Prepared by Linan Qiu <lq2137@columbia.edu>

Queues

A queue is well…a queue. Things that go in first gets out first. Think of a queue anywhere – say at the dinining hall or even in a printer (printer jobs!). This behavior is called *first-in-first-out* in contrast to stack's *first-in-last-out*.

We call adding a new element to the back of a queue `enqueue`-ing and removing an element from the front of a queue `dequeue`-ing.

We can easily make an interface:

`AwsmQueue`

```
// AwsmQueue.java

public interface AwsmQueue<T> {

  public void enqueue(T item);

  public T dequeue();

  public int size();

}
```

`AwsmLinkedList` as an Implementation of `AwsmQueue`

Again, we can implement queues using lists. However, this time, we either use `addFirst` and `removeLast` (since we are adding from the front and removing from the back) or `addLast` and `removeFirst`. In our case, we use `addLast` and `removeFirst` for linked lists. Why? Because `addLast` can be made $O(1)$ though in the naive implementation it is $O(N)$ (with only a head node and no tail node). `removeFirst` is $O(1)$. If we had chosen the other combination (`addFirst` and `removeLast`), `addFirst` would be $O(1)$ but `removeLast` would certainly be $O(N)$.

The implementation would be like this:

```
// AwsmLinkedQueue.java

public class AwsmLinkedQueue<T> implements AwsmQueue<T> {
  private AwsmLinkedList<T> list;

  public AwsmLinkedQueue() {
    list = new AwsmLinkedList<>();
  }

  @Override
  public void enqueue(T item) {
    list.addLast(item);
  }

  @Override
  public T dequeue() {
    T data = list.getFirst();
    list.removeFirst();
    return data;
  }

  @Override
  public int size() {
    return list.size();
  }
}
```

`AwsmArrayList` as an Implementation of `AwsmQueue`

As for `AwsmArrayList`, we have no choice but to accept the $O(N)$ penalty on one side.

```
// AwsmArrayQueue.java

public class AwsmArrayQueue<T> implements AwsmQueue<T> {

  private AwsmArrayList<T> list;

  public AwsmArrayQueue() {
    list = new AwsmArrayList<>();
  }

  @Override
  public void enqueue(T item) {
    list.addLast(item);
  }

  @Override
  public T dequeue() {
    T data = list.getFirst();
    list.removeFirst();
    return data;
  }

  @Override
  public int size() {
    return list.size();
  }
}
```

`CircularArray` as an Implementation of `AwsmQueue`

Since `AwsmArrayList` doesn't quite seem to do the job, we can instead try another method: using a circular array.

This is an array with two pointers: front and back. Initially, front (f) and back (b) point to the same index in the array.

```
[ ][ ][ ][ ][ ]
fb
```

As we `enqueue` elements, `b` increases. `b` basically marks the next available index.

```
enqueue(A)
[A][ ][ ][ ][ ]
 f  b

enqueue(B)
[A][B][ ][ ][ ]
 f     b
```

Now if we want to `dequeue` items, we simply shift `f` forward as well. This would ensure that we return the oldest item.

```
dequeue() return A
[ ][B][ ][ ][ ]
    f  b
```

Now the tricky part is that `b` and `f` wraps around. Let's say we have this arrangement elements:

```
enqueue(C)
[ ][B][C][ ][ ]
    f     b

enqueue(D)
[ ][B][C][D][ ]
    f        b

enqueue(E)
[ ][B][C][D][E]
 b  f

dequeue() return B
[ ][ ][C][D][E]
 b     f
```

See how `b` wraps around?

Now the circular array also has a limited size – once `b` is one item before `f`, we are out of space and we need to expand the array.

This is how we implement one:

```java
// AwsmCircularQueue.java


public class AwsmCircularQueue<T> implements AwsmQueue<T> {

  private T[] array;
  private int back;
  private int front;
  private int size;
  public static final int INITIAL_SIZE = 8;

  @SuppressWarnings("unchecked")
  public AwsmCircularQueue() {
    array = (T[]) new Object[INITIAL_SIZE];
    back = 0;
    front = 0;
    size = 0;
  }

  @Override
  public void enqueue(T item) {
    expand();
    array[back] = item;
    back = (back + 1) % array.length;
    size++;
  }

  @Override
  public T dequeue() {
    if (size == 0) {
      throw new IndexOutOfBoundsException();
    }
    T data = array[front];
    front = (front + 1) % array.length;
    size--;
    return data;
  }

  @SuppressWarnings("unchecked")
  private void expand() {
    if (size == array.length) {
      T[] newArray = (T[]) new Object[array.length * 2];
      for (int i = 0; i < size; i++) {
        newArray[i] = array[(front + i) % array.length];
      }
      array = newArray;
      front = 0;
      back = front + size;
    }
  }

  @Override
  public int size() {
    return size;
  }

  public static void main(String[] args) {
    AwsmCircularQueue<Integer> queue = new AwsmCircularQueue<>();

    for (int i = 0; i < 100; i++) {
      queue.enqueue(i);
    }

    for (int i = 0; i < 50; i++) {
      System.out.println(queue.dequeue());
    }

    for (int i = 0; i < 100; i++) {
      queue.enqueue(i);
    }

    while (queue.size() > 0) {
      System.out.println(queue.dequeue());
    }
  }
}
```

I will not go through this in detail. However, I'd highly encourage you to read through this and try to understand this code. If you don't, feel free to post questions on piazza!