

The Queue

Queues

The Queue ADT

Implementing a queue

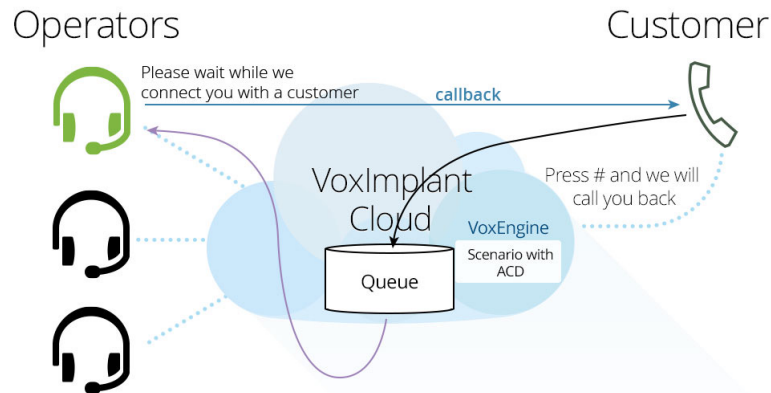
Improving the efficiency

Runtime example

A *queue* is a collection of objects, where :

- if we want to take an item, we take it from the *front*
- if we want to add an item, we add it onto the *back*

A queue is *first-in, first-out (FIFO)*



Queues in Computer Science

Queues are essential data structures in computing:

- packets being transmitted across the internet are queued for retransmission at each router
 - input and output buffers are queues – what you type first appears on the screen first
 - path planning algorithms maintain a queue of edges or locations to explore next
 - cloud computing maintains queues of work, queries, updates and access requests
- (and many more)

The Queue ADT

enqueue: add an item to the queue

dequeue: remove and return the item that has been in queue for longest time

front: report the item that has been in queue for longest time

length: report how many items are in the queue

The Queue ADT in Python

<code>enqueue(element)</code>	<code>#add element to the queue</code>
<code>dequeue()</code>	<code>#remove and return element that has been #in queue for longest time #(None if empty)</code>
<code>front()</code>	<code>#report element that has been in queue #for longest time (None if empty)</code>
<code>length()</code>	<code>#return the number of elements in queue</code>

Implementing the Queue

We will define a class, offering those methods.

How should we represent the data in the class?

The elements of the queue clearly have an order, and so we could use a sequence.

We will be doing a lot of adding elements to the sequence, and a lot of removing elements.

To add and delete most efficiently, we should do it at the end of the list ... but now we need to work at both ends?

```
class QueueV0:
    def __init__(self):
        self.body = []

    def enqueue(self, element):
        self.body.append(element)

    def dequeue(self):
        if len(self.body) == 0:
            return None
        return self.body.pop(0)

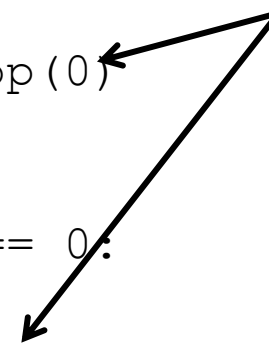
    def front(self):
        if len(self.body) == 0:
            return None
        return self.body[0]

    def length(self):
        return len(self.body)
```

Version 0

- internal python list
- add at end
- delete from front

Note differences to Stack



basic implementation

- no private variables
- no `__str__()` method

What is happening inside?

```
def test_queue():  
    myqueue = QueueV0()  
  
    myqueue.enqueue(1)  
  
    myqueue.enqueue('a')  
  
    myqueue.dequeue()  
  
    myqueue.enqueue(2)  
  
    myqueue.enqueue(3)  
  
    myqueue.enqueue(4)  
  
    myqueue.enqueue('b')  
  
    myqueue.enqueue('c')  
  
    myqueue.dequeue()
```

Queue: myqueue: []

Queue: myqueue: [1]

Queue: myqueue: [1, 'a']

Queue: myqueue: ['a']

Queue: myqueue: ['a', 2]

Queue: myqueue: ['a', 2, 3]

Queue: myqueue: ['a', 2, 3, 4]

Queue: myqueue: ['a', 2, 3, 4, 'b']

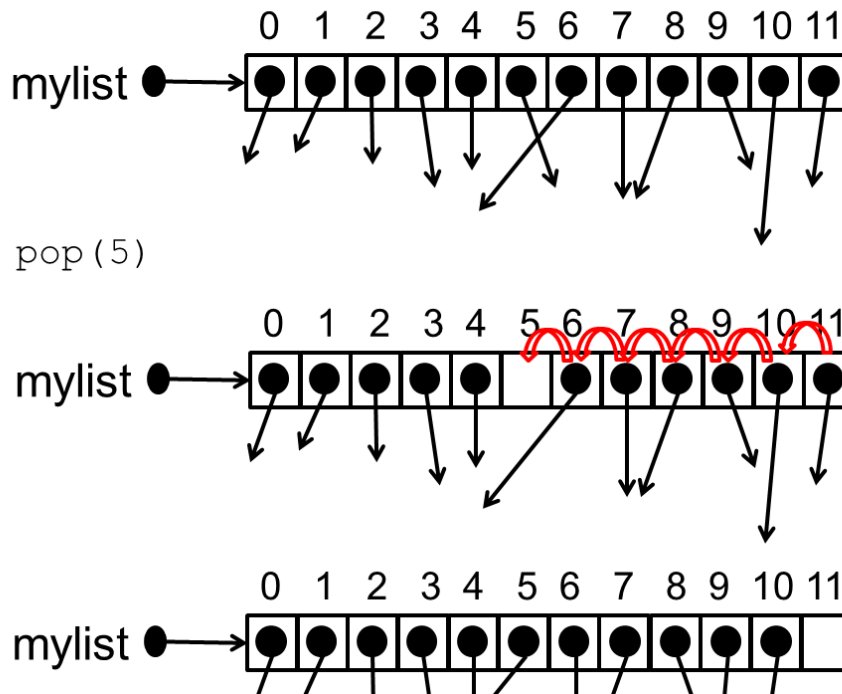
Queue: myqueue: ['a', 2, 3, 4, 'b', 'c']

Queue: myqueue: [2, 3, 4, 'b', 'c']

BUT: removing an element from the front of a list is not efficient.
See lecture 4 ...

Cost of popping an element

Python's lists have no spaces – when we pop an element, the list must close up, pushing space to the end.



for a list of length n ,
`pop(i)` takes $(n-1-i)$ copies.

`pop(0)` takes $n-1$ copies

`pop` is $O(n)$

Complexity of operations

```
class QueueV0:
    def __init__(self):
        self.body = []

    def enqueue(self, element):
        self.body.append(element)

    def dequeue(self):
        if len(self.body) == 0:
            return None
        return self.body.pop(0)

    def front(self):
        if len(self.body) == 0:
            return None
        return self.body[0]

    def length(self):
        return len(self.body)
```

List.append is $O(1)$ *on average*

List.pop(0) is $O(n)$
- always has to copy the remainder of the list

List index lookup is $O(1)$

List length is $O(1)$

Avoiding copying elements

Avoiding copying elements

Let's avoid using `List.pop(0)`

Instead, maintain our own internal reference to the first element in the list, which we will call `head`.

To dequeue, record the element which is referred to by `head`, reassign `None` to that cell, update our `head` pointer to the next index, and return the recorded element.

We need to fix our `length()` method, since we can't just return the length of the list – instead we first subtract `head`.

These operations take constant time, so `dequeue()` is $O(1)$ (and we haven't changed `enqueue()`)

```
class QueueV1:
    def __init__(self):
        self.body = []
        self.head = 0

    def enqueue(self, item):
        self.body.append(item)

    def dequeue(self):
        if self.length() == 0:
            return None
        item = self.body[self.head]
        self.body[self.head] = None
        self.head = self.head + 1
        return item

    def length(self):
        return len(self.body) - self.head

    def front(self):
        if self.length() == 0:
            return None
        return self.body[self.head]
```

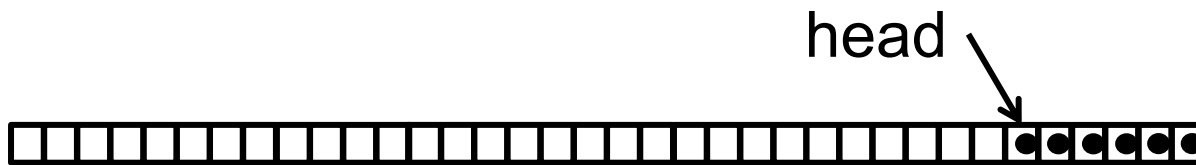
Version 1

basic implementation

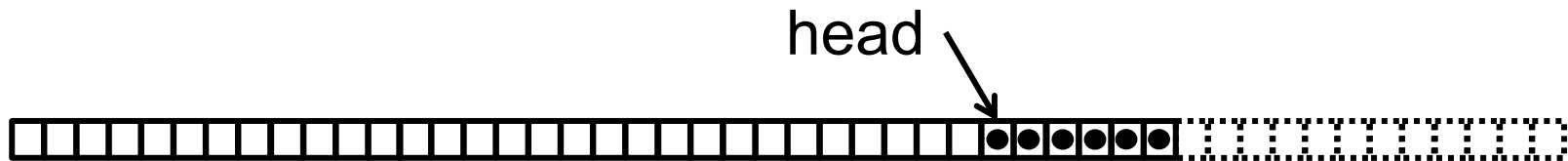
- no private variables
- no `__str__()` method

Exploding the space

If we do many enqueue and dequeue operations on our queue, we may end up with a very short queue, but with Python maintaining space for a very large list



(and Python may have created space at the end as part of the dynamic list growth ...)



How do we avoid exploding the space?

How do we avoid exploding the space?

When we reach the end of the list, instead of letting Python extend it, we will wrap around, and start placing new items at the front.

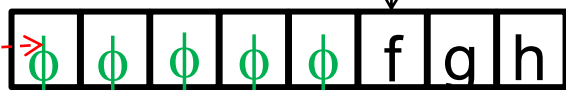
We only expand the list space when we have filled all cells in our current list.

We must maintain a head and a tail index, and update them carefully when we enqueue and dequeue.

Each operation will be slower than before, but still $O(1)$, and we will only require space for the biggest queue we have maintained at any one time.

(from now on, just write the elements in place in the list ...)

head

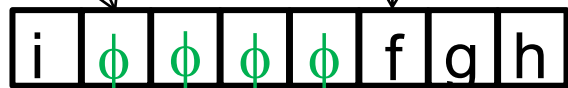


enqueue(i)

- how do we stop python wasting space by extending the list?
- we know we have spare space, so why not put it here?

tail

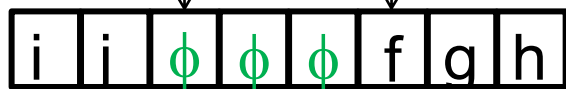
head



enqueue(j)

tail

head



enqueue(k)

enqueue(l)

enqueue(m)

tail



now just about to change tail to point to next cell ...

we notice that tail will coincide with head, so list must be full, so now we create a bigger list, and copy everything across.

Take the chance to put the head back to position 0

head

tail



enqueue pseudocode:

place new item in tail
if list is now full,

grow the list, copy and re-balance

else if tail is at end of list

tail = 0

else


tail = tail+1

Also need to
update head
properly in dequeue

Our version will
maintain head, tail
and size as
variables

Efficient Version

```
class QueueV2:
    def __init__(self):
        self.body = [None]*10
        self.head = 0    #index of 1st elt, unless empty, then 0
        self.tail = 0    #index of next cell to be filled
        self.size = 0

    def enqueue(self, item):
        if self.size == 0:                #redundant, but explicit
            self.body[0] = item           #empty queue has head at 0
            self.size = 1
            self.tail = 1
        else:
            self.body[self.tail] = item
            self.size = self.size + 1
            if self.size == len(self.body): #list is now full
                self.grow()
            elif self.tail == len(self.body)-1: #no room at end
                                                    #but must be room at front
                self.tail = 0
            else:
                self.tail = self.tail + 1
```

```
def dequeue(self):
    if self.size == 0:      #empty queue
        return None
    item = self.body[self.head]
    self.body[self.head] = None
    if self.size == 1:      #removed last element, so rebalance
        self.head = 0
        self.tail = 0
        self.size = 0
    elif self.head == len(self.body) - 1:  #head was at end
        self.head = 0      #we must have wrapped round
        self.size = self.size - 1
    else:
        self.head = self.head + 1
        self.size = self.size - 1
    return item

def length(self):
    return self.size

def front(self):
    return self.body[self.head]
```

```
def grow(self):
    oldbody = self.body
    self.body = [None] * (2*self.size)
    oldpos = self.head
    pos = 0
    if self.head < self.tail:
        while oldpos <= self.tail:
            self.body[pos] = oldbody[oldpos]
            oldbody[oldpos] = None
            pos = pos + 1
            oldpos = oldpos + 1
    else:
        while oldpos < len(oldbody):
            self.body[pos] = oldbody[oldpos]
            oldbody[oldpos] = None
            pos = pos + 1
            oldpos = oldpos + 1
        oldpos = 0
        while oldpos <= self.tail:
            self.body[pos] = oldbody[oldpos]
            oldbody[oldpos] = None
            pos = pos + 1
            oldpos = oldpos + 1
    self.head = 0
    self.tail = self.size
```

Note: there are more elegant ways of coding this class using modular arithmetic, but the procedure is the same

Note: we should also shrink the space for the list if the size of the queue is very much smaller than the list.

Complexity of operations

<code>class QueueV2:</code>	
<code>def __init__(self):</code>	$O(1)$ <i>on average</i>
<code>def enqueue(self, element):</code>	
<code>def dequeue(self):</code>	$O(1)$ <i>on average</i>
<code>def front(self):</code>	$O(1)$
<code>def length(self):</code>	$O(1)$

and our space requirement is determined by the biggest number of elements ever stored in our queue at any one time (and not by the number of `enqueue()` method calls)

```
>>> test_queues(1000000)
Creating a queue of each type.
Empty lengths: 0 0 0
First basic enqueueing and dequeuing:
q 0 : <-2-3-4-b-c-<
q 1 : <-2-3-4-b-c-<      Head:2; length:5
q 2 : <-2-3-4-b-c-<      Head:2; tail:7; size:5
enqueueing n items, then dequeuing n in each queue:
0 took 390.3035932019702    and has size      144 but length 5
1 took   1.037035487999674 and has size 8697464 but length 5
2 took   3.165163889992982 and has size 10485824 but length 5
now starting again, and maintaining a list of size 20 through n
operations
0 took 0.8664442739682272 and has size 264 but length 20
1 took 1.0124022209784016 and has size 8697464 but length 20
2 took 2.9212797950021923 and has size 384 but length 20
```

Basic version slower by a factor of x100

V1 can't reduce space, because we filled with None

For V2 we didn't implement the ability to shrink

```
>>> test_queues(1000000)
Creating a queue of each type.
Empty lengths: 0 0 0
First basic enqueueing and dequeuing:
q 0 : <-2-3-4-b-c-<
q 1 : <-2-3-4-b-c-<      Head:2; length:5
q 2 : <-2-3-4-b-c-<      Head:2; tail:7; size:5
enqueueing n items, then dequeuing n in each queue:
0 took 390.3035932019702 and has size 144 but length 5
1 took 1.037035487999674 and has size 8697464 but length 5
2 took 3.165163889992982 and has size 10485824 but length 5
now starting again, and maintaining a list of size 20 through n
operations
0 took 0.8664442739682272 and has size 264 but length 20
1 took 1.0124022209784016 and has size 8697464 but length 20
2 took 2.9212797950021923 and has size 384 but length 20
```

Why does V2 use 20% more space than V1?

V2 implements a 'doubling' policy, while V1 uses Python's built-in policy, which is less than doubling, but still $O(1)$ on average.

Basic version slower by a factor of x100

```
>>> test_queues(1000000)
Creating a queue of each type.
Empty lengths: 0 0 0
First basic enqueueing and dequeuing:
q 0 : <-2-3-4-b-c-<
q 1 : <-2-3-4-b-c-<      Head:2; length:5
q 2 : <-2-3-4-b-c-<      Head:2; tail:7; size:5
enqueueing n items, then dequeuing n in each queue:
0 took 390.3035932019702 and has size 144 but length 5
1 took 1.037035487999674 and has size 8697464 but length 5
2 took 3.165163889992982 and has size 10485824 but length 5
now starting again, and maintaining a list of size 20 through n
operations
0 took 0.8664442739682272 and has size 264 but length 20
1 took 1.0124022209784016 and has size 8697464 but length 20
2 took 2.9212797950021923 and has size 384 but length 20
```

V1 can't reduce space, because we filled with None

But V2 never has to grow the space beyond 20 cells.

V0 can be significantly slower

V1 occupies significantly more space

V2 is a little slower (x3 for this instance), and can reclaim space

Next lecture ...

More on sequences and ADTs