# Simulating Stacks and Queues

Let's play a little game. We will pretend that we are using a stripped down version of python that has only one collection data structure. We'll use it to build another. In the first case, we will use a queue to build a stack. In the second case, we will use a stack (and recursion) to build a queue. Lastly, we will use two stacks to build a queue.

## From Queue to Stack

We can use a queue to simulate a stack. It's not pretty. It's not efficient. But it does work.

```
from ds2.queue import ListQueue as Queue

class SimulatedStack:
    def __init__(self):
        self._q = Queue()

    def push(self, item):
        the_end_of_the_queue = "This is the end"
        self._q.enqueue(the_end_of_the_queue)
        self._q.enqueue(item)
        nextitem = self._q.dequeue()
        while nextitem is not the_end_of_the_queue:
            self._q.enqueue(nextitem)
            nextitem = self._q.dequeue()

    def pop(self):
          return self._q.dequeue()


S = SimulatedStack()
[S.push(i) for i in [2,4,6,8,10]]
print([S.pop() for i in range(5)])
```

```
[10, 8, 6, 4, 2]
```

## From Stack to Queue

We can use a stack to simulate a queue. This will look somewhat terrible. The key idea is to use recursion. The `dequeue` operation is trying to get the bottom item of the stack. It does this by

popping one item. If it's the last item, it returns it. Otherwise, it recursively calls `dequeue` while carefully pushing the popped item before returning the result.

```
from ds2.stack import ListStack as Stack

class SimulatedQueue:
    def __init__(self):
        self._s = Stack()

    def enqueue(self, item):
        self._s.push(item)

    def dequeue(self):
        x = self._s.pop()
        if self._s.isempty():
            return x
        else:
            y = self.dequeue()
            self._s.push(x)
            return y


Q = SimulatedQueue()

[Q.enqueue(i) for i in [2,4,6,8,10]]
print([Q.dequeue() for i in range(5)])


[2, 4, 6, 8, 10]
```

There is a sense in which we really haven't used just one stack to do this implementation. It is really using two stacks: one is the stack stored as `self._s` ; the other is the function call stack. There is a variable `x` that is local to every call to `dequeue()` .

# A Queue from Two Stacks

Could we do this without recursion and without some other kind of collection data structure? The answer is **no**. We could use the intuition from the recursive algorithm do the Queue simulation with two stacks and no recursion.

```
from ds2.stack import ListStack as Stack

class TwoStackQueue:
    def __init__(self):
        self._stack1 = Stack()
        self._stack2 = Stack()

    def enqueue(self, item):
        self._stack1.push(item)

    def dequeue(self):
        # Move everything from stack 1 to stack 2.
        while not self._stack1.isempty():
            x = self._stack1.pop()
            self._stack2.push(x)
        # Pick out the last item to be returned later.
        y = self._stack2.pop()
        # Move everything from stack 2 back to stack 1.
        while not self._stack2.isempty():
            x = self._stack2.pop()
            self._stack1.push(x)
        return y


Q = TwoStackQueue()

[Q.enqueue(i) for i in [2,4,6,8,10]]
print([Q.dequeue() for i in range(5)])


[2, 4, 6, 8, 10]
```

As with other inefficient Queue implementations, we could rearrange the inefficiency, making the enqueue operation slow at the expense of the dequeue operation.