Using the push() and pop() operations from stacks, we could implement enqueueing and dequeueing, so it is possible ot create queues using stacks, keeping in mind the FIFO(first-in first-out) principle. This implementation can be done in two forms:

- the enqueue operation would have to be O(n) in time complexity;
- the dequeue operation would have to be O(n) in time complexity;

To perform the first option, it is going to take two stacks(box1 and box2) that need to be kept in a way that allows the element inserted first to be always at the top of box1, and that we can simply use pop() for dequeue on box1. However, we would have to make the enqueued element reach the bottom of the stack for enqueueing. To do so, we will have to pop the elements of box1 one by one and push them onto box2. Then, we add the new element to box1. Finally, we pop everything from box2 again and push it back to box1, so that the new element gets placed at the bottom of the stack.

See the outcome below:

In [9]:

```
class Queue():
    def __init__(self):
        self.box1 = []
        self.box2 = []
# We start by adding two empty lists to the constructor.

    def peek(self):
        if len(self.box1) == 0:
            print("Oops! This queue is empty.")
        else:
            return self.box1[len(self.box1)-1]
# Then, we create the peek method.
# Because stacks follow the LIFO principle, we have to tell the method to output our list from last to first so th


    def enqueue(self, data):
        for i in range(len(self.box1)):
            item = self.box1.pop()
            self.box2.append(item)
        self.box1.append(data)
        for i in range(len(self.box2)):
            item = self.box2.pop()
            self.box1.append(item)
```

```
        return
# The next method is enqueue().
# Here we are loop traversing through both boxes, first to assign all elements from one box to another, then to ad
# As per above, the time complexity of this operation is of O(n).

    def dequeue(self):
        if len(self.box1)==0:
            print("Oops! This queue is empty.")
            return
        else:
            return self.box1.pop()
# Up next, dequeue().
# This operation will simply pop() the last element of the list, which is the first to ever be inserted into the l

    def get_queue(self):
        if len(self.box1) == 0:
            print("Oops! This queue is empty.")
            return
        for i in range(len(self.box1) - 1,0,-1):
            print(f'{self.box1[i]} <<-- ',end='')
        print(self.box1[0])
        return
```

To test our new blueprint:

In [10]:

```
new_queue = Queue()

new_queue.enqueue(1)
new_queue.enqueue(2)
new_queue.enqueue(3)

new_queue.get_queue()
```

```
1 <<-- 2 <<-- 3
```

In [11]:

```
new_queue.dequeue()
new_queue.get_queue()
```

```
2 <<-- 3
```

In [12]:
```
print(new_queue.peek())
```

```
2
```

In [13]:
```
new_queue.enqueue(4)
new_queue.get_queue()
```

```
2 <<-- 3 <<-- 4
```

In [14]:
```
new_queue.dequeue()
new_queue.dequeue()
new_queue.dequeue()

new_queue.get_queue()
```

```
Oops! This queue is empty.
```

Regarding the second option, we can make the dequeue method at a time complexity of O(n) just as it was done when building the enqueue method above. While for enqueueing we push into box1, for the dequeueing method, we have to pop all elements except the last one in box1. Next, those elements get push into box2, and the last element (which we left in box1 to get removed) gets popped. Finnaly, the elements in box2 are popped and pushed back into box1. This process is what makes the dequeue operation O(n), while enqueue and peek remain O(1).