

## Laboratory Assignment 13

## Objectives

- Queues Implemented as a Vector

## Activities

1. Recall the *Queue* abstract data type, which provides the following basic operations:

*empty?* Returns a Boolean value (**#t** or **#f**) depending on whether the queue is empty;

*enqueue* Adds a new element to the “end” of the queue;

*dequeue* Removes (and returns) the element at the “front” of the queue.

For simplicity, we just consider queues containing numbers. In this case, you can think of a queue as a row of cells—each containing a number—with a distinguished “front” and “back.” Our convention shall be to place the “front” of the queue on the left, and the “back” of the queue on the right. Thus, the diagram

|   |   |   |   |   |
|---|---|---|---|---|
| 3 | 8 | 2 | 1 | 9 |
|---|---|---|---|---|

represents a queue where 3 is at the front and 9 is at the back. The enqueue operation adds a new cell to the back of the queue; the dequeue operation removes a cell from the front of the queue. Thus the result of an enqueue operation with the number 6 results in the queue

|   |   |   |   |   |   |
|---|---|---|---|---|---|
| 3 | 8 | 2 | 1 | 9 | 6 |
|---|---|---|---|---|---|

and a subsequent dequeue operation results in the queue

|   |   |   |   |   |
|---|---|---|---|---|
| 8 | 2 | 1 | 9 | 6 |
|---|---|---|---|---|

and returns the value 3.

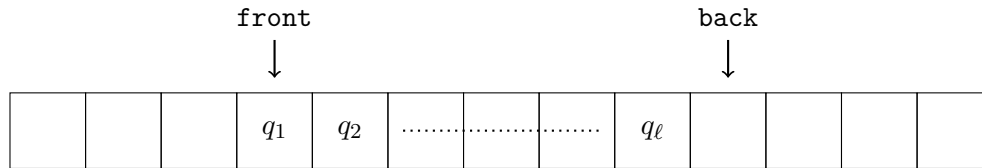
In this problem, you will implement a queue using a *vector*. The idea will be to set up a vector (with a large number of cells) and maintain the queue in an adjacent sequence of cells in the vector.

- (a) Perhaps the most natural approach would be to simply maintain the queue in positions  $0, 1, \dots, \ell - 1$  of the vector (assuming that the queue is currently containing  $\ell$  numbers). Thus, the queue pictured on the previous page would be maintained in the vector by placing the numbers 8,  $\dots$ , 6 in cells  $0, \dots, 4$  of the vector:

|   |   |   |   |   |   |   |   |   |   |    |
|---|---|---|---|---|---|---|---|---|---|----|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| 8 | 2 | 1 | 9 | 6 |   |   |   |   |   |    |

You may assume that `maxsize` is significantly larger than the size of any queue you ever wish to maintain, so that you will never run out of room. Even so, this approach has a serious shortcoming when it comes to the *time required for the basic operations*. What's the problem? (Keep in mind that this convention requires that a queue with  $\ell$  numbers is always held in the first  $\ell$  cells of the vector.)

- (b) Instead of the proposal above, consider the following convention for maintaining the queue: as above, the queue will be maintained in a collection of adjacent cells of the vector; however the positions of the “front” and the “back” may drift over time. More concretely, the implementation will maintain two indices, `front` and `back`, so that the queue occupies cells `front`, ..., `back - 1`, as in the following diagram:



Concretely,

- When the first element is placed in the queue (via `enqueue`), it is placed at position 0 of the vector and `front` and `back` are set appropriately.
- The enqueue operation places its argument at position `back` and increments `back`.
- The dequeue operation returns the element at position `front` and increments `front`.

Of course, you should encapsulate all your code in a Scheme object which exposes methods that reflect the abstract datatype operations mentioned above. The function that constructs your queue object should take one parameter: the size of the vector used to represent the queue. Thus, your object should have the form:

```
(define (make-queue maxsize)
  (let ((...))                ;; internal queue variables
    (define (empty?) ...)    ;; queue methods
    (define (enqueue x) ...)
    (define (dequeue) ...)
    (define (dispatcher ...) ...) ;; the dispatcher
    dispatcher))
```

**Remark.** Make sure your object behaves in a reasonable way if `dequeue` is called on an empty queue. As mentioned above, you may pretend that the vector has an infinite number of cells, so there is no risk of `back` ever exceeding `maxsize`.