Efficiency & Side-Effects

Features of Side-Effects:

- Can avoid copying with side-effects, more efficient.
- Program analysis is more difficult.
- Introduces bugs.
- Separate copies must be kept of "old" versions of data.

Queue 1 – Side-Effects

Implementation: head and tail pointers.

```
(define (queue-create) (cons (list) (list)))
(define (queue-empty? q) (null? (car q)))

(define (queue-dequeue! q)
   (let ((head (caar q)))
        (set-car! q (cdar q))
        (if (queue-empty? q)
            (set-cdr! q (list))
        )
        head
   )
)
```


Queue 1 (cont)

```
(define (queue-enqueue! q data)
   (if (queue-empty? q)
       (begin
           (set-car! q (list data))
           (set-cdr! q (car q))
       )
       (begin
           (set-cdr! (cdr q) (list data))
           (set-cdr! q (cddr q))
       )
   )
   'ok
)
```


Efficiency

What are the runtimes of

queue-create

queue-empty?

queue-dequeue!

queue-enqueue!

Seems pretty good.

Drawbacks

Drawbacks of implementation of queue with side-effects:

- There is only one “live” copy of each queue.
- To share the queue, it must be copied.
- Accidental sharing of queue might cause bugs.

Queue 2 – Pure-Functional

Implementation: Single List

```
(define (queue-create) (list))

(define queue-empty? null?)

(define queue-head car)

(define queue-dequeue cdr)

(define (queue-enqueue q data) (append q (list data)))
```


Efficiency

What are the runtimes of

queue-create

queue-empty?

queue-head

queue-dequeue

queue-enqueue

Amortized Analysis

(an introduction)

To find “average” (amortized) runtime.

- Charge a certain amount for each operation.
- For cheap operations, we make a profit.
- For expensive operations, we have a loss.
- The trick is to make sure we never go negative.
- Success implies amortized runtime is amount we charged.

Queue 3 – Amortized

Implementation: With *front* list and *rear* list.

- first element of *front* is head of queue.
- first element of *rear* is last element in queue – it's backwards.
- "proper" non-empty queue never has empty *front*.

```
(define (queue-create) (list (list) (list)))
(define (queue-empty? q) (null? (car q)))

(define (queue-head q) (caar q))
```


Queue 3 (cont)

```
(define (queue-make-proper q)
   (if (null? (car q))
       (list (reverse (cadr q)) (list))
       q
   )
)

(define (queue-dequeue q)
   (queue-make-proper (list (cdar q) (cadr q))))

(define (queue-enqueue q data)
   (queue-make-proper (list (car q) (cons data (cadr q)))))
```


Efficiency

What are the worst-case runtimes of:

queue-create

queue-empty?

queue-make-proper

queue-head

queue-dequeue

queue-enqueue

Better Analysis

Let n be number of items in queue.

- Charge \$2 for each `queue-make-proper` call.
- If `(queue-make-proper q)` returns `q`, cost is \$1.
- If `(queue-make-proper q)` calls `reverse`, cost is $\$n$.

If we start with an empty queue, after any set of operations, will we always have profit?

The answer is yes*!

*if we never use "old" queues.

Example 1

```
(define q1 (foldl queue-enqueue (queue-create) '(1 2 3 4 5 6)))

;; what does q1 look like?  How much $ do we have?

(define q2 (queue-dequeue q1))

;; what does q2 look like?  How much $ do we have?

(define q3 (queue-dequeue (queue-dequeue q2)))

;; what does q3 look like?  How much $ do we have?
```


Using "Old" Queues

What can happen if we use "old" queues?

```
(define q1 (foldl queue-enqueue (queue-create) '(1 2 3 4 5 6)))

;; what does q1 look like?  How much $ do we have?

(define q2 (queue-dequeue q1))

;; what does q2 look like?  How much $ did this operation cost?

(queue-dequeue q1)
(queue-dequeue q1)
(queue-dequeue q1)

;; how much $ do we have now?
```


Queue 1 vs Queue 3

Recall, Queue 1, using side-effects.

For Queue 1 once a dequeue happens the original queue is no longer accessible.

Thus, if usage patterns of queue do not use "old" copies, then Queue 1 and Queue 3 are comparable: Their average runtime is about the same.

If usage patterns do use "old" copies, then Queue 1 breaks unless explicit copies are made.

Queue 3 will not break, but we might pay for expensive operation twice.

Persistence

Two types of data structures

ephemeral: Only access to a single version of data structure, old copies not accessed.

persistent: Access to past copies of data structure allowed.

Can we build an efficient persistent queue?