Queues

Missed stacks – need to catch up.

A queue is a collection of objects where:

- We take items from the front
- We add items to the back

Queues are FIFO (First-in, First-out).

Queues in Computer Science

Queues are essential in CS:

- Packets 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 mantain a queue of edges or locations to explore next
- Grid and cloud computing maintain queues of queries, updates, and access requests

The Queue ADT (Abstract Datatype)

- enqueue: place an element onto the end of the queue
- dequeue: take the first element off of the front of the queue (and remove it from the queue)
- front: report the front element from the queue (but leave it on the queue)
- length: report how many elements are in the queue
- is empty: report whether or not the queue is empty

Implementing the Queue

We will define a class offering these methods.

How should we represent the data in the class?

The elements clearly have an order, so we could use a sequence. We'll 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. We need to work at both ends though, which causes difficulty for efficiency.

First Implementation

```
class QueueV1:
    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 first(self):
        if len(self.body) == 0:
            return None
        return self.body[0]

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

    def is_empty(self):
        return len(self.body) == 0
```

Note we're popping from position 0, not from the end.

Problems

Rmoving an element from the front of a list is not efficient.

On average, everything is O(1) except for the dequeue method, which is O(n).

Avoiding Copying Elements

When we use `dequeue`, instead of removing the first object (and forcing Python to reorganise the space), we change it to `None`, and record that the real start of the list is now one element later.

There are now 4 simple operations in the process, meaning that `dequeue` is now O(1).

We'll call our reference to the first element `head`.

We now need to fix our `length` method by subtracting `head` from it.

Second Implementation

```
class QueueV2:
    def __init__(self):
        self.body = []
        self.head = 0

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

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

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

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

    def is_empty(self):
        return self.length() == 0
```

Avoiding 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 possibly reserving space at the end as part of dynamic list growth.

To save space, we reuse the slots at the beginning that contain `None`. We loop back around.

We now need to record the tail as well as the head, so that we know where the next element goes.

Once our list fills up (we're about to overwrite a head value with a new tail element), we need to grab new space and copy all the elements across. When we're doing this, we may as well move the head to be at the beginning again.

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

Third Implementation [get from slides]

```
class QueueV3:
    def __init__(self):
        self.body = [None] * 10
        self.head = 0
        self.tail = 0
        self.size = 0

    def enqueue(self, element):
        if self.size == 0:    #redundant but explicit
            self.body[0] = element
            self.size = 1
            self.tail = 1
        else:
            self.body[self.tail] = element
            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
                self.tail = 0
            else:
                self.tail += 1

    def dequeue(self):
        if self.size == 0:
            reutrn None
        item = self.body[self.head]
        self.body[self.head] = None
        if self.size == 1
```

Although the code is now long, there's nothing very complicated in it. It just needs to be very cautious.

You can do it more elegantly using modular arithmetic, but the procedure is the same. It is not any more efficient.

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

`enqueue` and `dequeue` are O(1) on average, and everything else is O(1).

Protecting

We should declare private variables by using underscores at the beginning of the variable names, to indicate that those variables are not to be modified outside of the methods.