Data Structures
Lesson 2

BSc in Computer Science
University of New York, Tirana

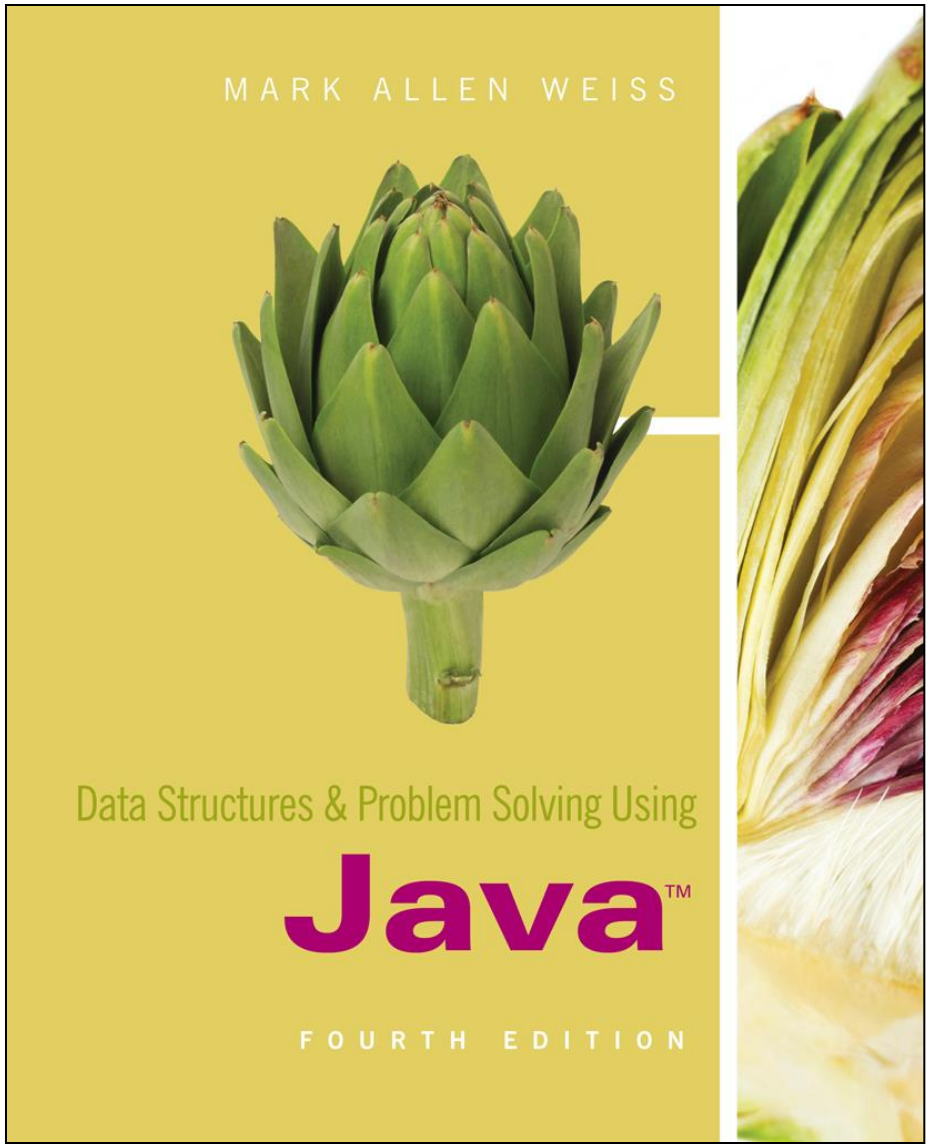
Assoc. Prof. Marenglen Biba

Outline

- Queues
- Queues: Implementation with Array
- Queues: Implementation with Linked List

Chapter 16

Queues



Addison-Wesley
is an imprint of
PEARSON

Queue

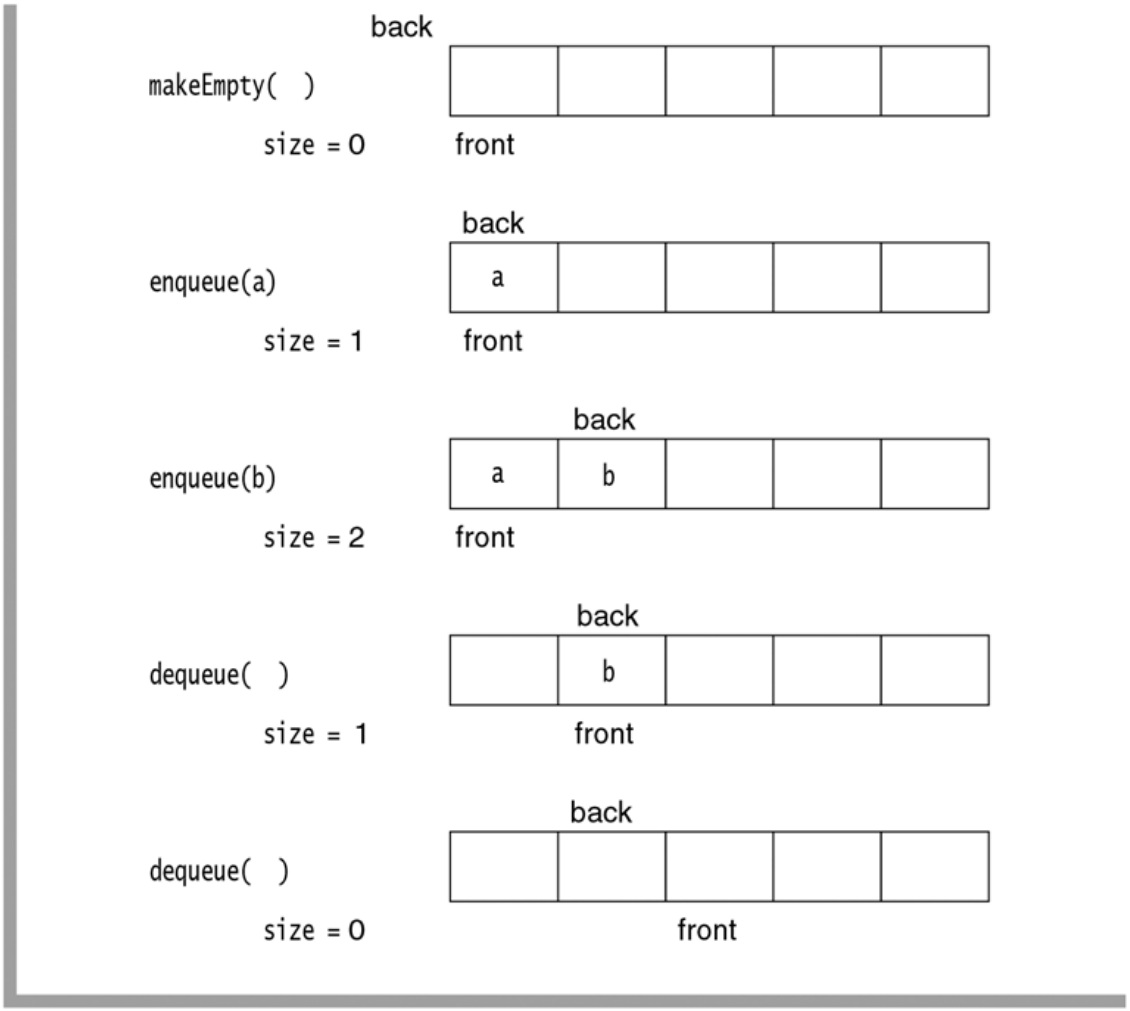
- The queue is another constrained linear data structure.
- The elements in a queue are ordered from least recently added (the front) to most recently added (the rear).
- Insertions are performed at the rear of the queue, and deletions are performed at the front.
- You use the enqueue operation to insert elements and the dequeue operation to remove elements.

Queue: FIFO mode of operation

- The movement of elements through a queue reflects the First in, First out (FIFO) behavior that is characteristic of the flow of customers in a line or the transmission of information across a data channel.
- Queues are routinely used to regulate the flow of physical objects, information, and requests for resources (or services) through a system.
- Operating systems, for example, use queues to control access to system resources such as printers, files, and communications lines.

figure 16.8

Basic array implementation of the queue



Basic array implementation

- The fundamental problem with the basic array approach is that after some enqueue operations, we cannot add any more items even though the queue is not really full.
  - See Line 1 of Figure 16.9

Circular array implementation

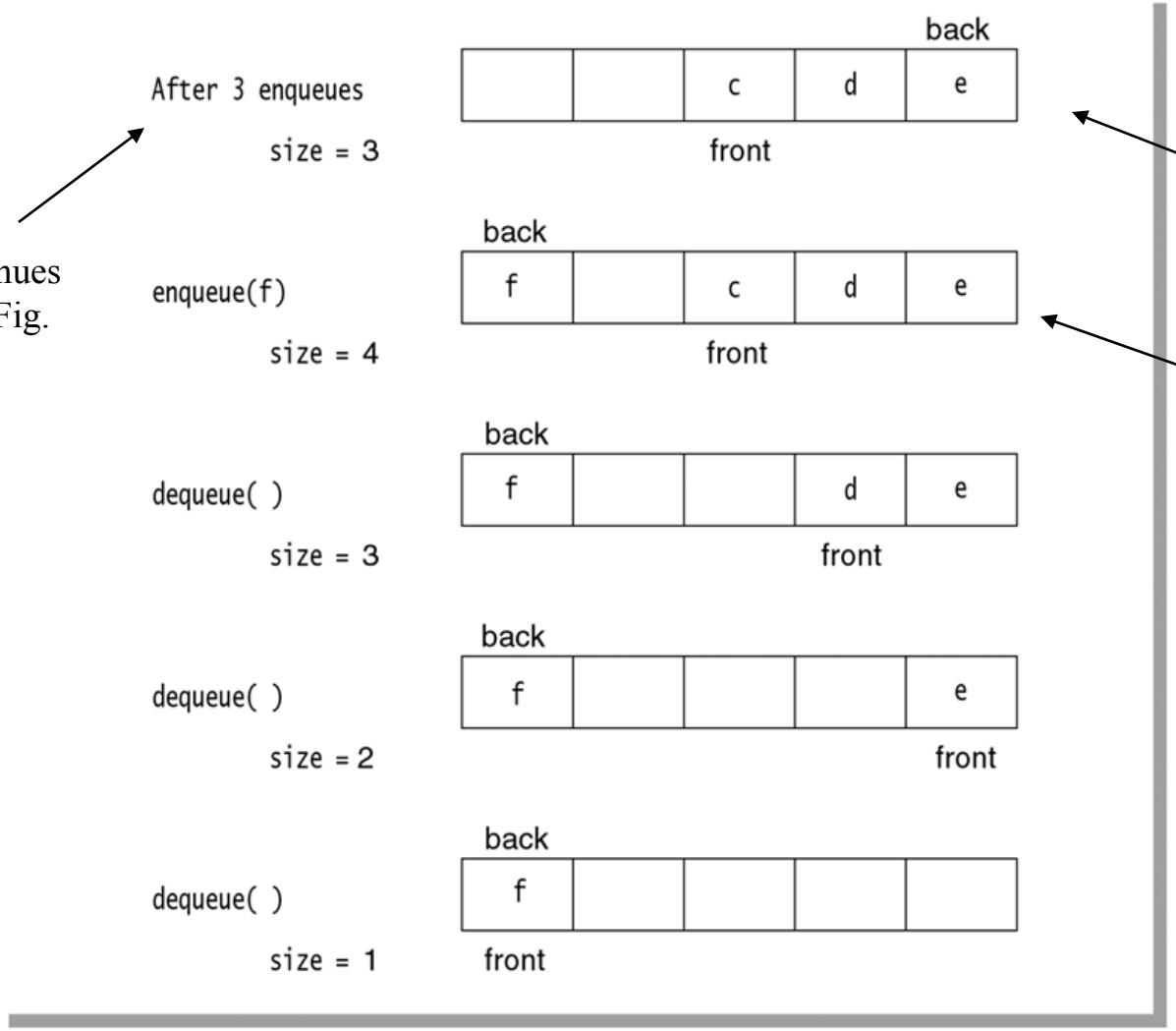


figure 16.9

Array implementation of the queue with wraparound

Continues from Fig. 16.8

Cannot add any more if we use the basic array implementation

Solution: Wraparound, when either front or back reaches the end of the array, reset it to the beginning.

This is called circular array implementation.

figure 16.10

Skeleton for the array-based queue class

```
package weiss.nonstandard;

// ArrayQueue class
//
// CONSTRUCTION: with no initializer
//
// ******************PUBLIC OPERATIONS*********************
// void enqueue( x )      --> Insert x
// AnyType getFront( )    --> Return least recently inserted item
// AnyType dequeue( )     --> Return and remove least recent item
// boolean isEmpty( )     --> Return true if empty; else false
// void makeEmpty( )      --> Remove all items
// ******************ERRORS********************************
// getFront or dequeue on empty queue

public class ArrayQueue<AnyType>
{
    public ArrayQueue( )
      { /* Figure 16.12 */ }

    public boolean isEmpty( )
      { /* Figure 16.13 */ }
    public void makeEmpty( )
      { /* Figure 16.17 */ }
    public AnyType dequeue( )
      { /* Figure 16.16 */ }
    public AnyType getFront( )
      { /* Figure 16.16 */ }
    public void enqueue( AnyType x )
      { /* Figure 16.14 */ }

    private int increment( int x )
      { /* Figure 16.11 */ }
    private void doubleQueue( )
      { /* Figure 16.15 */ }

    private AnyType [ ] theArray;
    private int         currentSize;
    private int         front;
    private int         back;

    private static final int DEFAULT_CAPACITY = 10;
}
```

Array of the queue

Actual size of the queue

Back and front of the queue

Capacity of the queue

Increment( )

```
/**
 * Internal method to increment with wraparound.
 * @param x any index in theArray's range.
 * @return x+1, or 0 if x is at the end of theArray.
 */
private int increment( int x )
{
    if( ++x == theArray.length )
        x = 0;
    return x;
}
```

figure 16.11

The wraparound routine

Constructor( )

```
/**
 * Construct the queue.
 */
public ArrayQueue( )
{
    theArray = (AnyType []) new Object[ DEFAULT_CAPACITY ];
    makeEmpty( );
}
```

figure 16.12

The constructor for the `ArrayQueue` class

isEmpty( )

```
/**
 * Test if the queue is logically empty.
 * @return true if empty, false otherwise.
 */
public boolean isEmpty( )
{
    return currentSize == 0;
}
```

figure 16.13

The `isEmpty` routine for the `ArrayQueue` class

Enqueue(x)

figure 16.14

The `enqueue` routine for the `ArrayQueue` class

```
/**
 * Insert a new item into the queue.
 * @param x the item to insert.
 */
public void enqueue( AnyType x )
{
    if( currentSize == theArray.length )
        doubleQueue( );
    back = increment( back );
    theArray[ back ] = x;
    currentSize++;
}
```

Increment back

DoubleQueue( )

```
/**
 * Internal method to expand theArray.
 */
private void doubleQueue( )
{
    AnyType [ ] newArray;

    newArray = (AnyType []) new Object[ theArray.length * 2 ];

        // Copy elements that are logically in the queue
    for( int i = 0; i < currentSize; i++, front = increment( front ) )
        newArray[ i ] = theArray[ front ];

    theArray = newArray;
    front = 0;
    back = currentSize - 1;
}
```

figure 16.15

Dynamic expansion for the `ArrayQueue` class

Copy all the elements:
increment count i and front

Dequeue( ) and getFront( )

```
/**
 * Return and remove the least recently inserted item
 * from the queue.
 * @return the least recently inserted item in the queue.
 * @throws UnderflowException if the queue is empty.
 */
public AnyType dequeue( )
{
    if( isEmpty( ) )
        throw new UnderflowException( "ArrayQueue dequeue" );
    currentSize--;

    AnyType returnValue = theArray[ front ];
    front = increment( front );
    return returnValue;
}

/**
 * Get the least recently inserted item in the queue.
 * Does not alter the queue.
 * @return the least recently inserted item in the queue.
 * @throws UnderflowException if the queue is empty.
 */
public AnyType getFront( )
{
    if( isEmpty( ) )
        throw new UnderflowException( "ArrayQueue getFront" );
    return theArray[ front ];
}
```

figure 16.16

The `dequeue` and `getFront` routines for the `ArrayQueue` class

Increment front

makeEmpty( )

```
/**
 * Make the queue logically empty.
 */
public void makeEmpty( )
{
    currentSize = 0;
    front = 0;
    back = -1;
}
```

figure 16.17

The `makeEmpty` routine for the `ArrayQueue` class

For an empty queue, back must be initialized to -1.

Linked List Implementation

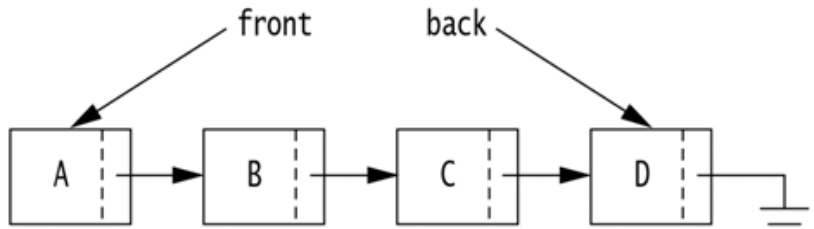


figure 16.22

Linked list implementation of the queue class

figure 16.23

Skeleton for the linked list-based queue class

```
package weiss.nonstandard;

// ListQueue class
//
// CONSTRUCTION: with no initializer
//
// ******************PUBLIC OPERATIONS*********************
// void enqueue( x )      --> Insert x
// AnyType getFront( )    --> Return least recently inserted item
// AnyType dequeue( )     --> Return and remove least recent item
// boolean isEmpty( )     --> Return true if empty; else false
// void makeEmpty( )      --> Remove all items
// ******************ERRORS********************************
// getFront or dequeue on empty queue

public class ListQueue<AnyType>
{
    public ListQueue( )
      { /* Figure 16.24 */ }
    public boolean isEmpty( )
      { /* Figure 16.27 */ }
    public void enqueue( AnyType x )
      { /* Figure 16.25 */ }
    public AnyType dequeue( )
      { /* Figure 16.25 */ }
    public AnyType getFront( )
      { /* Figure 16.27 */ }
    public void makeEmpty( )
      { /* Figure 16.27 */ }

    private ListNode<AnyType> front;
    private ListNode<AnyType> back;
}
```

Two listnodes: back and front

Constructor( )

figure 16.24

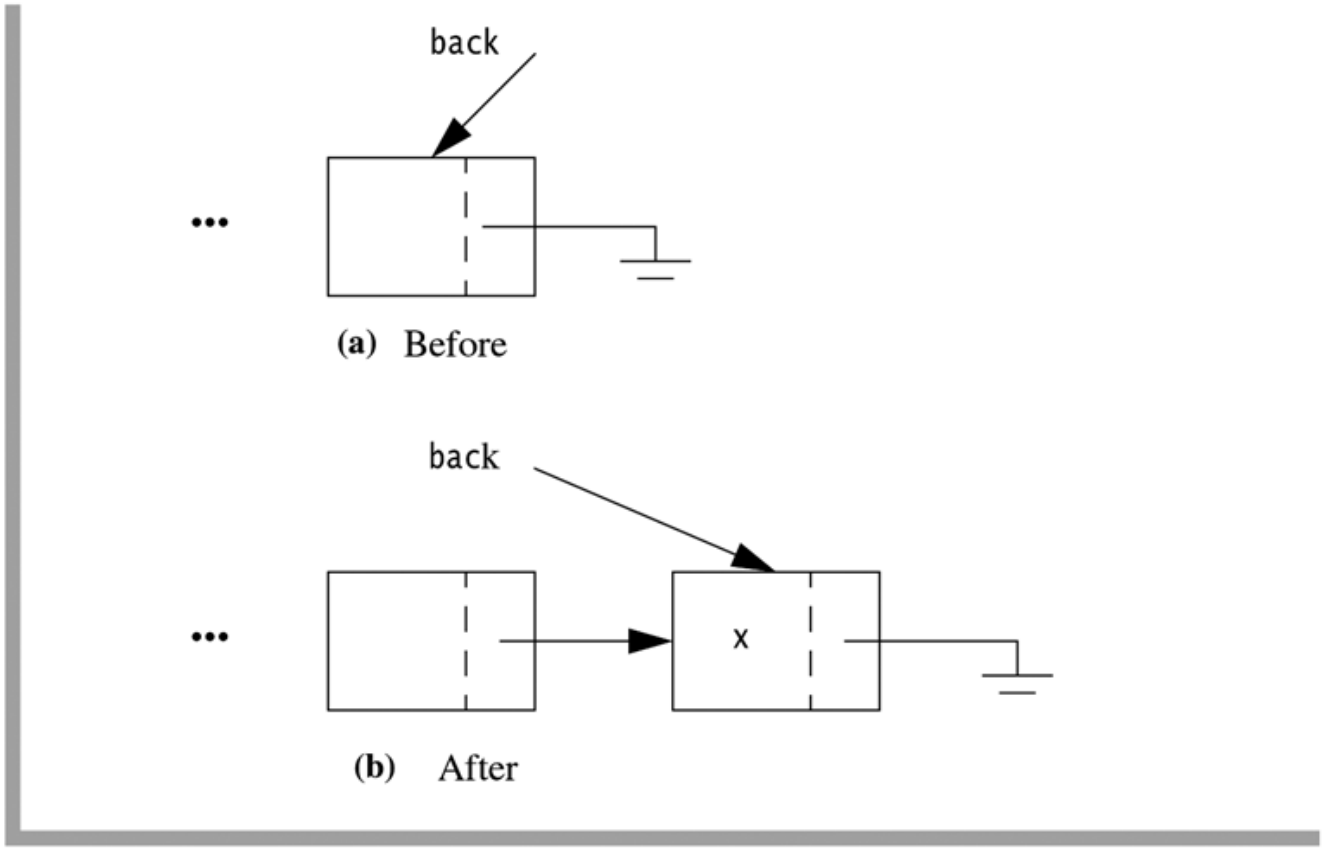
Constructor for the linked list-based `ListQueue` class

```
/**
 * Construct the queue.
 */
public ListQueue( )
{
    front = back = null;
}
```

Enqueue

figure 16.26

The **enqueue** operation for the linked list-based implementation



Enqueue( ) and Dequeue( )

```
/**
 * Insert a new item into the queue.
 * @param x the item to insert.
 */
public void enqueue( AnyType x )
{
    if( isEmpty( ) )    // Make a queue of one element
        back = front = new ListNode<AnyType>( x );
    else                // Regular case
        back = back.next = new ListNode<AnyType>( x );
}

/**
 * Return and remove the least recently inserted item
 * from the queue.
 * @return the least recently inserted item in the queue.
 * @throws UnderflowException if the queue is empty.
 */
public AnyType dequeue( )
{
    if( isEmpty( ) )
        throw new UnderflowException( "ListQueue dequeue" );

    AnyType returnValue = front.element;
    front = front.next;
    return returnValue;
}
```

figure 16.25

The `enqueue` and `dequeue` routines for the `ListQueue` class

Two cases: empty and non-empty

Makes back.next point to the new node

Removes the front element and makes front point to the next listnode

getFront( ), makeEmpty() and isEmpty( )

figure 16.27

Supporting routines for the `ListQueue` class

```
/**
 * Get the least recently inserted item in the queue.
 * Does not alter the queue.
 * @return the least recently inserted item in the queue.
 * @throws UnderflowException if the queue is empty.
 */
public AnyType getFront( )
{
    if( isEmpty( ) )
        throw new UnderflowException( "ListQueue getFront" );
    return front.element;
}

/**
 * Make the queue logically empty.
 */
public void makeEmpty( )
{
    front = null;
    back = null;
}

/**
 * Test if the queue is logically empty.
 */
public boolean isEmpty( )
{
    return front == null;
}
```

Readings

- Book
  - Chapter 16

Laboratory Exercises

- Add to the ADT, for both implementations that we have defined, the following methods:
  - ShowElements: shows all the elements in the queue in FIFO order
  - ShowInverse: show the elements in inverse order.
  - New constructor which specifies size of queue (for the array implementation) as parameter.
  - Clone: replicate a queue in another queue.
  - FindMinimum: find the smallest element in the queue
    - Hint: you need to use the following signature:

    public AnyType findMinimum(Comparator<AnyType> cmp)

    and use compare( ) of cmp to compare two AnyType objects.
    - Hint: define interface Comparator<AnyType> and class BookComparator<Book> implements Comparator<Book>
    - Hint: then define in BookComparator the method compare() that returns -1, 0, or 1. Class book must implement the interface Comparator<Book>.
- Test all these in a testing class