Implementing Queues in Lisp

Peter Norvig

Sun Microsystems Laboratories
Two Federal Street
Billerica MA 01821

Richard C. Waters

Mitsubishi Electric Research Laboratories
201 Broadway
Cambridge MA 02139

A queue is a data structure where items are entered one at a time and removed one at a time in the same order—i.e., first in first out. They are the same as stacks except that in a stack, items are removed in the reverse of the order they are entered—i.e., last in first out. Queues are most precisely described by the functions that act on them:

`(make-queue)` Creates and returns a new empty queue.

`(queue-elements` *queue*`)` Returns a list of the elements in *queue* with the oldest element first. The list returned may share structure with *queue* and therefore may be altered by subsequent calls on `enqueue` and/or `dequeue`.

`(empty-queue-p` *queue*`)` Returns `t` if *queue* does not contain any elements and `nil` otherwise.

`(queue-front` *queue*`)` Returns the oldest element in *queue* (i.e., the element that has been in the *queue* the longest). When *queue* is empty, the results are undefined.

`(dequeue` *queue*`)` *Queue* is altered (by side-effect) by removing the oldest element in *queue*. The removed element is returned. When *queue* is empty, the results are undefined.

`(enqueue` *queue item*`)` *Queue* is altered (by side-effect) by adding the element *item* into *queue*. The return value (if any) is undefined.

```
(empty-queue-p (setq q (make-queue))) ⇒ t
(progn (enqueue q 'a) (enqueue q 'b) (queue-front q)) ⇒ a
(progn (enqueue q 'c) (enqueue q 'd) (dequeue q)) ⇒ a
(queue-elements q) ⇒ (b c d)
```

Having `enqueue` and `dequeue` alter *queue* by side-effect is convenient for most uses of queues and allows for efficient implementations. However, it means that care must be taken when queues are manipulated. For instance, if the output of `queue-elements` must be preserved beyond a subsequent use of `enqueue` or `dequeue` it must be copied (e.g., with `copy-list`).

Queues Implemented With Lists

Lisp's eponymous data structure, the list, can be used to represent a wide variety of data structures including queues. The implementation of queues in Figure 1 represents a queue as a cons cell whose car is a list of the elements in the queue, ordered with the oldest first.

The implementation in Figure 1 is simple and easy to understand. The close similarity of queues and stacks is highlighted by the fact that `dequeue` is implemented using `pop` and `enqueue` is implemented in a way that is very similar to `push`.

The one thing that may not be immediately clear about the implementation in Figure 1 is the reason why a header cell is necessary, instead of just using the list of elements in the queue to represent the queue. The header cell is needed so that an element can be added into an empty queue (and the last element removed from a one-element queue) purely by side-effect. For this to work, an empty queue must be some kind of mutable structure that can be pointed to (e.g., not just `nil`).

```
(defun make-queue () (list nil))
(defun queue-elements (q) (car q))
                                                                    ;space time
(defun empty-queue-p (q) (null (car q)))                            ;  2    2
(defun queue-front (q) (caar q))                                    ;  2    2
(defun dequeue (q) (pop (car q)))                                   ;  4    4
(defun enqueue (q item) (setf (car q) (nconc (car q ) (list item)))) ;  4  O(n)

(setq q (make-queue)) ⇒ (nil)
(progn (enqueue q 'a) (enqueue q 'b) (enqueue q 'c)  q) ⇒ ((a b c))
```

Figure 1: Queue implementation using nconc.

The functions in Figure 1 are divided into two groups to reflect the fact that the last four functions are called much more often than the first two. As a result, it is more important that they be efficient.

The first column of numbers on the right of Figure 1 shows the size of the code required if the corresponding function is compiled in line at the point of use. The size is specified as the number of primitive operations (car, cdr, cons, list, null, rplaca, rplacd, setq, branching, generating a constant nil, and calling an out-of-line function) that are necessary. For instance, dequeue requires 4 basic operations (a car, two cdrs and a rplacd).

The space numbers cannot be taken as exactly reflecting any particular Lisp implementation, because a given Lisp compiler may create code that performs unnecessary operations, and a given hardware platform may require multiple instructions to support some of the primitive operations. However, this does not matter a great deal, because the relative code size of functions is the key thing that is important in the context of this paper. (The validity of the numbers in Figure 1 as a basis for this kind of comparison has been verified by looking at the code produced by the compilers for the TI Explorer and the Symbolics Lisp Machine.)

An important virtue of the implementation of queues in Figure 1 is that the functions are coded compactly enough that it is practical to compile all of them in line (i.e., by declaring them inline). In most Common Lisp implementations, this is significantly more efficient then using out-of-line function calls.

The second column of numbers on the right of Figure 1 shows the number of basic operations that have to be executed at run time. If there is any branching required, the number reflects the control path that is most likely to be taken. These numbers reveal that there is a problem with the implementation. Most of the functions have small fixed costs that are independent of the size of the queue. However, the time required to perform the nconc in enqueue is proportional to the size of the queue.

Keeping a Pointer to the End of the Queue

The problem with nconc is not that it makes an expensive change (it merely performs one rplacd), but that it has to search down the entire list to locate the cons cell containing the last queue element. This inefficiency can be overcome by maintaining a pointer to the end of the list of queue elements.

In particular, BBN Lisp supported a queue data structure exactly like the one in Figure 1 except that the cdr of the header cell was used as a pointer to the list cell containing the last element in the queue (if any). Using this pointer, the six queue functions can be supported as shown in Figure 2. (In BBN Lisp, the function enqueue was called tconc.)

The only difference between Figure 2 and Figure 1 is in the implementation of enqueue. It is transformed into a constant-time operation and is therefore very much faster. Unfortunately,

```
(defun make-queue () (list nil))
(defun queue-elements (q) (car q))
                                                          ;space time
(defun empty-queue-p (q) (null (car q)))                  ;  2    2
(defun queue-front (q) (caar q))                          ;  2    2
(defun dequeue (q) (pop (car q)))                         ;  4    4
(defun enqueue (q item)                                   ;  9    8
  (let ((new-last (list item)))
    (if (null (car q))
        (setf (car q) new-last)
        (setf (cddr q) new-last))
    (setf (cdr q) new-last)))

(setq q (make-queue)) ⇒ (nil)
(progn (enqueue q 'a) (enqueue q 'b) (enqueue q 'c)  q) ⇒ ((a b . #1=(c)) . #1#)
```

Figure 2: Simple queue implementation using an end pointer.

enqueue is now too large to be comfortably compiled in line.

The implementation of **enqueue** in Figure 2 is larger than in Figure 1 primarily because it has to test for a special boundary condition. When the input queue is empty, **enqueue** has to do a **rplaca** to insert the (one element) list of queue elements in the car of the header cell; otherwise it has to do a **rplacd** to extend the list of queue elements.

Moving the Boundary Test to a Better Place

It is possible to remove the boundary test from **enqueue** by rearranging the queue data structure as follows. First, the two components of the header cell are interchanged, putting the pointer to the end of the queue in the car. Second, a convention can be adopted that an empty queue's end pointer points to the queue itself. These two changes allow the same code to be used for inserting an element into a queue whether or not the queue is empty, see Figure 3.

Unfortunately, while the two changes above simplify **enqueue**, they make it more difficult to implement **dequeue**. The problem is that **dequeue** now has a special boundary condition to test for—if the queue becomes empty, the queue's last pointer has to be made to point to the queue itself. However, because this is a simpler special case than the one in **enqueue** in Figure 2, it does not lead to as much overhead. Also, since some applications do significantly more **enqueues** than **dequeues** and no application does more **dequeues**, the trade-off is worthwhile.

The implementation approach in Figure 3 takes subtle advantage of the typeless nature of Lisp. In most other languages, the header cell for a queue would be a different type of structure from the

```
(defun make-queue () (let ((q (list nil))) (setf (car q) q)))
(defun queue-elements (q) (cdr q))
                                                          ;space time
(defun empty-queue-p (q) (null (cdr q)))                  ;  2    2
(defun queue-front (q) (cadr q))                          ;  2    2
(defun dequeue (q)                                        ;  7    6
  (let ((elements (cdr q)))
    (unless (setf (cdr q) (cdr elements))
      (setf (car q) q))
    (car elements)))
(defun enqueue (q item) (setf (car q) (setf (cdar q) (list item))))  ;  4    4

(setq q (make-queue)) ⇒ #1=(#1#)
(progn (enqueue q 'a) (enqueue q 'b) (enqueue q 'c)  q) ⇒ (#1=(c) a b . #1#)
```

Figure 3: A compact and efficient queue implementation.

```
(defun make-queue () (let ((q (list nil))) (cons q q)))
(defun queue-elements (q) (cdar q))
                                                                    ;space time
(defun empty-queue-p (q) (null (cdar q)))                           ;  3    3
(defun queue-front (q) (cadar q))                                   ;  3    3
(defun dequeue (q) (car (setf (car q) (cdar q))))                   ;  4    4
(defun enqueue (q item) (setf (cdr q) (setf (cddr q) (list item)))) ;  4    4

(setq q (make-queue)) ⇒ (#1=(nil) . #1)
(progn (enqueue q 'a) (enqueue q 'b) (enqueue q 'c)  q) ⇒ ((nil a b . #1=(c)) . #1#)
```

Figure 4: Another compact and efficient queue implementation.

cells forming the linked list of queue elements. This would block **enqueue** from treating the cdr of the header cell the same as the cdr of a linked list cell. (In some languages, this problem could be overcome by judicious use of type unioning or type-check bypassing.)

Eliminating the Boundary Test by Adding a Cell

A different way to improve on Figure 2 is to eliminate the need for any boundary tests at all, by adding a dummy cell into the list holding the elements in the queue as shown in Figure 4. This allows **enqueue** and **dequeue** to operate essentially as if the queue were never empty. However, the other functions have to be adjusted to skip over the dummy cell, and therefore become a bit longer.

Whether or not the implementation in Figure 4 is better than the one in Figure 3 depends on the details of your Lisp implementation and which queue operations you use most. For instance, if calls on **dequeue** are particularly infrequent (e.g., because a list of the items queued is the primary result desired), then the implementation in Figure 3 is better. In contrast, if the Lisp Implementation has special hardware support for following chains of pointers through cons cells (e.g., the TI Explorer), Figure 4 is better.

Queues Implemented With Vectors

Lists are a convenient basis for queues. In particular, the interaction of **cons** and garbage collection provides support for queues of unbounded length without any special provisions having to be made. However, list-based implementations are wasteful of memory, because an entire cons cell has to be used to store each element in the queue, and as elements are enqueued and dequeued, new cons cells continually have to be allocated.

Memory efficient implementations of queues are possible using vectors. This approach is often taken in other languages. Figure 5 shows an implementation like those usually shown in introductory data-structure texts. The basic approach is to store the elements of a queue as a section of a vector treated as a ring. The elements are stored in reverse order in the vector so that a comparison with zero can be used to detect when either the front or end pointers reach the edge of the vector.

The primary advantage of a vector-based implementation is that it requires only about half the memory to store the contents of the queue. If the queue elements are shorter than a word (e.g., characters or bits) even more savings are possible. In addition, enqueuing and dequeuing elements does not generate any garbage at all (unless the queue size gets so large that an enlarged vector has to be allocated).

The primary disadvantage of a vector-based implementation is that it is more complicated. In particular, it has to do all its own memory management. This means that the queue still takes up a lot of space even when it is empty. In addition, provision has to be made for extending the vector holding the queue if it becomes full. (In figure 5, this is supported by the function **extend-queue** and a fullness test in **enqueue**.)

```
(defstruct q front end size elements)

(defun make-queue (&optional (size 20))
  (make-q :front (1- size) :end (1- size) :size size
          :elements (make-sequence 'simple-vector size)))
(defun queue-elements (q)
  (when (not (empty-queue-p q))
    (do ((i (1+ (q-end q)) (1+ i))
         (result nil))
        (nil)
      (when (= i (q-size q)) (setq i 0))
      (push (svref (q-elements q) i) result)
      (when (= i (q-front q)) (return result)))))
                                                          ;space time
(defun empty-queue-p (q) (= (q-front q) (q-end q)))       ;  3    3
(defun queue-front (q) (svref (q-elements q) (q-front q))) ;  3    3
(defun dequeue (q)                                        ;  7    6
  (let ((front (q-front q)))
    (prog1 (svref (q-elements q) front)
           (when (zerop front) (setq front (q-size q)))
           (setf (q-front q) (1- front)))))
(defun enqueue (q item)                                   ; 10    8
  (let ((end (q-end q)))
    (setf (svref (q-elements q) end) item)
    (when (zerop end) (setq end (q-size q)))
    (when (= (setf (q-end q) (1- end)) (q-front q)) (extend-queue q))))

(defun extend-queue (q)
  (let* ((elements (q-elements q))
         (size (q-size q))
         (new-size (* 2 size))
         (divide (1+ (q-front q)))
         (new-end (+ divide size -1))
         (new (make-sequence 'simple-vector new-size)))
    (replace new elements :end2 divide)
    (replace new elements :start1 (1+ new-end) :start2 divide)
    (setf (q-elements q) new)
    (setf (q-end q) new-end)
    (setf (q-size q) new-size)))

(progn (setq q (make-queue))
       (dotimes (i 17) (enqueue q '-) (dequeue q))
       (dotimes (i 5) (enqueue q i)) q)
 ⇒ #S(queue front 17 end 2 size 20 elements
       #(2 1 0 - - - - - - - - - - - - - - - - - 4 3))
```

Figure 5: A traditional vector-based queue implementation.

Whenever possible, it is good to start the queue at a size that is sufficient to hold the maximum expected size, rather than starting at an arbitrary size like 20. For this reason the function **make-queue** is extended by giving it an optional size argument. Given firm maximum-size information one could go further and dispense with **extend-queue** and the fullness test in **enqueue**. However, this is a dangerous practice and saves relatively little.

It is worthy of note that it would be a mistake to use an adjustable array in the queue data structure. This would make extending the array a little bit easier, but would slow up all of the other operations on the vector. Adjustable arrays are only helpful when there may be many pointers directly to the array that has to be extended. Whenever, as here, there is known to be only one pointer, it is much better to change the pointer to point to a new array, than to extend the array itself.

Another problem is that **queue-elements** becomes an **O(n)** operation, since it has to copy the

```
(defstruct q front end size elements)

(defun make-queue (&optional (size 20))
  (make-q :front (- size 1) :end (- size 1) :size size
          :elements (make-sequence 'simple-vector size)))
(defun queue-elements (q)
  (do ((i (1+ (q-end q)) (1+ i))
       (result nil))
      ((> i (q-front q)) result)
    (push (svref (q-elements q) i) result)))
                                                          ;space time
(defun empty-queue-p (q) (= (q-front q) (q-end q)))       ;  3    3
(defun queue-front (q) (svref (q-elements q) (q-front q))) ;  3    3
(defun dequeue (q)                                        ;  5    5
  (prog1 (svref (q-elements q) (q-front q)) (decf (q-front q))))
(defun enqueue (q item)                                   ;  8    7
  (setf (svref (q-elements q) (q-end q)) item)
  (when (minusp (decf (q-end q))) (shift-queue q)))

(defun shift-queue (q)
  (let* ((elements (q-elements q))
         (new elements))
    (when (> (q-front q) (/ (q-size q) 2))
      (setq new (make-sequence 'simple-vector (* 2 (q-size q))))
      (setf (q-elements q) new)
      (setf (q-size q) (* 2 (q-size q))))
    (setf (q-end q) (- (q-size q) 2 (q-front q)))
    (replace new elements :start1 (1+ (q-end q)))
    (setf (q-front q) (1- (q-size q)))))

(progn (setq q (make-queue))
       (dotimes (i 17) (enqueue q '-) (dequeue q))
       (dotimes (i 5) (enqueue q i)) q)
 ⇒ #S(queue front 19 end 14 size 20 elements
       #(2 1 0 - - - - - - - - - - - - - 4 3 2 1 0))
```

Figure 6: A faster vector-based queue implementation.

queue contents into a list. If you want to be able to easily get a list of the elements in a queue, it is better to start with a list-based implementation.

A final problem with Figure 5 is the inefficiency of some of the key operations. The functions **empty-queue** and **queue-front** are small and could be coded in line. However, **dequeue** is on the borderline in size and **enqueue** is quite large.

Shifting Is Better Than Using a Ring

Figure 6 shows the kind of improvements than can be obtained using a little ingenuity. The key difference between Figure 6 and Figure 5 is that the implementation does not treat the vector as a ring. Rather, whenever the queue reaches the end of the vector, it is shifted over (by the function **shift-queue**, which also extends the vector if necessary).

One might well imagine that operating on the vector as a ring had to be better than shifting everything over every time the queue reaches the edge of the vector. However, as long as the queue is significantly shorter than the vector (say only 2/3 the length or less) then shifting does not have to occur very often, and performing occasional shifts ends up being cheaper than complex decrementing of the pointers all of the time. **Dequeue** and **enqueue** both become significantly more efficient, and **dequeue** becomes short enough to easily code in line.

All in all, except for the fact that the queue structure has to be a bit bigger for things to work out efficiently, the implementation in Figure 6 is better than the one in Figure 5 in all respects.

Given its memory efficiency and quite reasonable speed, it is worth considering Figure 6 as an alternative to a list-based implementation in any situation where the function `queue-elements` is not used.

Conclusion

Lisp provides an all-purpose data structure—the list—which is often adequate for rapid prototyping. But when an efficient solution is required, Lisp programmers must choose their data structures carefully. Figures 3–6 show two efficient list based implementations of queues and two efficient vector-based implementations. Which is appropriate to use depends on the details of the exact situation in question.

The various implementations presented above illustrate several general issues to keep in mind when seeking efficient algorithms. Introducing alignments of components can often eliminate special cases (e.g., the way the queue data structure is rearranged in Figure 3). Sometimes a computation can be moved from an expensive context to a less expensive one (e.g., moving the boundary test from **enqueue** to **dequeue** in Figure 3). Many times, it is better to do a little extra work all the time, then do an expensive check to determine when extra work is really needed (e.g., indexing through the extra cell in Figure 4 is better in many situations than testing for whether the list is empty). Other times, it is better to introduce extra work some of the time to eliminate a steady background of work (e.g., occasional wholesale shifting in Figure 6 is better than continual performing complex pointer stepping). Slimming functions down to in-line-able size can pay big pragmatic dividends. Above all, the only way to get a really efficient algorithm is to experiment with many alternatives.

From: "Damaris M. Ayuso" <dayuso@BBN.COM>; Date: Wed, 4 Sep 91 13:42:38 EDT

HOW TO DETERMINE WHICH PROGRAMMING LANUAGE YOU ARE USING

C: You shoot yourself in the foot.

APL: You hear a gunshot, and there's a hole in your foot, but you don't remember enough linear algebra to understand what the hell happened.

C++: You accidently create a dozen instances of yourself and shoot them all in the foot. Providing emergency medical care is impossible since you can't tell which are bitwise copies and which are just pointing at others and saying, "that's me, over there."

Modula/2: After realizing that you can't actually accomplish anything in the language, you shoot yourself in the head.

Smalltalk: You spend so much time playing with the graphics and windowing system that your boss shoots you in the foot, takes away your workstation, and makes you develop in COBOL on a character cell terminal.

FORTRAN: You shoot yourself in each toe, iteratively, until you run out of toes; then you read in the next foot and repeat. If you run out of bullets, you continue anyway because you have no exception-processing ability.

Algol: You shoot yourself in the foot with a musket. The musket is esthetically fascinating, and the wound baffles the adolescent medic in the emergency room.

COBOL: USEing a COLT45 HANDGUN, AIM gun at LEG.FOOT, THEN place ARM.HAND.FINGER on HANDGUN.TRIGGER, and SQUEEZE. THEN return HANDGUN to HOLSTER. Check whether shoelace needs to be retied.

BASIC: Shoot self in foot with water pistol. On big systems, continue until entire lower body is waterlogged.

PL/I: You consume all available system resources, including all offline bullets. The Data Processing & Payroll Department doubles its size, triples its budget, acquires four new mainframes, and drops the original one on your foot.

SNOBOL: You grab your foot with your hand, then rewrite your hand to be a bullet. The act of shooting the original foot then changes your hand/bullet into yet another foot (a left foot).

LISP: You shoot yourself in the appendage which holds the gun with which you shoot yourself in the appendage which holds the gun with which you shoot yourself in the appendage which holds the gun with which you shoot yourself in the appendage which holds the gun with which you ...