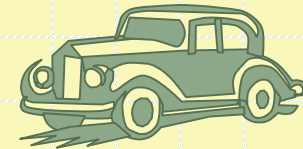
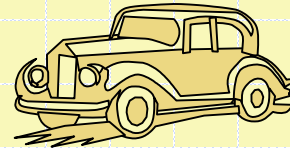
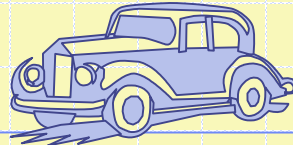


Queues



Dr. Aiman Hanna

**Department of Computer Science & Software Engineering
Concordia University, Montreal, Canada**

These slides have been extracted, modified and updated from original slides of :

Data Structures and Algorithms in Java, 5th edition. John Wiley & Sons, 2010. ISBN 978-0-470-38326-1.

Data Structures and the Java Collections Framework by William J. Collins, 3rd edition, ISBN 978-0-470-48267-4.

Both books are published by Wiley.

Copyright © 2010-2011 Wiley

Copyright © 2010 Michael T. Goodrich, Roberto Tamassia

Copyright © 2011 William J. Collins

Copyright © 2011-2024 Aiman Hanna

All rights reserved

The Queue ADT

- ❑ The **Queue** ADT stores arbitrary objects.
- ❑ Insertions and deletions follow the first-in first-out (FIFO) scheme.
- ❑ Insertions are at the rear of the queue and removals are at the front of the queue.
- ❑ Main queue operations:
 - **enqueue**(object): inserts an element at the end of the queue
 - object **dequeue**(): removes and returns the element at the front of the queue

The Queue ADT

- ❑ Auxiliary queue operations:
 - object **front()**: returns the element at the front without removing it.
 - integer **size()**: returns the number of elements stored.
 - boolean **isEmpty()**: indicates whether no elements are stored.
- ❑ Exceptions
 - Attempting the execution of dequeue or front on an empty queue throws an **EmptyQueueException**

Example

<i>Operation</i>	<i>Output</i>	<i>Q Contents</i>
enqueue(5)	--	[5]
enqueue(3)	--	[5, 3]
dequeue()	5	[3]
enqueue(7)	--	[3, 7]
dequeue()	3	[7]
front()	7	[7]
dequeue()	7	[]
dequeue()	"error"	[]
enqueue(9)	--	[9]
isEmpty()	false	[9]
size()	1	[9]
dequeue()	9	[]

Applications of Queues

□ Direct applications

- Waiting lists, bureaucracy
- Access to shared resources (e.g., printer)
- Multiprogramming

□ Indirect applications

- Auxiliary data structure for algorithms
- Component of other data structures

Array-based Queue

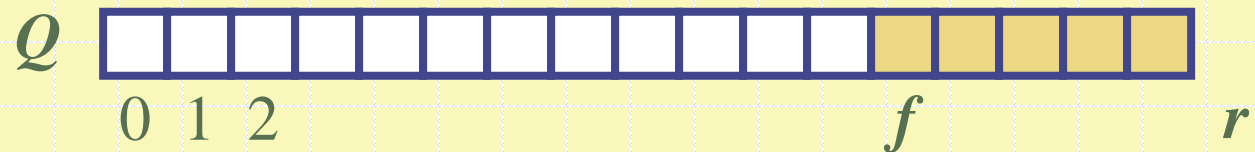
- ❑ Can be implemented using an array, Q , of fixed capacity.
- ❑ Use an array of size N in a circular fashion.
- ❑ We can let $Q[0]$ be the front of the queue, however this is inefficient since each `dequeue()` operation would result in moving all remaining elements forward.
- ❑ That is, each `dequeue()` operation would have a complexity of $O(n)$.

Array-based Queue

- Instead, two variables keep track of the front and rear:
 - f index of the front element
 - r index of the next available element in the array (that is the one immediately past the rear/last element)
- Initially, we assign $f = r = 0$, which indicates that the queue is empty (generally $f = r$ indicates empty queue).
- Index r is kept empty. Insertion is made into $Q[r]$ then r is incremented.

Array-based Queue

- This configuration would allow enqueue(), dequeue() and front() to be performed in constant time, that is $O(1)$.
- However, this configuration has a serious problem.
- For example, if we repeatedly enqueue and dequeue a single element N times, then we end up with assign $f = r = N$ (indicating an empty queue, which is correct). Below is another example.

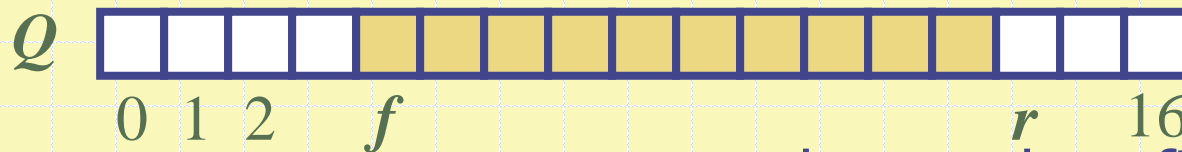


- While the array clearly has many empty elements, an insertion cannot be made as it would result in array-out-of-bounds error.

Array-based Queue

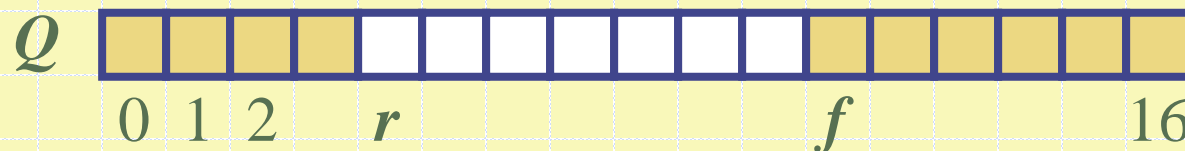
- ❑ Instead of having this normal configuration, we can let the indices f and r wrap around the end of the array.
- ❑ That is view the array as a “circular array” that goes from $Q[0]$ to $Q[N - 1]$ then back to $Q[0]$.
- ❑ Notice that Index r is kept empty, which means that the queue can hold a maximum of $N-1$ elements.

normal configuration



- ❑ For example, $N=17$, queue can hold a maximum of 16 elements.

wrapped-around configuration



Queue Operations

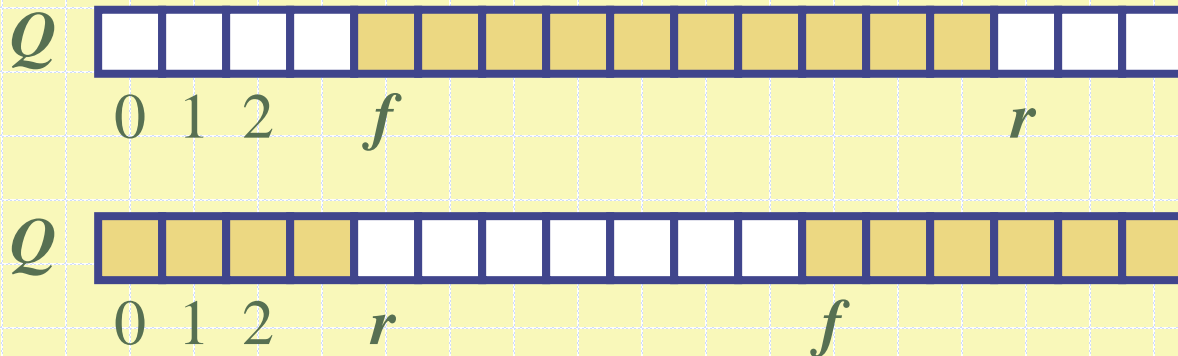
- We use the modulo operator (remainder of division)

Algorithm *size()*

return $((N - f) + r) \bmod N$

Algorithm *isEmpty()*

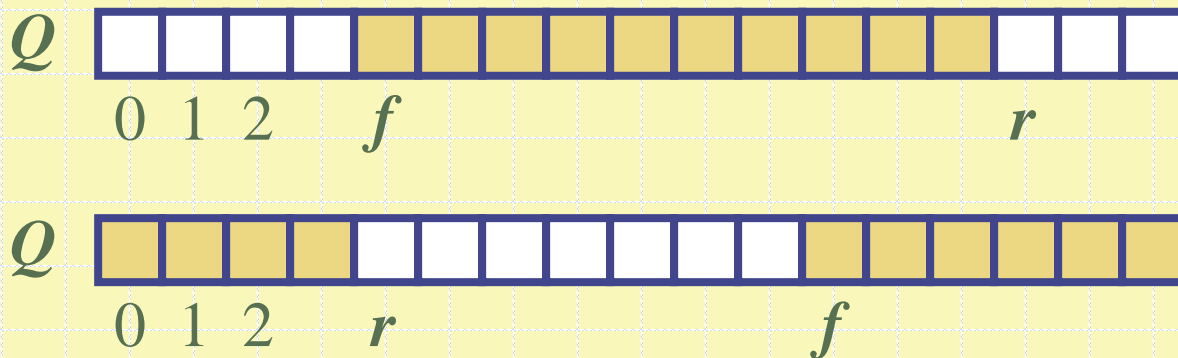
return $(f = r)$



Queue Operations (cont.)

- ❑ Operation enqueue throws an exception if the array is full
- ❑ This exception is implementation-dependent

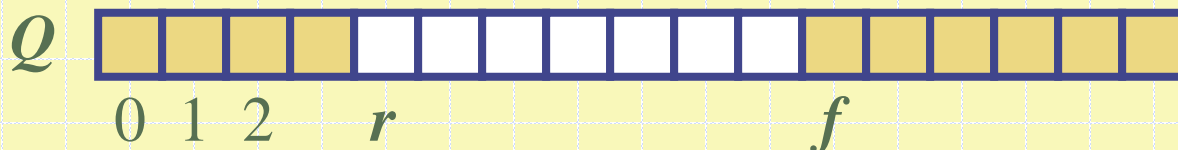
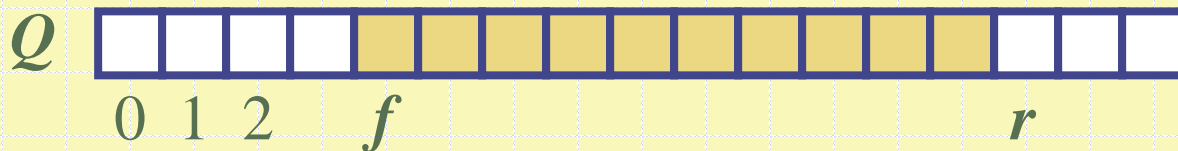
```
Algorithm enqueue(o)  
  if  $size() = N - 1$  then  
    throw FullQueueException  
  else  
     $Q[r] \leftarrow o$   
     $r \leftarrow (r + 1) \bmod N$ 
```



Queue Operations (cont.)

- ❑ Operation `dequeue` throws an exception if the queue is empty
- ❑ This exception is specified in the queue ADT

```
Algorithm dequeue()  
  if isEmpty() then  
    throw EmptyQueueException  
  else  
     $o \leftarrow Q[f]$   
     $f \leftarrow (f + 1) \bmod N$   
  return  $o$ 
```



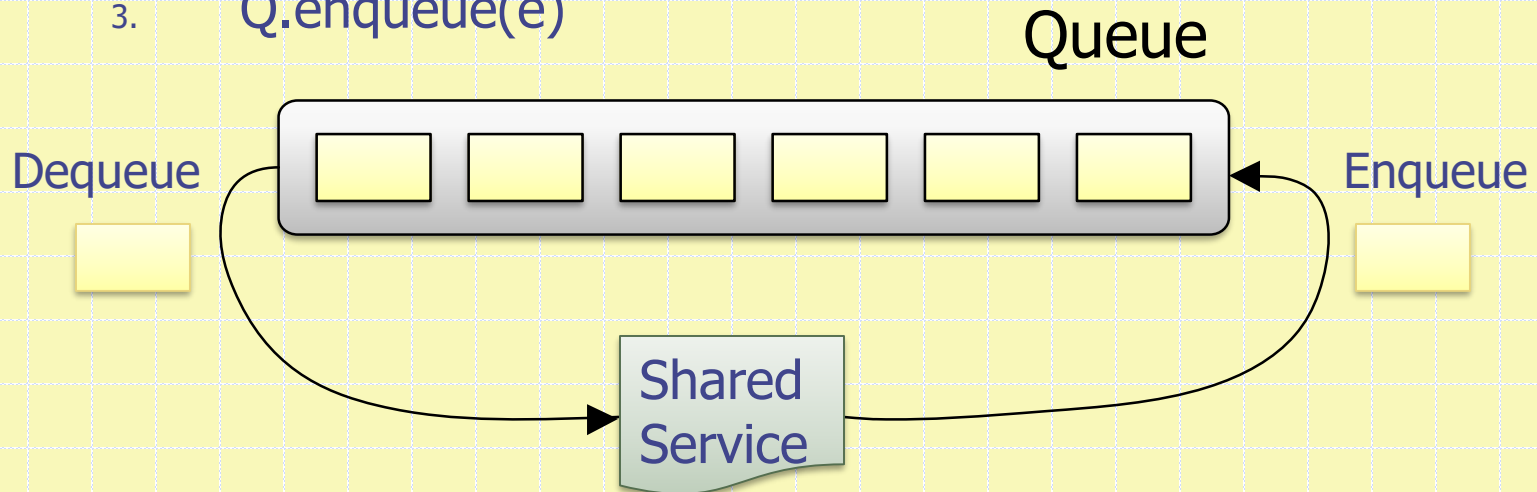
Queue Interface

- ❑ Java interface
corresponding to
our Queue ADT
(Note: not identical to the
shown interface)
- ❑ Requires the
definition of class
`EmptyQueueException`
- ❑ There is no
corresponding
built-in Java class

```
public interface Queue<E> {  
    public int size();  
    public boolean isEmpty();  
    public E front()  
        throws EmptyQueueException;  
    public void enqueue(E element);  
    public E dequeue()  
        throws EmptyQueueException;  
}
```

Application: Round Robin Schedulers

- We can implement a round robin scheduler using a queue Q by repeatedly performing the following steps:
 1. $e = Q.dequeue()$
 2. Service element e
 3. $Q.enqueue(e)$



Growable Array-based Queue

- ❑ In an enqueue() operation, when the array is full, we can replace the array with a larger one instead of throwing an exception.
- ❑ Similar to what have been explained for growable array-based stack, the enqueue() operation has amortized running time of:
 - ***$O(n)$ with the incremental strategy***
 - ***$O(1)$ with the doubling strategy***

Double-Ended Queues

- ❑ A *double-ended queue* (dequeue, or D.Q.) ADT is richer than the stack ADT and the queue ADT.
- ❑ It supports insertion and deletion at both ends.
- ❑ Elements can only be added to or removed from the front (head) or the back (tail).

Double-Ended Queues

- The fundamental operations allowed by such queue are:
 - **addFirst(*e*)**: Insert as new element at the head of the queue.
 - **addLast(*e*)**: Insert as new element at the tail of the queue.
 - **removeFirst()**: Remove and return the first element of the D.Q., or an error if the queue is empty.
 - **removeLast()**: Remove and return the last element of the D.Q., or an error if the queue is empty.
- Other operations may include: **getFirst()**, **getLast()**, **size()** and **isEmpty()**

Example

<i>Operation</i>	<i>Output</i>	<i>D.Q. Contents</i>
addFirst(3)	--	[3]
addFirst(5)	--	[5, 3]
removeFirst()	5	[3]
addLast(7)	--	[3, 7]
addLast(9)	--	[3, 7, 9]
addLast(12)	--	[3, 7, 9, 12]
addFirst(8)	--	[8, 3, 7, 9, 12]
removeLast()	12	[8, 3, 7, 9]
removeFirst()	8	[3, 7, 9]
isEmpty()	<i>false</i>	[3, 7, 9]
addFirst(6)	--	[6, 3, 7, 9]
size()	4	[6, 3, 7, 9]