Lecture 7

Queue
Another limited data structure and FIFO

Last time:

We looked at Stack:
- A limited data structure that has mainly two operations, push and pop.
- Push inserts a new item onto the top of the stack and pop **gets and removes** the item from the top of the stack.
- Push and pop's running time complexity is O(1).
- It can be used by a programmer in applications as an aiding tool.
- Java's LinkedList and ArrayDeque provide a few methods that let programmers to use them as if it is Stack.
- Example usages of stack are reversing a word and back button of a web browser, etc. But, there are many others.

Now, I work at a Starbucks coffee shop.

In the US, we say, "are you in the line?"
In the GB, they say, "are you in the queue?"

Working at a Starbucks coffee shop in London, what do I do?
I serve anyone next in the queue.

Step 1: Conceptual View

A queue is a container of objects that are inserted and removed based on FIFO (First-in First-out) principle.

In the queue, there are two major operations, enqueue and dequeue.
- Enqueue means inserting a new item to the back of the queue.
- Dequeue means removing the first item from the queue.

Front   Back

Stack VS. Queue
Conceptually, you can think of Stack as if it has only one door on the top whereas queue has two doors at both ends (the front and the back).

The door on the top of a stack can be opened on both directions. But, the back door of a queue can be pushed from the outside so that people can walk in but cannot come out. Similarly, the front door of a queue can be pushed and opened from the inside so that people inside can come out but no one can go in through the front door.

Due to the fact of having two doors at both ends, queue gets more complicated than stack.

Step 2: Implementation View

Just like Stack, Queue is also a data structure that is built on top of other data structures (array, ArrayList, or LinkedList). No matter what underlying structure it uses, we want to make sure that a queue implements the same following functionalities. We can achieve this using an interface.

```
public interface QueueInterface<AnyType> {
    void enqueue(AnyType item);  // O(1)
    AnyType dequeue();           // O(1)
    AnyType peekFront();         // O(1)
    boolean isEmpty();           // O(1)
}
```

Array-based implementation

We need some fields:
- An "array **elements**" which has a fixed length
- Variable "**back**" that refers to the back of the queue
- Variable "**front**" that refers to the front of the queue
- Additionally, "**nItems**" variable to keep track of current number of items

Every time a new item *is added* (enqueued), the "back" index should be **increased.**

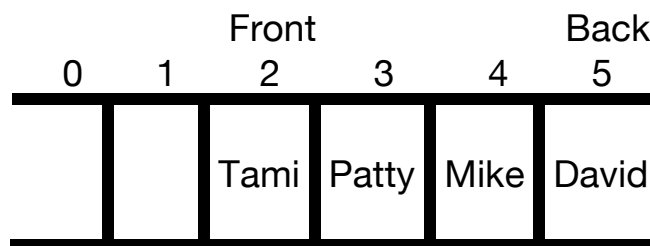And, when the front item *is removed* (dequeued), the "front" index should also be **INCREASED.**
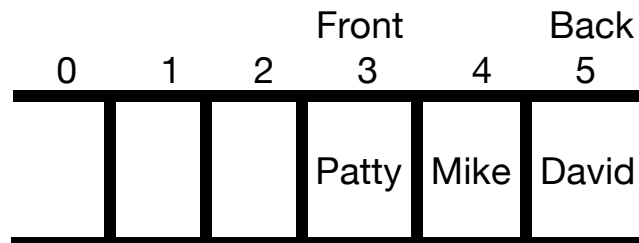
A queue with already three people in
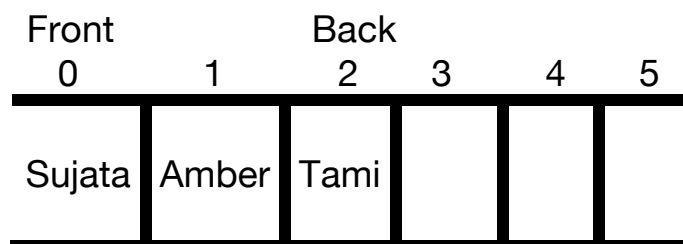
Patty and Mike came into the Starbucks coffee shop

Front				Back	
0	1	2	3	4	5
Sujata	Amber	Tami	Patty	Mike	
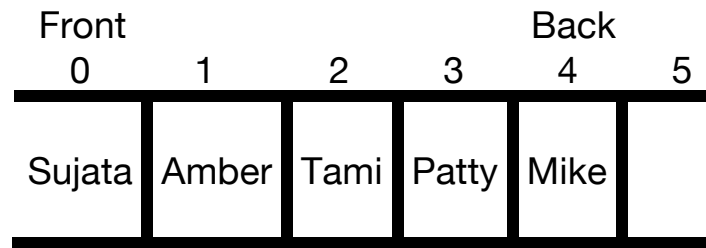
Now, Terry was able to serve Sujata and Amber quickly but David came in to the shop.

		Front		Back	
0	1	2	3	4	5
		Tami	Patty	Mike	David

Now, Terry was able to serve Tami but this time Greg came in.

			Front		Back
0	1	2	3	4	5
			Patty	Mike	David

Now the question is where Greg goes? We know we have some space available in the coffee shop queue.
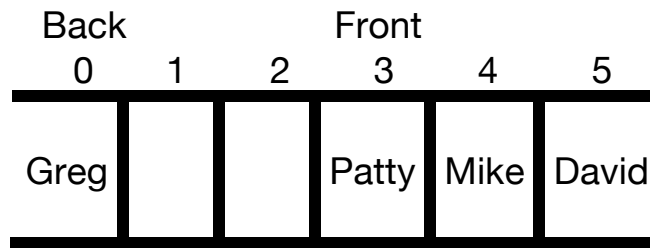
Circular Queue
In reality, people move forward as they are being served but array does not know what it means to move forward (Not that smart, huh?).

But, it's OK because we are smart, right?

What we need to do is to have "front" and "back" variables *wrap around* the array. The result is the circular queue.

Now, where does Greg go?

	Back			Front		
	0	1	2	3	4	5
	Greg			Patty	Mike	David

Now **back** is smaller than **front** (Or **back** is before **front**). Sounds strange and interesting.

But, how does this work? Let's try to implement, shall we?

Skeleton: ArrayQueue class

```
public class ArrayQueue<AnyType> implements
QueueInterface<AnyType> {
    private static final int DEFAULT_CAPACITY = 6;
    private _____ elements;
    private int front; // front index
    private int back; // back index
    private int nItems; //current number of items


    public ArrayQueue() {
        elements = new _____[DEFAULT_CAPACITY];
        front = 0;
        back = _____;
        nItems = 0;
    }

    // TODO Implements all of the core methods here
}
```
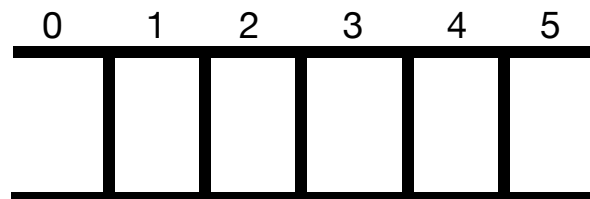
Enqueue

```
/**
 * Inserts a new item into the back of the queue.
 * @param e the new item to be inserted.
 */
@Override
public void enqueue(AnyType item) {
    if (isFull()) {
        throw new RuntimeException("Queue is full");
    }
    back++;
    int index = _____;
    elements[index] = item;
    nItems++;
}
```

Another look at wrapping around!

Initial queue
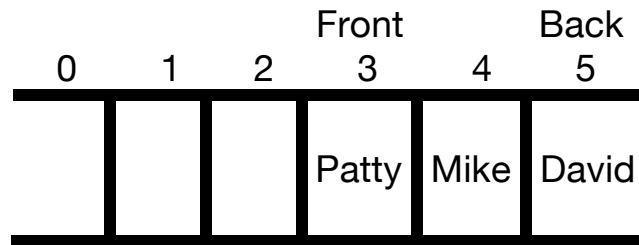

Sujata arrives at the coffee shop.
elements.length = 6;
back++; → 0
back % _____; → 0 % 6 → _____
elements[0] = "Sujata";

After a few minutes, this is what the queue looks like as we have seen before.

```
                          Front        Back
         0     1     2     3     4     5
       ┌─────┬─────┬─────┬─────┬─────┬─────┐
       │     │     │     │Patty│Mike │David│
       └─────┴─────┴─────┴─────┴─────┴─────┘
```

Now Greg arrives.
elements.length = 6;
back++; → _____
back % _____; -> _____ % 6 -> _____
elements[_____] = "Greg";

Do you see how it works?

Dequeue

```java
/**
 * Returns and removes the item at the front.
 * @return the item at the front of the queue
 * @throws NuSuchElementException if it is empty
 */
@Override
public AnyType dequeue() {
    if (isEmpty()) {
        throw new NoSuchElementException();
    }
    int index = _____;

    AnyType result = _____ elements[index];
    elements[index] = _____; // remove it
    front = _____;
    nItems--; // one less in the queue
    return result;
}
```

Peek Front

```
/**
 * Returns the first item in the queue without removing it.
 * @return the first item
 * @throws NoSuchElementException if it is empty
 */


@Override
public AnyType peekFront() {
    if (isEmpty()) {
        throw new NoSuchElementException();
    }
    return _____ elements[_____];
}
```

isEmpty

```
@Override
public boolean isEmpty() {



}
```

Revisit to the time complexity (Big O):
Enqueue:

Dequeue:

Thinking back again!
Now, what if you do not want to create your own Queue interface
and a Queue class that implements the interface but still want to
have FIFO type of data structure in your code?

What would you use?

Again, what are some of the methods that are offered in the LinkedList class in Java?

```
addLast(Object) : void
removeFirst() : Object
```

```
LinkedList<Integer> theQueue = new LinkedList<Integer>();
// enqueue into the queue
theQueue.addLast(5);
// dequeue from the queue
theQueue._____;
```

Do not be confused! You are using addLast and _____ methods and Queue is FIFO.

Also there are other methods in LinkedList and ArrayDeque such as offer(E e), pollFirst(), and pollLast(), etc.