CS313E: Elements of Software Design

Another ADT: Queues

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

Last updated: October 4, 2012 at 16:50

Queues

Another useful Abstract Data Type is the *queue*. The queue is a FIFO (first in, first out) structure; items are put onto the *back* of the queue, and removed from the *front*.

Queues are useful in a wide variety of 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 might the interface for the Queue ADT contain?

Queue Interface

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

Queue()	$\longrightarrow$ Queue
isEmpty()	$\longrightarrow$ boolean
enqueue(item)	$\longrightarrow$ Queue
dequeue()	$\longrightarrow$ item
peek()	$\longrightarrow$ item
len()	$\longrightarrow$ integer
str()	$\longrightarrow$ string

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

Python3 has a defined queue class, but we'll be defining our own.
Fill in the missing implementations.

```
class MyQueue:
    def __init__(self):
        self.items = []
    def __str__(self):
        ...
    def enqueue(self, item):
        self.items.insert(0, item)
    def dequeue(self):
        return self.items.pop()
    def __len__(self):
        ...
    def isEmpty(self):
        ...
    def peek(self):
        ...
```

Using MyQueue

```
from MyQueue import *
>>> q = MyQueue()
>>> q.enqueue( 3 )
>>> q.enqueue( 5 )
>>> q.enqueue( 7 )
>>> print( q )
[ 3 5 7 ]
>>> q.dequeue()
3
>>> print( q )
[ 5 7 ]
>>> len( q )
2
>>> q.enqueue( 9 )
>>> q
<MyQueue.MyQueue object at 0xb750cd2c>
>>> print( q )
[ 5 7 9 ]
```

Using MyQueue

```
>>> print( q )
[ 5 7 9 ]
>>> q.peek()
5
>>> print( q )
[ 5 7 9 ]
>>> q.dequeue()
5
>>> q.dequeue()
7
>>> q.dequeue()
9
>>> q.dequeue()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "MyQueue.py", line 23, in dequeue
    return self._items.pop()
IndexError: pop from empty list
```

Queue Application: Simulation

Constructive simulation means building a computer model of a real situation. Some characteristics are:

1. A clock "ticks" to keep track of the passing of time.
2. At each tick, the system "state" is updated to reflect changes during that time interval.
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. *Simplifying assumptions are always necessary.*

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 Design

Let's think about this problem in an object-oriented fashion. What are the Objects relevant to this problem?

Market Simulation Design

Let's think about this problem in an object-oriented fashion. What are the Objects relevant to this problem?

Perhaps:

- Customers
- Checkout lines
- Market (the entire system)

To make it realistic, we'll add an element of randomness in:

- how often customers arrive at the checkout lines;
- how many items they have in their baskets.

Customer Interface

Of course, we'll define Customer as an ADT (class). What is the interface?

Customer Interface

Of course, we'll define Customer as an ADT (class). What is the interface?

Customer()	$\longrightarrow$ Customer object
getCustomerNumber()	$\longrightarrow$ integer
getItemsCount()	$\longrightarrow$ integer
decrementItemsCount()	$\longrightarrow$ Customer
str()	$\longrightarrow$ string

Market Simulation: Customer Class

```
import random

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."""

    CustomerNumber = 0

    def __init__(self):
        """The variable _itemsCount indicates not only the number
        of items in the customer's basket, but also the time to
        check the customer out.  We assume that a cashier can process
        one item per tick of the clock."""

        self._customerNumber = Customer.CustomerNumber
        self._itemsCount = random.randint(1, 10)
        Customer.CustomerNumber += 1
```

Market Simulation: Customer Class (2)

```
# Customer class continued

def getCustomerNumber(self):
    return self._customerNumber

def getItemsCount(self):
    return self._itemsCount

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

def __str__(self):
    return "C" + str( self.getCustomerNumber() ) + \
           "(" + str( self.getItemsCount() ) + ")"
```

Line Class Interface

Another ADT is the Line. What is the interface? That is, what actions does a line take?

Line Class Interface

Another ADT is the Line. What is the interface? That is, what actions does a line take?

Line()	$\longrightarrow$ Line object
isEmpty()	$\longrightarrow$ boolean
customerArrives()	$\longrightarrow$ Line object
customerLeaves()	$\longrightarrow$ Line object
firstInLine()	$\longrightarrow$ Customer object
len()	$\longrightarrow$ integer
str()	$\longrightarrow$ string

Market Simulation: Line Class

```
from MyQueue import *
class Line:
    """A market line acts like a Queue.  Customers enter and move through
    the line.  At each tick, one item is removed from the basket of the
    first customer in line.  When the basket is empty, the customer pays
    and departs the line."""

    def __init__(self, num):
        """Open a new line, with no customers initially."""
        self._line = MyQueue()
        self._lineNumber = num

    def isEmpty(self):
        return self._line.isEmpty()

    def customerArrives(self, c):
        """Add a new customer to the line."""
        self._line.enqueue(c)
        print( "Customer joining Line", self._lineNumber)
```

Market Simulation: Line Class (2)

```
# Line class continued

def customerLeaves(self):
    """The first customer in line departs."""
    self._line.dequeue()

def firstInLine(self):
    return self._line.peek()

def __str__(self):
    return "Line " + str(self._lineNumber) + ":" + str(self._line)

def __len__(self):
    return len(self._line)
```

Market Simulation

```
class Market:
    """Our goal is to simulate a market with two checkout lines.
    Customers are added randomly according to a percentage
    possibility that we specify on the simulate method.
    Each customer has 1 to 10 items in his basket and each cashier
    can scan one item per tick of the simulator clock."""

    def __init__(self):
        self._line1 = Line(1)
        self._line2 = Line(2)

    def getLine(self, k):
        if k == 1:
            return self._line1
        elif k == 2:
            return self._line2
        else:
            print("Error: No such line number")
```

Market Simulation

```
# Continuation of the Market class

def joinLine (self, c):
    """Add a new customer to the shortest line.  If they're the
    same length, choose one at random."""
    if len( self.getLine(1) ) < len( self.getLine(2) ):
        self.getLine(1).joinLine( c )
    elif len( self.getLine(1) ) > len( self.getLine(2) ):
        self.getLine(2).joinLine( c )
    else:
        r = random.choice([1, 2])
        self.getLine(r).joinLine( c )

def shouldIAddCustomer(self, k):
    """Given a number k, return a Boolean where the
    probability of True is k/10.  Thus if k = 3,
    the probability of True is 30%."""
    return random.randint(0, 9) < k

def __str__(self):
    """The state of the market is just the state of the
    two checkout lines."""
    return str(self._line1) + "\n" + str(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

```
# Continuation of the Market class

def advanceLine( self, lineNum ):
    line = self.getLine( lineNum )
    if ( not line.isEmpty() ):
        # See if the first customer in line is almost finished.
        if ( line.firstInLine().getItemsCount() <= 1 ):
            line.customerLeaves()
        else:
            # If not, scan one item.
            line.firstInLine().decrementItemsCount()
```

Market Simulation

```
    def simulate (self, steps, k):
        """Simulate is the driver for this system. ... """

        print( "The simulation will run for ", steps, " steps.", sep="")
        print( "A new customer is added appr. every", k, "of 10 steps." )
        custNumber = 1
        for i in range(1, steps+1):
            print ( "\nStep: ", i )
            self.advanceLine( 1 )
            self.advanceLine( 2 )

            # Decide whether to add a new customer
            if self.shouldIAddCustomer(k):
                # If so, create a new customer.
                newCustomer = Customer()
                custNumber += 1
                # Add customer to the shortest line.
                self.joinLine( newCustomer )
            # Finally, print the new state of the Market.
            print( self )

if __name__ == '__main__':
    M = Market()
    M.simulate(20, 3)
```

Running the Simulation

```
felix:~/cs313e/python> python Market.py
The simulation will run for 20 steps.
A new customer is added approximately every 3 of 10 steps.

Step:  1
Customer joining Line 1
Line 1:[ C0(6) ]
Line 2:[ ]

Step:  2
Line 1:[ C0(5) ]
Line 2:[ ]

...

Step:  5
Customer joining Line 2
Line 1:[ C0(2) ]
Line 2:[ C1(7) ]

Step:  6
Line 1:[ C0(1) ]
Line 2:[ C1(6) ]
```

Running the Simulation (2)

```
Step:  7
Line 1:[ ]
Line 2:[ C1(5) ]

...

Step:  10
Customer joining Line 1
Line 1:[ C2(6) ]
Line 2:[ C1(2) ]

Step:  11
Line 1:[ C2(5) ]
Line 2:[ C1(1) ]

Step:  12
Line 1:[ C2(4) ]
Line 2:[ ]

Step:  13
Customer joining Line 2
Line 1:[ C2(3) ]
Line 2:[ C3(8) ]
```

Running the Simulation (3)

```
...

Step:  17
Customer joining Line 1
Line 1:[ C4(6) ]
Line 2:[ C3(4) ]

Step:  18
Customer joining Line 1
Line 1:[ C4(5) C5(3) ]
Line 2:[ C3(3) ]

Step:  19
Customer joining Line 2
Line 1:[ C4(4) C5(3) ]
Line 2:[ C3(2) C6(6) ]

Step:  20
Line 1:[ C4(3) C5(3) ]
Line 2:[ C3(1) C6(6) ]
felix:~/cs313e/python>
```

Why Simulate?

You can see why a person opening a Market might want to do this sort of simulation.

- If you have too few checkout lines, the line length will grow longer and longer.
- If you have too many lines, most of the time checkers will be idle.

Analysis assumes that the model parameters (how often customers arrive, how many items they buy, how quickly the checkers scan) reflect the reality of the situation being modeled.

Similar (but much more complicated) techniques are use for battlefield simulation, training airline pilots, weather modeling, on-line gaming, and a million other applications.

Event Based Simulation

The type of simulation illustrated above is pretty standard, but not very efficient. The only times you *really care* what's going on in the simulation is when something interesting happens:

- a new customer arrives;
- a customer finishes checking out.

In *event-based simulation*, you keep a list (queue) of interesting upcoming events. Then, instead of ticking the clock, you compute what the state will be at the next event and advance the clock to that point. How could you turn our grocery store simulation into an event based simulation?

Inheritance: The Bounded Queue ADT

Suppose you wanted to limit a queue so that it never contained more than n elements. That's called a *Bounded Queue*. What would that interface look like?

The first thing to note is that a Bounded Queue *is just a Queue*, with 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.* Inheritance just means that I'm extending an existing class rather than defining one from scratch.

Bounded Queue Interface

Below is the Queue interface. For a BoundedQueue which things need to change, and which stay the same?

Queue()	$\longrightarrow$ Queue
isEmpty()	$\longrightarrow$ boolean
enqueue(item)	$\longrightarrow$ Queue
dequeue()	$\longrightarrow$ item
peek()	$\longrightarrow$ item
len()	$\longrightarrow$ integer
str()	$\longrightarrow$ string

Bounded Queue Interface

Below is the Bounded Queue interface. Notice that it's identical to the Queue interface except for one new function `isFull`.

Queue()	$\longrightarrow$ Queue
isEmpty()	$\longrightarrow$ boolean
isFull?()	$\longrightarrow$ boolean
enqueue(item)	$\longrightarrow$ Queue
dequeue()	$\longrightarrow$ item
peek()	$\longrightarrow$ item
len()	$\longrightarrow$ integer
str()	$\longrightarrow$ string

Is that really the only change? Which functions will be identical to the Queue implementation? Which will have to change?

Bounded Queue Interface

Below is the Bounded Queue interface. Notice that it's identical to the Queue interface except for one new function `isFull`.

Queue()	$\longrightarrow$ Queue
isEmpty()	$\longrightarrow$ boolean
isFull?()	$\longrightarrow$ boolean
enqueue(item)	$\longrightarrow$ Queue
dequeue()	$\longrightarrow$ item
peek()	$\longrightarrow$ item
len()	$\longrightarrow$ integer
str()	$\longrightarrow$ string

Is that really the only change? Which functions will be identical to the Queue implementation? Which will have to change?

`isFull` is new. `Queue` (i.e., `__init__`) and `enqueue` will have to change. The others can remain the same.

Inheritance: Rectangle Class

Let's start with a simpler example than the Bounded Queue, our familiar 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
```

Inheritance: Rectangle Class

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

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

def area(self):
    return self._height * self._width

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

Inheritance: The Square Class

Notice a Square *just is a Rectangle*, with the special property that the width equals the height. So, why not not use that fact? That's what inheritance is all about.

```
# This line tells us that Square is a "subclass" of Rectangle.
class Square (Rectangle):
    """Define a class of squares. Squares are just
    rectangles, but height and width are equal."""

    def __init__(self, side):
        # When creating a Square, first create a Rectangle
        # Could use super( Square, self ).__init__( side, side )
        Rectangle.__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**
`__init__`	overridden
`__str__`	overridden
`getSide`	new
`getHeight`	inherited
`getWidth`	inherited
`perimeter`	inherited
`area`	inherited
`diagonalLen`	inherited

Using the Classes

Here Square and Rectangle are both defined in the Rectangle module (file).

```
>>> 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(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?

Below is the Queue class (without implementations).

```
class MyQueue:
    def __init__(self):
        ...
    def __str__(self):
        ...
    def __len__(self):
        ...
    def isEmpty(self):
        ...
    def enqueue(self, item):
        ...
    def dequeue(self):
        ...
```

Bounded Queue

Below is the BoundedQueue class definition. Note: there's no need to redefine any method that doesn't change.

```
from MyQueue import *
# This is an extension to the MyQueue ADT
class BoundedQueue(MyQueue):
    """A BoundedQueue is just a Queue that is not allowed
       to grow beyond a specified length."""

    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.

Bounded Stack

Now that you know how to extend MyQueue to BoundedQueue, think about how you'd define BoundedStack as an extension to your Stack class.

By the way, every programming language implementation contains a BoundedStack. If you have an infinite recursion in your program and it overflows the run-time stack, you've hit the Stack bound (which may be several thousand frames).

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:

Method	**from class**
`__init__`	BoundedPriorityQueue
`__str__`	MyQueue
`__len__`	MyQueue
`enqueue`	BoundedPriorityQueue
`dequeue`	MyQueue
`isEmpty`	MyQueue
`isFull`	BoundedQueue