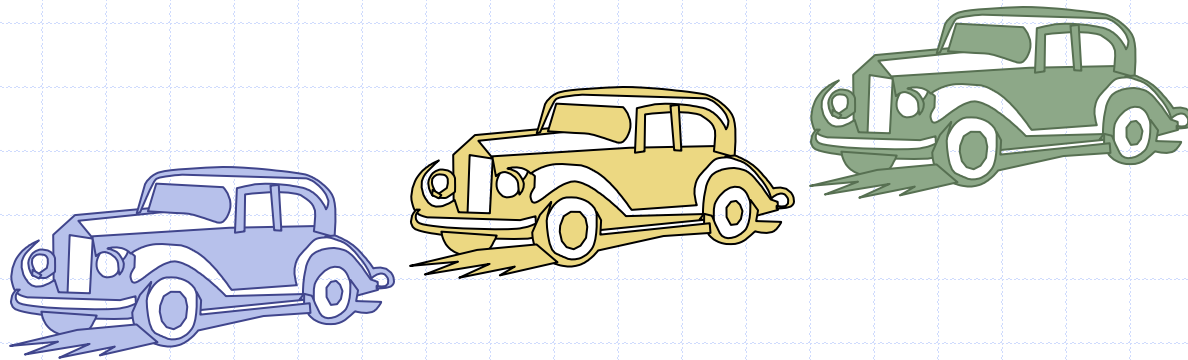


Presentation for use with the textbook **Data Structures and Algorithms in Java, 6th edition**, by M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, Wiley, 2014

Queues



The Queue ADT

- The **Queue** ADT stores arbitrary objects
- Insertions and deletions follow the first-in first-out 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**
- Auxiliary queue operations:
 - object **first**(): 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
- Boundary cases:
 - Attempting the execution of dequeue or first on an empty queue returns **null**

Example

<i>Operation</i>		<i>Output</i>	<i>Q</i>
enqueue(5)	–	(5)	
enqueue(3)	–	(5, 3)	
dequeue()	5	(3)	
enqueue(7)	–	(3, 7)	
dequeue()	3	(7)	
first()	7	(7)	
dequeue()	7	()	
dequeue()	<i>null</i>	()	
isEmpty()	<i>true</i>	()	
enqueue(9)	–	(9)	
enqueue(7)	–	(9, 7)	
size()	2	(9, 7)	
enqueue(3)	–	(9, 7, 3)	
enqueue(5)	–	(9, 7, 3, 5)	
dequeue()	9	(7, 3, 5)	

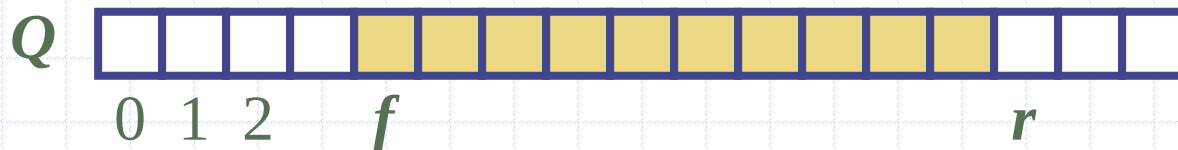
Applications of Queues

- 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

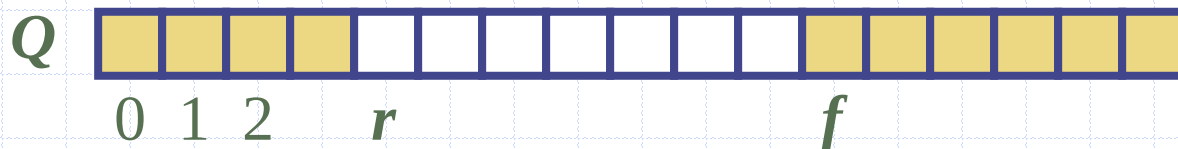
Array-based Queue

- Use an array of size N in a circular fashion
- Two variables keep track of the front and size
 - f index of the front element
 - sz number of stored elements
- When the queue has fewer than N elements, array location $r = (f + sz) \bmod N$ is the first empty slot past the rear of the queue

normal configuration



wrapped-around configuration

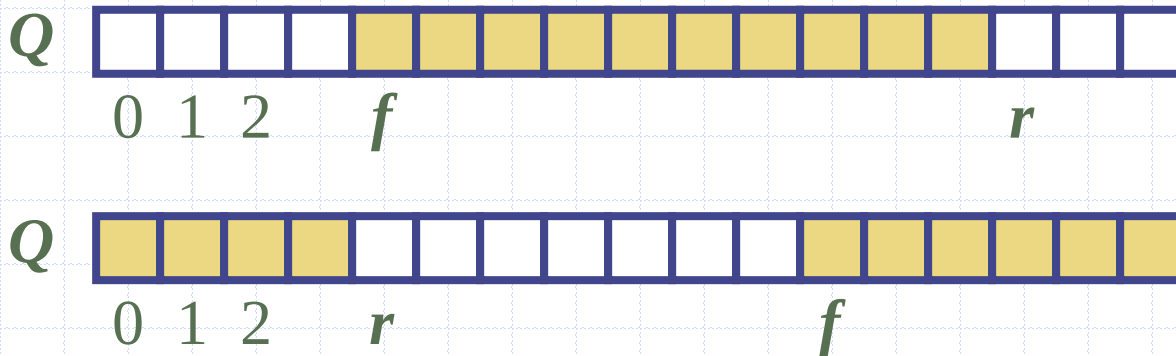


Queue Operations

- We use the modulo operator (remainder of division)

Algorithm *size()*
return *sz*

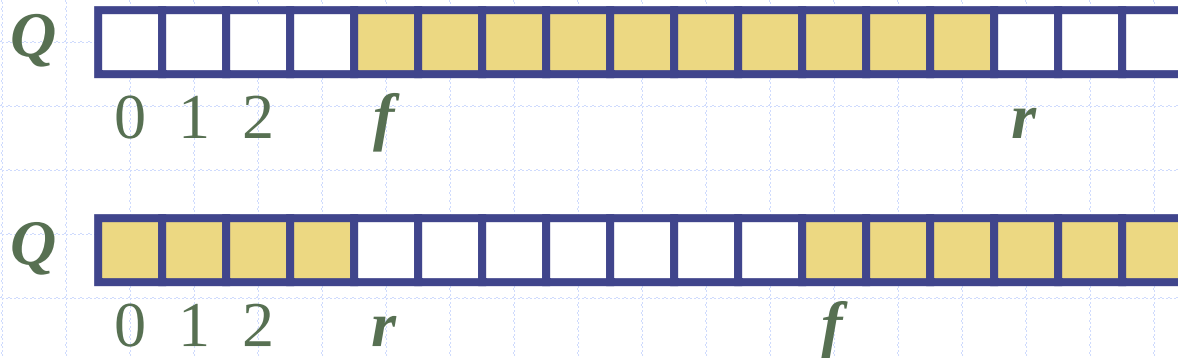
Algorithm *isEmpty()*
return (*sz* == 0)



Queue Operations (cont.)

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

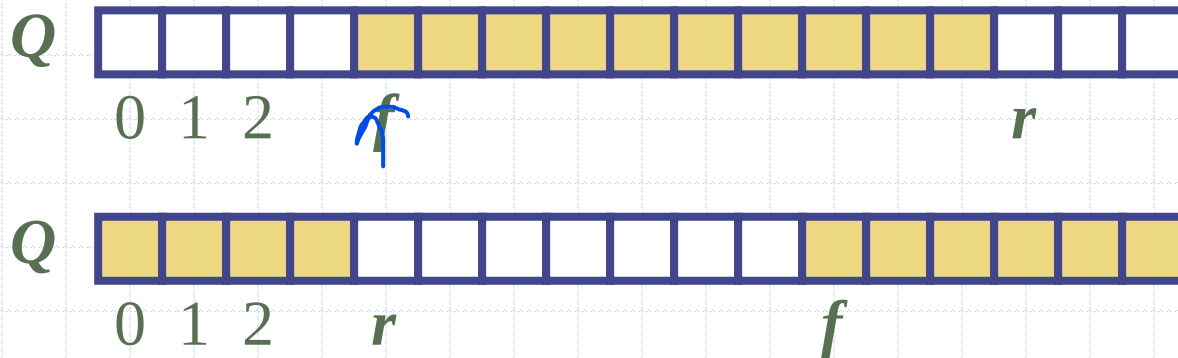
```
Algorithm enqueue(o)  
  if  $size() = N - 1$  then  
    throw IllegalStateException  
  else  
     $r \leftarrow (f + sz) \bmod N$   
     $Q[r] \leftarrow o$   
     $sz \leftarrow (sz + 1)$ 
```



Queue Operations (cont.)

- Note that operation `dequeue` returns null if the queue is empty

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



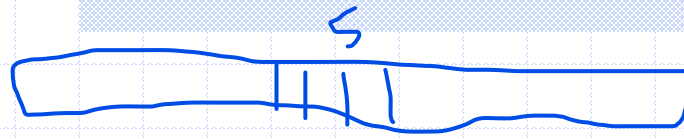
Queue Interface in Java

- ❑ Java interface corresponding to our Queue ADT
- ❑ Assumes that `first()` and `dequeue()` return null if queue is empty

```
public interface Queue<E> {  
    int size();  
    boolean isEmpty();  
    E first();  
    void enqueue(E e);  
    E dequeue();  
}
```

Array-based Implementation

```
1  /** Implementation of the queue ADT using a fixed-length array. */
2  public class ArrayQueue<E> implements Queue<E> {
3      // instance variables
4      private E[] data;                // generic array used for storage
5      private int f = 0;                // index of the front element
6      private int sz = 0;               // current number of elements
7
8      // constructors
9      public ArrayQueue() {this(CAPACITY);} // constructs queue with default capacity
10     public ArrayQueue(int capacity) {      // constructs queue with given capacity
11         data = (E[]) new Object[capacity]; // safe cast; compiler may give warning
12     }
13
14     // methods
15     /** Returns the number of elements in the queue. */
16     public int size() { return sz; }
17
18     /** Tests whether the queue is empty. */
19     public boolean isEmpty() { return (sz == 0); }
20
```



Array-based Implementation (2)

```
21  /** Inserts an element at the rear of the queue. */
22  public void enqueue(E e) throws IllegalStateException {
23      if (sz == data.length) throw new IllegalStateException("Queue is full");
24      int avail = (f + sz) % data.length;    // use modular arithmetic
25      data[avail] = e;
26      sz++;
27  }
28
29  /** Returns, but does not remove, the first element of the queue (null if empty). */
30  public E first() {
31      if (isEmpty()) return null;
32      return data[f];
33  }
34
35  /** Removes and returns the first element of the queue (null if empty). */
36  public E dequeue() {
37      if (isEmpty()) return null;
38      E answer = data[f];
39      data[f] = null;                // dereference to help garbage collection
40      f = (f + 1) % data.length;
41      sz--;
42      return answer;
43  }
```

Comparison to java.util.Queue

- Our Queue methods and corresponding methods of **java.util.Queue**:

Our Queue ADT	Interface java.util.Queue	
	throws exceptions	returns special value
enqueue(<i>e</i>)	add(<i>e</i>)	offer(<i>e</i>)
dequeue()	remove()	poll()
first()	element()	peek()
size()	size()	
isEmpty()	isEmpty()	

Analyzing the Efficiency of an Array-Based Queue

Method	Running Time
size	$O(1)$
isEmpty	$O(1)$
first	$O(1)$
enqueue	$O(1)$
dequeue	$O(1)$

Table 6.4: Performance of a queue realized by an array. The space usage is $O(N)$, where N is the size of the array, determined at the time the queue is created, and independent from the number $n < N$ of elements that are actually in the queue.

Implementing a Queue with a Singly Linked List

- As we did for the stack ADT, we can adapt a singly linked list to implement the queue ADT while supporting worst-case $O(1)$ -time for all operations, and without any artificial limit on the capacity.
- The natural orientation for a queue is to align the front of the queue with the front of the list, and the back of the queue with the tail of the list, because the only update operation that singly linked lists support at the back end is an insertion.
- Our Java implementation of a `LinkedList` class is given in Code 6.11.

Implementing a Queue with a Singly Linked List

```
1 /** Realization of a FIFO queue as an adaptation of a SinglyLinkedList. */
2 public class LinkedQueue<E> implements Queue<E> {
3     private SinglyLinkedList<E> list = new SinglyLinkedList<>( ); // an empty list
4     public LinkedQueue( ) { } // new queue relies on the initially empty list
5     public int size( ) { return list.size( ); }
6     public boolean isEmpty( ) { return list.isEmpty( ); }
7     public void enqueue(E element) { list.addLast(element); }
8     public E first( ) { return list.first( ); }
9     public E dequeue( ) { return list.removeFirst( ); }
10 }
```

Code Fragment 6.11: Implementation of a Queue using a SinglyLinkedList.

A Circular Queue

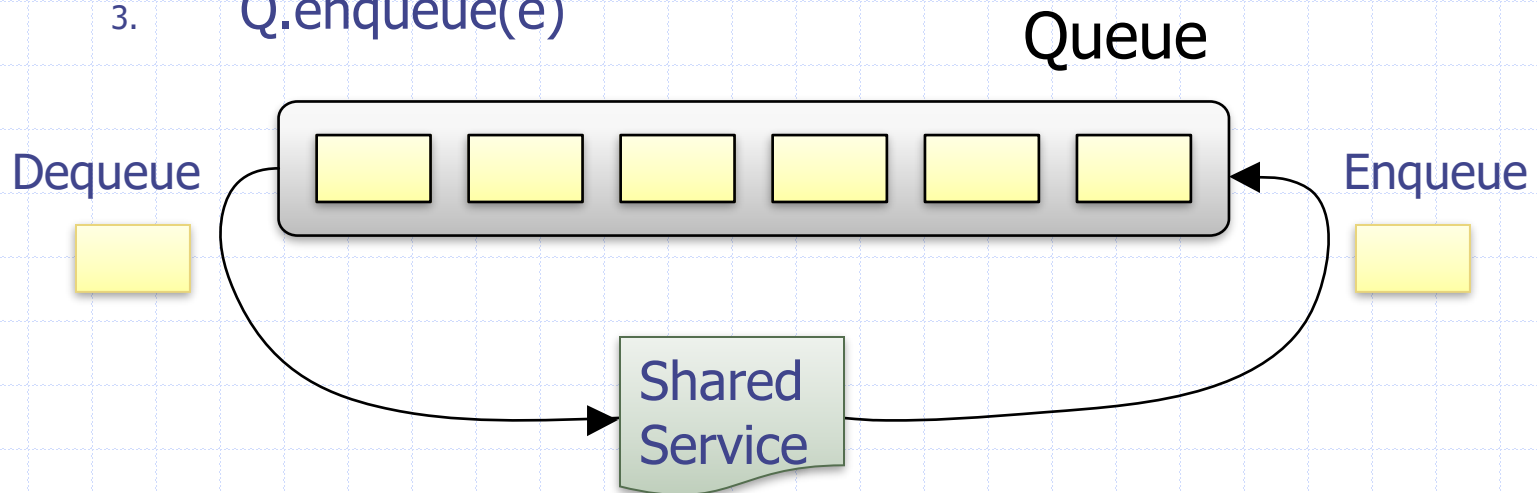
- In Section 3.3, we implemented a ***circularly linked list*** class that supports all behaviors of a singly linked list, and an additional rotate() method that efficiently moves the first element to the end of the list.
- We can generalize the Queue interface to define a new CircularQueue interface with such a behavior, as shown in Code Fragment 6.12.

```
1 public interface CircularQueue<E> extends Queue<E> {  
2     /**  
3     * Rotates the front element of the queue to the back of the queue.  
4     * This does nothing if the queue is empty.  
5     */  
6     void rotate( );  
7 }
```

Code Fragment 6.12: A Java interface, CircularQueue, that extends the Queue ADT with a new rotate() method.

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. $Q.enqueue(e)$



6.3 Double-Ended Queues

- We next consider a queue-like data structure that supports insertion and deletion at both the front and the back of the queue. Such a structure is called a ***double-ended queue***
- The deque abstract data type is more general than both the stack and the queue ADTs. The extra generality can be useful in some applications.

The Deque Abstract Data Type

- **The deque abstract data type is more general than both the stack and the queue ADTs. The extra generality can be useful in some applications.**
- **The deque abstract data type is richer than both the stack and the queue ADTs. To provide a symmetrical abstraction,**

The Deque Abstract Data Type

- The deque ADT is defined to support the following update methods:

addFirst(<i>e</i>):	Insert a new element <i>e</i> at the front of the deque.
addLast(<i>e</i>):	Insert a new element <i>e</i> at the back of the deque.
removeFirst():	Remove and return the first element of the deque (or null if the deque is empty).
removeLast():	Remove and return the last element of the deque (or null if the deque is empty).

Additionally, the deque ADT will include the following accessors:

first():	Returns the first element of the deque, without removing it (or null if the deque is empty).
last():	Returns the last element of the deque, without removing it (or null if the deque is empty).
size(): Returns the number of elements in the deque.	
isEmpty(): Returns a boolean indicating whether the deque is empty.	

The deque ADT with the Java interface

```
1 /**
2  * Interface for a double-ended queue: a collection of elements that can be inserted
3  * and removed at both ends; this interface is a simplified version of java.util.Deque.
4  */
5 public interface Deque<E> {
6     /** Returns the number of elements in the deque. */
7     int size( );
8     /** Tests whether the deque is empty. */
9     boolean isEmpty( );
10    /** Returns, but does not remove, the first element of the deque (null if empty). */
11    E first( );
12    /** Returns, but does not remove, the last element of the deque (null if empty). */
13    E last( );
14    /** Inserts an element at the front of the deque. */
15    void addFirst(E e);
16    /** Inserts an element at the back of the deque. */
17    void addLast(E e);
18    /** Removes and returns the first element of the deque (null if empty). */
19    E removeFirst( );
20    /** Removes and returns the last element of the deque (null if empty). */
21    E removeLast( );
22 }
```

Code Fragment 6.14: A Java interface, Deque, describing the double-ended queue ADT. Note the use of the generic parameterized type, E, allowing a deque to contain elements of any specified class..

Implementing a Deque

- We can implement the deque ADT efficiently using either
 - an array or
 - a linked listfor storing elements.

Implementing a Deque with a Circular Array

- If using an array, we recommend a representation similar to the ArrayQueue class, treating the array in circular fashion and storing
 - the index of the first element and
 - the current size of the deque as fields;
- the index of the last element can be calculated, as needed, using modular arithmetic.
- One extra concern is avoiding use of negative values with the modulo operator.
- When removing the first element, the front index is advanced in circular fashion, with the assignment $f = (f+1) \% N$.
- But when an element is inserted at the front, the first index must effectively be decremented in circular fashion and it is a mistake to assign $f = (f-1) \% N$.
- The problem is that when f is 0, the goal should be to “decrement” it to the other end of the array, and thus to index $N-1$.
- However, a calculation such as $-1 \% 10$ in Java results in the value -1 . A standard way to decrement an index circularly is instead to assign $f = (f-1+N) \% N$.
- Adding the additional term of N before the modulus is calculated assures that the result is a positive value.

Implementing a Deque with a Doubly Linked List

- Because the deque requires insertion and removal at both ends, a doubly linked list is most appropriate for implementing all operations efficiently.
- In fact, the `DoublyLinkedList` class from Section 3.4.1 already implements the entire `Deque` interface; we simply need to add the declaration `implements Deque<E>` to that class definition in order to use it as a deque.

Performance of the Deque Operations

- Table 6.5 shows the running times of methods for a deque implemented with a doubly linked list.
- Note that every method runs in $O(1)$ time.

Method	Running Time
size, isEmpty	$O(1)$
first, last	$O(1)$
addFirst, addLast	$O(1)$
removeFirst, removeLast	$O(1)$

Table 6.5: Performance of a deque realized by either a circular array or a doubly linked list. The space usage for the array-based implementation is $O(N)$, where N is the size of the array, while the space usage of the doubly linked list is $O(n)$ where $n < N$ is the actual number of elements in the deque.

Dequeues in the Java Collections Framework

- The Java Collections Framework includes its own definition of a deque, as the `java.util.Deque` interface
- The Java Collections Framework also includes several implementations of the interface including:
 - one based on use of a circular array (`java.util.ArrayDeque`) and
 - one based on use of a doubly linked list (`java.util.LinkedList`).
- So, if we need to use a deque and would rather not implement one from scratch, we can simply use one of those built-in classes.
- As is the case with the `java.util.Queue` class, the `java.util.Deque` provides duplicative methods that use different techniques to signal exceptional cases. A summary of those methods is given in Table 6.6.

Dequeues in the Java Collections Framework

Our Deque ADT	Interface <code>java.util.Deque</code>	
	throws exceptions	returns special value
<code>first()</code>	<code>getFirst()</code>	<code>peekFirst()</code>
<code>last()</code>	<code>getLast()</code>	<code>peekLast()</code>
<code>addFirst(<i>e</i>)</code>	<code>addFirst(<i>e</i>)</code>	<code>offerFirst(<i>e</i>)</code>
<code>addLast(<i>e</i>)</code>	<code>addLast(<i>e</i>)</code>	<code>offerLast(<i>e</i>)</code>
<code>removeFirst()</code>	<code>removeFirst()</code>	<code>pollFirst()</code>
<code>removeLast()</code>	<code>removeLast()</code>	<code>pollLast()</code>
<code>size()</code>	<code>size()</code>	
<code>isEmpty()</code>	<code>isEmpty()</code>	

Table 6.6: Methods of our deque ADT and the corresponding methods of the `java.util.Deque` interface.