Lecture 7

Queue
Another limited data structure and FIFO

Last time:

We looked at a Stack:

- A limited data structure that has mainly two operations, push and pop.
- Push inserts a new element onto the top of the stack and pop gets and removes the element from the top of the stack.
- Push and pop's time complexity is O(1).
- It can be used by a programmer in applications as an aiding tool.
- Java's LinkedList provides a few methods that enable a programmer to use it as a Stack.
- Example usages of a stack is reversing a word or back button of a web browser, etc.

Now I work at a Starbucks coffee shop!

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

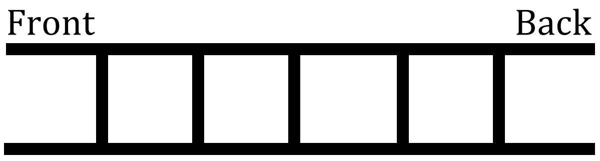
Working at a Starbucks coffee shop in the UK, what do I do?
I serve anyone next in the queue. (First-in First-out)

Step 1: Conceptual View

A queue is a container of objects that are inserted and removed based on FIFO principle.

In the queue, there are two major operations, enqueue and dequeue.

- Enqueue means inserting a new item into the back of the queue.
- Dequeue means removing the first item from the queue.



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 the back and the front.

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 go 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, a queue gets more complicated than a 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, LinkedList, etc.) No matter what underline structure it uses, a queue must implement the same following functionalities. We can achieve this using an interface.

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

Array-based implementation

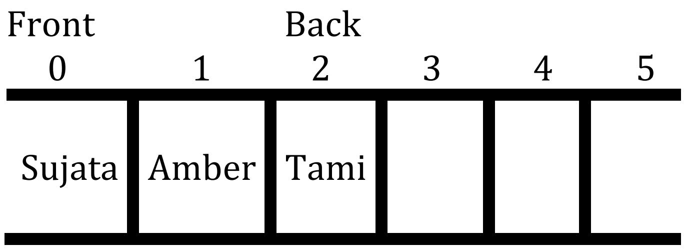
We know we need at least three fields:

- An "array A" which has a fixed-size
- Variable "back" that refers to the back of the queue
- Variable "front" that refers to the front of the queue
- Additionally, some others such as "nItems" variable to keep track of current number of items

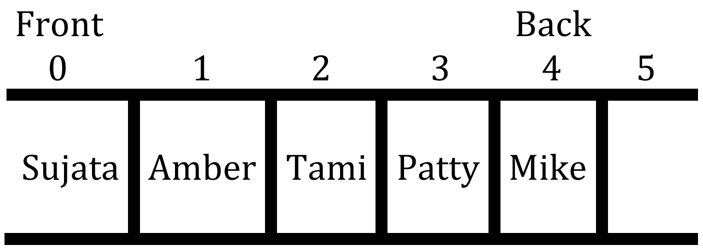
Every time a new element *is added* (enqueued), the "back" index should be **increased.**

And, when the front element *is removed* (dequeued), the "front" index should be **increased.**

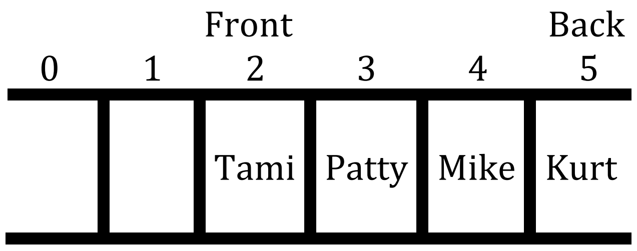
A queue



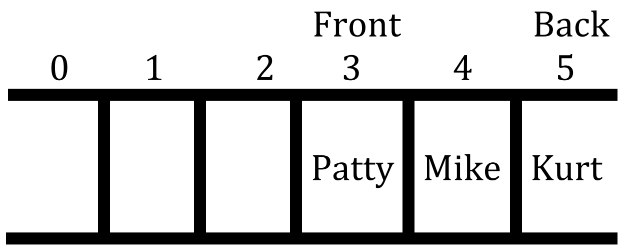
Patty and Mike came into the Starbucks coffee shop



Now, Terry was able to serve Sujata and Amber quickly but Kurt came in to the shop.



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



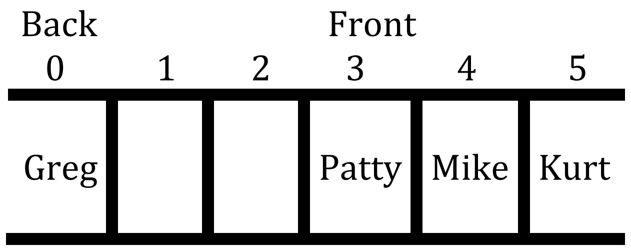
Now the question is where Greg goes? We know we have some space available in the coffee shop queue.

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?



Now back is smaller than front. Sounds interesting.
But, how does it work? Let’s try to implement, shall we?

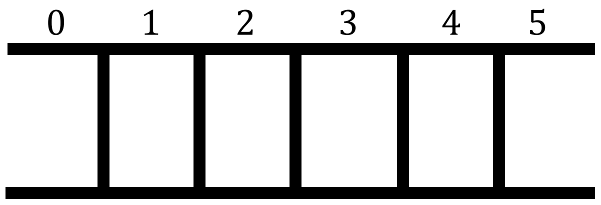
Skeleton: ArrayQueue class

```
public class ArrayQueue<AnyType> implements
QueueInterface<AnyType> {


     private AnyType[] A;
     private static final int DEFAULT_CAPACITY = 15;
     private int front; // front index
     private int back; // back index
     private int max; // maximum number of elements
     private int nItems; //current number of elements

     @SuppressWarnings("unchecked")
     public ArrayQueue() {
          max = DEFAULT_CAPACITY;
          A = (AnyType[]) new Object[max];
          back = -1; // initially set to -1
          front = 0; // initially set to 0
          nItems = 0; // set number of items to 0
     }

      // TODO Implements all of the core methods here

}
```

Enqueue

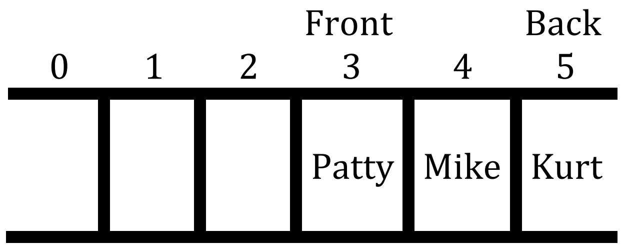
```
/**
 * Inserts a new element into the back of the queue.
 * @param e the new item to be inserted.
 */
@Override
public void enqueue(AnyType e) {
     if(isFull()) throw new RuntimeException("Queue is full");
     // increase back index
     back++;
     // wrap around!
     int index = ___________________;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

Another look at wrapping around!
Initial queue



Sujata arrives at the coffee shop.
max = 6;
back++; → 0
back % max; → 0 % 6 → __________
A[0] = "Sujata";

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



Now Greg arrives.
max = 6;
back++; → ______________
back % max; -> _____________ % 6 -> _____________
A[________] = “Greg”;

Do you see how it works?

Dequeue

```
/**
 * Returns and removes the element at the front.
 * @return the element at the front of the queue
 * @throws NuSuchElementException if it is empty
 */
@Override
public AnyType dequeue() {
     if(isEmpty()) throw new NoSuchElementException();
     int index = ____________________;
     AnyType element = A[index];
     A[index] = null; // make sure to remove the element
     front++; // front increases
     nItems--; // one less element in the queue
     return element;
}
```

Peek Front

```
/**
 * Returns the first element in the queue without removing it.
 * @return the first element
 * @throws NoSuchElementException if it is empty
 */
@Override
public AnyType peekFront() {
     if(isEmpty()) throw new NoSuchElementException();
     return A[front%max];
}
```

Now, would you make any change into the previous code?

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?

What are some of the major methods that are offered in the LinkedList in Java?

+addFirst(element: Object) : void
+removeLast() : Object

```
LinkedList<Integer> theQueue = new LinkedList<Integer>();

// enqueue into the queue
theQueue.addLast(5);

// dequeue from the queue
theStack.removeFirst();
```

Do not be confused: You are using addFirst and removeLast methods and Queue is FIFO.