Using Queues: Coded Messages

A **repeating key** is a sequence of integers that determine by how much each character in a message is shifted. Consider the repeating key 3 1 7 4 2 5

message: knowledge
encoded message:

queue: 3 1 7 4 2 5
Using Queues: Coded Messages

A **repeating key** is a sequence of integers that determine by how much each character in a message is shifted. Consider the repeating key:

\[
3 \quad 1 \quad 7 \quad 4 \quad 2 \quad 5
\]

message: knowledge
encoded
message: n
dequeued: 3

queue:

\[
1 \quad 7 \quad 4 \quad 2 \quad 5
\]
A **repeating key** is a sequence of integers that determine by how much each character in a message is shifted. Consider the repeating key

\[3 \ 1 \ 7 \ 4 \ 2 \ 5\]

message: knowledge
encoded
message: n

queue: 1 7 4 2 5 3
Using Queues: Coded Messages

A *repeating key* is a sequence of integers that determine by how much each character in a message is shifted. Consider the repeating key

```
3 1 7 4 2 5
```

message: knowledge
encoded
message: no
dequeued: 1

queue: 7 4 2 5 3
Using Queues: Coded Messages

A repeating key is a sequence of integers that determine by how much each character in a message is shifted. Consider the repeating key

\[ 3 \ 1 \ 7 \ 4 \ 2 \ 5 \]

message: knowledge
encoded
message: no

queue: 7 4 2 5 3 1
Using Queues: Coded Messages

A repeating key is a sequence of integers that determine by how much each character in a message is shifted. Consider the repeating key

\[
3 \ 1 \ 7 \ 4 \ 2 \ 5
\]

message: knowledge
encoded
message: novangjhl
Algorithm in Pseudocode for the Dequeue Operation Using a Circular Array Representation of a Queue

```pseudocode
Algorithm dequeue() {
    if queue is empty then ERROR
    result = queue[front]
    count = count – 1
    queue[front] = null
    front = (front + 1) mod (size of array queue)
    return result
}
```

Where `mod` is the modulo operator (or modulus or remainder), denoted `%` in Java.
public T dequeue() {
    if (queue.isEmpty())
        throw new EmptyQueueException();
    result = queue[front];
    count = count - 1;
    queue[front] = null;
    front = (front + 1) % queue.length;
    return result;
}
Enqueue Operation Using a Circular Array Implementation of a Queue

**Algorithm** enqueue(element)
- if queue is full then expandQueue()
- rear = (rear + 1) mod size of queue
- queue[rear] = element
- ++count

**Algorithm** expandQueue()
- q = new array of size 2 * size of queue
- copied = 0 // number of elements copied to the larger array
- i = 0 // index of next entry in array q
- j = front // index of next entry in array queue
- while copied < count do { // copy data to new array
  q[i] = queue[j]
  ++i
  j = (j + 1) mod size of queue
  ++copied
}
- rear = i – 1 // position of last element in the queue
- front = 0
- queue = q
public void enqueue(T element) {
    if (count == queue.length) expandQueue();
    rear = (rear + 1) % queue.length;
    queue[rear] = element;
    ++count;
}

private void expandQueue() {
    T[] q = (T[]) new Object[2*queue.length];
    copied = 0;  // number of elements copied to the larger array
    i = 0;       // index of next entry in array q
    j = front;   // index of next entry in array queue
    while (copied < count) {
        q[i] = queue[j];
        ++i;
        j = (j + 1) % queue.length;
        ++ copied;
    }
    rear = count - 1;
    front = 0;
    queue = q;
}