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:

![Queue ADT Diagram]

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);

Enqueue 5  Dequeue
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);

Enqueue  7  5  Dequeue

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(8);

Enqueue: 2 7 8
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);
enqueue(7);
enqueue(2);
int k = dequeue();
enqueue(8);
enqueue( 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.....
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
```

Enqueue ➔ Dequeue

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();       // Return true if the queue is empty and false otherwise
    int size();              // returns number of integers in queue
}
```

Enqueue ➔ Dequeue

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.....

```
int 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!

```
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.

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:

\[
\begin{array}{c}
\text{front} = 0 \\
\text{next} = 0 \\
\text{size} = 0 \
\end{array}
\]

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

\[
\text{size} = 0
\]

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**:

```c
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) {
    if(size != A.length) {
        A[next] = n;
        next = nextSlot(next);
        ++size;
    }
}
```

Array-based Implementation of Queues

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

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

// To move a pointer:

int front = 0;
int next = 0;
int size = 1;

// enqueue(5);

front = 0
next = 1
size = 1

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

// To move a pointer:

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

int front = 0;
int next = 0;
int size = 2;

// enqueue(5);
enqueue(7);

front = 0
next = 2
size = 2

// To move a pointer:
Array-based Implementation of Queues

```c
void enqueue(int n) {
    if (size != A.length) {
        A[next] = n;
        next = nextSlot(next);
        ++size;
    }
}
```

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

`A:`

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

### Example

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

```
// To move a pointer:
```
Array-based Implementation of Queues

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

```java
next = 8
front = 1
size = 7
```

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

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

```java
int size() {
    return size;
}
```

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

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

---

Array-based Implementation of Queues

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

```java
int enqueue(int n) {
    if(size != A.length) {
        A[next] = n;
        next = nextSlot(next);
        ++size;
    }
}
```

```java
size = 2
```

```java
dequeue(); => 5
dequeue(); => 7
dequeue(); => 12
dequeue(); => -3
dequeue(); => 5
dequeue(); => 0
```

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

```java
int enqueue(int n) {
    if(size != A.length) {
        A[next] = n;
        next = nextSlot(next);
        ++size;
    }
}
```

```java
size = 2
```

```java
dequeue(); => 5
dequeue(); => 7
dequeue(); => 12
dequeue(); => -3
dequeue(); => 5
dequeue(); => 0
```

```java
int[] A = new int[10];
int size = 0;
int front = 0; int next = 0;
```
Array-based Implementation of Queues

```java
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) {
    if(size != A.length) {
        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!

```java
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) {
    if(size != A.length) {
        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!
```
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
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:

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

Priority Queue ADT

- **enqueue(5):**

```
  enqueue  5  dequeue
```
Priority Queue ADT

enqueue(5);
enqueue(7);

Priority Queue ADT

enqueue(5);
enqueue(7);
enqueue(2);
enqueue(2);
Priority Queue ADT

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

enqueue 2 5 dequeue
k = 7

Priority Queue ADT

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

enqueue 2 5 8 dequeue
k = 7
Priority Queue ADT

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

Queue ADT: Two Important Variations

The Priority Queue ADT 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())