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

| a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z |

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

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

message: **knowledge**

encoded

message: **n**

dequeued: **3**

queue: **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 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

| a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z |
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

queue: \[ 4 \ 2 \ 5 \ 3 \ 1 \ 7 \]
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)

1. **if** queue is full **then** expandQueue()
2. rear = (rear + 1) mod size of queue
3. queue[rear] = element
4. ++count

**Algorithm** expandQueue()

1. q = new array of size 2 * size of queue
2. copied = 0 // number of elements copied to the larger array
3. i = 0 // index of next entry in array q
4. j = front // index of next entry in array queue
5. **while** copied < count **do** { // copy data to new array
   1. q[i] = queue[j]
   2. ++i
   3. j = (j + 1) mod size of queue
   4. ++copied
}
6. rear = i – 1 // position of last element in the queue
7. front = 0
8. 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;
}