Linear Structures

Queues

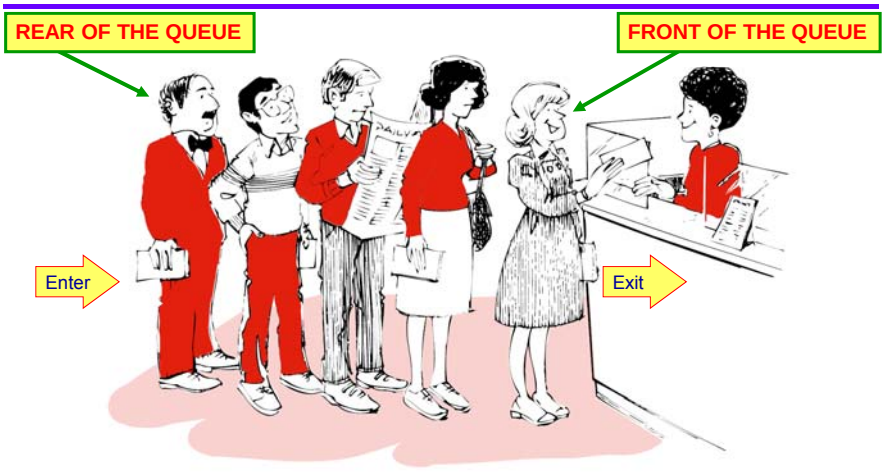
Abstract Definition

A Queue is an ordered collection of homogeneous elements (i.e. a list), from which elements may be deleted at one end called the front (head) of the queue, and into which elements may be inserted at the other end called the rear (tail) of the queue.

Specifications

- A queue is a FIFO "first in, first out" structure, which contains elements of some data type.
- The order of arrival of elements into the queue is determined by its FIFO structure.

A Queue of People



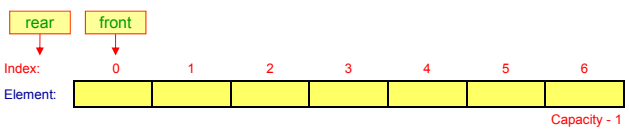
Operations

Remember that a queue is a specialized list.

Constructor()	Constructs an empty queue.
clear()	Sets the queue to an empty state.
isEmpty()	Check if the queue is empty.
isFull()	Check if the queue is full.
enqueue(entry)	Adds entry at the rear of the queue.
serve(element)	Removes and returns the element at the front of the queue.
peek(element)	Retrieves and returns the element at the front of the queue. The queue is unchanged.

Queue Implementation Using Arrays – First try

- Use a partially filled array of fixed capacity
- Use two integer variables:
  - one is called front, which points to the first element of the queue.
  - The other is called rear, which points to the last element of the queue.
- An empty queue is initialized by setting:
  front = 0, and rear = -1.
- The queue is empty if the condition: rear < front is true.
- What is the condition for a full queue? Rear == capacity - 1

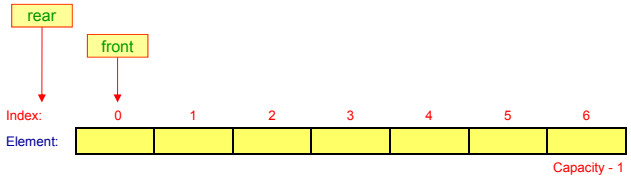


Queue Implementation
Using Arrays – First try (cont.)

- The enqueue(entry) operation is done as follows:
  - If not full, then Increment rear.
  - Store the entry in the array at rear.
- The serve(element) operation is done as follows:
  - Retrieve the element of the array at front.
  - Increment front.

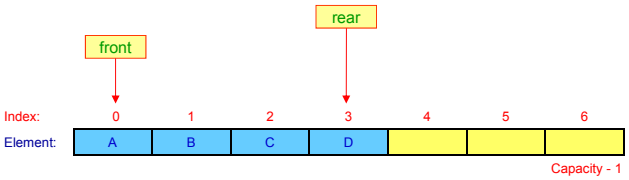
Queue Implementation
Using Arrays – First try (cont.)

- Example: Start with an empty queue.
  - enqueue(A); enqueue(B); enqueue(C); enqueue(D).



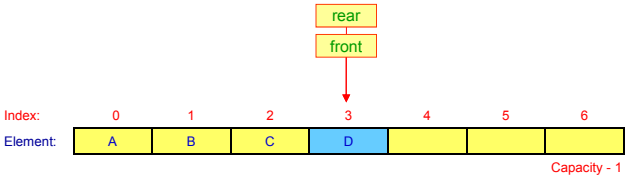
Queue Implementation
Using Arrays – First try (cont.)

- Example: Start with an empty queue.
  - enqueue(A); enqueue(B); enqueue(C); enqueue(D).
  - serve(A); serve(B); serve(C).



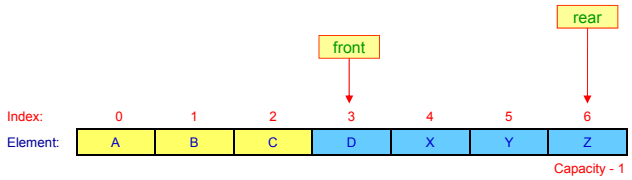
Queue Implementation
Using Arrays – First try (cont.)

- Example: Start with an empty queue.
  - enqueue(A); enqueue(B); enqueue(C); enqueue(D).
  - serve(A); serve(B); serve(C).
  - enqueue(X); enqueue(Y); enqueue(Z).



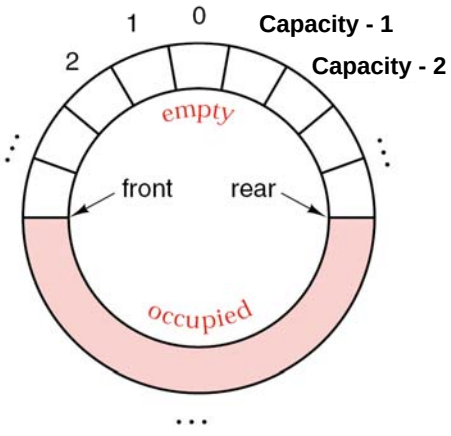
Queue Implementation
Using Arrays – First try (cont.)

- Example: Start with an empty queue.
  - enqueue(A); enqueue(B); enqueue(C); enqueue(D).
  - serve(A); serve(B); serve(C).
  - enqueue(X); enqueue(Y); enqueue(Z).
  - Can we enqueue another element?
- Problem: both front and rear are always increasing.

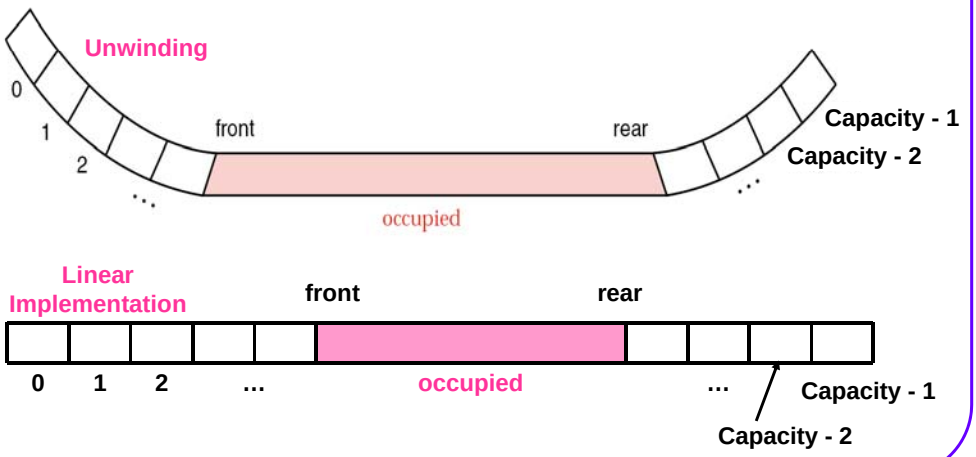


Queue Implementation
Using a Circular Array

Circular Queue



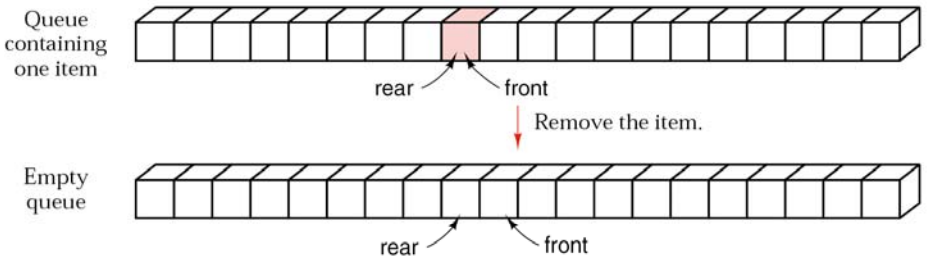
Queue Implementation Using a Circular Array (cont.)



Circular Arrays in Java

- Equivalent methods to increment an index i in a circular array:
  - if ((i+1) == capacity) i = 0; else i = i + 1;
  - i = ((i+1) == capacity) ? 0 : (i + 1);
  - i = (i+1) % capacity;

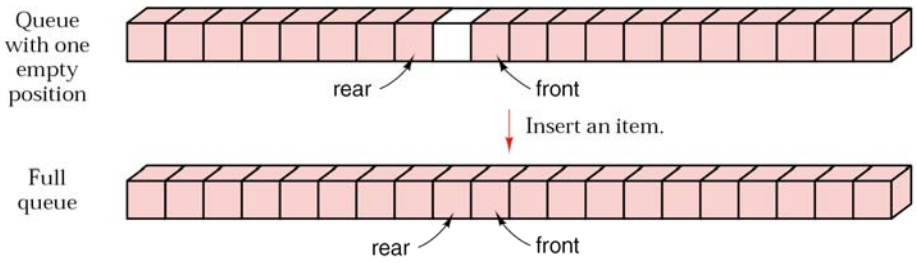
Boundary Conditions



Note: Queue is empty when rear = front - 1

r Lsbm

Boundary Conditions (cont.)



Note: Queue is full when rear = front – 1 also!

Boundary Conditions (cont.)

To resolve the boundary conditions conflict:

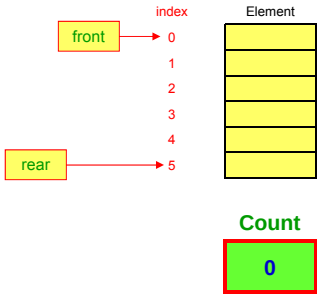
- Use front and rear indices and a boolean flag to indicate fullness (or emptiness).
- Use front and rear indices and an integer counter of entries called count.
- Use front and rear indices taking special values to indicate emptiness.

Empty Slide

Mapping Operation: Constructor ... *O*(1)

- **Postcondition**:
  - The queue has been initialized as an empty queue.
- **Code**:

```
count = 0;
front = 0;
rear = CAPACITY - 1;
```

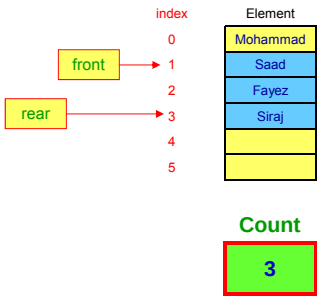


CAPACITY = 6

Mapping Operation: clear() ... *O*(1)

- **Postcondition**:
  - The queue has been emptied.
- **Code**:

```
count = 0;
front = 0;
rear = CAPACITY - 1;
```

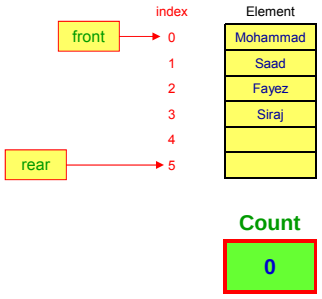


CAPACITY = 6

Mapping Operation: clear() ... *O*(1)

- **Postcondition**:
  - The queue has been emptied.
- **Code**:

```
count = 0;
front = 0;
rear = CAPACITY - 1;
```



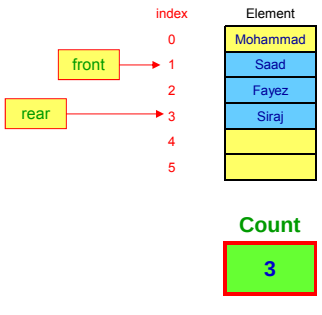
Mapping Operation: isEmpty() ... *O*(1)

- **Postcondition**:
  - The return value is true if the queue has no items, otherwise, it is false.
- **Code**:

```
return (count == 0);
```

- **Example**:

isEmpty(); ➔ returns false.



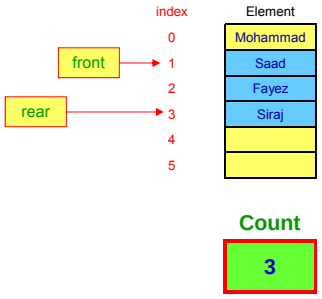
Mapping Operation: isFull() ... *O*(1)

- **Postcondition**:
  - The return value is true if the queue has a number of items equal to its capacity, otherwise it is false.
- **Code**:

```
return (count == CAPACITY);
```

- **Example**:

  isFull(); ➔ returns false.



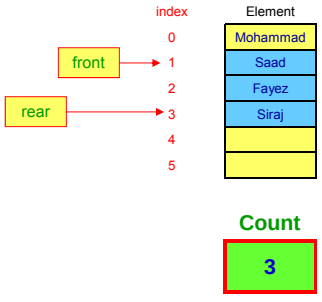
Mapping Operation: peek() ... *O*(1)

- **Precondition:**
  - isEmpty() returns false.
- **Postcondition**:
  - A copy of the front element of the queue has been returned, and the queue remains unchanged.
- **Code**:

```
assert !isEmpty();
return data[front];
```

- **Example**:

  peek(); ➔ returns 'Saad'.



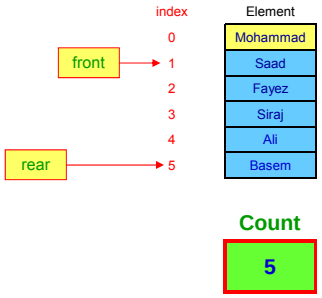
Mapping Operation: enqueue(entry) ... *O*(1)

- **Precondition:**
  - isFull() returns false.
- **Postcondition**:
  - A new copy of entry has been inserted at the rear of the queue.
- **Code**:

```
assert !isFull();
rear = (rear + 1) % CAPACITY;
data[rear] = entry;
count++;
```

- **Example**:

```
enqueue('Fouad');
```



CAPACITY = 6

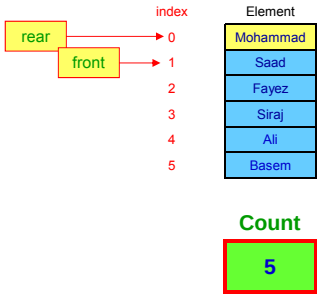
Mapping Operation: enqueue(entry) ... *O*(1)

- **Precondition:**
  - isFull() returns false.
- **Postcondition**:
  - A new copy of entry has been inserted at the rear of the queue.
- **Code**:

```
assert !isFull();
rear = (rear + 1) % CAPACITY;
data[rear] = entry;
count++;
```

- **Example**:

```
enqueue('Fouad');
```



CAPACITY = 6

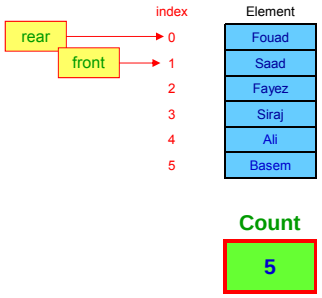
Mapping Operation: enqueue(entry) ... *O*(1)

- **Precondition:**
  - isFull() returns false.
- **Postcondition**:
  - A new copy of entry has been inserted at the rear of the queue.
- **Code**:

```
assert !isFull();
rear = (rear + 1) % CAPACITY;
data[rear] = entry;
count++;
```

- **Example**:

```
enqueue('Fouad');
```



Count

5

CAPACITY = 6

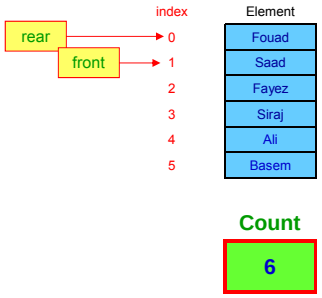
Mapping Operation: enqueue(entry) ... *O*(1)

- **Precondition:**
  - isFull() returns false.
- **Postcondition**:
  - A new copy of entry has been inserted at the rear of the queue.
- **Code**:

```
assert !isFull();
rear = (rear + 1) % CAPACITY;
data[rear] = entry;
count++;
```

- **Example**:

```
enqueue('Fouad');
```



Count

6

CAPACITY = 6

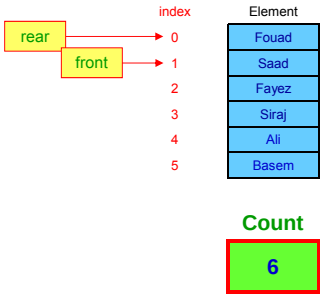
Mapping Operation: serve() ... *O*(1)

- **Precondition:**
  - isEmpty() returns false.
- **Postcondition**:
  - The front item of the queue has been removed and returned.
- **Code**:

```
assert !isEmpty());
temp = data[front];
front = (front + 1) % CAPACITY;
count--;
return temp;
```

- **Example**:

  serve(); ➔ returns 'Saad'.



Count

6

CAPACITY = 6

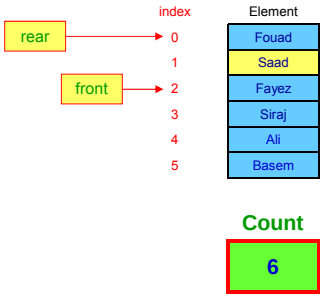
Mapping Operation: serve() ... *O*(1)

- **Precondition:**
  - isEmpty() returns false.
- **Postcondition**:
  - The front item of the queue has been removed and returned.
- **Code**:

```
assert !isEmpty();
temp = data[front];
front = (front + 1) % CAPACITY;
count--;
return temp;
```

- **Example**:

  serve(); ➔ returns 'Saad'.



Count

6

CAPACITY = 6

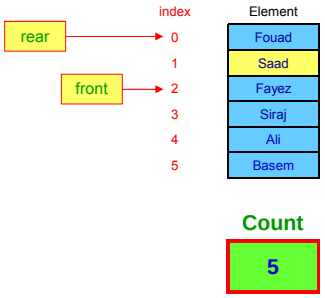
Mapping Operation: serve() ... *O*(1)

- **Precondition:**
  - isEmpty() returns false.
- **Postcondition**:
  - The front item of the queue has been removed and returned.
- **Code**:

```
assert !isEmpty();
temp = data[front];
front = (front + 1) % CAPACITY;
count--;
return temp;
```

- **Example**:

  serve(); ➔ returns 'Saad'.



CAPACITY = 6

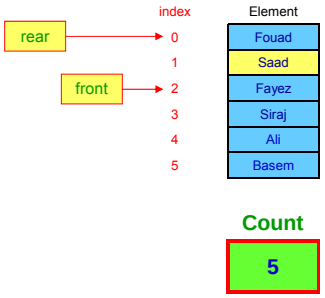
Mapping Operation: size() ... *O*(1)

- **Postcondition**:
  - The return value is the number of items in the queue.
- **Code**:

```
return count;
```

- **Example**:

  size(); ➔ returns 5.



CAPACITY = 6

Queue Implementation in Java Using a Circular Array and Generics

```
public class ArrayQueue<E> {
   private static final int CAPACITY = 30;          // default array capacity
   private int front, rear;                         // Index for the front an rear of the queue
   private int count;                               // The number of elements in the queue
   private E data[ ];                               // Generic array used for the queue

   // HELPER METHOD
   private int nextIndex(int i) { return (i+1) % CAPACITY; }

   // DEFAULT CONSTRUCTOR
   public ArrayQueue() {
      data = (E[ ]) new Object[CAPACITY];           // Array of default capacity
      count =  0; front = 0; rear = CAPACITY – 1; // Indicates empty queue
   }
```

Queue Implementation in Java Using a Circular Array and Generics

```
   // MUTATOR METHODS
   public void enqueue(E entry)  {  ……….  }
   public E serve();             {  ……….  }
   public void clear()           { count = 0; front = 0; rear = capacity -1 }


   // OBSERVER METHODS
   public boolean isEmpty()      { return count == 0; }
   public boolean isFull()       { return count == CAPACITY; }
   public int size()             { return count; }
   public E peek()               { assert !isEmpty(); return data[front]; }
}
```

Queue Applications

- Scheduling Problems:
  - CPU Scheduling
  - Disk I/O Scheduling
- Simulation of real systems:
  - Network Simulation
  - Airport Simulation, Run Executable.

Advantages and Disadvantages

- Time complexity of all queue operations is O(1).
  - A very efficient data structure.
- Fixed Storage space must be reserved in advance
  - Queue length is limited to the array size that was reserved.
  - May overflow or may waste unused space.
- What is a Double Queue (deque)?

The Deque Interface (ADT)

- A Deque is a generalization of both the FIFO (Queue) and the LIFO (Stack) structures.
- A Deque is an ordered collection of homogeneous elements (i.e. a list), with a front and a rear, where:
  - Elements can be added at the front of the list or at the rear of the list.
  - Elements can be removed at the front of the list or at the rear of the list.

Deque Operations

Remember that a deque is a specialized list.

Constructor()	Constructs an empty deque.
clear()	Sets the deque to an empty state.
isEmpty()	Checks if the deque is empty.
isFull()	Checks if the deque is full.
addFirst(entry)	Adds entry at the front of the deque.
removeFirst()	Removes and returns element at front of the deque.
addLast(entry)	Adds entry at the rear of the deque
removeLast()	Removes and returns element at rear of the deque.
getFirst()	Returns the element at front of the deque.
getLast()	Returns the element at rear of the deque.

NOTES:

- Given a particular Deque implementation, we can use it directly to implement a Stack or a Queue as follows:
  - For a Stack, use:
    - addFirst(x), for the push(x) operation,
    - removeFirst(), for the pop() operation, and
    - getFirst(), for the peek() operation.
  - For a Queue, use:
    - addLast(x), for the enqueue(x) operation,
    - removeFirst(), for the serve() operation, and
    - getFirst(), for the peek() operation.