Queues

Queue ADT: First-In First-Out

```
void enqueue (ElementType  x)
ElementType  dequeue( )
-------------------------------------
ElementType  front( )
boolean isEmpty( )
int size( )
```

Standard data structures for Queue ADT:

Circular array
Singly-linked list

In efficient implementations of a Queue, every operation is O(1) time

Queue implemented as a Circular Array

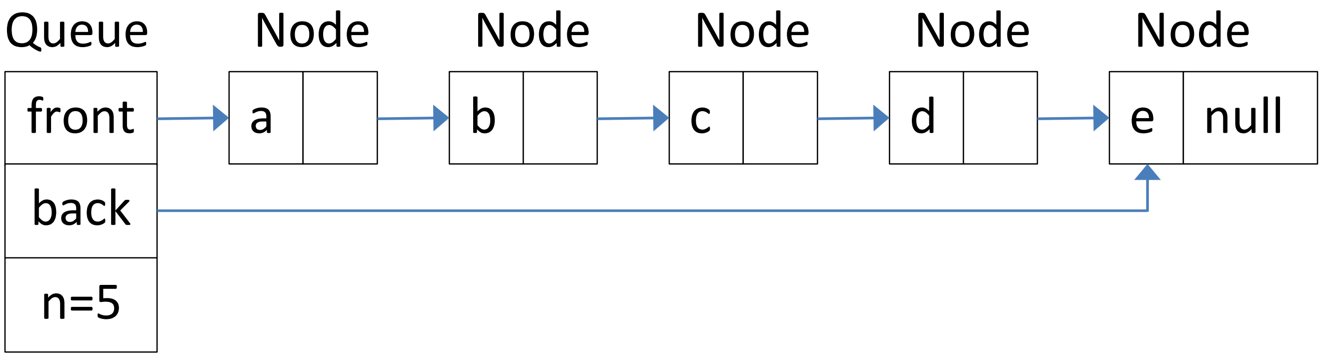
	0	1	2	3	4	5	6	7	…	MAX–1
array			a	b	c	d	e			
			front				back			

	0	1	2	3	…	MAX–3	MAX–2	MAX–1
array	d	e				a	b	c
		back				front		

```
class Queue {
        ElementType  array[MAX];
        int front, back, n;
        Queue( ) {
                front = 0;  back = MAX–1;  n = 0;
        }
        void enqueue (ElementType  x) {
                if (isFull( )) throw exception;
                back = (back + 1) % MAX;
                array[back] = x;
                n += 1;
        }
```

```
    ElementType  dequeue( ) {
        if (isEmpty( )) throw exception;
        ElementType x = array[front];
        front = (front + 1) % MAX;
        n –= 1;
        return x;
    }
    ElementType  front( ) {
        if (isEmpty( )) throw exception;
        return array[front];
    }
    boolean isEmpty( ) { return n == 0; }
    boolean isFull( ) { return n == MAX; }
    int size( ) { return n; }
}
```

Queue implemented as a Singly-Linked List



```
class Node {
        ElementType  data;
        Node next;
        Node (ElementType x, Node q) { data = x;  next = q; }
}
class Queue {
        Node front, back;
        int n;
        Queue( ) {
                front = null;  back = null;  n = 0;
        }
        void enqueue (ElementType  x) {
                Node p = new Node (x, null);
                if (isEmpty( ))  front = p;  else  back.next = p;
                back = p;
                n += 1;
        }
```

```
    ElementType  dequeue( ) {
        if (isEmpty( )) throw exception;
        ElementType x = front.data;
        front = front.next;
        if (front == null)  back = null;
        n –= 1;
        return x;
    }
    ElementType  front( ) {
        if (isEmpty( )) throw exception;
        return front.data;
    }
    boolean isEmpty( ) { return front == null; }
    int size( ) { return n; }
}
```

Applications of Queues:

- Buffer accessed by both producers and consumers

```
Queue Q( );

void producer( ) {
        while (true) {
                data = produceData( );   // application-dependent
                while (Q.isFull( ))  wait( );
                Q.enqueue (data);
        }
}

void consumer( ) {
        while (true) {
                while (Q.isEmpty( ))  wait( );
                data = Q.dequeue( );
                consumeData (data);   // application-dependent
        }
}
```

There are also several other issues (which are outside the scope of this course). For example, must be able to enforce that only one producer or consumer can modify the queue at a time, to maintain consistency.

- Simulation of transportation network (vehicles waiting at a traffic light, airport arrivals and departures)

- Simulation of customers waiting in line at a bank or supermarket

- Operating systems (process scheduler, print spooler)

- Routers in computer networks

- Traversing the nodes of a tree or graph (later in this course)