CS 112 – Introduction to Computing II
Wayne Snyder
Computer Science Department
Boston University

Today
Queues
Implementing a queue using a Circular (or Ring) Buffer;
Deques
Priority Queues
Reading: Wikipedia article on “Circular Buffers”

Next:
Analysis of Algorithms: How to measure the running time of algorithms
Iterative sorting: Selection Sort and Insertion Sort

Queue ADT

The Queue ADT is a simple variant of a stack which makes a simple change which in fact changes everything: instead of moving items in and out of the same “end” of the list, as in a stack:

Instead you use different ends of the list:
Queue ADT

This means that instead of reversing the order of the items, as with a stack, they remain in the same order; since you have stood in lines many times at Starbucks (or outside my office!), I'll only give a brief example:

enqueue(5);
Queue ADT

This means that instead of reversing the order of the items, as with a stack, they remain in the same order; since you have stood in lines many times at Starbucks (or outside my office), I’ll only give a brief example:

enqueue(5);
enqueue(7);

Enqueue 7 5 Dequeue

enqueue(2);

Enqueue 2 7 5 Dequeue
Queue ADT

This means that instead of reversing the order of the items, as with a stack, they remain in the same order; since you have stood in lines many times at Starbucks (or outside my office), I’ll only give a brief example:

enqueue(5);
enqueue(7);
enqueue(2);
int k = dequeue();

Enqueue 2 7 Dequeue

k = 5

Queue ADT

This means that instead of reversing the order of the items, as with a stack, they remain in the same order; since you have stood in lines many times at Starbucks (or outside my office), I’ll only give a brief example:

tenue(5);
enqueue(7);
enqueue(2);
int k = dequeue();
enqueue(8);

Enqueue 8 2 7 Dequeue

k = 5
Queue ADT

This means that instead of reversing the order of the items, as with a stack, they remain in the same order; since you have stood in lines many times at Starbucks (or outside my office), I'll only give a brief example:

enqueue(5);
debuee(7);
enqueue(2);
int k = dequeue();
debuee(8);
debuee( dequeue() )

Enqueue  7  8  2  Dequeue

k = 5

Queue ADT

Queues occur all the time, in real life:

And in computer systems (CPUs and Networks):

In fact, anywhere where one service is desired by many, and must be fairly distributed... there is a whole branch of math called “queueing theory” which you will learn about in CS 237 and CS 350.....
Queue ADT

The informal interface for a Queue is similar to that for a stack:

```java
public void enqueue(int n) -- Insert n at the read of the queue
public int dequeue() -- Remove the integer at the front of the queue and return it
public int peek() -- Return the number at the front of the queue without removing it
public int size() -- Return the number of integers in the queue
public boolean isEmpty() -- Return true if the queue is empty and false otherwise
```

Array-based Implementation of Queues

The Java Interface (subject of today's lab) for such an ADT is as follows:

```java
// Queueable Interface

public interface Queueable {
    void enqueue(int n); // insert at the rear of the queue
    int dequeue(); // Remove and return head of queue
    int peek(); // Return head of queue without removing
    boolean isEmpty();
    int size(); // returns number of integers in queue
}
```

How to implement this with arrays?
Array-based Implementation of Integer Queues

To implement an array-based queue for ints, here is the first thing you might think of....

```java
to dequeue() {
    int temp = A[front];
    ++front;
    return temp;
}

boolean isEmpty() {
    return (size() == 0);
}
```

But there is an obvious problem, and not so trivial..... running off the end of the array!

```java
void enqueue(int k) {
    A[next] = k;
    ++next;
}

int dequeue() {
    int temp = A[front];
    ++front;
    return temp;
}

boolean isEmpty() {
    return (size() == 0);
}
```
Array-based Implementation of Integer Queues

What solutions could we come up with for this problem?

Well, there are several:

Bad: Resize the array so you don’t run off the end. But then your array grows and grows and grows!

Good: Each time you dequeue, shift all the data over (similarly with how a queue is managed in Starbucks: when the person at the head of the line leaves, everyone moves up!). A natural solution, but if the queue is very large, each dequeue takes a long time, since you have to touch every data item and move it.

So, if you have:

A:

And you dequeue the -3, you need to shift the queue members to the right (towards 0) one slot:
Array-based Implementation of Integer Queues

Good: Each time you dequeue, shift all the data over (similarly with how a queue is managed in Starbucks: when the person at the head of the line leaves, everyone moves up!). A natural solution, but if the queue is very large, each dequeue takes a long time, since you have to touch every data item and move it.

Problem: For EVERY dequeue, you have to move EVERY number; we would like to avoid constantly moving the items..... so:

Best: Consider the array to be in a circle, with each end "glued" together, so that you never run off the array....

Array-based Implementation of Queues

In the ring or circular buffer approach, when we reach the end of the array we wrap around to the beginning:

In the fill count version of circular buffer, we keep track of the number of elements:

int size = 0;
int front = 0;
int next = 0;

How do we move the pointers front and next around the ring?
Array-based Implementation of Queues

The **standard solution** is to wrap around to the beginning of the array, creating a **circular buffer**:

```java
int size = 0;
int front = 0;
int next = 0;

// To move a pointer:
int nextSlot(int k) {
    return ((k + 1) % A.length);
}

void enqueue(int n) {
    int nextSlot = nextSlot(next);
    A[nextSlot] = n;
    ++size;
    next = nextSlot;
}
```
Array-based Implementation of Queues

```java
void enqueue(int n) {
    A[next] = n;
    next = nextSlot(next);
    ++size;
}

// To move a pointer:
int nextSlot(int k) {
    return ((k + 1) % A.length);
}
```

Array-based Implementation of Queues

```java
void enqueue(int n) {
    A[next] = n;
    next = nextSlot(next);
    ++size;
}

// To move a pointer:
int nextSlot(int k) {
    return ((k + 1) % A.length);
}
```
Array-based Implementation of Queues

next = 8  
front = 0  
size = 8

enqueue(5);  
enqueue(7);  
enqueue(12);  
enqueue(-3);  
enqueue(5);  
enqueue(0);  
enqueue(34);  
enqueue(9);

// To move a pointer:

int nextSlot(int k) {
    return ((k + 1) % A.length);
}

void enqueue(int n) {
    A[next] = n;
    next = nextSlot(next);
    ++size;
}

int dequeue() {
    int temp = A[front];
    front = nextSlot(front);
    --size;
    return temp;
}

// To move a pointer:

int nextSlot(int k) {
    return ((k + 1) % A.length);
}

void enqueue(int n) {
    A[next] = n;
    next = nextSlot(next);
    ++size;
}
Array-based Implementation of Queues

```java
// To move a pointer:
int nextSlot(int k) {
    return ((k + 1) % A.length);
}

int dequeue() {
    int temp = A[front];
    front = nextSlot(front);
    --size;
    return temp;
}

void enqueue(int n) {
    A[next] = n;
    next = nextSlot(next);
    ++size;
}
```

Array-based Implementation of Queues

```java
// can still underflow!
int dequeue() {
    int temp = A[front];
    front = nextSlot(front);
    --size;
    return temp;
}

boolean isEmpty() {
    return (size() == 0);
}
```
Array-based Implementation of Queues

int [] A = new int[10];
int size = 0;
int front = 0; int next = 0;

int nextSlot(int k) {
  return ((k + 1) % A.length);
}

void enqueue(int n) {
  A[next] = n;
  next = nextSlot(next);
  ++size;
}

int size() {
  return size;
}
// can still underflow!
int dequeue() {
  int temp = A[front];
  front = nextSlot(front);
  --size;
  return temp;
}

boolean isEmpty() {
  return (size() == 0);
}

Note: Can’t distinguish full or empty from the pointers alone, that is why we keep track of the size!
Array-based Implementation of Queues

Note: Can't distinguish full or empty from the pointers alone, that is why we keep track of the size!

int[] A = new int[10];
int size = 0;
int front = 0; int next = 0;

int nextSlot(int k) {
    return ((k + 1) % A.length);
}

void enqueue(int n) {
    A[next] = n;
    next = nextSlot(next);
    ++size;
}

int size() {
    return size;
}

// can still underflow!
int dequeue() {
    int temp = A[front];
    front = nextSlot(front);
    --size;
    return temp;
}

boolean isEmpty() {
    return (size() == 0);
}

Can solve overflow by resizing but it can still underflow!
Array-based Implementation of Queues

Circular or ring buffers are the standard technique for implementing queues and buffers in operating systems and many, many other applications!

Queue ADT: Two Important Variations

The Deque ("deck") ADT is a “double-ended queue” in which you can insert or remove from either end; it is either a queue going in both directions, or two stacks stuck together:

- enqueueRear(k): Insert the key k in the rear
- dequeueRear(): Remove and return the item from the rear of the list
- enqueueFront(k): Insert the key k in the front
- dequeueFront(): Remove and return the item from the front of the list

enqueueRear  dequeueRear
       ARRAY    dequeueFront
       enqueueFront
Queue ADT: Two Important Variations

The Priority Queue ADT is a queue in which the list is always kept ordered; this is useful when elements in the queue have a different need or right for service; the only change is in the enqueue method; they are typically called by different names:

- **insert(k):** Insert the key k in order in the list
- **getMax():** Remove and return the item in the front of the list

Queue ADT: Two Important Variations

The Priority Queue ADT is a queue in which the list is always kept ordered; this is useful when elements in the queue have a different need or right for service; the interface is usually defined with somewhat different names for the two basic operations, depending on whether it is a "maxQueue" (ordered so that biggest go to the front) or "minQueue" (smallest go to front).

- **insert(k):** Insert the key k in order in the list (cf. enqueue(k))
- **getMax() or getMin():** Remove and return the item in the front of the list (cf. dequeue())
Priority Queue ADT

insert(5);

Priority Queue ADT

insert(5);
insert(7);

insert(5) 5 getMax

insert 5 7 getMax
Priority Queue ADT

insert(5);
insert(7);
insert(2);

Priority Queue ADT

insert(5);
insert(7);
insert(2);
int k = getMax();

\[ k = 7 \]
Priority Queue ADT

```c
insert(5);
insert(7);
insert(2);
int k = getMax();
enqueue(8);
```

insert 2 5 8 getMax

k = 7

---

Priority Queue ADT

```c
insert(5);
insert(7);
insert(2);
int k = getMax();
insert(8);
insert(getMax());
```

insert 2 5 8 getMax

k = 7