

# CS313E: Elements of Software Design

## Another ADT: Queues

Dr. Bill Young  
Department of Computer Sciences  
University of Texas at Austin

Last updated: October 10, 2011 at 13:51

The queue is a FIFO structure. Items are put onto the *back* of the queue, and removed from the *front*.

Queues are useful in a wide variety of different applications. We'll see several later in this section. But they are also useful for system applications such as printer queues, job scheduling, etc.

What does the interface for the Queue ADT contain?

# Queue Interface

The following are possible operations in a queue interface. What are the semantics and how would they be implemented in Python?

Queue()	→ Queue
IsEmpty?()	→ Boolean
Enqueue(ItemType)	→ Queue
Dequeue()	→ ItemType
Len()	→ Integer

You might also want to ask what is the first element of the queue without removing it. Are there other useful operations you can imagine?

# Queues in Python

```
class MyQueue:
    def __init__(self):
        self.items = []

    def __str__(self):
        output = ""
        for x in self.items:
            output = output + str(x) + " "
        return "[" + output + "]"

    def __len__(self):
        return len(self.items)

    def isEmpty(self):
        return self.items == []

    def enqueue(self, item):
        self.items.insert(0, item)

    def dequeue(self):
        return self.items.pop()
```

Simulation is often used to study real situations. Some characteristics are:

- 1 A global clock “ticks” to keep track of the passing of time.
- 2 At each tick, the system “state” is updated to reflect changes.
- 3 Often, randomness is used to introduce unpredictable events into the simulation.
- 4 The simulation focuses on aspects of interest and ignores most details of the real system.

For example, if simulating a grocery store, customers may enter checkout lines randomly, move through the lines, and exit the simulation.

# Simulating a Neighborhood Market

Our goal is to simulate a neighborhood market with two checkout lines. Customers are added randomly according to some percentage possibility. Each customer has 1 to 10 items in his basket and each cashier can scan one item per tick of the simulator clock.

Step: 13

Customer joining Line 2

Line 1: [ C1(5) ]

Line 2: [ C2(9) ]

Step: 14

Customer joining Line 1.

Line 1: [ C1(4) C3(5) ]

Line 2: [ C2(8) ]

...

Step: 17

Line 1: [ C1(1) C3(5) ]

Line 2: [ C2(5) ]

Step: 18

Line 1: [ C3(5) ]

Line 2: [ C2(4) ]

# Market Simulation: Customer

```
import random
from Queue import *

class Customer:
    """A customer is generated with a random number of items
    in his basket. We assume it takes a cashier 1 tick to
    process each item."""

    def __init__(self, num):
        """The variable _itemsCount indicates not only the number of items
        in the customers basket, but also the time to check the customer
        out. We assume that a cashier can process one item per tick."""
        self._customerNumber = num
        self._itemsCount = random.randint(1, 10)

    def getCustomerNumber(self):
        return self._customerNumber

    def getItemsCount(self):
        return self._itemsCount

    def decrementItemsCount(self):
        self._itemsCount -= 1
```

# Market Simulation

```
    # This is the __str__ for Customers
def __str__(self):
    return "C" + str( self.getCustomerNumber() ) + \
           "(" + str( self.getItemsCount() ) + ")"
```

```
class Market:
```

```
    def __init__(self):
        self._line1 = MyQueue()
        self._line2 = MyQueue()

    def addToLine (self, q, c):
        """Add customer c to Line q."""
        if q == 1:
            self._line1.enqueue(c)
            print("Customer joining Line 1")
        else:
            self._line2.enqueue(c)
            print("Customer joining Line 2")
```

# Market Simulation

```
def chooseLine(self):
    """Select the shortest line, or randomly if equal."""
    if len( self._line1 ) < len( self._line2 ):
        return 1
    elif len( self._line1 ) > len( self._line2 ):
        return 2
    else:
        return random.choice([1, 2])

def shouldIAddCustomer(self, k):
    # k gives the percentage probability of
    # adding a customer this round. Example,
    # if k = 3, there's a 30% chance.
    return random.randint(0, 9) < k

def printMarketState(self):
    """Print the current state of the market's two lines."""
    print (" Line 1: ", self._line1)
    print (" Line 2: ", self._line2)
```

# Testing Our Customer Generator Approach

To test our approach to generating new Customers according to a given percentage, I wrote the following function:

```
>>> import random
>>> def TestRandomness( k, tests ):
    succeed = 0
    for i in range(tests):
        if random.randint(0, 9) < k:
            succeed += 1
    print ("For k = ", k, ": succeeded ", (succeed / tests) * 100, "%")

... .. >>>
>>> TestRandomness( 3, 10000)
For k = 3 : succeeded 30.3 %
>>> TestRandomness( 5, 10000)
For k = 5 : succeeded 50.59 %
>>> TestRandomness( 7, 100000)
For k = 7 : succeeded 70.191 %
>>> TestRandomness( 1, 100000)
For k = 1 : succeeded 10.121 %
>>> TestRandomness( 0, 100000)
For k = 0 : succeeded 0.0 %
```

# Market Simulation

```
def simulate (self, steps, k):
    custNumber = 0
    for i in range(steps):
        print ("\nStep: ", i)
        if ( not self._line1.isEmpty() ):
            if ( self._line1.peek().getItemsCount() <= 1 ):
                c = self._line1.dequeue()
            else:
                self._line1.peek().decrementItemsCount()
        if ( not self._line2.isEmpty() ):
            if ( self._line2.peek().getItemsCount() <= 1 ):
                c = self._line2.dequeue()
            else:
                self._line2.peek().decrementItemsCount()
        # Decide whether to add a customer
        if self.shouldIAddCustomer(k):
            newCustomer = Customer(custNumber)
            custNumber += 1
            self.addToLine( self.chooseLine(), newCustomer )
        self.printMarketState()
```

# Bounded Queue

Suppose you wanted never to allow a queue to grow to more than  $n$  elements. That's called a "Bounded Queue." *What would that ADT look like?*

The first thing to note is that a Bounded Queue *is* a Queue, but it has some additional constraints. So, why define a completely new ADT? Why not just *extend* the one we already have.

In Object Oriented Programming (OOP) we call that *inheritance*.

*But what things need to change, and which stay the same?*

# Inheritance: Rectangle Class

```
import math
class Rectangle(object):
    """Define a class of rectangles. Rectangles have an associated
    height and width."""

    def __init__(self, height, width):
        self._height = height
        self._width = width

    def __str__(self):
        return "Rectangle with height " + str(self._height) \
            + " and width " + str(self._width)

    def getHeight(self):
        return self._height

    def getWidth(self):
        return self._width

    def perimeter(self):
        return 2 * (self._height + self._width)
```

# Inheritance: The Square Class

```
# These are from the Rectangle class.
def area(self):
    return self._height * self._width

def diagonalLen(self):
    return math.sqrt( math.pow( self._height, 2) \
                      + math.pow( self._width, 2) )
```

```
class Square (Rectangle):
    """Define a class of squares. Squares are just
    rectangles, but height and width are equal."""

    def __init__(self, side):
        # Could use Rectangle.__init__( side, side )
        super( Square, self ).__init__( side, side )
        self._side = side

    def __str__(self):
        return "Square with side " + str(self._side)

    def getSide(self):
        return self._side
```

# Inheriting and Overriding

A subclass may have:

- methods new in this class;
- methods *inherited* from the parent class;
- methods that *override* (redefine) parent methods.

Class Square has the following methods:

Method	new/inherited/overridden
<code>__init__</code>	overridden
<code>__str__</code>	overridden
<code>getSide</code>	new
<code>getHeight</code>	inherited
<code>getWidth</code>	inherited
<code>perimeter</code>	inherited
<code>area</code>	inherited
<code>diagonalLen</code>	inherited

# Using the Classes

```
>>> from Rectangle import *
>>> r = Rectangle(4, 5)
>>> r.area()
20
>>> r.perimeter()
18
>>> r.getWidth()
5
>>> r.getSide()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Rectangle' object has no attribute 'getSide'
>>> s = Square(5)
>>> s.getWidth()
5
>>> s.getSide()
5
```

# Using the Classes

```
>>> s.area()
25
>>> s.perimeter()
20
>>> print(s)
Square with side 5
>>> print(s)
Square with side 5
>>> print(r)
Rectangle with height 4 and width 5
>>> s2 = Square(1)
>>> s2.diagonalLen()
1.4142135623730951
>>> x = s.diagonalLen()
>>> math.pow(x, 2)
2.0000000000000004
```

# The Queue ADT: What Must Change?

```
class MyQueue:
    def __init__(self):
        self._items = []

    def __str__(self):
        output = ""
        for x in self._items:
            output = output + str(x) + " "
        return "[" + output + "]"

    def __len__(self):
        return len(self._items)

    def isEmpty(self):
        return self._items == []

    def enqueue(self, item):
        self._items.insert(0, item)

    def dequeue(self):
        return self._items.pop()
```

# Bounded Queue

```
from MyQueue import *

class BoundedQueue(MyQueue):
    # This is an extension to the MyQueue ADT

    def __init__(self, bound):
        # When creating a BoundedQueue, must first create
        # a MyQueue.
        MyQueue.__init__(self)
        self._bound = bound

    def isFull(self):
        return len(self._items) == self._bound

    def enqueue(self, item):
        if self.isFull():
            print ("Enqueue failed because queue is full")
        else:
            self._items.insert(0, item)
```

# Bounded Queue

```
>>> q = BoundedQueue(4)
>>> q.enqueue("a")
>>> q.enqueue("b")
>>> q.enqueue("c")
>>> q.enqueue("d")
>>> q.enqueue("e")
Enqueue failed because queue is full
>>> print q
[ d c b a ]
>>> q.dequeue()
'a'
>>> print q
[ d c b ]
>>> q.isFull()
False
>>> q.enqueue("e")
>>> print q
[ e d c b ]
>>> q.isFull()
True
```

# Extending the Market Simulation

Suppose you wanted to extend our Market Simulation example to make each line into a `BoundedQueue`: only 5 Customers can be in a line before the management opens another checkout line.

What changes would you have to make to the program?

- 1 Market would have to have a list of lines, rather than a fixed number of lines.
- 2 Deciding which line to join would be more complicated.
- 3 The simulator would have to loop over all lines.
- 4 We'd have to decide if we ever close a line and how.

# Priority Queue

Suppose you wanted to have items in the queue of several different *priorities*. E.g., a hospital patient queue where critical patients are seen ahead of urgent patients who are seen ahead of routine patients.

It would be easy to have three queues. But how could you deal with this in *one* queue?

The answer is to define a *Priority Queue*, where higher priority items are inserted closer to the front of the queue. (Notice, it's no longer a FIFO structure.)

# Priority Queue

```
from MyQueue import *

class PriorityQueue(MyQueue):
    # This is an extension to the Queue ADT

    def __init__(self):
        MyQueue.__init__(self)

    def enqueue(self, item):
        """Insert at the point where we find something of
        greater priority."""
        if self.isEmpty():
            self._items.insert(0, item)
            return
        point = len(self)
        for i in range( len(self) ):
            if self._items[i] >= item:
                point = i
                break;
        self._items.insert(point, item)
```

# Comparing Arbitrary Objects

Suppose you have a list of objects that you'd like to sort. No problem if they are integers, or floats, or strings, or other types on which there is a “natural” order.

If not, you can define an order by defining the `__le__` function to compare them.

Typically, you define `__le__` within a class as:

```
def __le__(self, other):
```

```
    ...
```

**Note:** you also may have to define `__lt__` and/or `__eq__` to make this work.

# Defining the Comparison

```
class Widget:
    def __init__(self, tag, priority):
        self._tag = tag
        self._priority = priority

    def __str__(self):
        return "< " + self._tag + ", " + \
            str(self._priority) + " >"

# By defining these methods, we allow comparisons between
# Widgets using the standard notation: w1 < w2, w1 >= w2, etc.

    def __lt__(self, other):
        return self._priority < other._priority

    def __le__(self, other):
        return self._priority <= other._priority

# Without this "w1 == w2" does component-wise
# equality.

    def __eq__(self, other):
        return self._priority == other._priority
```

# And Using It

```
>>> from PriorityQueue import *
>>> from Widget import *
>>> pq = PriorityQueue()
>>> w1 = Widget("foo", 2)
>>> print (w1)
< foo, 2 >
>>> w2 = Widget("bar", 3)
>>> w3 = Widget("baz", 1)
>>> pq.enqueue(w1)
>>> print (pq)
[ < foo, 2 > ]
>>> pq.enqueue(w2)
>>> print (pq)
[ < foo, 2 > < bar, 3 > ]
>>> pq.enqueue(w3)
>>> print (pq)
[ < baz, 1 > < foo, 2 > < bar, 3 > ]
```

# Using a Priority Queue

Suppose in your Widget Works, some special, longtime customers should get their orders processed faster than newer customers.

How might you use a `PriorityQueue` to implement this?

Add an additional *priority* field to the `Order` type and add the functions necessary to compare `Orders` according to priority.

# Can We Do Both? Bounded Priority Queue

There's nothing to prevent defining a Queue that is both Bounded and a Priority Queue. That is easy in Python with *multiple inheritance*.

```
import ParentClass1
import ParentClass2

class ChildClass (ParentClass1, ParentClass2):
    ...
```

# BoundedPriorityQueue

```
from BoundedQueue import *
from PriorityQueue import *

class BoundedPriorityQueue(BoundedQueue, PriorityQueue):

    def __init__(self, bound):
        BoundedQueue.__init__(self, bound)
        PriorityQueue.__init__(self)

    def enqueue(self, item):
        """Insert at the point where we find something of greater priority.
        But also take the boundedness into account."""
        if self.isFull():
            print ("Enqueue failed because queue is full")
            return
        # otherwise, insert the item at the appropriate spot.
        point = len(self)
        for i in range( len(self) ):
            if self._items[i] >= item:
                point = i
                break;
        self._items.insert(point, item)
```

# Using the BoundedPriorityQueue

```
>>> from Widget import *
>>> from BoundedPriorityQueue import *
>>> bpq = BoundedPriorityQueue(3)
>>> w1 = Widget("red", 2)
>>> print (w1)
< red, 2 >
>>> w2 = Widget("blue", 3)
>>> w3 = Widget("white", 1)
>>> w4 = Widget("mauve", 2)
>>> bpq.enqueue(w1)
>>> print (bpq)
[ < red, 2 > ]
>>> bpq.enqueue(w2)
>>> print (bpq)
[ < red, 2 > < blue, 3 > ]
>>> bpq.enqueue(w3)
>>> print (bpq)
[ < white, 1 > < red, 2 > < blue, 3 > ]
>>> bpq.enqueue(w4)
Enqueue failed because queue is full
>>> print (bpq)
[ < white, 1 > < red, 2 > < blue, 3 > ]
```

# Where the Methods are Defined

A BoundedPriorityQueue has the following methods, defined in:

<b>Method</b>	<b>from class</b>
<code>__init__</code>	BoundedPriorityQueue
<code>__str__</code>	MyQueue
<code>__len__</code>	MyQueue
<code>enqueue</code>	BoundedPriorityQueue
<code>dequeue</code>	MyQueue
<code>isEmpty</code>	MyQueue
<code>isFull</code>	BoundedQueue