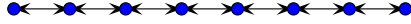


## Notes on Queues



*Queues* are also easy to implement in Scheme, though they are not so clean as *stacks*. Let's define a queue constructor:

```
(define (Queue)
  (define store nil)

  (define (this msg . args)
    (cond
      ((eq? msg 'enqueue) (apply enqueue args))
      ((eq? msg 'dequeue) (apply dequeue args))
      ((eq? msg 'empty?) (apply empty? args))
      (else (error "queue message not understood: " msg))
    )
  )

  (define (last x)
    (if (null? (cdr x))
        x
        ;else
        (last (cdr x))
    )
  )

  (define (enqueue x) ; add to the back
    (if (empty?)
        (set! store (list x))
        ;else
        (set-cdr! (last store) (list x))
    )
  )

  (define (dequeue) ; remove from the front
    ; user is responsible ensuring queue is non empty
    (define tmp (car store))
    (set! store (cdr store))
    tmp
  )

  (define (empty?)
    (eq? store nil)
  )

  this
)
```

Note that since we add to the back of the store and remove from the front, we get FIFO behavior. This is not a particularly efficient implementation. While removal takes a constant amount of time, adding items takes  $O(n)$  since the last function walks down the storage list every time. If we keep a back pointer, this will save us the walk.

```
(define (Queue)
  (define front nil)
  (define back nil)

  (define (this msg . args)
    (cond
      ((eq? msg 'enqueue) (apply enqueue args))
      ((eq? msg 'dequeue) (apply dequeue args))
      ((eq? msg 'empty?) (apply empty? args))
      (else (error "queue message not understood: " msg))
    )
  )

  (define (enqueue x) ; add to the back
```

```

(define tmp (list x))
(if (empty?)
    (begin (set! front tmp) (set! back tmp))
    ;else
    (begin (set-cdr! back tmp) (set! back tmp))
    )
)
(define (dequeue) ; remove from the front
  ; user is responsible ensuring queue is non empty
  (define tmp (car front))
  (set! front (cdr front))
  tmp
)
(define (empty?)
  (eq? front nil)
)

this
)

```

Note that the code for *enqueue* has to check whether the queue is empty. What if we could guarantee that the queue was never empty. Then the *enqueue* code would be simpler. How do we keep that guarantee? By creating a head item at the very start. Here's the implementation:

```

(define (Queue)
  (define front (list 'head))
  (define back nil)

  (define (this msg . args)
    (cond
      ((eq? msg 'enqueue) (apply enqueue args))
      ((eq? msg 'dequeue) (apply dequeue args))
      ((eq? msg 'empty?) (apply empty? args))
      (else (error "queue message not understood: " msg))
    )
  )

  (define (enqueue x) ; add to the back
    (set-cdr! back (list x))
    (set! back (cdr back))
  )

  (define (dequeue) ; remove from the front
    ; user is responsible ensuring queue is non empty
    (define tmp (cadr front))
    (set-cdr! front (cddr front))
    (if (null? (cdr front))
        (set! back front)
    )
    tmp
  )

  (define (empty?)
    (eq? (cdr front) nil)
  )

  (set! back front)
  this
)

```

In this implementation, note that *front* is initially bound to this head object. Just before the constructor returns, *back* is bound to this head object as well (the reason *back* wasn't bound to *front* when it was first defined has to do with the fact that Scheme may do the definitions in any order, even intermixed). Note also that the definition of *empty?* has changed to reflect the presence of a head node as has the definition of *dequeue*. We have also introduced a special case into the dequeue algorithm. Can we remove the special cases from both the *enqueue* and *dequeue* routine at the same time? The superior student will reflect and come to enlightenment on this issue.