Queues (Chapter 5)

Eleni Mangina

Room B2.05
School of Computer Science and Informatics
University College Dublin, Ireland

Queues: Concept

- A queue is a container of objects / values.
- Insertion and removal based on the first-in-first-out (FIFO) principle.
  - Objects can be inserted at any time, but only the object that has been in the queue the longest can be removed.
- Terminology:
  - Items can be “Enqueued” (insertion).
  - Items can be “Dequeued” (removal).
  - The “Front” of the queue is the next item to be dequeued
  - The “Rear” of the queue is where the last item was enqueued
- NOTE: We insert and remove from different places!

Example: Bread Queue

enter at end of line

exit from front of queue

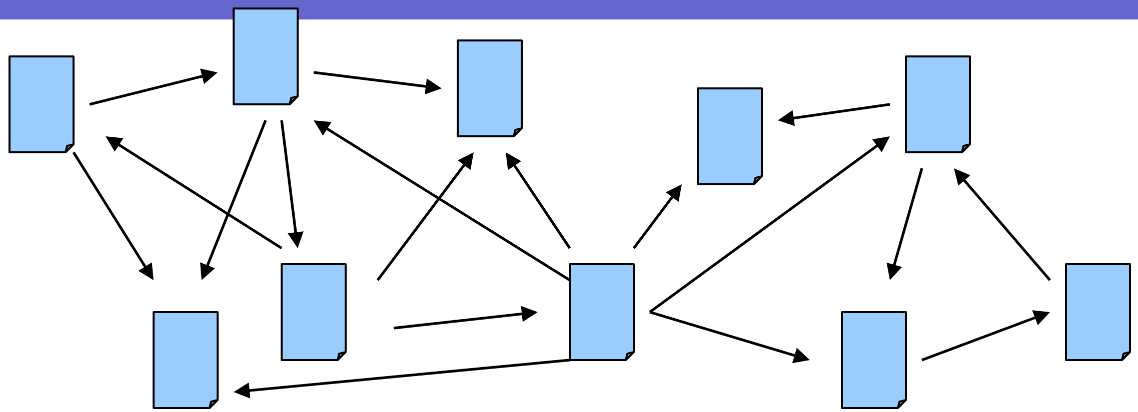
The Queue ADT

- 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 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

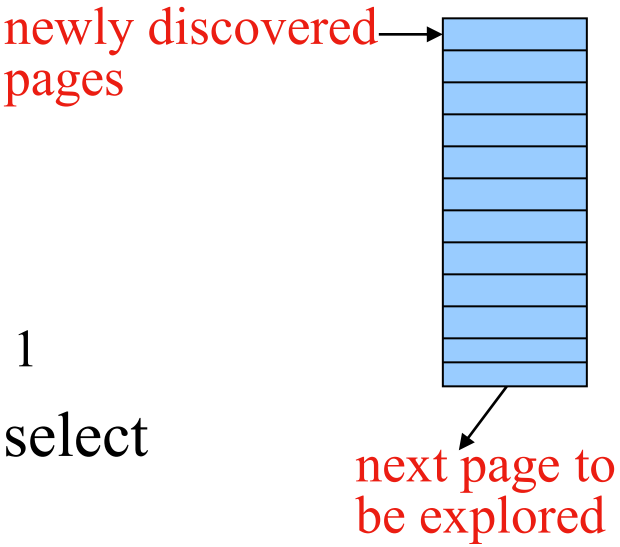
Operation	*Output*	*Q*
enqueue(5)	–	(5)
enqueue(3)	–	(5, 3)
dequeue()	*5*	(3)
enqueue(7)	–	(3, 7)
dequeue()	*3*	(7)
front()	*7*	(7)
dequeue()	*7*	()
dequeue()	*"error"*	()
isEmpty()	*true*	()
enqueue(9)	–	(9)
enqueue(7)	–	(9, 7)
size()	*2*	(9, 7)
enqueue(3)	–	(9, 7, 3)
enqueue(5)	–	(9, 7, 3, 5)
dequeue()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

Example: Web spider



- Search engines (e.g. Google) use programs called <u>spiders</u> to discover new pages on the web

- Input = a seed page
- Repeatedly
  1. Parse the page and extract hyperlinks
  2. Follow one of the hyperlinks and go to 1
- Spiders employ a Queue to store & select hyperlinks (Breadth First Search)



Queues: Functional Specification

- Queues should work with objects.

- Core Operations:
  - enqueue(o): Inserts object o onto rear of queue
  - dequeue(): Removes the object at the front of the queue and returns it

- Support Operations:
  - size(): Returns the number of objects in the queue
  - isEmpty(): Return a boolean indicating if the queue is empty.
  - front(): Return the front object in the queue, without removing it

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

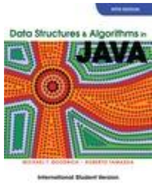
Queue Interface in Java

- 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;
}
```

A FIFO Queue Interface in Java

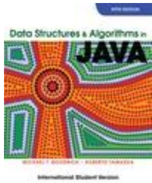
```
public interface Queue<E> {
/** *Returns the number of elements in the queue.
*@return number of elements in the queue. */
public int size();

/** * Returns whether the queue is empty.
*@return true if the queue is empty, false otherwise. */
public boolean isEmpty();

/** * Inspects the element at the front of the queue.
*@return element at the front of the queue.
*@exception EmptyQueueException if the queue is empty. */
public E front() throws EmptyQueueException;

/** * Inserts an element at the rear of the queue.
*@param element new element to be inserted. */
public void enqueue (E element);

/** * Removes the element at the front of the queue.
*@return element removed.
*@exception EmptyQueueException if the queue is empty. */
public E dequeue() throws EmptyQueueException; }
```



Queues: Impl. Strategies

- Array-based Implementation:
  - Array holds the objects pushed onto the queue
    - Insertion begins at index 0.
    - Auxiliary values needed to keep track of the “front” and “rear” of the queue of the queue.
  - Finite Capacity

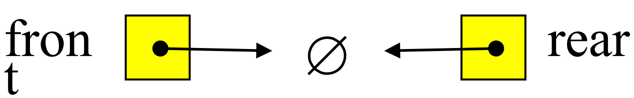
- Link-based Implementation:
  - Objects stored in special “nodes”
  - Nodes maintain ordering information
    - Link to the next object in the stack.
    - Need auxiliary references for “front” and “rear” nodes.
  - Infinite Capacity



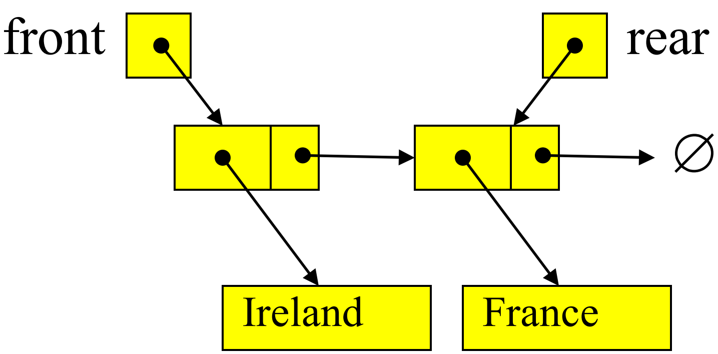
Link-Based Queue

- Auxiliary Data Structure: **Node**
  - A reference to the object being stored in the queue
  - A link to the next node in the queue (this is the link).

- Key Nodes / Data:
  - The “front” node of the queue
  - The “rear” node of the queue
  - Need to keep track of the “size” of the queue



Empty Queue



Queue of size 2

Link-Based Queue

Algorithm enqueue(o):

Input: an object o

Output: none

node ← new Node(o)

if (rear = null) **then** front ← node

else rear.next ← node

rear ← node

size ← size + 1

Algorithm size():

Input: none

Output: count of objects on the stack

return size

Algorithm isEmpty():

Input: none

Output: true if the stack is empty, false otherwise

return size = 0

Algorithm dequeue():

Input: none

Output: the top object

e ← front.element

front ← front.next

if (front = null) rear ← null

size ← size-1

return e

Algorithm front():

Input: none

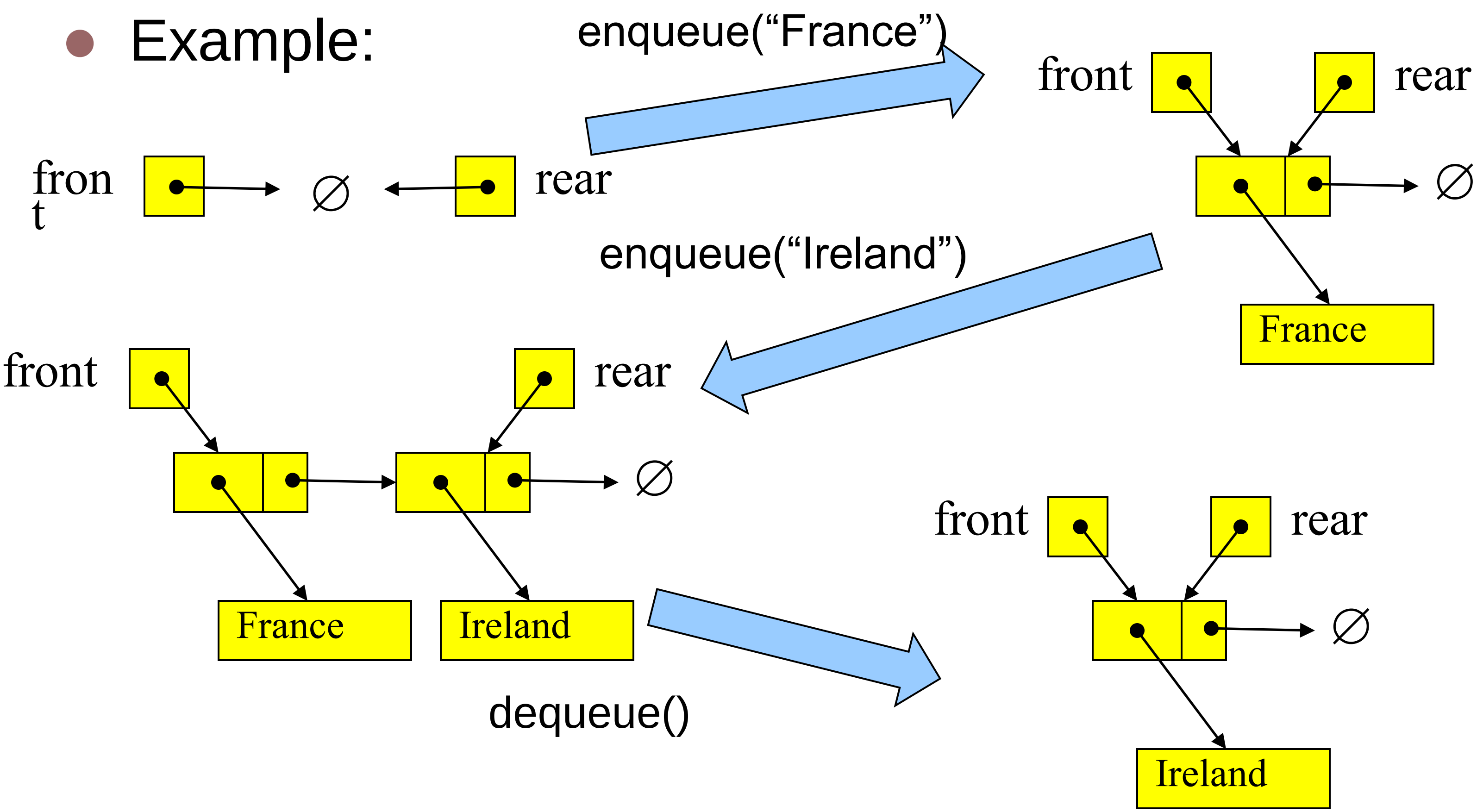
Output: the top object

return front.element

Link-Based Queues: Dry Runs

- View operations as atomic
  - Show the state of the Linked List after each operation

- Example:



Link-Based Queues: Impl.

- Class name: `LinkedQueue`
  - Fields:
    - An inner class `Node` (see Worksheet)
    - An integer, size (number of objects in the stack)
    - Two `Node` fields, front and rear
  - Constructors
    - Default Constructor (sets front and rear to `null` and `size` to `0`)
  - Methods:
    - 1 per operation: methods names should match operation names
    - Implement methods based on pseudo code

- This is part of your next worksheet

Link-Based Queues: Analysis

- Operation Running Times:

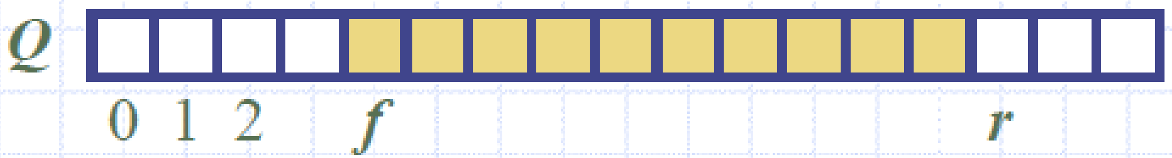
Operation	Running Time
enqueue(o)	O(1)
dequeue()	O(1)
front()	O(1)
isEmpty()	O(1)
size()	O(1)

- Issues:
  - What happens if we dequeue from an empty queue?

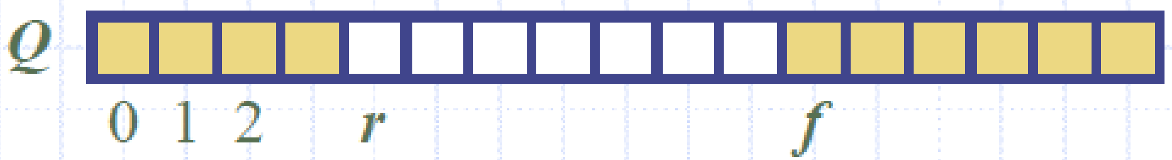
Array-based Queue

- 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

normal configuration



wrapped-around configuration

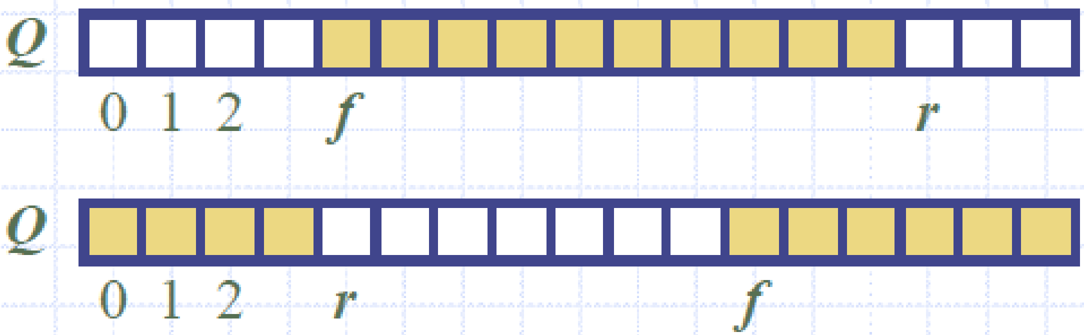


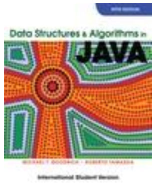
Queue Operations

- We use the modulo operator (remainder of division)

Algorithm *size*()
  return $(N - f + r) \bmod N$

Algorithm *isEmpty*()
  return $(f = r)$

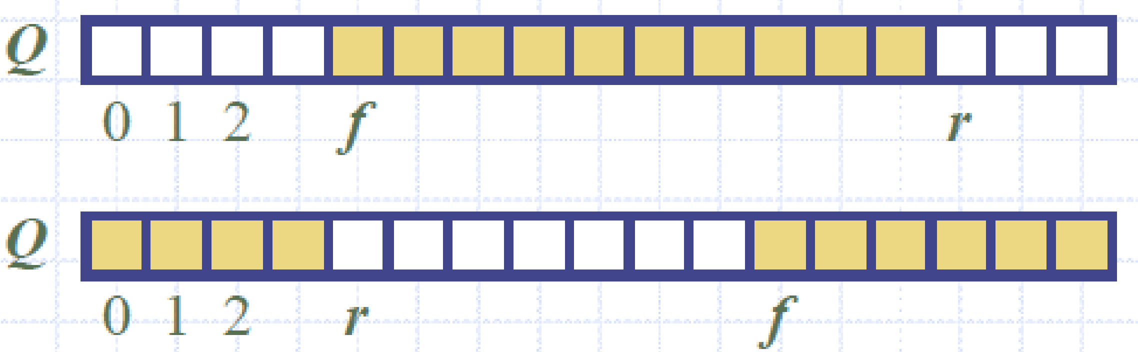




Queue Operations (cont.)

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

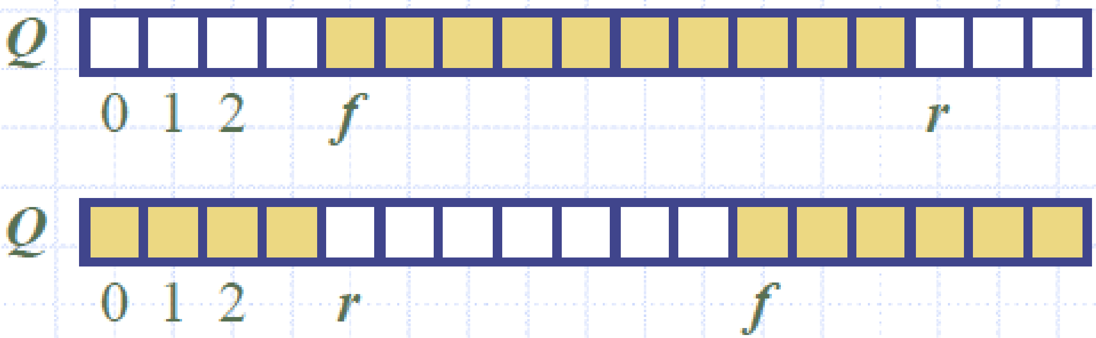
Algorithm *enqueue*(*o*)
  if *size*() = $N - 1$ **then**
    throw *FullQueueException*
  else
    $Q[r] \leftarrow o$
    $r \leftarrow (r + 1) \bmod N$



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

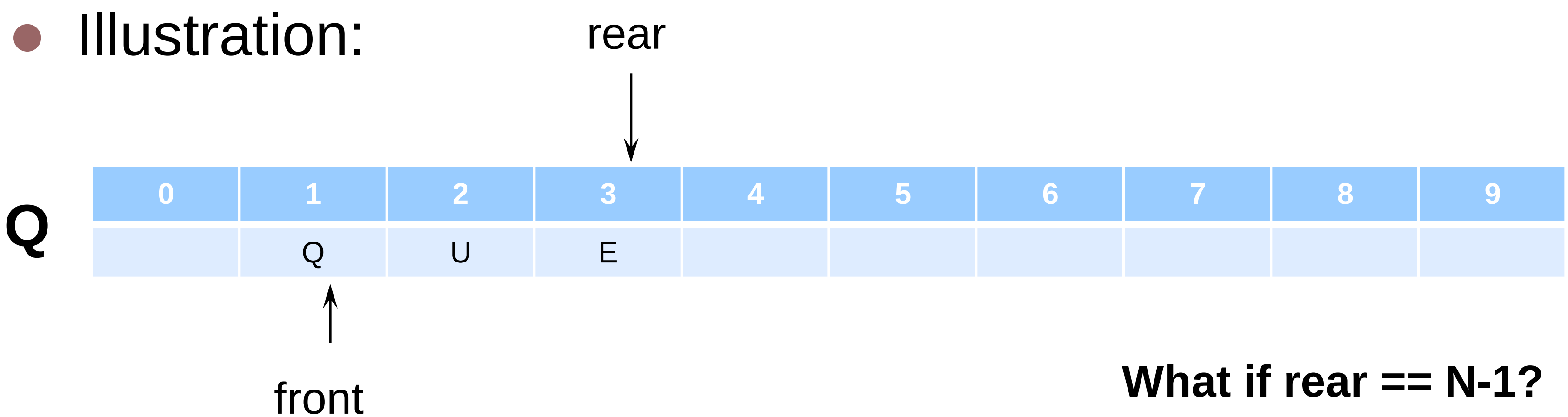


Naïve Array-Based Queue

- Create a queue using an array by specifying a maximum size N for our stack, e.g., N = 1000.

- The queue consists of:
  - an N-element array Q and
  - Two integer variables *front* and *rear*, the index of the front and rear elements in array Q respectively.

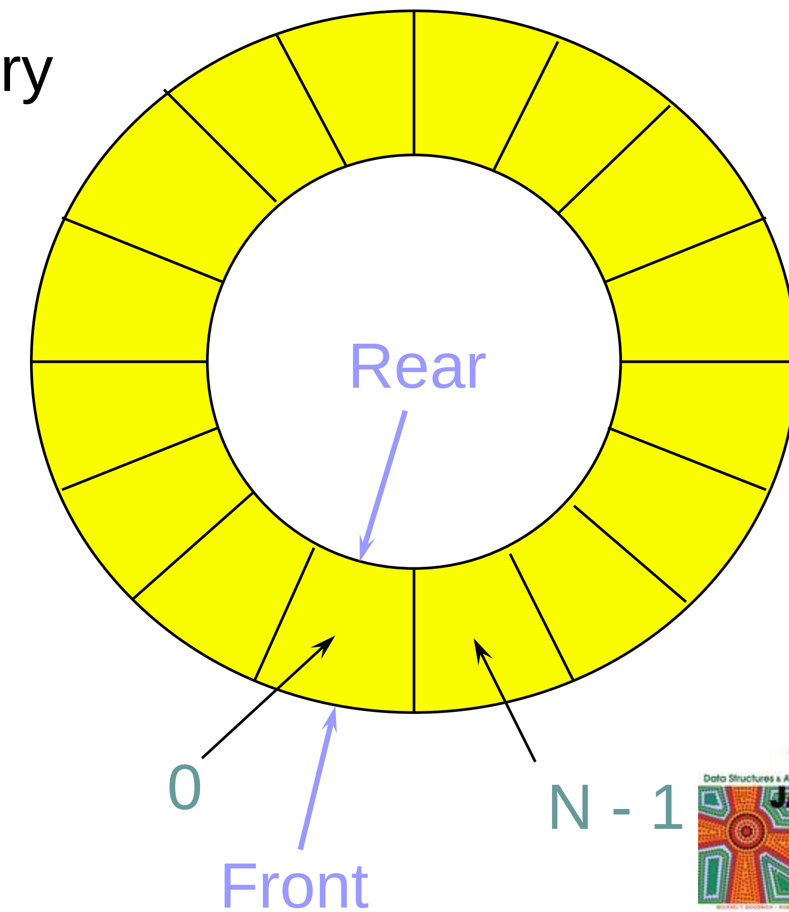
- Illustration:



What if rear == N-1?

Circular Arrays

- Naïve approach suffers from wasted space:
  - Dequeued indices are not reused.
- Could do an array copy with every dequeue – O(n)!
- Solution: Logically join the first and last cells in the array:
  - When front (or rear) reaches the end of the array, perform a “wraparound” by setting it to 0…



Circular Array Queue Size

- In the basic configuration, we can calculate the number of items in the queue as follows:
  - Size = rear - front

- In a wrap-around configuration, this equation gives us a negative value whose value represents the number of empty positions in the array…
  - By adding N on to this value we get the number of filled positions in the array!

- Hence, for configuration 2:
  - Size = N + rear - front

- Finally, we can use the fact the N mod N = 0 to show that, in either case, the size of the queue can be found as follows:
  - Size = (N + rear - front) mod N

Array-Based Queue

Algorithm enqueue(o):

Input: an object o

Output: none

Q[rear] ← 0

rear ← (rear + 1) % N

Algorithm size():

Input: none

Output: count of objects on the stack

return (N + rear – front) % N

Algorithm isEmpty():

Input: none

Output: true if the stack is empty, false otherwise

return rear = front

Algorithm dequeue():

Input: none

Output: the front object

e ← Q[front]

Q[front] ← null

front ← (front+1) % N

return e

Algorithm front():

Input: none

Output: the top object

return Q[front]

Array-Based Queues: Dry Runs

- View operations as atomic
  - Show the state of the array, S, and top element index, t, after each operation

- Example:

operation	f	r	0	1	2	3	4	5	6	7	8	9
Initial State	0	0										
enqueue(H)	0	1	H									
enqueue(A)	0	2	H	A								
enqueue(A)	0	3	H	A	A							
dequeue()	1	3		A	A							
enqueue(P)	1	4		A	A	P						
enqueue(P)	1	5		A	A	P	P					
dequeue()	2	5			A	P	P					

Array-Based Queues: Impl.

- Class name: `ArrayQueue`
  - Fields:
    - An array of objects, Q
    - An integer, N (array size)
    - Two integers, front and rear
  - Constructors
    - Default Constructor (sets N to 1000)
    - Constructor with 1 integer parameter (used to set value of N)
  - Methods:
    - 1 per operation: methods names should match operation names – except for naming conventions (lower case first letter)
    - Implement methods based on pseudo code

- This is part of your next worksheet

Array-Based Queues: Analysis

- Operation Running Times:

Operation	Running Time	LBQ Running Time
enqueue(o)	O(1)	O(1)
dequeue()	O(1)	O(1)
front()	O(1)	O(1)
isEmpty()	O(1)	O(1)
size()	O(1)	O(1)

- Issues:
  - What happens if we dequeue from an empty queue?
  - What happens if we enqueue on to a full queue?

- Which Implementation Strategy is better?

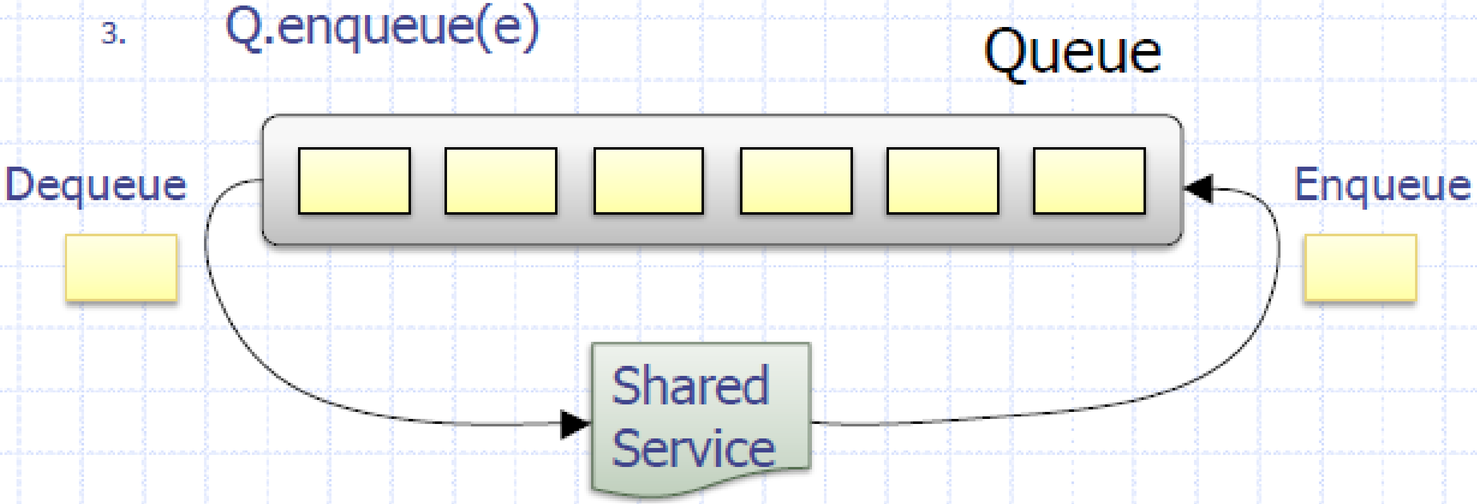
Enqueue Dequeue with Single LL

```
public void enqueue(E elem) {
    Node<E> node = new Node<E>();
    node.setElement(elem);
    node.setNext(null); // node will be new tail node
    if (size == 0) head = node;
    // special case of a previously empty queue
    else tail.setNext(node);
    // add node at the tail of the list
    tail = node;
    // update the reference to the tail node
    size++; }
```

```
public E dequeue() throws EmptyQueueException {
    if(size == 0) throw new
    EmptyQueueException("Queue is empty.");
    E tmp = head.getElement();
    head = head.getNext();
    size--;
    if (size == 0) tail = null;
    // the queue is now empty
    return tmp; }
```

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)

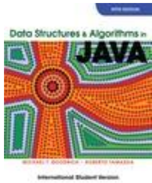


The Josephus problem...

In the children’s game “hot potato”, a group of n children sit in a circle passing an object, called the “potato”, around the circle. The potato begins with a starting child in the circle, and the children continue passing the potato until a leader rings a bell, at which point the child holding a potato must leave the game after handing the potato to the next child in the circle. After the selected child leaves, the other children close up the circle. This process is then continued until there is only one child remaining, who is declared the winner. **If the leader always uses the strategy of ringing the bell after the potato has been passed k times, for some fixed value k, then determining the winner for a given list of children is known as the Josephus problem.**

Using a Queue..

We can solve the Josephus problem for a collection of n elements using a queue, by associating the potato with the element at the front of the queue and storing elements in the queue according to their order around the circle. Thus, **passing the potato is equivalent to dequeuing an element and immediately enqueuing it again**. After this process has been processed k times, we remove the front element by dequeuing it from the queue and discarding it.



The code..

```
public class Josephus {
   /** Solution of the Josephus problem using a queue. */
   public static <E> E Josephus(Queue<E> Q, int k) {
     if (Q.isEmpty()) return null;
     while (Q.size() > 1) {
       System.out.println("  Queue: " + Q + "  k = " + k);
       for (int i=0; i < k; i++)
         Q.enqueue(Q.dequeue());  // move the front element to the end
       E e = Q.dequeue(); // remove the front element from the collection
       System.out.println("    " + e + " is out");
       }
     return Q.dequeue();  // the winner
   }
   /** Build a queue from an array of objects */
   public static <E> Queue<E> buildQueue(E a[]) {
     Queue<E> Q = new NodeQueue<E>();
     for (int i=0; i<a.length; i++)
       Q.enqueue(a[i]);
     return Q;
   }
   /** Tester method */
   public static void main(String[] args) {
     String[] a1 = {"Alice", "Bob", "Cindy", "Doug", "Ed", "Fred"};
     String[] a2 = {"Gene", "Hope", "Irene", "Jack", "Kim", "Lance"};
     String[] a3 = {"Mike", "Roberto"};
     System.out.println("First winner is " + Josephus(buildQueue(a1), 3));
     System.out.println("Second winner is " + Josephus(buildQueue(a2), 10));
     System.out.println("Third winner is " + Josephus(buildQueue(a3), 7));
   }
  }
```

What is the running time of this algorithm?

Deques (Chapter 5)

Eleni Mangina

Room B2.05
School of Computer Science and Informatics
University College Dublin, Ireland

Deque: Concept

- A deque (pronounced “deck”) is a container of objects / values.
- Insertion and removal based on the first or last-in first or last-out (FLI-FLO) principle.
- Terminology:
  - Items can be “inserted” at the front or back of the deque.
  - Items can be “removed” at the front or back of the deque
- NOTE: We insert and remove from **both** the front and the back.

Deques: Functional Specification

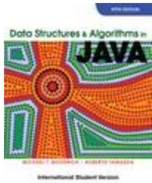
- Core Operations:
  - insertFirst(o): Inserts object o onto front of the deque
  - insertLast(o): Inserts object o onto the rear of the deque
  - removeFirst(): Removes the object at the front of the deque and returns it
  - removeLast(): Removes the object at the rear of the deque and returns it

- Support Operations:
  - size(): Returns the number of objects in the deque
  - isEmpty(): Return a boolean indicating if the deque is empty
  - front(): Return the front object in the deque
  - rear(): Return the rear object in the deque

Deques: Impl. Strategies

- Array-based Implementation: (Not covered)
  - Use a circular array
    - Need to handle transitions between normal and wraparound modes.
  - Finite Capacity

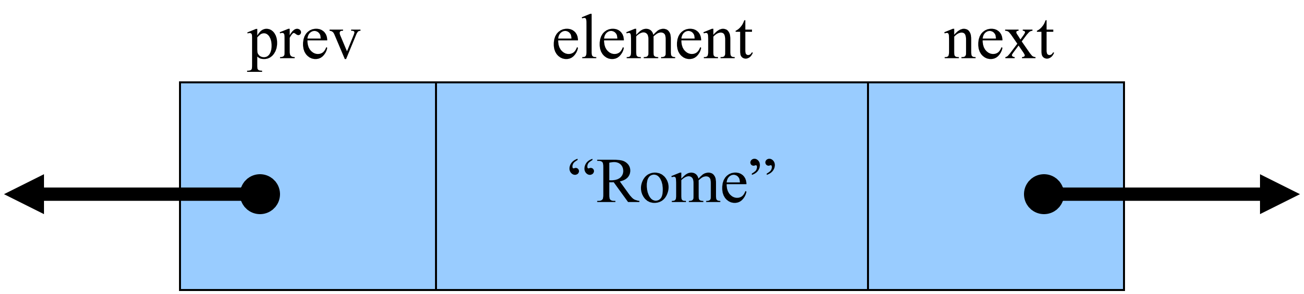
- Link-based Implementation:
  - Objects stored in special “nodes”
  - Nodes maintain ordering information
    - Link to the next **<u>and previous</u>** objects in the deque.
    - Need auxiliary references for “front” and “rear” nodes.
  - Infinite Capacity



Doubly Linked Lists

- In a Doubly Linked List, the objects are stored in **nodes**.

- Each nodemaintains:
  - A reference to the object
  - A reference to the next node in the list
  - A reference to the previous node in the list

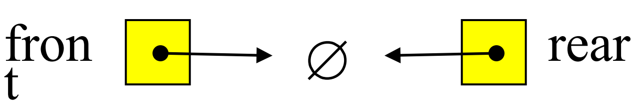


- Again, we store references to “key” nodes / entry points.
  - These provide us with a way of accessing the list

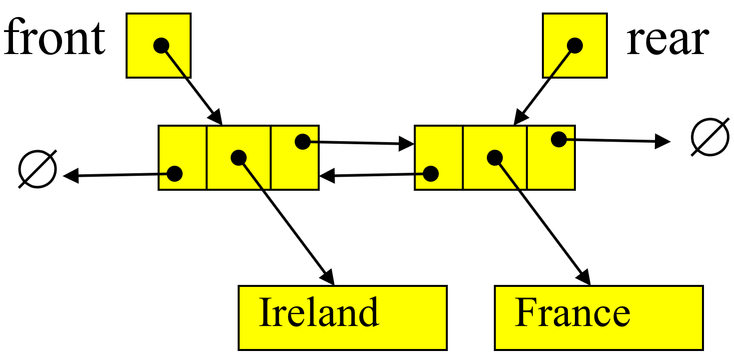
Link-BasedDeque

- Auxiliary Data Structure: **Node**
  - A reference to the object being stored in the deque
  - A link to the next node in the deque

- Key Nodes / Data:
  - The “front” node of the deque
  - The “rear” node of the deque
  - Need to keep track of the “size” of the deque



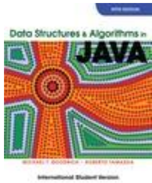
EmptyDeque



Deque of size 2

Interface for Deque

```
        /** * Interface for a deque: a collection of objects that are inserted
        * and removed at both ends; a subset of java.util.LinkedList methods. *
        * @author Roberto Tamassia
        * @author Michael Goodrich */

public interface Deque<E> {
        /** * Returns the number of elements in the deque. */
        public int size(); /** * Returns whether the deque is empty. */
        public boolean isEmpty(); /** * Returns the first element; an exception is thrown if deque is empty. */
        public E getFirst() throws EmptyDequeException; /** * Returns the last element; an exception is thrown if deque is empty. */
        public E getLast() throws EmptyDequeException; /** * Inserts an element to be the first in the deque. */
        public void addFirst (E element); /** * Inserts an element to be the last in the deque. */
        public void addLast (E element); /** * Removes the first element; an exception is thrown if deque is empty. */
        public E removeFirst() throws EmptyDequeException; /** * Removes the last element; an exception is thrown if deque is empty. */
        public E removeLast() throws EmptyDequeException; }
```



NodeDeque

```
public class NodeDeque<E> implements Deque<E> {
        protected DLNode<E> header, trailer; // sentinels
        protected int size; // number of elements
        public NodeDeque() { // initialize an empty deque
                header = new DLNode<E>();
                trailer = new DLNode<E>();
                header.setNext(trailer); // make header point to trailer
                trailer.setPrev(header); // make trailer point to header
                size = 0; }

        public int size() { return size; }

        public boolean isEmpty() { if (size == 0) return true; return false; }

        public E getFirst() throws EmptyDequeException {
                if (isEmpty()) throw new EmptyDequeException("Deque is empty.");
                return header.getNext().getElement(); }

        public void addFirst(E o) {
                DLNode<E> second = header.getNext();
                DLNode<E> first = new DLNode<E>(o, header, second);
                second.setPrev(first);
                header.setNext(first);
                size++; }
        public E removeLast() throws EmptyDequeException { if (isEmpty()) throw new EmptyDequeException("Deque is empty.");
                DLNode<E> last = trailer.getPrev(); E o = last.getElement(); DLNode<E> secondtolast = last.getPrev();
                trailer.setPrev(secondtolast); secondtolast.setNext(trailer); size--; return o; }

}
```