CSE 1729:Principles of Programming

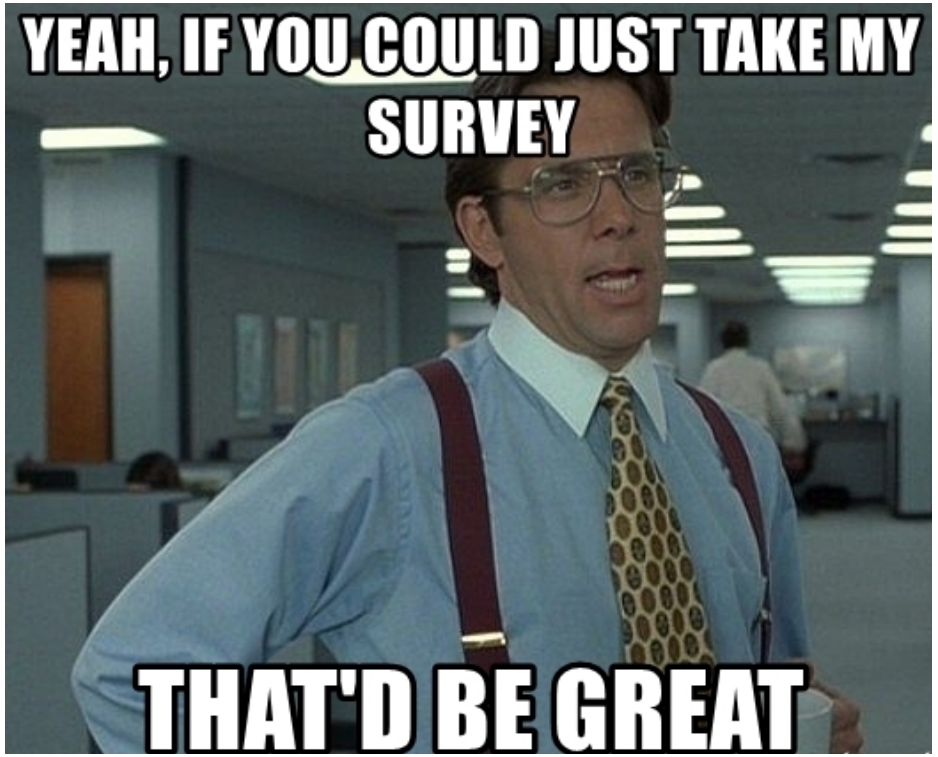
Lecture 22 : Queues with Pointers

Kaleel Mahmood

Department of Computer Science and Engineering

University of Connecticut

The SET Survey



- Please complete it here: https://blueapp.grove.ad.uconn.edu/Blue/
- If 70% of the class fills it out we will drop the lowest TWO homework grades!!!

What you should consider:

- Lectures NEVER went overtime, always short and to the point.
- We were the ONLY section of CSE 1729 to give students exposure to Python AND Scheme.
- Only section of CSE 1729 taught using memes (probably).
- Your feedback is appreciated.

Review from last lecture

```
x = 5
y = x
x = 3
print(y)
```

```
b1 = BankAccount()
b2 = b1
b1.addBalance(500)
b2.checkBalance()
```

- Pointers are used to access a certain place in memory.
- When dealing with a primitive, using the equality symbol you get a NEW pointer and a NEW place in memory with the value copied over.
- When dealing with objects, using the equality symbol gets you a NEW pointer to the SAME place in memory.

Review from last lecture (2)

```
(define tail (cons 0 1))
```

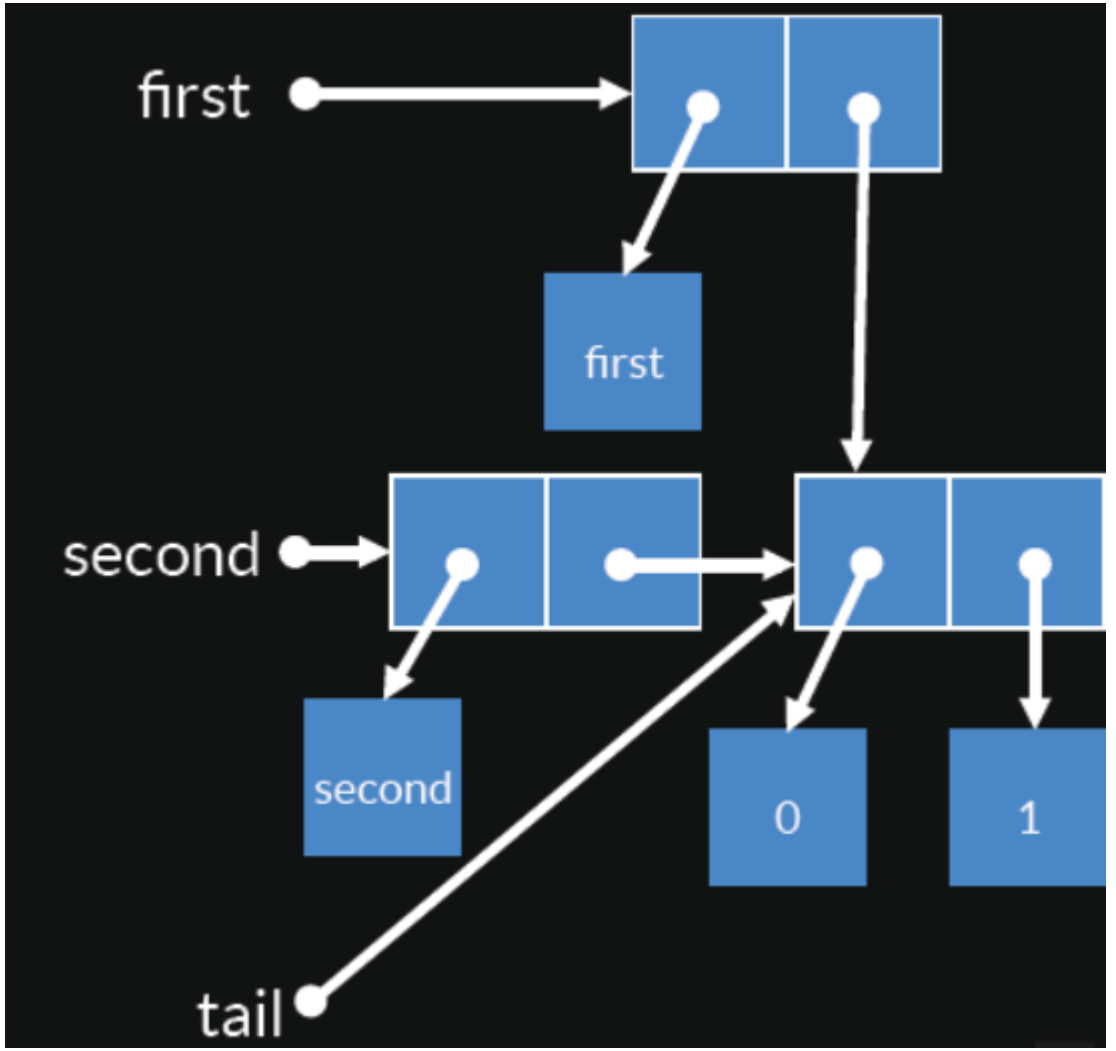
```
(define first (cons 'first tail))
```

We now have linked first and tail together.

```
(define second (cons 'second tail))
```

We now have linked second and tail together.

```
> first
(first 0 . 1)
> second
(second 0 . 1)
```



BEWARE: THE SEMANTICS OF VARIABLE ASSIGNMENT *DEPEND ON THE VALUE*

- `(define var <expr>)` sets `var` to be the value returned by `<expr>`.
- If the value is an atomic scheme type (number, Boolean, etc.): a new cell with the value is set up & the variable is directed there.

```
1  > (define x 0)
2  > (define y x)
3  > x
4  0
5  > y
6  0
7  > (set! x 1)
8  > x
9  1
10 > y
```

Now what is the value of y?

0

If this is confusing go back and review the example from last lecture where we did the same thing in Python.

BEWARE: THE SEMANTICS OF VARIABLE ASSIGNMENT *DEPEND ON THE VALUE*

```
 1 > (define x (cons 0 0))
 2 > (define y x)
 3 > x
 4 (0 . 0)
 5 > y
 6 (0 . 0)
 7 > (set-car! x 1)
 8 > x
 9 (1 . 0)
10 > y
```

What is the value of “y”? To check this you should first ask, when “y” was defined, was defined based on a primitive or object?

```
10 > y
11 (1 . 0)
```

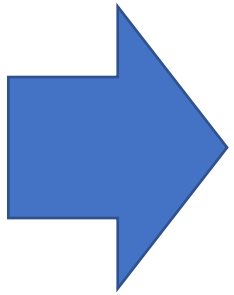
MUTABLE STRUCTURED DATA IN SCHEME AND RACKET

- Scheme and Racket handle mutable structured data a little differently.
- To obtain the behavior we have discussed in class (and read about in SICP), you'll need to start your Racket buffer with

#lang racket

- The Racket designers mean business...a pair is either *mutable* or *functional* and never the twain shall meet.

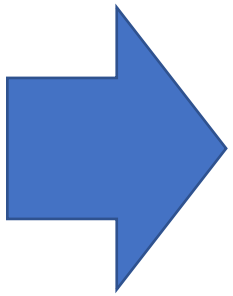
```
1 #lang racket
2 (define x (cons 1 2))
3 x
4 (car x)
5 (cdr x)
```



```
Welcome to DrRacket, version 8.3 [cs].
Language: racket, with debugging; memory limit: 128 MB.
'(1 . 2)
1
2
>
```

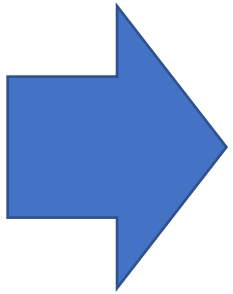
- The Racket designers mean business...a pair is either *mutable* or *functional* and never the twain shall meet.

```
1  #lang racket
2  (define x (cons 1 2))
3  x
4  (car x)
5  (cdr x)
```



```
Welcome to DrRacket, version 8.3 [cs].
Language: racket, with debugging; memory limit: 128 MB.
'(1 . 2)
1
2
>
```

```
1  #lang racket
2  (define x (cons 1 2))
3  x
4  (car x)
5  (cdr x)
6  (set-car! x 2)
```



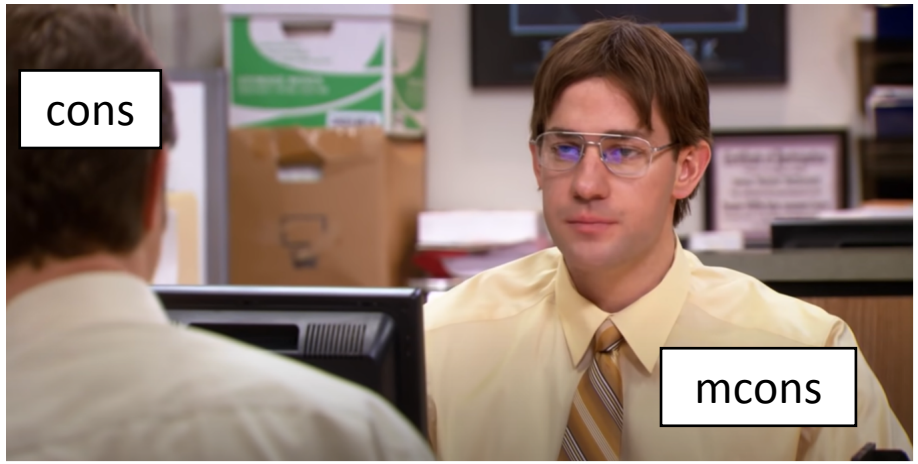
```
Welcome to DrRacket, version 8.3 [cs].
Language: racket, with debugging; memory limit: 128 MB.
set-car!: unbound identifier in: set-car!
>
```

How can we change pairs and lists in non-Scheme languages?

Answer: Have a completely SEPARATE datatype.

From the Racket documentation:

> A mutable pair is not a pair; they are completely separate datatypes.



Source:
https://docs.racket-lang.org/reference/mpairs.html#%28def._%28%28quote._~23~25kernel%29._set-mcar%21%29%29

MUTABLE PAIRS IN RACKET

- The Racket designers mean business...a pair is either *mutable* or *functional* and never the twain shall meet.
- Pairs built with `cons` cannot be destructively assigned.
- A mutable pair is built with mcons, accessed with `mcar` and `mcdr`, destructive assignment via `set-mcar!` and `set-mcdr!`.

```
1 > (define a (mcons 1 2))
2 > (car a)
3 . . car: expects argument of type <pair>; given (mcons 1 2)
4 > (mcar a)
5 1
6 > (set-mcar! a 3)
7 > a
8 (mcons 3 2)
```

Can't mix and match

MUTABLE LIST INFRASTRUCTURE

- Racket has infrastructure for mutable lists. As with pairs, lists and mutable lists (built from mutable pairs) must be kept separate.
- The infrastructure is loaded with:

```
> (require compatibility/mlist)
```

Then mutable operations can be called:

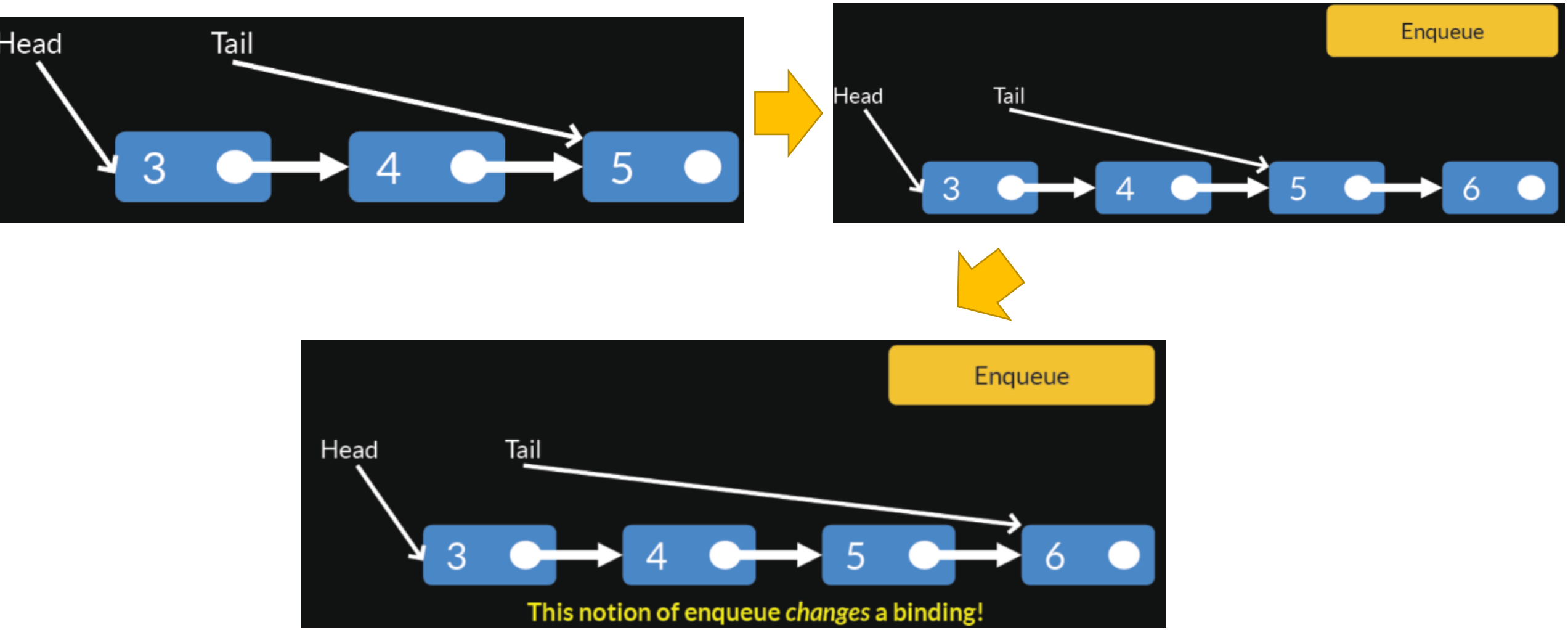
```
> (mlist 1 2 3 4)
(mcons 1 (mcons 2 (mcons 3 (mcons 4 '()))))
> (null? (mlist 1 2))
#f
```

```
> (mappend (mlist 1 2) '())
(mcons 1 (mcons 2 '()))
```

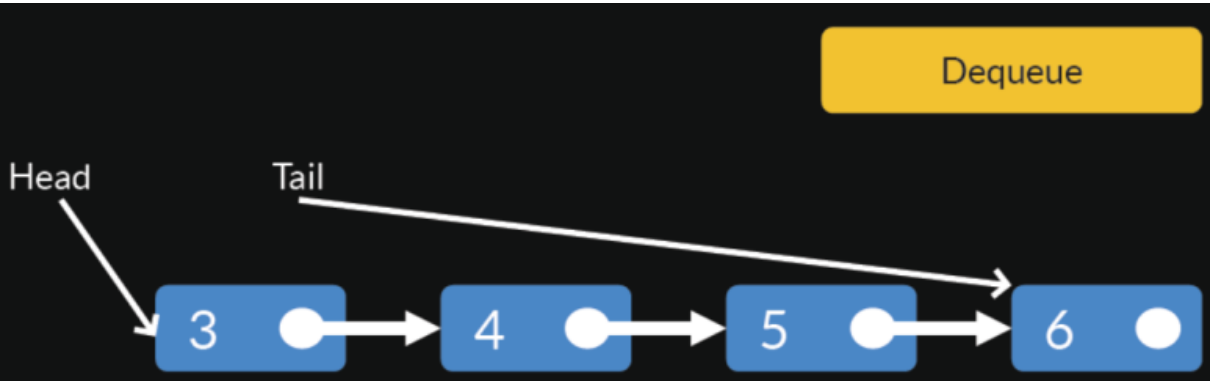
A MORE SOPHISTICATED QUEUE DATA STRUCTURE

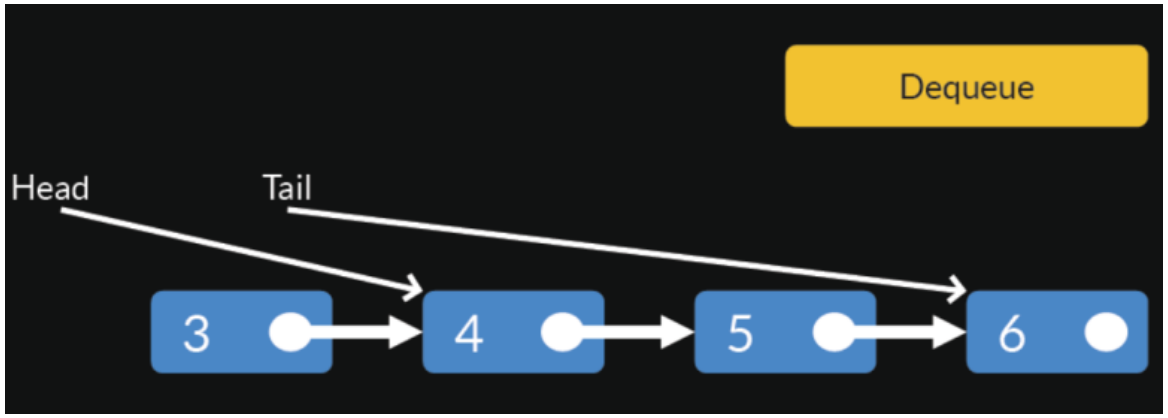
- We will maintain a queue as a linked list that “remembers” its tail.
- Notice that we can enqueue and dequeue in a fixed amount of time (independent of number of elements in the queue) if we remember the tail.

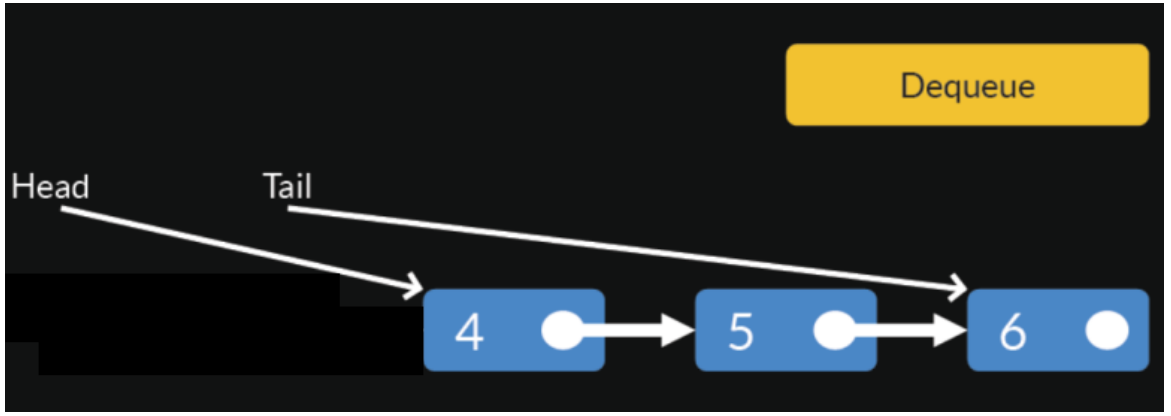
Pictorial example of what we want to accomplish with enqueue:



Pictorial example of what we want to accomplish with dequeue:







IMPLEMENTING A QUEUE AS A LINKED LIST IN SCHEME

- In this picture: each "node" of our "linked list" must remember two things
  - A **value**,
  - **A pointer** to its successor.
- We implement this as a **pair** containing the value and a pointer to the next node.

```
(value . next)
```

- We thus define the accessor functions:

```
(define (value n) (car n))
(define (next n) (cdr n))
```

AS FOR THE QUEUE ITSELF, WE NEED TO RETAIN TWO POINTERS

- To maintain the queue, we maintain two pointers: **head** and **tail**. We will maintain the invariant that **head** *points to the front of the queue*; **tail** *points to the end of the queue*.
- To check emptiness:

```
(define (empty?) (null? head))
```

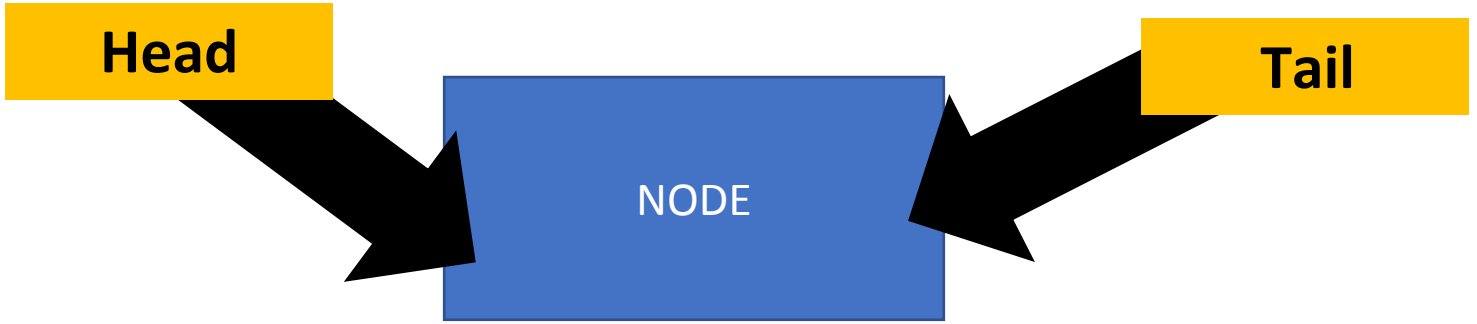
- To return the value at the head:

```
(define (front) (value head))
```

Assume an empty queue to start…

A MORE SOPHISTICATED QUEUE DATA STRUCTURE

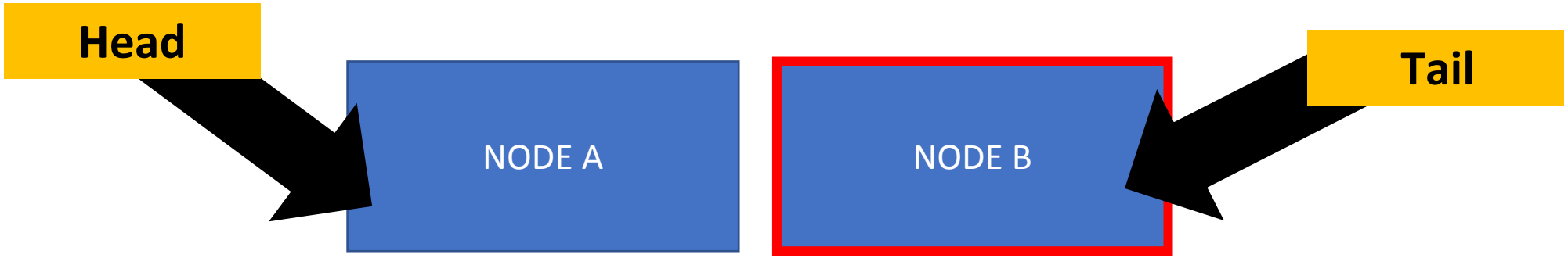
```
1  (define (enqueue x)
2    (let ((new-node (cons x '())))
3      (begin
4        (if (empty?)
5            (set! head new-node)
6            (set-cdr! tail new-node))
```



What if the queue is not empty?

A MORE SOPHISTICATED QUEUE DATA STRUCTURE

```
1  (define (enqueue x)
2    (let ((new-node (cons x '())))
3      (begin
4        (if (empty?)
5            (set! head new-node)
6            (set-cdr! tail new-node))
7        (set! tail new-node))))
```



Would the following NEW code work instead?

Previous code:

```
1 (define (enqueue x)
2   (let ((new-node (cons x '())))
3     (begin
4       (if (empty?)
5           (set! head new-node)
6           (set-cdr! tail new-node))
7       (set! tail new-node))))
```

NEW code:

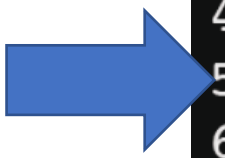
```
1 (define (enqueue x)
2   (begin
3     (if (empty?)
4         (set! head (cons x '()))
5         (set-cdr! tail (cons x '())))
6     (set! tail (cons x '()))))
```

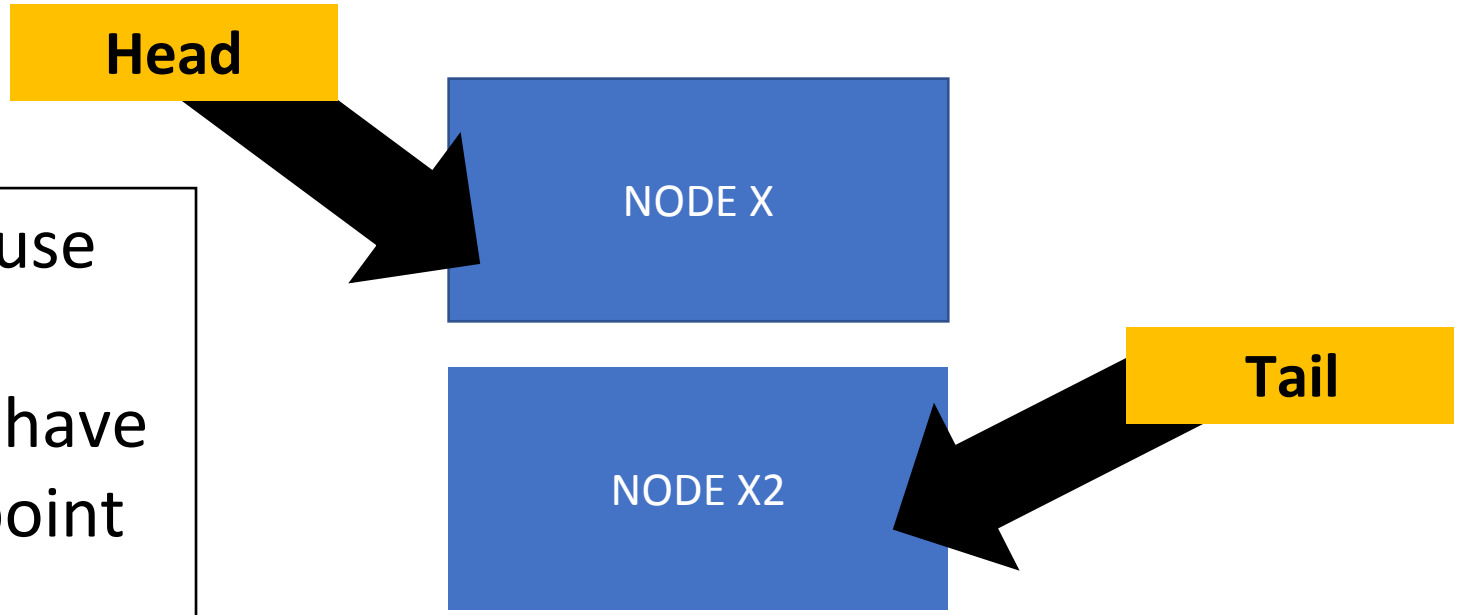
Consider starting the list…

```
1  (define (enqueue x)
2    (begin
3      (if (empty?)
4          (set! head (cons x '()))
5          (set-cdr! tail (cons x '())))
6      (set! tail (cons x '()))))
```

Consider starting the queue…

```
1  (define (enqueue x)
2    (begin
3      (if (empty?)
4          (set! head (cons x '()))
5          (set-cdr! tail (cons x '())))
6      (set! tail (cons x '()))))
```

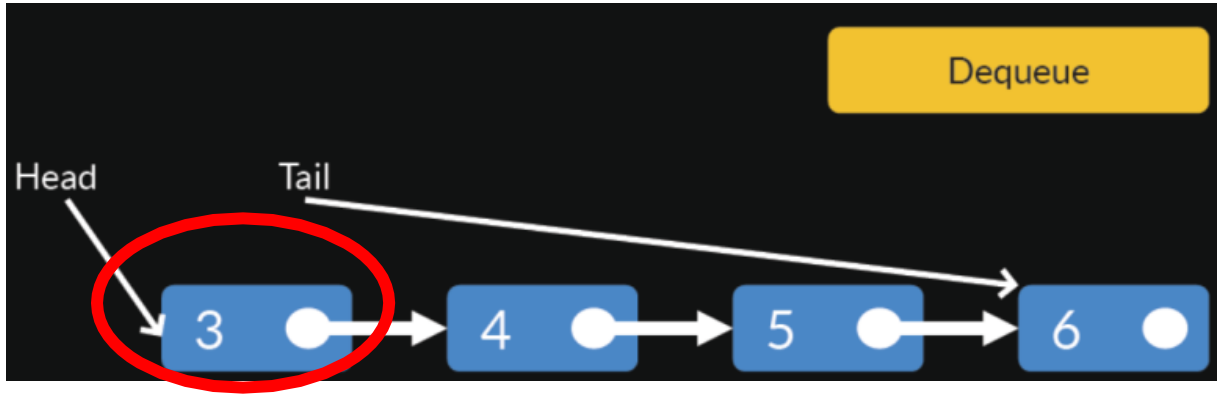




This is wrong because when starting the queue we need to have the head and tail point to the SAME node.

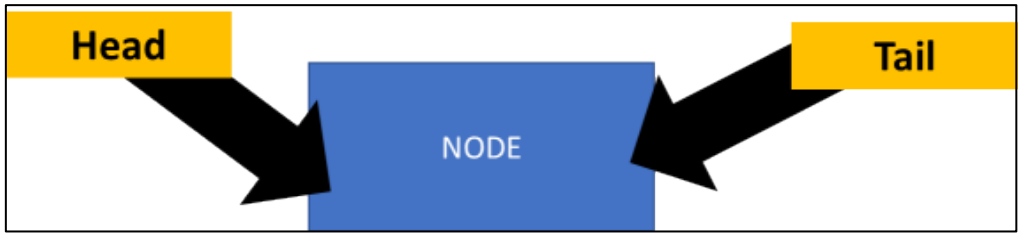
We can also have the dequeue operation

```
1  (define (dequeue)
2    (let ((return (value head)))
```

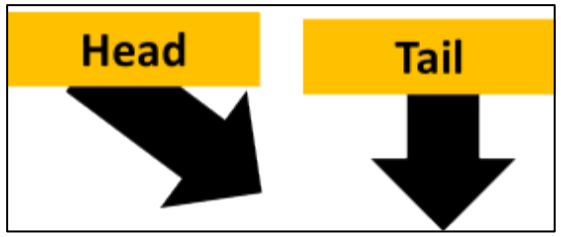


Case: Only one element in the queue

Before dequeue operation



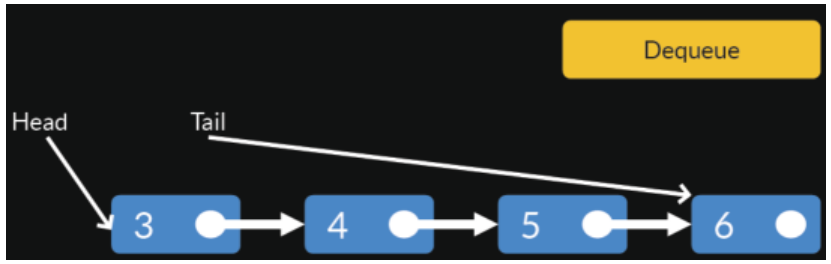
After dequeue operation
(head and tail point to null)



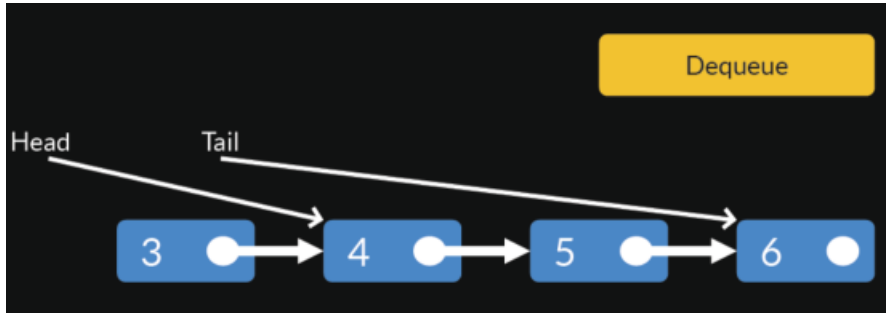

```
1  (define (dequeue)
2    (let ((return (value head)))
3      (if (eq? head tail)
4          (begin (set! head '())
5                 (set! tail '())
6                 return)
```

Case: multiple elements in the queue

Before dequeue operation



After dequeue operation




```
1  (define (dequeue)
2    (let ((return (value head)))
7          (begin (set! head (next head))
8                 return))))
```

```
(define (next n) (cdr n))
```

Putting everything together…

Putting everything together…

Putting everything together…

Putting everything together…

```
(define (make-queue)
  (let ((head '())
        (tail '()))
    (define (value n) (car n))
    (define (next n)  (cdr n))
    (define (empty?) (null? head))
    (define (front) (value head))
    (define (enqueue x)
      (let ((new-node (cons x '())))
        (begin
          (if (empty?)
              (set! head new-node)
              (set-cdr! tail new-node))
          (set! tail new-node))))
    (define (dequeue)
      (let ((return (value head)))
        (if (eq? head tail)
            (begin (set! head '())
                   (set! tail '())
                   return)
            (begin (set! head (next head))
                   return))))
```

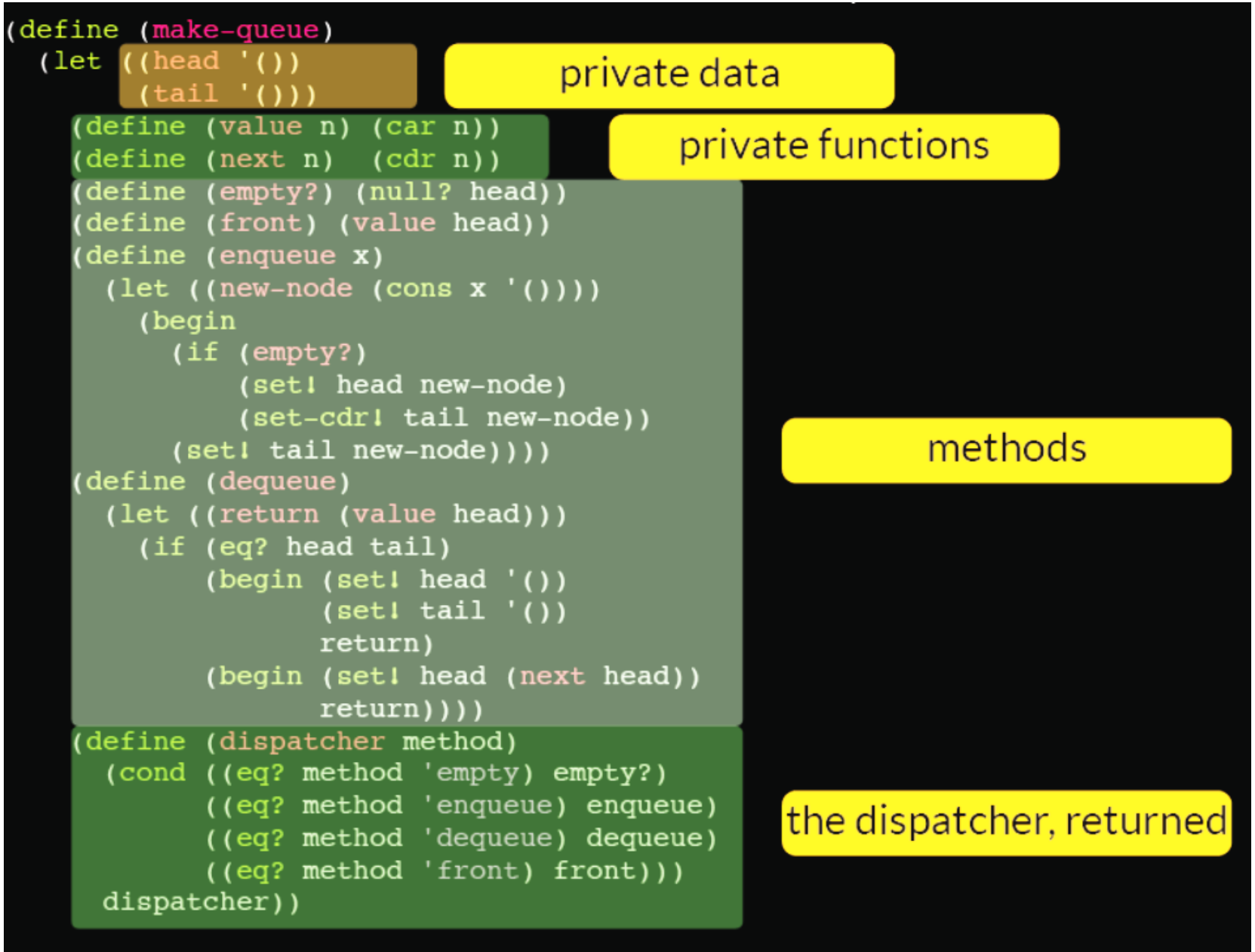
private data

private functions

methods

Are we done?

Putting everything together…done!



THE QUEUE, IN ACTION

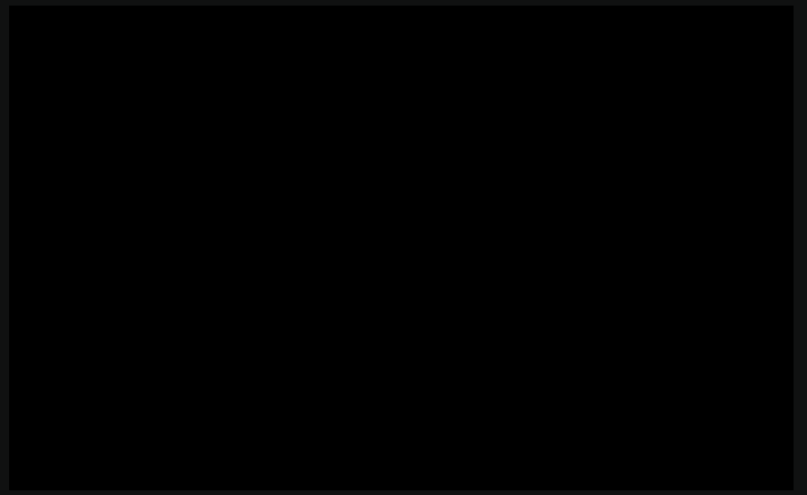
```
1  > (define q (make-queue))
2  > ((q 'enqueue) 5)
3  > ((q 'enqueue) 6)
4  > ((q 'enqueue) 7)
5  > ((q 'dequeue))
```

Now what will be the value returned in line 5?

```
6  5
7  > ((q 'enqueue) 8)
8  > ((q 'dequeue))
9  6
10 > ((q 'dequeue))
```

THE QUEUE, IN ACTION

```
> (define q (make-queue))
> ((q 'enqueue) 5)
> ((q 'enqueue) 6)
> ((q 'enqueue) 7)
> ((q 'dequeue))
5
> ((q 'enqueue) 8)
> ((q 'dequeue))
6
> ((q 'dequeue))
7
> ((q 'dequeue))
8
> ((q 'empty))
#t
```



At the end we can check and see the queue is indeed empty.

We build a Queue with pointers.
Why did we build it this way?


```
(define (make-queue)
  (let ((head '())
        (tail '()))
    (define (value n) (car n))
    (define (next n)  (cdr n))
    (define (empty?) (null? head))
    (define (front) (value head))
    (define (enqueue x)
      (let ((new-node (cons x '())))
        (begin
          (if (empty?)
              (set! head new-node)
              (set-cdr! tail new-node))
          (set! tail new-node))))
    (define (dequeue)
      (let ((return (value head)))
        (if (eq? head tail)
            (begin (set! head '())
                   (set! tail '())
                   return)
            (begin (set! head (next head))
                   return))))
    (define (dispatcher method)
      (cond ((eq? method 'empty) empty?)
            ((eq? method 'enqueue) enqueue)
            ((eq? method 'dequeue) dequeue)
            ((eq? method 'front) front)))
      dispatcher))
```

private data

private functions

methods

the dispatcher, returned

A queue built with pointers

	 Array		 Queue
Insertion	O(n)	Insertion (Enqueue)	O(1)
Deletion	O(n)	Deletion (Dequeue)	O(1)

Figure Sources

- https://memegenerator.net/img/instances/48666909.jpg
- https://i.imgflip.com/34y4s4.jpg