

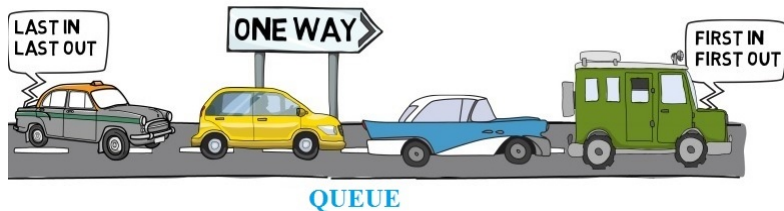
# CSI33 DATA STRUCTURES

Department of Mathematics and Computer Science  
Bronx Community College

## OUTLINE

## 1 CHAPTER 5: STACKS AND QUEUES

- Queues
- In-Class work



# A QUEUE ADT

## A CONTAINER CLASS FOR FIRST-IN-FIRST-OUT ACCESS

A **queue** is a **first-in-first-out** structure, i.e. it is a list-like container with access restricted to both ends of the list.

Items are added at the **back** of the queue and removed from the **front**. In real life: “waiting in line” or “taking a number”.

- The **enqueue** method puts an item at the back of the queue.
- The **dequeue** method returns the item at the front, and removes it from the queue.  
(precondition: queue is not empty— $\text{size} > 0$ )
- The **front** method returns that item  
(precondition: queue is not empty— $\text{size} > 0$ )
- The **size** method returns the number of items in the queue.

# A QUEUE ADT

## SPECIFICATION FOR A TYPICAL QUEUE

```
class Queue(object):
    def __init__(self):
        """ post:  creates an empty FIFO queue"""
    def enqueue(self,x):
        """post:  adds x at the back of the stack"""
    def dequeue(self):
        """pre:  self.size()>0
        post:  removes and returns the front item"""
    def front(self):
        """pre:  self.size()>0
        post:  returns the first item in the queue"""
    def size(self):
        """post:  returns number of elements in the queue"""
```

# A QUEUE ADT

## QUEUES APPLICATIONS

Queues used in computer programming as a sort of buffer between different phases of a computing process, for example:

- when printing documents  
(job requests are placed on a queue in OS),
- when compiling/interpreting a single program  
(**lexical analysis** splits the program into its meaningful pieces, the **tokens** that can be stored in a queue for subsequent processing by next phase, which is often some sort of grammar-based syntactic analysis)
- when determining whether or not a phrase is a palindrome  
(i.e. has the same sequence of letters when read either forward or backward).

## SIMPLE QUEUE APPLICATION: A PALINDROME RECOGNIZER

## A PALINDROME EXAMPLES

race car

Madam I'm Adam

I prefer PI

Never odd or even

**tricky part:** the palindromeness of a phrase is determined only by the letters; spaces, punctuation, and capitalization don't matter.

## A PALINDROME RECOGNIZER - SPECIFICATION

```
def isPalindrome(phrase):  
    """pre:  phrase is a string  
       post:  return True if the alphabetic characters  
in phrase form the same sequence reading either  
left-to-right or right-to-left."""
```

## SIMPLE QUEUE APPLICATION: A PALINDROME RECOGNIZER

## A PALINDROME RECOGNIZER

```
from MyQueue import Queue
from Stack import Stack

def isPalindrome(phrase):
    forward = Queue()
    reverse = Stack()
    extractLetters(phrase, forward, reverse)
    return sameSequence(forward, reverse)
```

## SIMPLE QUEUE APPLICATION: A PALINDROME RECOGNIZER

## PHASE I: EXTRACT LETTERS

```
def extractLetters(phrase, q, s):  
    for ch in phrase:  
        if ch.isalpha():  
            ch = ch.lower()  
            q.enqueue(ch)  
            s.push(ch)
```



## SIMPLE QUEUE APPLICATION: A PALINDROME RECOGNIZER

## PHASE II: SAME SEQUENCE

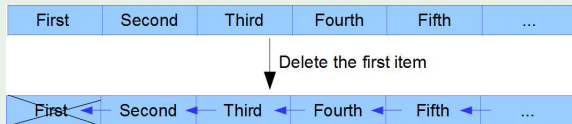
```
def sameSequence(q, s):  
    while q.size() > 0:  
        ch1 = q.dequeue()  
        ch2 = s.pop()  
        if ch1 != ch2:  
            return False  
    return True
```

# QUEUE IMPLEMENTATIONS

## A PYTHON LIST IS NOT AN EFFICIENT QUEUE

Implementation of **queue** with Python's built-in list, implemented as an array, is straightforward: we need to insert at one end, and remove from the other one. If we decide that the first item is at the:

- beginning of the list, then to remove an item from the queue (dequeue) is  $\Theta(n)$  – every item must be moved down to delete the first item,

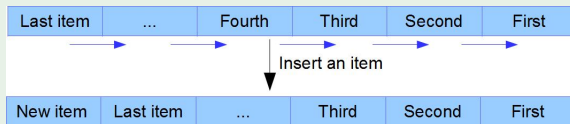


# QUEUE IMPLEMENTATIONS

## A PYTHON LIST IS NOT AN EFFICIENT QUEUE

Implementation of **queue** with Python's built-in list, implemented as an array, is straightforward: we need to insert at one end, and remove from the other one. If we decide that the first item is at the:

- end/back of the list, then to insert an item into the queue (enqueue) is  $\Theta(n)$  – every item must be moved up to insert the item.



# QUEUE IMPLEMENTATIONS

## LINKED LIST

An alternative: to use linked implementation.

This is the most flexible representation of a Queue ADT.

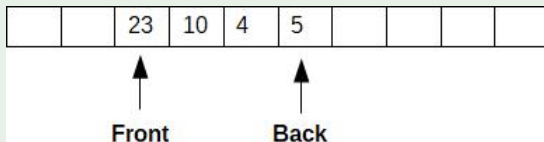
Use references to to the first node (front) and last node (back), a singly-linked list can perform enqueue and dequeue in constant time ( $\Theta(1)$ ).

However, the links take up extra memory space.

# QUEUE IMPLEMENTATIONS

## CIRCULAR ARRAY

A **circular array** avoids both the space inefficiency of links and the time inefficiency of the Python list (array) representation by not moving items. Instead, the front and back of the queue move, using changing indexes as markers.

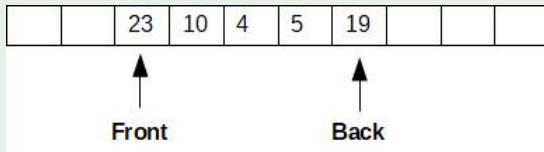


# QUEUE IMPLEMENTATIONS

## CIRCULAR ARRAY : ENQUEUE

To enqueue a new item: the index marking the back of the queue is increased by one.

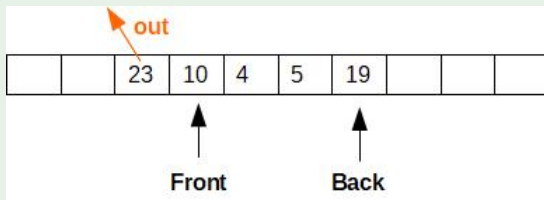
The item that was in the back stays in the same position, but the new item goes behind it, into the new “back” position.



# QUEUE IMPLEMENTATIONS

## CIRCULAR ARRAY : DEQUEUE

To dequeue an item: the front of the queue is moved to the next item behind it.

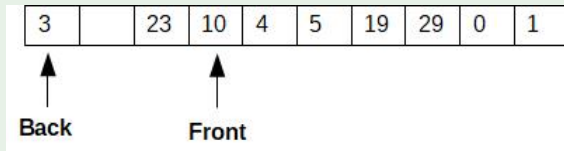


# QUEUE IMPLEMENTATIONS

## CIRCULAR ARRAY

The array is “circular” because when either marker goes past the end of the array, it is put back at index zero

$(\text{tail} = (\text{tail}+1) \% \text{size})$ .



check out this place: <http://www.yashcode.com/2017/11/queue-in-data-structure.html>



## SIMULATION OF RETAIL STORE WITH ONE CHECKOUT REGISTER

## A TYPICAL QUEUING SIMULATION

Queues can model the behavior of real-world queues. Consider a grocery store with one check-out register. To measure efficiency of service delivery, one runs a program that simulates these events:

- Random arrival times  
(customers finish shopping; they don't arrive at a constant rate)
- Waiting for service (grocery checkout register)
- Random time to be serviced  
(depends on number and kind of items)

## SIMULATION OF RETAIL STORE WITH ONE CHECKOUT REGISTER

Here is a function that will generate test data for us. Each customer has arrivalTime and itemCount.

```
from random import random, randrange

def genTestData(filename, totalTicks, maxItems, arrivalInterval):
    outfile = open(filename, "w")
    # step through the ticks
    for t in range(1, totalTicks):
        if random() < 1./arrivalInterval:
            # a customer arrives this tick
            # with a random number of items
            items = randrange(1, maxItems+1)
            outfile.write("%d %d\n" % (t, items))
    outfile.close()
```

totalTicks - length of time over which the simulation will be run  
arrivalInterval average number of ticks between arrivals

## SIMULATION OF RETAIL STORE WITH ONE CHECKOUT REGISTER

```
genTestData("data.txt", 2*60*60, 40, 180)
```

Generate data for two-hour simulation with up to 40 items for purchase and 3 minute average interval between customer arrivals.

```
from random import random, randrange
```

```
def genTestData(filename, totalTicks, maxItems, arrivalInterval):  
    outfile = open(filename, "w")  
    # step through the ticks  
    for t in range(1, totalTicks):  
        if random() < 1./arrivalInterval:  
            # a customer arrives this tick  
            # with a random number of items  
            items = randrange(1, maxItems+1)  
            outfile.write("%d %d\n" % (t, items))  
    outfile.close()
```

totalTicks - length of time over which the simulation will be run

arrivalInterval average number of ticks between arrivals

## SIMULATION OF RETAIL STORE WITH ONE CHECKOUT REGISTER

```
class Customer(object):
    def __init__(self, arrivalTime, itemCount):
        self.arrivalTime = int(arrivalTime)
        self.itemCount = int(itemCount)
    def __repr__(self):
        return ("Customer(arrivalTime=%d, itemCount=%d)" %
                (self.arrivalTime, self.itemCount))
```

## SIMULATION OF RETAIL STORE WITH ONE CHECKOUT REGISTER

```
def createArrivalQueue(fname):  
    q = Queue()  
    infile = open(fname)  
    for line in infile:  
        time, items = line.split()  
        q.enqueue(Customer(time,items))  
    infile.close()  
    return q
```

## SIMULATION OF RETAIL STORE WITH ONE CHECKOUT REGISTER

The simulation will be carried out in CheckerSim object.

```
class CheckerSim(object):
    def __init__(self, arrivalQueue, avgTime):
        self.time = 0    # ticks so far
        self.arrivals = arrivalQueue # arrival events to process
        self.line = Queue()    # customers waiting in line
        self.serviceTime = 0    #time left for current customer
        self.totalWait = 0
        self.maxWait = 0
        self.customerCount = 0    # number of customers processed
        self.maxLength = 0    # max line length
        self.ticksPerItem = avgTime
```

## SIMULATION OF RETAIL STORE WITH ONE CHECKOUT REGISTER

```
def run(self):
    while (self.arrivals.size() > 0 or
           self.line.size() > 0 or
           self.serviceTime > 0):
        self.clockTick()
def averageWait(self):
    float(self.totalWait) / self.customerCount
def maximumWait(self):
    return self.maxWait
def maximumLineLength(self):
    return self.maxLength
```

## SIMULATION OF RETAIL STORE WITH ONE CHECKOUT REGISTER

```
def clockTick(self):
    self.time += 1
    while (self.arrivals.size() > 0 and
           self.arrivals.front().arrivalTime == self.time):
        self.line.enqueue(self.arrivals.dequeue())
        self.customerCount += 1
    self.maxLength = max(self.maxLength, self.line.size())
    if self.serviceTime > 0:
        self.serviceTime -= 1
    elif self.line.size() > 0:
        customer = self.line.dequeue()
        self.serviceTime = customer.itemCount * self.ticksPerItem
        waitTime = self.time - customer.arrivalTime
        self.totalWait += waitTime
        self.maxWait = max(self.maxWait, waitTime)
```



## SIMULATION OF RETAIL STORE WITH ONE CHECKOUT REGISTER

The simulation is executed by calling its `run` method, which in turn executes `clockTick` method until the simulation is complete.

This particular approach is called **time-driven simulation**, i.e. we increment the clock one tick at a time and do whatever has to be done in that tick.

Any events in the `arrivalQueue` that occur at a given tick are moved into the line.

**drawback:** many of our cycles around the tick-loop will essentially be idle time.

**alternative approach:** use **event-driven** simulation, i.e. we don't model each tick, but simply "jump ahead" to the next event to be processed.

## SIMULATION OF RETAIL STORE WITH ONE CHECKOUT REGISTER

## IN-CLASS ASSIGNMENT

Modify the code to an **event-driven** version of the checker simulation.