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

$$3 \ 1 \ 7 \ 4 \ 2 \ 5$$

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

| 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 |
| 7 | 4 | 2 | 5 | 3 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

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
```

| 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: novangjhl

queue: 4 2 5 3 1 7
Second Approach: Queue as a Circular Array

- *Circular array* is an array that conceptually loops around on itself

```
0  1  2  3  4  ...  n-2  n-1
```

- Conceptual diagram showing a circular array with indices 0 to n-1.
Circular Array

0 1 2 3 4  n-2  n-1

...
Circular Queue

- **cq**
  - front: 2
  - rear: 0
  - queue: 9
  - count: 0

- Array: [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ]
  - 0: yellow
  - 1: red
  - 2: empty
  - 3: empty
  - 4: purple
  - 5: black
  - 6: orange
  - 7: pink
  - 8: white
Algorithm in pseudocode for the dequeue operation on a circular array representation of a queue

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.
Example: full array

The array is now full
Example: full array

If we add another element we need to expand the array.
Example: full array

If we add another element we need to expand the array.
Example: full array

But we cannot just copy the data to the same positions in the larger array.
Example: full array

But we cannot just copy the data to the same positions in the larger array.

These locations should be in use.
We *could* build the new array, and copy the queue elements into contiguous locations beginning at location `front`:
Example: full array

Or we could copy the queue elements in order to the \textit{beginning} of the new array.
Algorithm enqueue(element) {
  if queue is full then {
    // Expand the array
    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
  }
  rear = (rear + 1) mod size of queue
  queue[rear] = element
  ++count