

Queues

- Implementations of the *Queue* ADT
- Queue specification
- Queue interface
- Block (array) representations of queues
- Recursive (linked) representations of queues

Reading: Weiss, Chapter 16

2. Specification

Recall that in a *queue*, or *FIFO*, elements are added to one end, and read/deleted from the other, in chronological order.

1. *Queue()*: create an empty queue
2. *isEmpty()*: return *true* if the queue is empty, *false* otherwise
3. *enqueue(e)*: *e* is added as the last item in the queue
4. *examine()*: return the first item, error if the queue is empty
5. *dequeue()*: remove and return first item, error if queue empty

1. Educational Aims

The aims of this topic are to:

1. Introduce two main ways of implementing collection classes:
 - block (array-based) implementations, and
 - linked (recursive) implementations
2. Introduce pros and cons of the two structures.
3. Develop basic skills in manipulating these two kinds of structures.

Note the ADT just specifies the operations available, and the results of applying those operations. There are many different ways to implement any given ADT.

2.1 Classification of ADT Operations:

constructors are used to create data structure instances

eg. *Queue*

checkers report on the “state” of the data structure

eg. *isEmpty*

manipulators examine and modify data structures

eg. *enqueue*, *examine*, *dequeue*

3. Interface

```
import CITS2200.*;

public interface Queue {    // some javadoc comments omitted

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

    /**
     * add a new item to the queue
     * @param a the item to add
     */
    public void enqueue (Object a);
```

© Tim French

CITS2200 Queues Slide 5

```
/**
 * examine the first item in the queue
 * @return the first item
 * @exception Underflow if the queue is empty
 */
public Object examine () throws Underflow;

/**
 * remove the first item in the queue
 * @return the first item
 * @exception Underflow if the queue is empty
 */
public Object dequeue() throws Underflow;
```

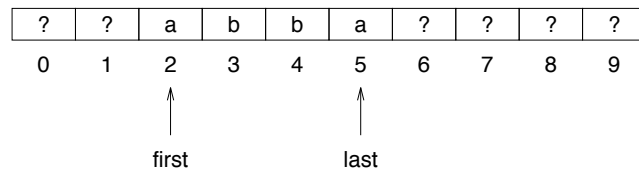
© Tim French

CITS2200 Queues Slide 6

4. Block Representations

Simplest representation:

- sequence of elements stored in array
- indices (counters) indicating first and last element



© Tim French

CITS2200 Queues Slide 7

Disadvantage: queue will be bounded! — can only implement a variation on the spec:

3. *enqueue(e)*: e is added as the last item in the queue, *or error if the queue is full*

For convenience, we will include another checker:

6. *isFull()*: return *true* if the queue is full, *false* otherwise

© Tim French

CITS2200 Queues Slide 8

4.1 Class Declaration

```
import CITS2200.*;

/**
 * Block representation of a queue.
 * The queue is bounded.
 */
public class QueueBlock implements Queue {
```

Notice implementing interface — class will only compile without error if it provides all methods specified in the interface. (Otherwise you can declare the class as *abstract*).

```
/**
 * an array of queue items
 */
private Object[] items;

/**
 * index for the first item
 */
private int first;

/**
 * index for the last item
 */
private int last;
```

4.2 Modifiers

enqueue, *examine* and *dequeue* are straightforward...

```
/**
 * add a new item to the queue
 * @param a the item to add
 * @exception Overflow if queue is full
 */
public void enqueue (Object a) throws Overflow {
    if (!isFull()) {
        last++;
        items[last] = a;
    }
    else throw new Overflow("enqueueing to full queue");
}
```

```
/**
 * examine the first item in the queue
 * @return the first item
 * @exception Underflow if the queue is empty
 */
public Object examine () throws Underflow {
    if (!isEmpty()) return items[first];
    else throw new Underflow("examining empty queue");
}
```

```

/**
 * remove the first item in the queue
 * @return the first item
 * @exception Underflow if the queue is empty
 */
public Object dequeue() throws Underflow {
    if (!isEmpty()) {
        char a = items[first];
        first++;
        return a;
    }
    else throw new Underflow("dequeuing from empty queue");
}

```

```

/**
 * test whether the queue is empty
 * @return true if the queue is empty, false otherwise
 */

public boolean isEmpty () {return first == last + 1;}

```

Java arrays number from 0, so `first` is initialised to 0...

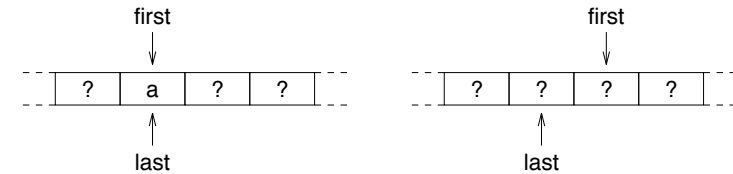
```

/**
 * initialise a new queue
 * @param size the size of the queue
 */
public QueueBlock (int size) {
    items = new char[size];
    first = 0;
    last = -1;
}

```

4.3 Constructors and Checkers

To see how to code the constructor and `isEmpty` consider successive deletions until `first` catches `last`.



The queue has one element if `first == last`, and is therefore empty when `first == last + 1`...

The queue is full if there is simply no room left in the array...

```

/**
 * test whether the queue is full
 * @return true if the queue is full, false otherwise
 */
public boolean isFull () {return last == items.length - 1;}

```

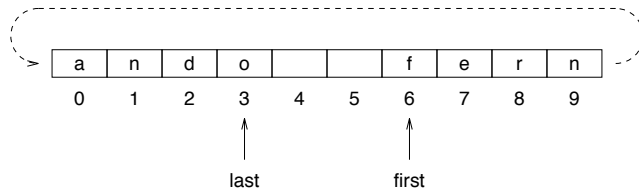
Notes

- `length` is an instance variable of an array object, and contains the size of the array.
- Since arrays number from 0, the n^{th} element has index $n - 1$.

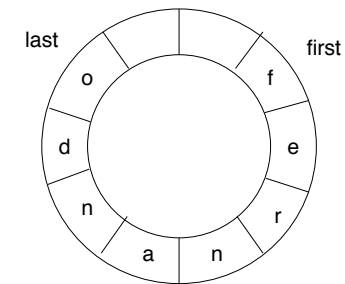
4.4 Alternative Block Implementations

Problem: as elements are deleted the amount of room left for the queue is eroded — the space in the array is not reused.

Solution: wrap queue around...



Conceptually, this forms a *cyclic queue* (or *cyclic buffer*)...



Effects on the above program...

- **first** and **last** must be incremented until they reach the end of the array, then reduced to 0. This can be achieved in a concise way using the % ("mod") operation. eg:

```
public void enqueue (Object a) {  
    if (!isFull()) {  
        last = (last + 1) % items.length;  
        items[last] = a;  
    }  
    else throw new Overflow("enqueueing to full queue");  
}
```

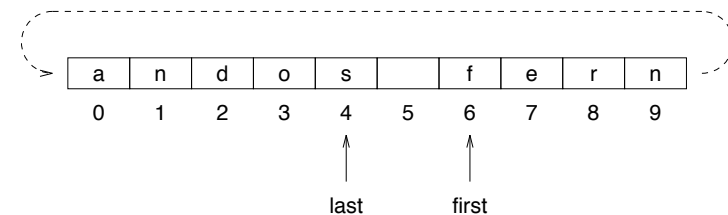
- A queue is now empty when:

```
first == (last + 1) % items.length
```

Problem: The above condition also represents a full queue!

One solution — define queue as full when it contains `items.length-1` elements and use the condition:

```
first == (last + 2) % items.length
```



But now a queue created to hold n objects only has room for $n - 1$ objects

⇒ modify the constructor...

```
public QueueCyclic (int size) {  
    items = new char[size+1];    // add 1 to array size  
    first = 0;  
    last = size;                // start last at end of block  
}
```

Another solution — instead of two indices, keep one index for the first element, and a count of the size of the queue.

⇒ Exercises!

5. Recursive (Linked) Representation

Biggest problem with block representation — predefined queue length

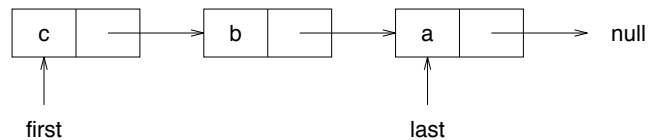
Solution: use a recursive structure!

Recall singly linked list...



For a queue, we need to be able to access both ends — one to insert and one to delete.

Although the end can be accessed by following the references down the list, it is more efficient to store references to both ends...



Note, it is important that the arrows point from first to last.

5.1 Class Declaration

```
import CITS2200.*;  
/**  
 * Linked list representation of a queue of characters.  
 */  
public class QueueLinked implements Queue {  
  
    /**  
     * the front of the queue, or null if queue's empty  
     */  
    private Link first;  
  
    /**  
     * the back of the queue, or null if queue's empty  
     */  
    private Link last;  
}
```

5.2 Constructors and Checkers

Empty queue:

first → null
last → null

Queue and *isEmpty* are easy...

```
/**
 * initialise a new Queue
 */
public QueueLinked () {
    first = null;
    last = null;
}

/**
 * test whether the queue is empty
 * @return true if the queue is empty, false otherwise
 */
public boolean isEmpty () {return first == null;}
```

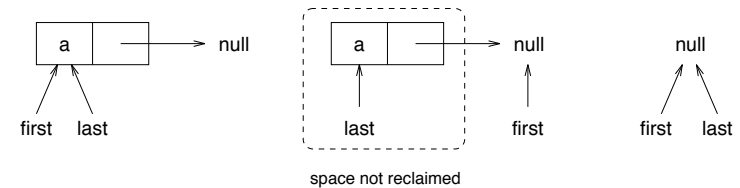
5.3 Examining and Dequeueing

Examining and dequeueing are easy!

Examining is the same as for the linked list...

```
public Object examine () throws Underflow {
    if (!isEmpty()) return first.item;
    else throw new Underflow("examining empty queue");
}
```

Dequeueing is the same as deleting in the linked stack, except that when the last item is dequeued, **last** must be assigned null...



```

public Object dequeue () throws Underflow {
    if (!isEmpty()) {
        Object o = first.item;
        first = first.successor;
        if (isEmpty()) last = null;
        return o;
    }
    else throw new Underflow("dequeuing from empty queue");
}

```

...unless the queue is empty, then **first** and **last** must both reference a new link...

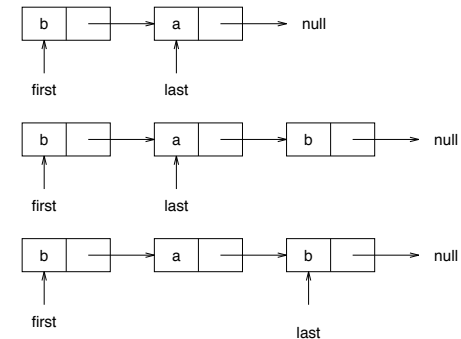
```

public void enqueue (Object a) {
    if (isEmpty()) {
        first = new LinkChar(a,null);
        last = first;
    }
    else {
        last.successor = new LinkChar(a,null);
        last = last.successor;
    }
}

```

5.4 Enqueueing

Enqueueing is also easy! Just reassign the null reference at the end of the queue to a reference to another link, and move **last** to the new last element. . .



6. Summary

- block (array with indices to endpoints)
 - bounded
 - may reserve space unnecessarily
 - ‘eroded’ with use
- block with wrap around (cyclic)
 - bounded
 - space reserved unnecessarily
 - not ‘eroded’
- recursive (linked list with references to endpoints)
 - unbounded
 - no unnecessary space wasted
 - no ‘erosion’ of space — garbage collection