CS 112 – Introduction to Computing II

Wayne Snyder
Computer Science Department
Boston University

Today
Stacks and Queues;
Priority Queues;
Queues implemented by Circular (or Ring) Buffers; [Reading: Wiki “Circular Buffers”]
Deques
Exceptions

Stack ADT

A Stack for integers could be defined by the following interface of public methods:

```
// Stack Interface

void push(int key);    // push the key onto the top of the stack
int pop();            // remove the top key and return it
int top();            // examine top element and return it without
                      // removing it from stack
boolean isEmpty();    // is the Stack empty?
int size();           // how many integers in the stack
```
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:

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:
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(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();

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 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);
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:

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.....

Array-based Implementation of Queues

A Queue for integers could be defined by the following interface:

```java
void enqueue(int key); // insert the key at the end of the queue
int dequeue(); // remove the key at front of the queue
boolean isEmpty(); // returns number of integers in queue
int size();
```

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.....

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

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

int size() {
    return (next - front);
}
```

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

```
void enqueue(int k) {
    if(size() != A.length) {
        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: Reallocate a bigger array so you don’t run off the end (we’ll talk about resizing arrays next week). But then your array grows and grows and grows!

Good: Each time you enqueue, 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. Enqueue takes $\Theta(1)$ but every dequeue takes $\Theta(N)$ time.

Best: Consider the array to be in a circle, with each end “glued” together, so that you never run off the array…. This will be $\Theta(1)$ for all operations!

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 diagram, the array is represented as a circle with elements 9, 8, 7, 6, 5, 4, 3, 2, 1, 0. The array is shown with slots labeled from 0 to 9, indicating the positions where elements are stored.

A:
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:

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

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

```
size = 0
```

How do we move the pointers `front` and `next` around the ring?

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

```c
#include <stdio.h>

#define A 10
#define nextSlot(k) ((k + 1) % A)

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

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

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

```
// enqueue operation
enqueue(5);
```

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

```
Array-based Implementation of Queues

enqueue(5);
enqueue(7);

// 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

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;
}
```
Array-based Implementation of Queues

next = 8
front = 0

A:

// 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;
}

size = 8

Array-based Implementation of Queues

dequeue() => 5

next = 8
front = 1

// 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;
}

size = 7
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);
}
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) {
    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);
}
```

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);
}
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
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 (and the names change):

There are two flavors: A MaxQueue or a MinQueue, depending on whether the element removed is the largest or smallest element:

- `put(k)`: Insert the key k into the priority queue
- `getMax()`: Remove and return the largest key in the queue
- `[ Or: `getMin()`: Remove and return the smallest key in the queue. ]

```
put(5);
putMax
```

Priority Queue ADT

```
put(5);
```

```
put 5 getMax
```
Priority Queue ADT

```plaintext
put(5);
put(7);
```

```
5 7
```

Priority Queue ADT

```plaintext
put(5);
put(7);
put(2);
```

```
2 5 7
```

getMax
Priority Queue ADT

```java
put(5);
put(7);
put(2);
int k = getMax();
```

![Priority Queue Diagram](image)

```
k = 7
```

---

Priority Queue ADT

```java
put(5);
put(7);
put(2);
int k = getMax();
put(8);
```

![Priority Queue Diagram](image)

```
k = 7
```
Priority Queue ADT

```java
put(5);
put(7);
put(2);
int k = getMax();
put(8);
put(getMax());
```

```
put 2 5 8 getMax
```

```
k = 7
```

Exceptions for Error Handling in Java

To this point, we have not dealt with how to report and recover from errors in Java; for example, with `IntStack.java`:

```java
public class IntStack {
    private int[] A = new int[20];
    private int next = 0;

    public void push(int n) {
        A[next] = n;
        ++next;
    }

    public int pop() {
        --next;
        return A[next];
    }
}
```

```java
IntStack S = new IntStack();
S.push(3);
3
```
To this point, we have not dealt with how to report and recover from errors in Java; for example, with IntStack.java:

```java
public class IntStack {
    private int[] A = new int[20];
    private int next = 0;

    public void push(int n) {
        A[next] = n;
        ++next;
    }

    public int pop() {
        --next;
        return A[next];
    }
}
```

IntStack S = new IntStack();
S.push(3);
S.push(4);
System.out.println(S.pop());

3
4

---
Exceptions for Error Handling in Java

To this point, we have not dealt with how to report and recover from errors in Java; for example, with `IntStack.java`:

```java
public class IntStack {
    private int[] A = new int[20];
    private int next = 0;

    public void push(int n) {
        A[next] = n;
        ++next;
    }

    public int pop() {
        --next;
        return A[next];
    }
}
```

```java
IntStack S = new IntStack();
S.push(3);
S.push(4);
System.out.println(S.pop());
System.out.println(S.pop());
```

This is called Stack Underflow.
Exceptions for Error Handling in Java

To this point, we have not dealt with how to report and recover from errors in Java; for example, with IntStack.java:

```java
public class IntStack {
    private int[] A = new int[20];
    private int next = 0;

    public void push(int n) {
        A[next] = n;
        ++next;
    }

    public int pop() {
        --next;
        return A[next];
    }
}
```

IntStack S = new IntStack();
for (int i = 1; i <= 20; ++i)
    S.push(i);

S.push(21)

```
java.lang.ArrayIndexOutOfBoundsException: 20
at IntStack.push(IntStack.java:52)
at IntStack.main(IntStack.java:133)
at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
at sun.reflect.NativeMethodAccessorImpl.invoke(Unknown Source)
at sun.reflect.DelegatingMethodAccessorImpl.invoke(Unknown Source)
at java.lang.reflect.Method.invoke(Unknown Source)
```
Exceptions for Error Handling in Java

Communicating errors and recovering from them is a big problem, and it is solved in Java by the mechanism of Exceptions. You have seen these already:

```
java.lang.ArrayIndexOutOfBoundsException: 20
 at IntStack.push(IntStack.java:52)
 at IntStack.main(IntStack.java:133)
 at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
 at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethod)
 at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl)
 at java.lang.reflect.Method.invoke(Method.java:597)
```

When a piece of code encounters a serious error, it throws an exception, which is an instance of a class that reports the error and terminates execution of that piece of code. By catching an exception, we can handle it and prevent the program itself from terminating.

Exceptions for Error Handling in Java

Exceptions are an essential way to deal with errors in Java, most commonly, you only have to deal with the simple case of an ADT throwing some exception that must be caught by the client. You have to remember a couple of things:

```java
public class IntStack {
    private int[] A = new int[20];
    private int next = 0;
    public void push(int n) {
        A[next] = n;
        ++next;
    }
    public int pop() {
        --next;
        return A[next];
    }
}
```

```java
class StackUnderflowException extends Exception {
    // could have members but usually not
}
```

1. An exception is an instance of a class, and can contain members; usually, the exception contains nothing, and the name itself is important.
Exceptions are an essential way to deal with errors in Java, most commonly, you only have to deal with the simple case of an ADT throwing some exception that must be caught by the client. You have to remember a couple of things:

1. An exception is an instance of a class, and can contain members; usually, the exception contains nothing, and the name itself is important.
2. You throw an exception when you encounter the condition/error by calling the constructor for the exception in a throw statement.
3. Any call to that method must be inside a try-catch block which catches that exception (or a superclass).

```java
public class IntStack {
    private int[] A = new int[20];
    private int next = 0;
    public void push(int n) {
        A[next] = n;
        ++next;
    }
    public int pop() {
        if (next == 0)
            throw new StackUnderflowException(); // default constructor for class
        --next;
        return A[next];
    }
}
class StackUnderflowException extends Exception {
    // could have members but usually not
}
```

```java
try {
    System.out.println(S.pop());
} catch (StackUnderflowException e) {
    System.out.println("Q underflew!");
}
```
Exceptions for Error Handling in Java

Exceptions are an essential way to deal with errors in Java, most commonly, you only have to deal with the simple case of an ADT throwing some exception that must be caught by the client. You have to remember a couple of things:

```java
public class IntStack {
    private int[] A = new int[20];
    private int next = 0;
    public void push(int n) {
        A[next] = n;
        ++next;
    }
    public int pop() throws StackUnderflowException {
        if (next == 0)
            throw new StackUnderflowException();
        --next;
        return A[next];
    }
}
class StackUnderflowException extends Exception {} // could have members but usually not
```

1. An exception is an instance of a class, and can contain members; usually, the exception contains nothing, and the name itself is important.
2. You throw an exception when you encounter the condition/error by calling the constructor for the exception in a throw statement.
3. Any call to that method must be inside a try-catch block which catches that exception (or a superclass).
4. The header of the method must list all exceptions that it throws.

```java
try {
    System.out.println(S.pop());
} catch (StackUnderflowException e) {
    System.out.println("Q underflow!");
}
```

```java
public class IntStack {
    private int[] A = new int[20];
    private int next = 0;
    public void push(int n) throws StackOverflowException {
        if (next == 20)
            throw new StackOverflowException();
        A[next] = n;
        ++next;
    }
    public int pop() throws StackUnderflowException {
        if (next == 0)
            throw new StackUnderflowException();
        --next;
        return A[next];
    }
}
class StackUnderflowException extends Exception {}
class StackOverflowException extends Exception {}
```

1. An exception is an instance of a class, and can contain members; usually, the exception contains nothing, and the name itself is important.
2. You throw an exception when you encounter the condition/error by calling the constructor for the exception in a throw statement.
3. Any call to that method must be inside a try-catch block which catches that exception (or a superclass).
4. The header of the method must list all exceptions that it throws.

```java
try {
    S.push(S.pop());
} catch (StackUnderflowException e) {
    System.out.println("Q underflow!");
} catch (StackOverflowException e) {
    System.out.println("Q overflow!");
}
```