Queues

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
- 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

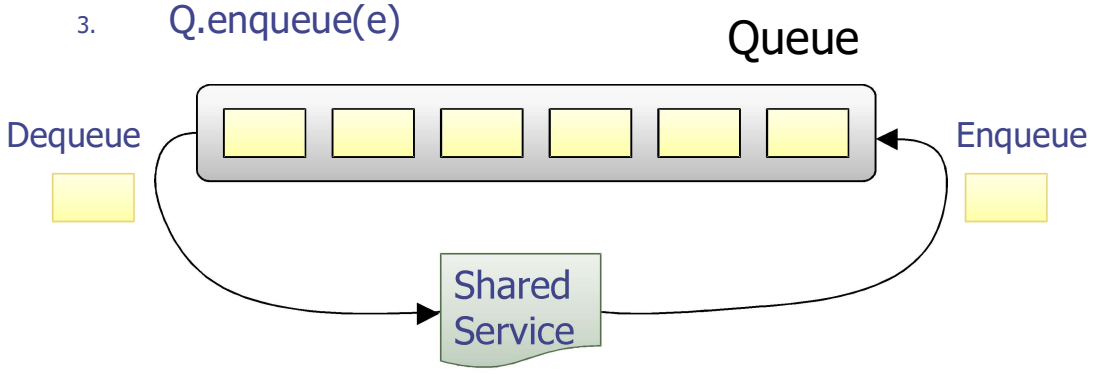
Operation	*Output*	*Queue*
enqueue(5)	–	(5)
enqueue(3)	–	(5, 3)
dequeue()	*5*	(3)
enqueue(7)	–	(3, 7)
dequeue()	*3*	(7)
front()	*7*	(7)
dequeue()	*7*	()
dequeue()	*"error"*	()
isEmpty()	*true*	()
enqueue(9)	–	(9)
enqueue(7)	–	(9, 7)
size()	*2*	(9, 7)
enqueue(3)	–	(9, 7, 3)
enqueue(5)	–	(9, 7, 3, 5)
dequeue()	*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

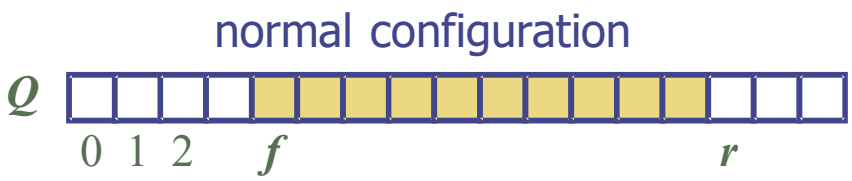
Application: Round Robin Schedulers

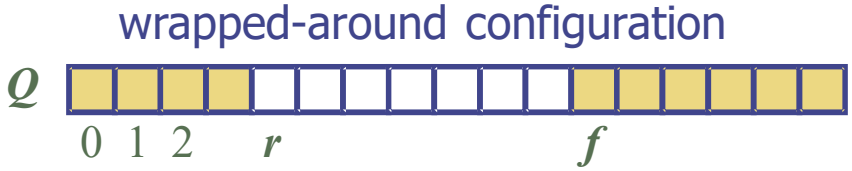
- We can implement a round robin scheduler using a queue Q by repeatedly performing the following steps:
  1. e = Q.dequeue()
  2. Service element e
  3. Q.enqueue(e)



Array-based Queue

- Use an array of size N in a circular fashion
- Two variables keep track of the front and rear
  - f index of the front element
  - r index immediately past the rear element
- Array location r is kept empty

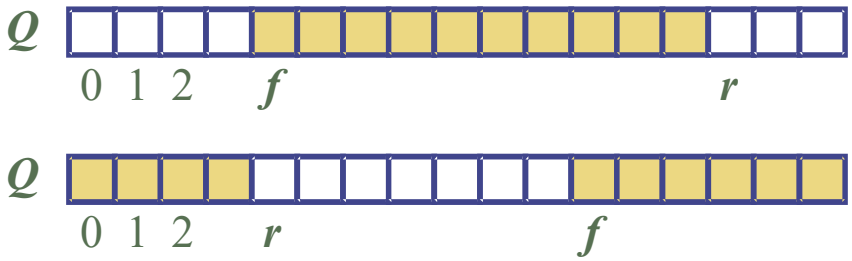




Queue Operations

- We use the modulo operator (remainder of division)

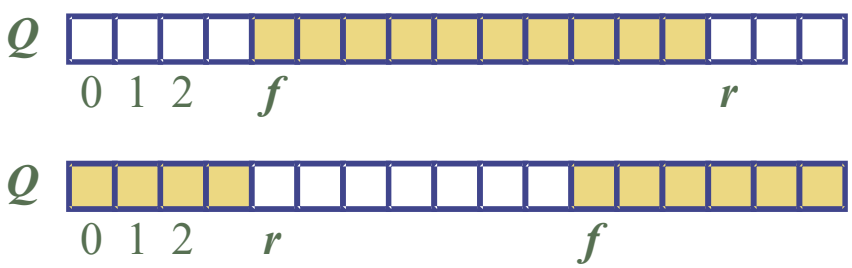
```
Algorithm size()
  return

Algorithm isEmpty()
  return
```



Queue Operations (cont.)

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

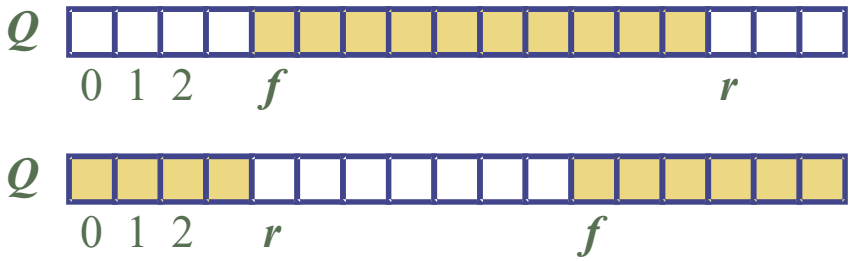
```
Algorithm enqueue(o)
  if size() = N – 1 then
    throw FullQueueException
  else
    Q[r] ← o
    r ← (r + 1) mod N
```



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 ← Q[f]
    f ← (f + 1) mod N
    return o
```



Queue Interface in Java

- Java interface corresponding to our Queue ADT
- Requires the definition of class EmptyQueueException
- No corresponding built-in Java class

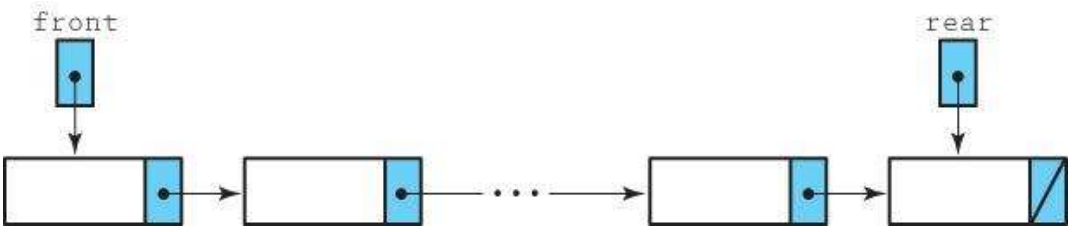
```
public interface Queue<E> {

  public int size();

  public boolean isEmpty();

  public E front()
      throws EmptyQueueException;

  public void enqueue(E element);

  public E dequeue()
      throws EmptyQueueException;
}
```

Linked-Based Implementations

- In this section we develop a link-based implementations of the Queue ADT.
- For nodes we use the same `Node` class we used for the linked implementation of stacks.
- After discussing the link-based approaches we compare all of our queue implementation approaches.

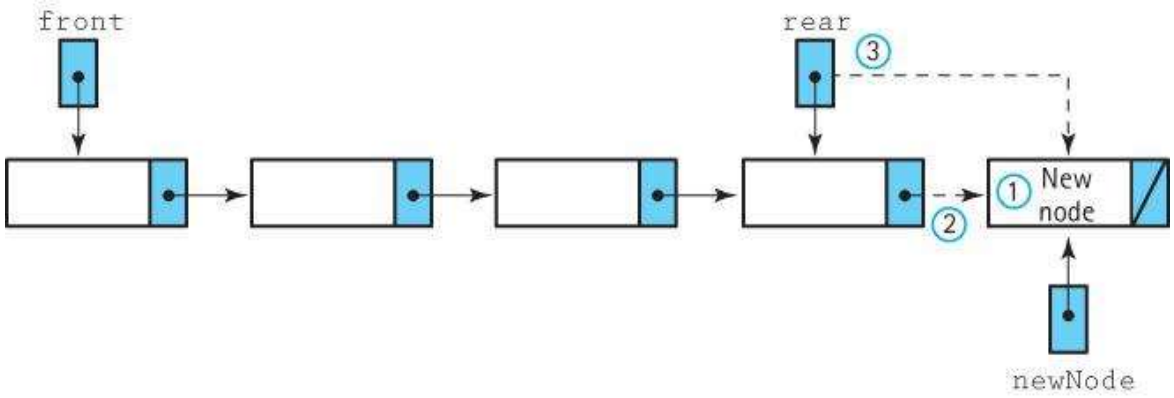
The LinkedUnbndQueue Class

```
public class LinkedListQueue<E> implements Queue<E>
{
  protected Node<E> front;    // front of this queue
  protected Node<E> rear;     // rear of this queue
  public LinkedListQueue()
  {
    front = null;
    rear = null;
  }
. . .
```



The enqueue operation

```
Enqueue (element)
   1. Create a node for the new element
   2. Insert the new node at the rear of the queue
   3. Update the reference to the rear of the queue
```

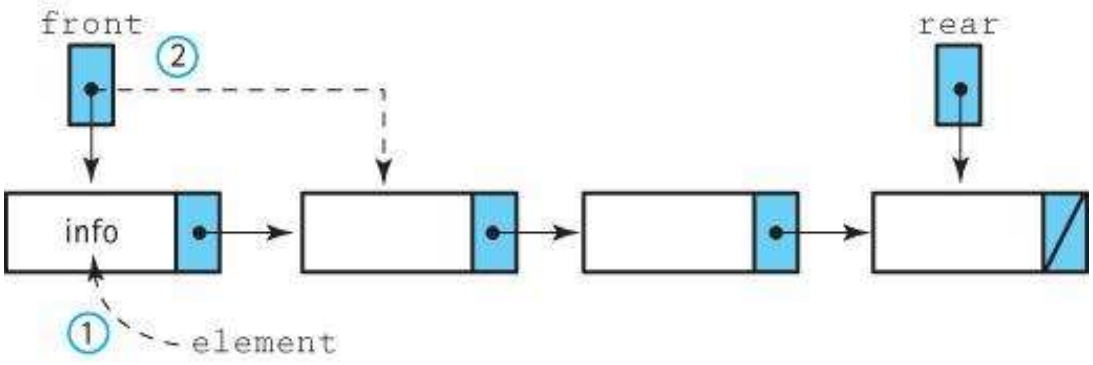


Code for the enqueue method

```
public void enqueue(E element)
// Adds element to the rear of this queue.
{
  Node<E> newNode = new Node<E>(element);
  if (rear == null)
    front = newNode;
  else
    rear.setNext(newNode);
  rear = newNode;
}
```

The dequeue operation

```
Dequeue: returns Object
    1. Set element to the information in the front node
    2. Remove the front node from the queue
       if the queue is empty
          Set the rear to null
       return element
```



Code for the dequeue method

```
public Object dequeue()
// Throws EmptyQueueException if this queue is empty,
// otherwise removes front element from this queue and returns it.
{
  if (isEmpty())
    throw new EmptyQueueException(
                "Dequeue attempted on empty queue.");
  else
  {
    E element;
    element = front.getInfo();

    front = front.getNext();
    if (front == null)
      rear = null;

    return element;
  }
}
```

Comparing Queue Implementations

- Storage Size
  - Array-based: takes the same amount of memory, no matter how many array slots are actually used, proportional to current capacity
  - Link-based: takes space proportional to actual size of the queue (but each element requires more space than with array approach)

- Operation efficiency
  - All operations, for each approach, are O(1)
  - Except for the Constructors:
    - Array-based: O(N)
    - Link-based: O(1)