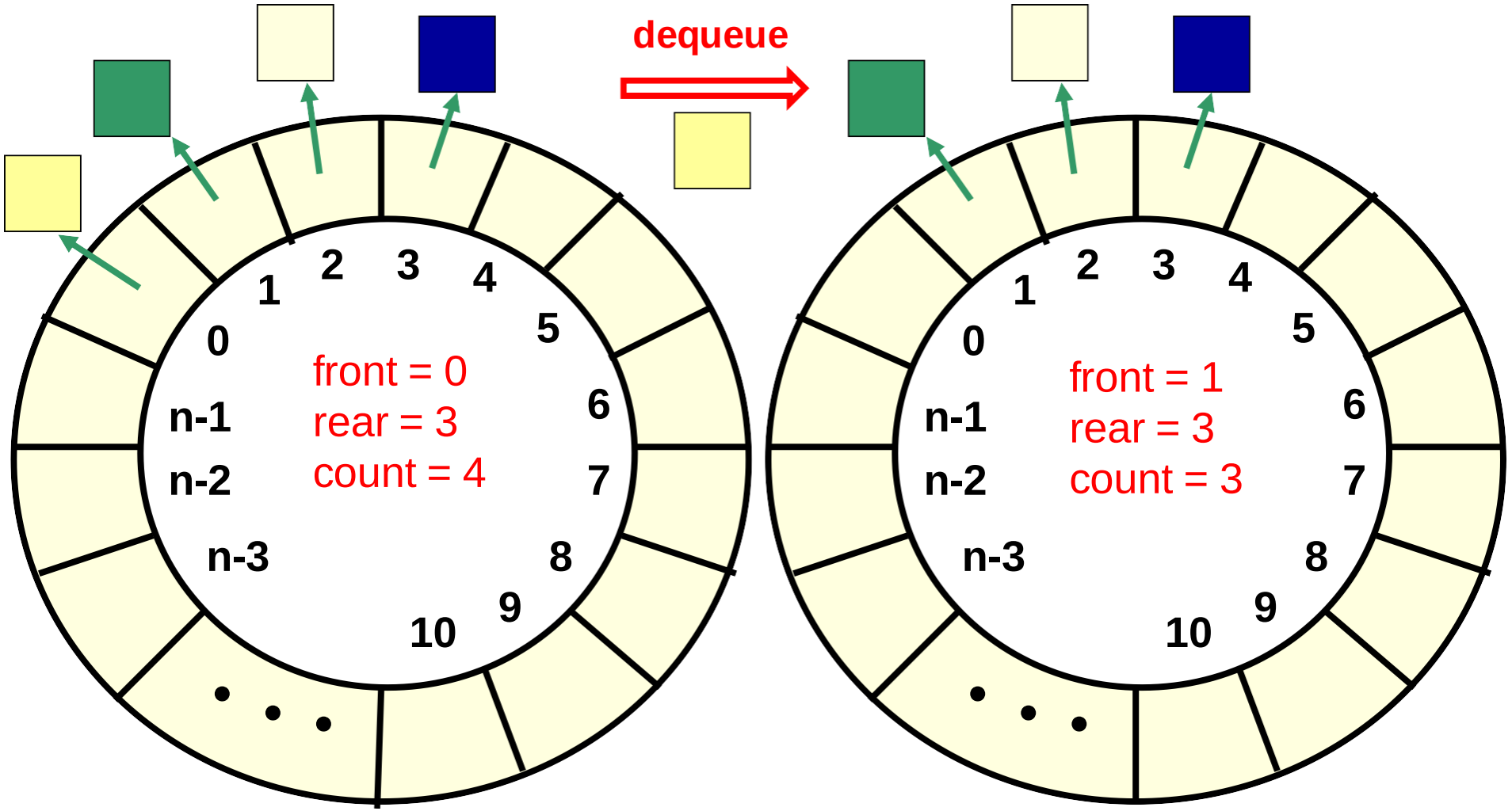
Conceptually

- A circular array, however, allows the rear to wrap around to the beginning of the array once it reaches the end, as long as there's available space, making better use of the allocated space.
- Both the front and rear are **updated in a circular fashion**. This means that when either pointer reaches the end of the array, it wraps back to the beginning using the modulo operation.
  - `front = (front + 1) % array.length;`
  - `rear = (rear + 1) % array.length;`

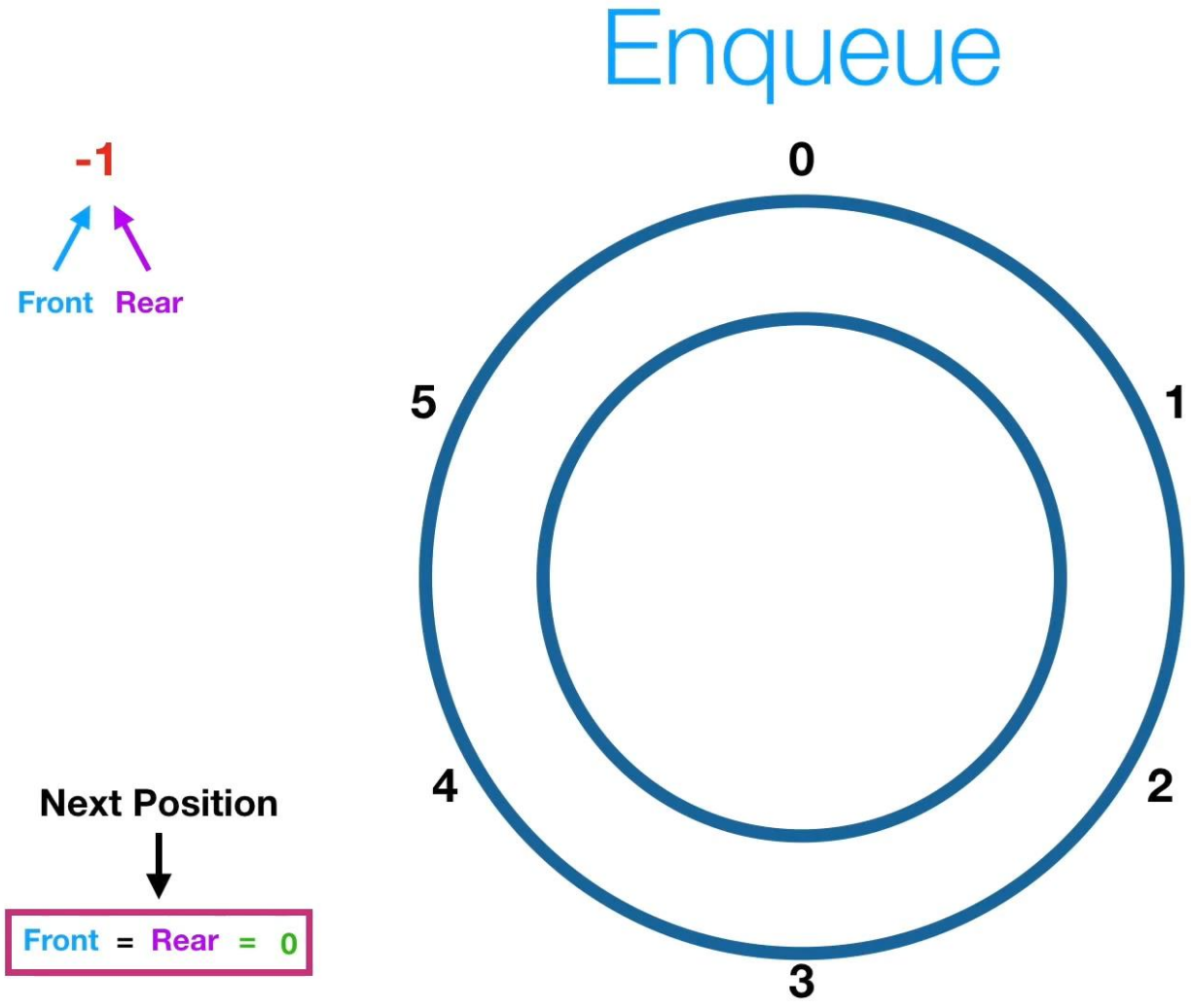
Remaining Queue Operations

- The resulting array is called a (conceptually) circular array.
- A circular array is an array that conceptually loops around itself.
    - The last index (n-1) is thought to precede index 0
    - The index after index n-1 is index 0
- The advantage of a circular array is that we do not need to shift data items when performing a dequeue operation.



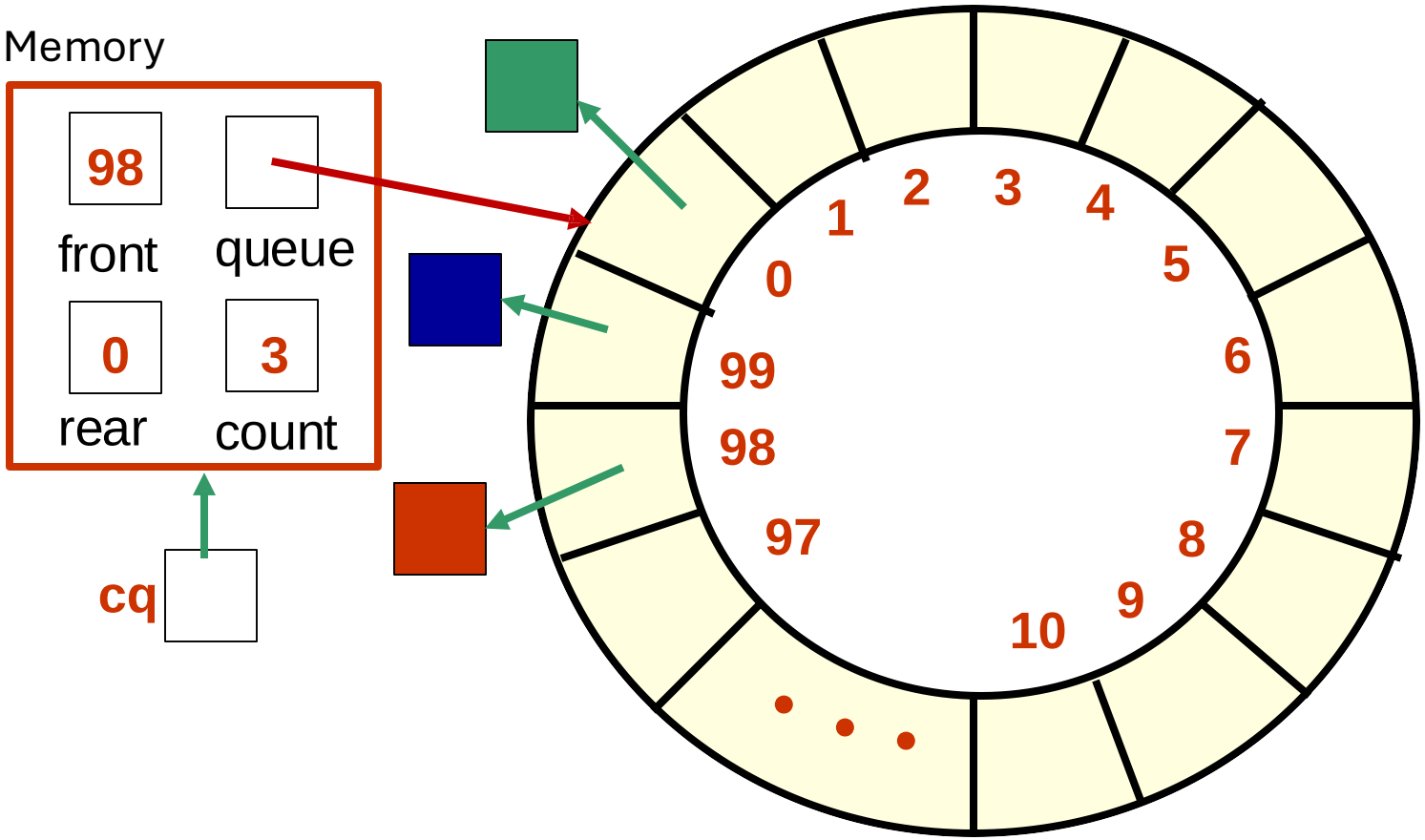

dequeue
front = 0
rear = 3
count = 4
front = 1
rear = 3
count = 3
0
1
2
3
4
5
6
7
8
9
10
n-1
n-2
n-3

Enqueue


-1
Front
Rear
0
1
2
3
4
5
Next Position
Front = Rear = 0
BLUE
TREE
CODE

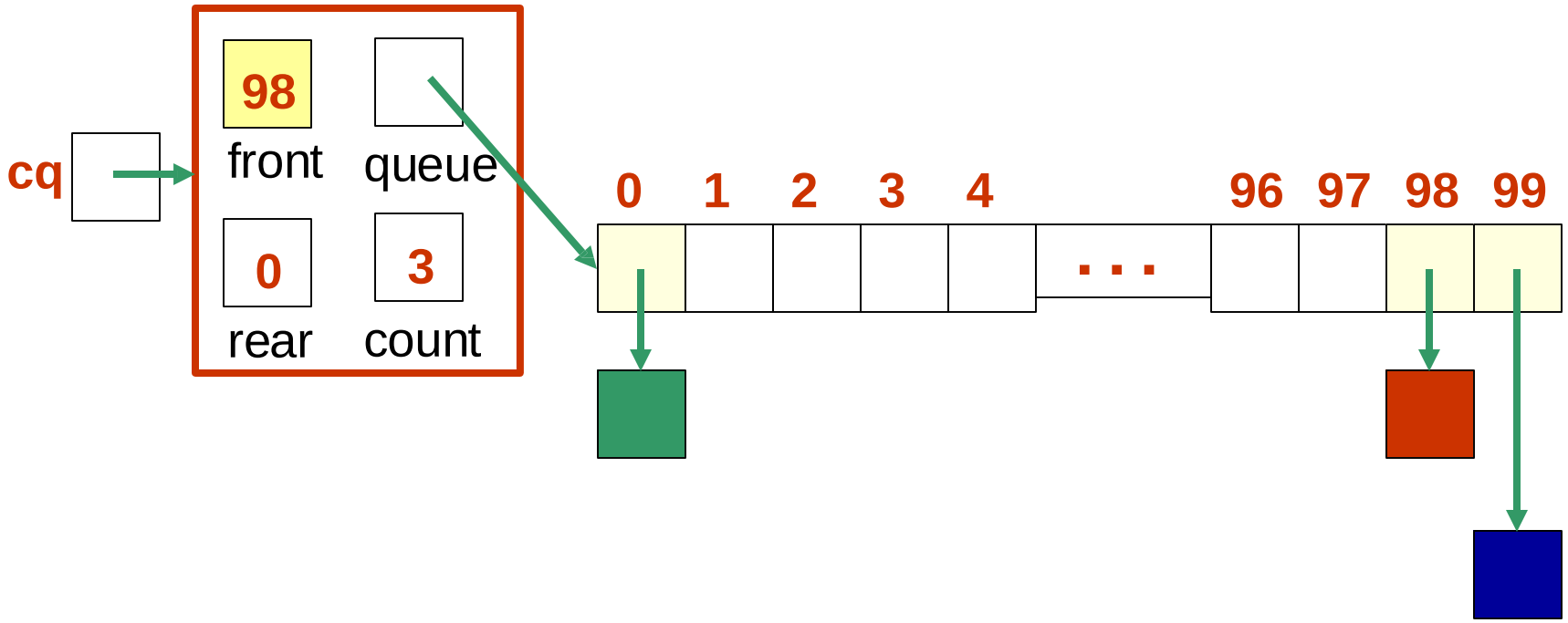
Circular Queues Explained

- If we reach the end of the array, the next data item is stored at index 0.



Circular Queue Drawn Linearly

Queue from the previous slide



Implementing a Circular Queue in Java

```
public class CircularArrayQueue<T> implements QueueADT<T> {
private final int DEFAULT_CAPACITY = 100;
private int count, front, rear;
private T[] queue;
```

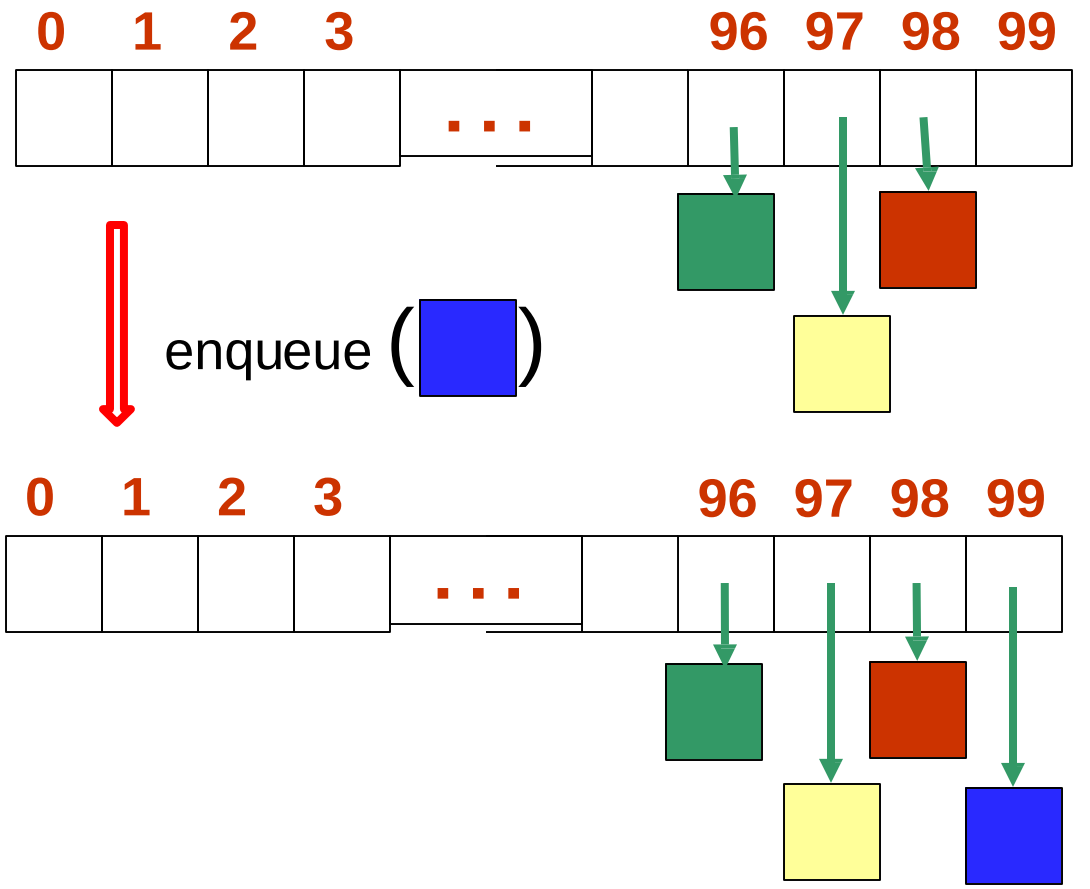
```
// Constructor with default capacity
public CircularArrayQueue() {
count = 0;
front = 0;
rear = -1;
queue = (T[])(new Object[DEFAULT_CAPACITY]);
}
```

Why is the rear initialized to -1?

```
// Constructor with a specified initial capacity
public CircularArrayQueue(int initialCapacity) {
count = 0;
front = 0;
rear = -1;
queue = (T[])(new Object[initialCapacity]);
}
```

Queue as a Circular Array

When an element is enqueued, the value of the rear is incremented, but if the rear reaches the last index of the array, it wraps around to the beginning (index 0) due to the circular array implementation.



Queue as a Circular Array

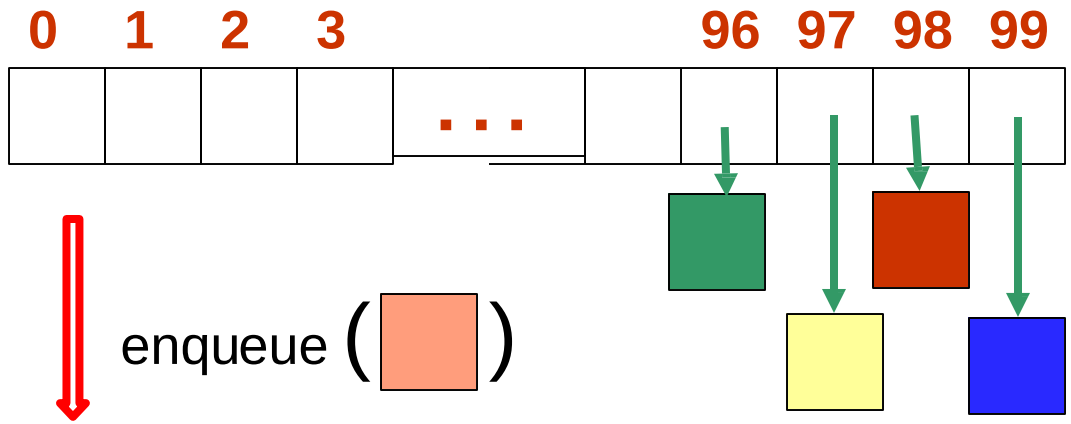
but it must consider the

need to loop back to

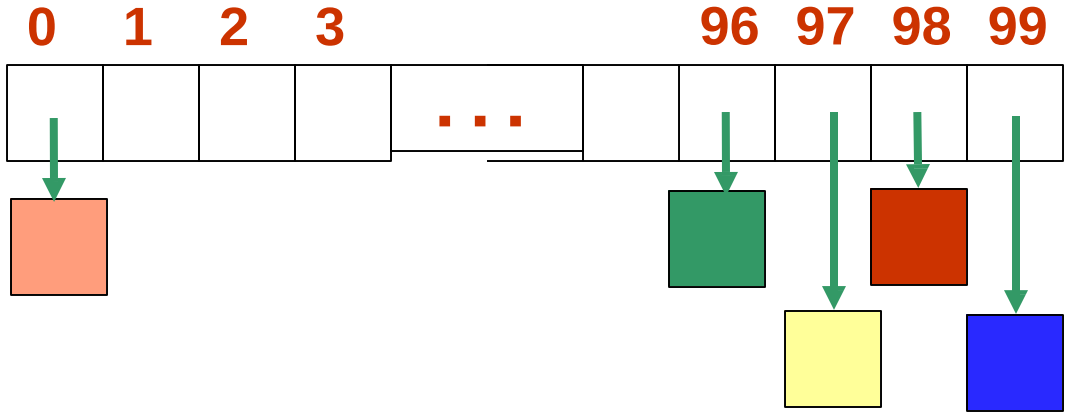
index 0: rear = (rear + 1)

% queue.length

front = 96
rear = 99



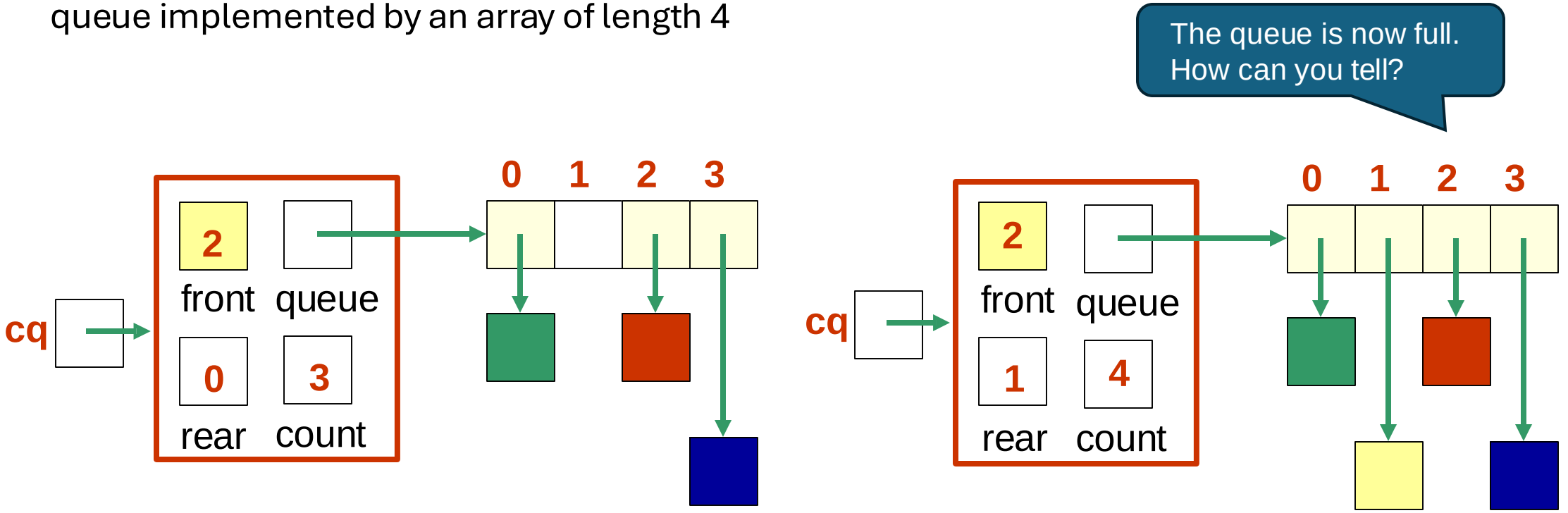
front = 96
rear = 0



Can this array implementation also reach capacity?

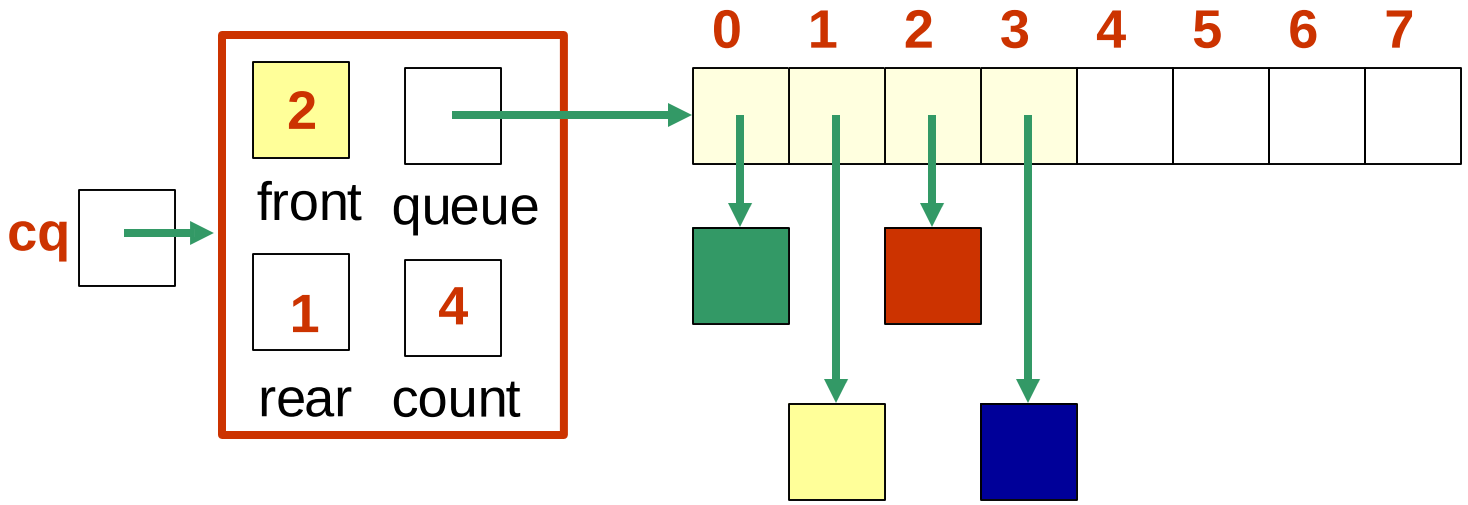
Identifying Capacity Limits in a Circular Queue

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



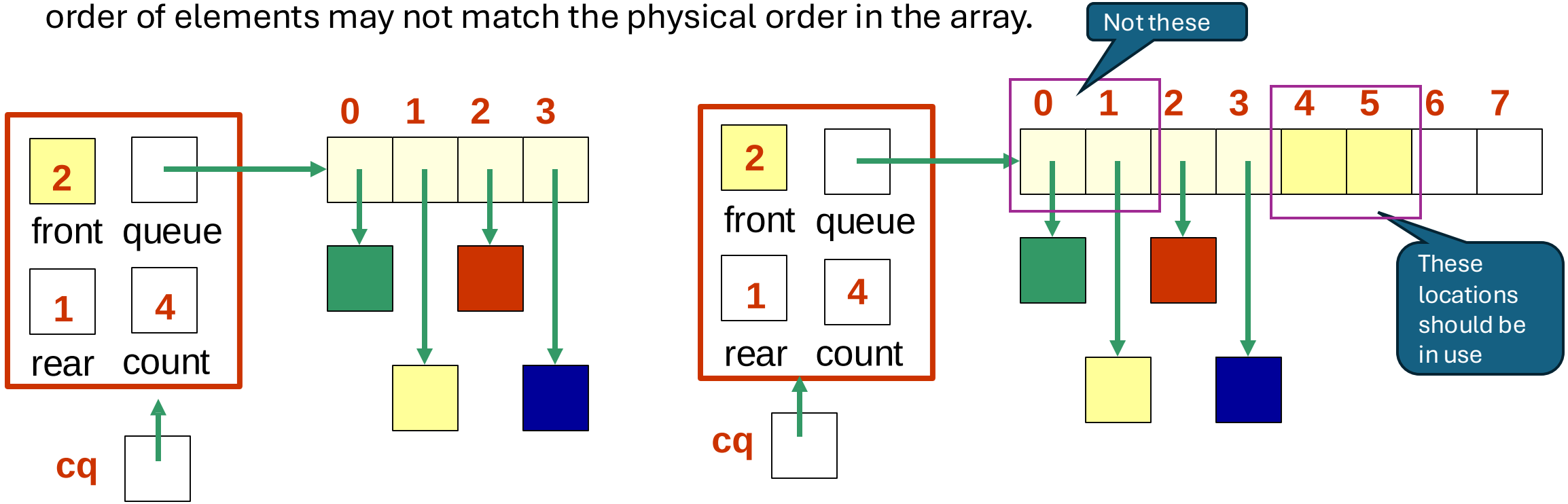
Challenges of Resizing a Circular Queue

- We can't just double the size of the array and copy values to the same positions as before because circular properties of the queue will be lost.



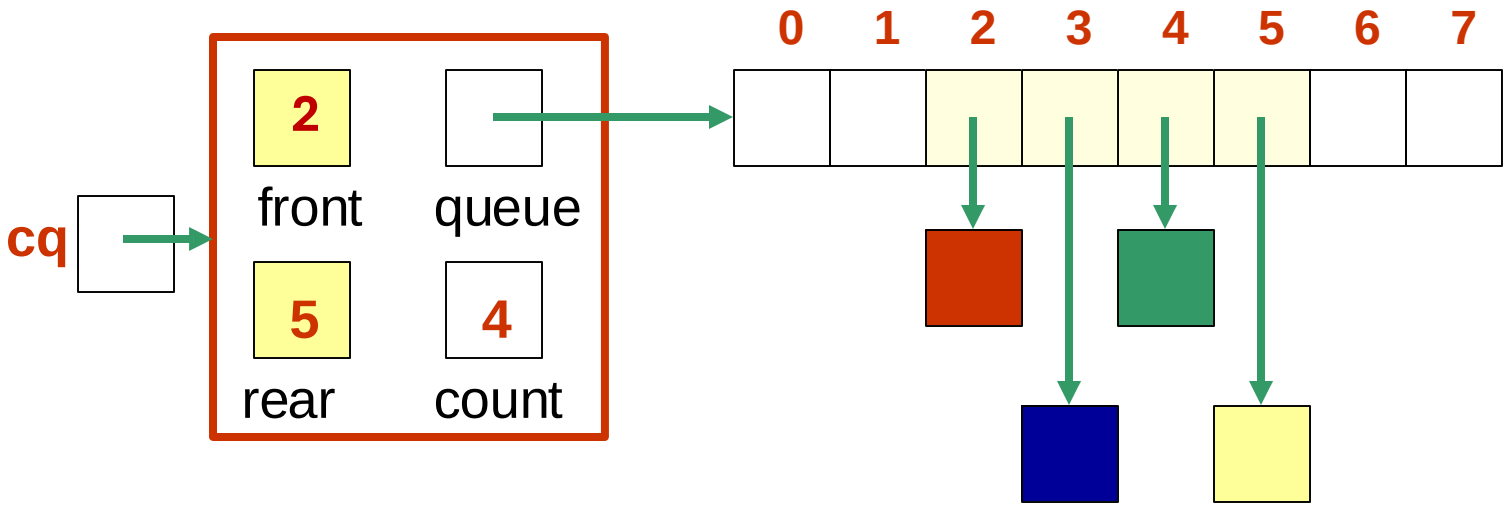
Challenges of Resizing a Circular Queue (cont.)

- When you double the size of a regular array, you can simply copy elements from the original array to the new, larger array in the same order. However, with a circular queue, the logical order of elements may not match the physical order in the array.



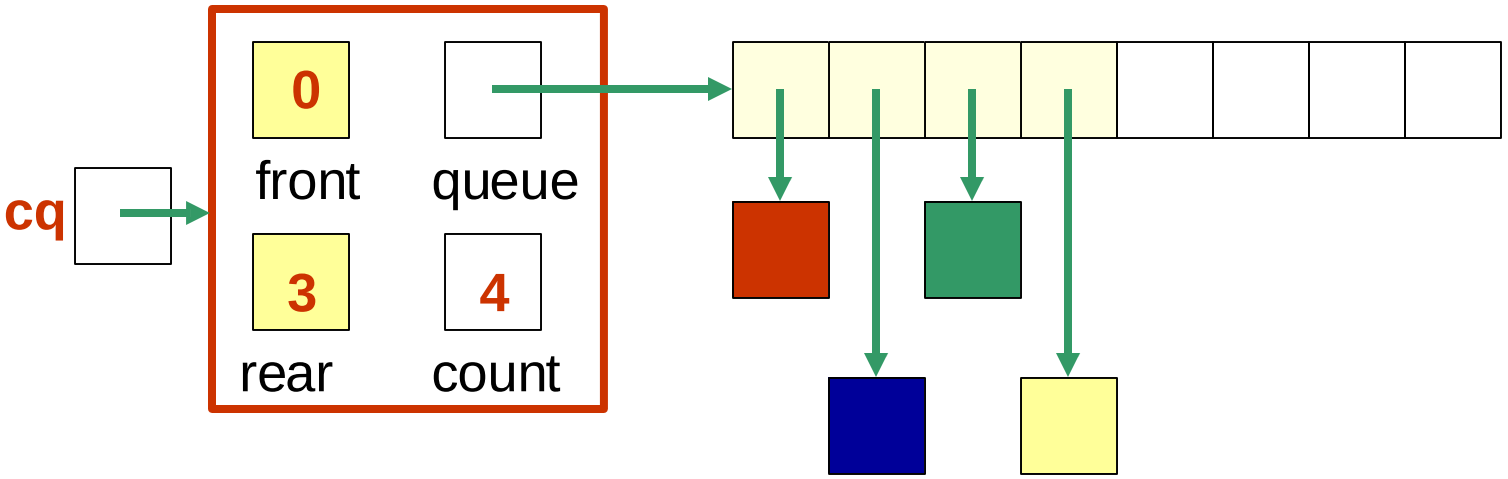
Challenges of Resizing a Circular Queue (cont.)

- We could build the new array and copy the queue data items into contiguous locations beginning at the index front:



Challenges of Resizing a Circular Queue (cont.)

- … or we could copy the queue data items to the beginning of the new array



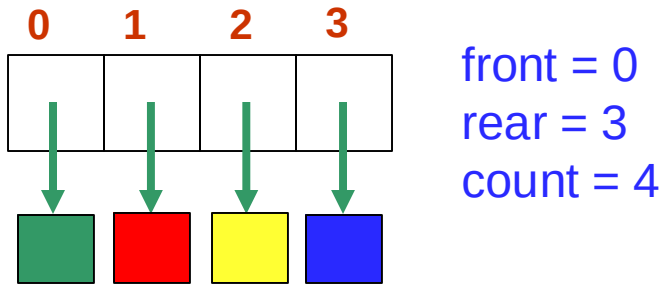
Algorithm Enqueue

In a circular queue, adding a new item involves adjusting pointers to maintain the circular structure, allowing elements to "wrap around" if the end of the array is reached.

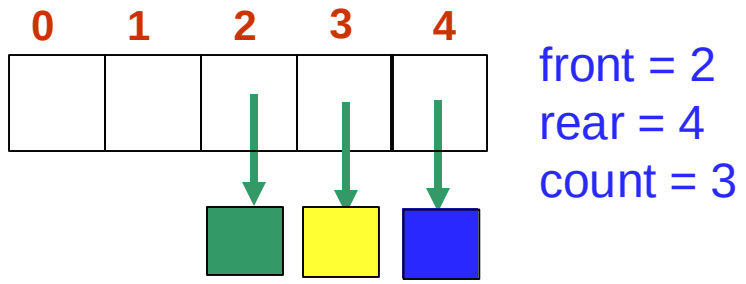
```
Algorithm enqueue(dataItem)
In: New data item to add to the queue
Out: Nothing, but dataItem is enqueued
    1. if count = length of array then
    expandQueue()
    2. rear = (rear + 1) modulo length of array
    queue[rear] = dataItem
    3. ++count
```

Need to consider two cases

1. The array is full

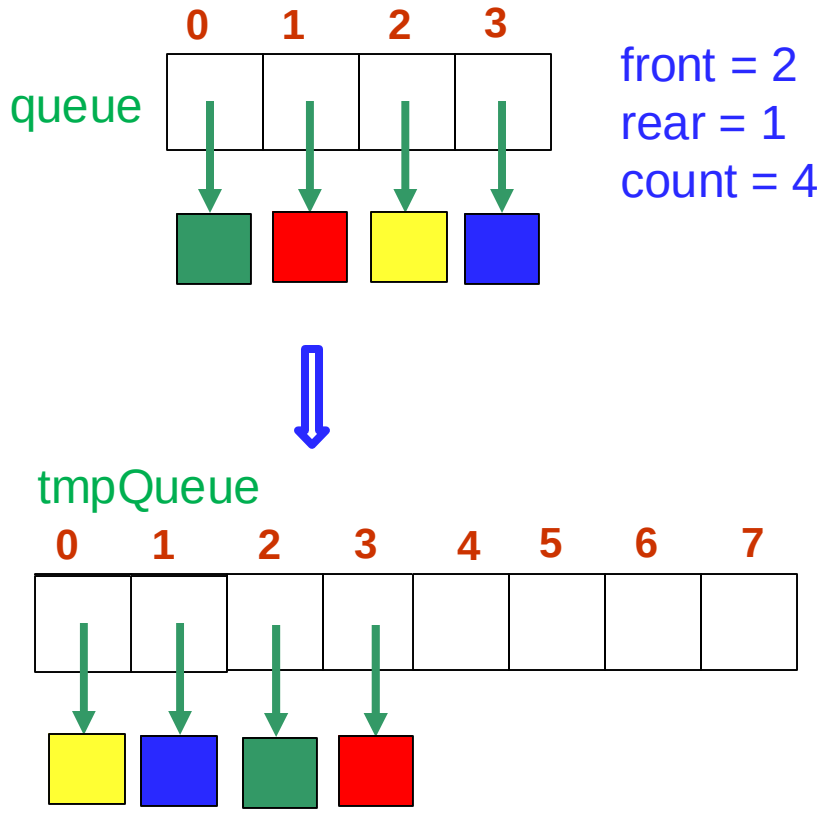


2. The array is not full



Algorithm Enqueue

```
Algorithm expandQueue()
Input: None (operates on the internal queue array, queue,
and related variables: front, rear, and count)
Output: None (but updates the queue to a larger array)
- Create a new array tmpQueue that is twice the size of the
  current queue.
- Copy each element from queue to tmpQueue in the correct
  order:
    - Start from front in queue.
    - Place each element into tmpQueue, starting from the
    beginning.
- Reset pointers: Set front to 0.
                  Set rear to count - 1.
- Replace queue with tmpQueue.
```



- Java code for expandQueue provided in the original course slides

Uses of Queues in Computing

Printer queue

Queue of documents to print

Uses of Queues in Computing

Keyboard input buffer

- A command is not processed until the user hits RETURN. The typed characters are stored in a queue.

```
javac test.java
```

Uses of Queues in Computing

GUI event queue (click on buttons, menu items)

Events are saved in a queue and processed when the computer is ready

Question!

In a circular array implementation of a queue, the variables front, rear, and count (the number of items in the queue) are tracked. Suppose the front is at index 0, and the rear is at the index that is one less than the **current capacity** of the array. What can be determined about the value of count?

A) count must be zero.

B) count must be equal to the current capacity.

C) count could be either zero or equal to the capacity, but no other values are possible.

D) None of the above.

Question!

- Consider the following code executed on a circular array-based integer queue:

```
IntQueue q = new IntQueue();
q.insert(1);
q.insert(2);
q.insert(3);
System.out.println(q.getFront());
```

- Suppose q is represented by a circular array with a capacity of 5. After executing the above code, what will be the state of the front, rear, and data array?

A) front = 0, rear = 3, data = [1, 2, 3, _, _]

B) front = 1, rear = 3, data = [_, 1, 2, 3, _]

C) front = 0, rear = 2, data = [1, 2, _, _, _]

D) front = 1, rear = 4, data = [_, 1, 2, 3, 4]


thank you