Programming Languages and Techniques (CIS120)

Lecture 15

October 4th, 2015

Mutable Queues

Lecture notes: Chapter 16

Announcements

- No Lab Sections this week
  - Fall break!
- HW 4: Mutable Queues is available
  - Due: Tuesday, October 13^{th} at 11:59 pm
- Midterm 1 is graded.
  - Scores available on the HW submission web page
  - More details on Weds. (after make-up exams have been taken)

Reference and Equality

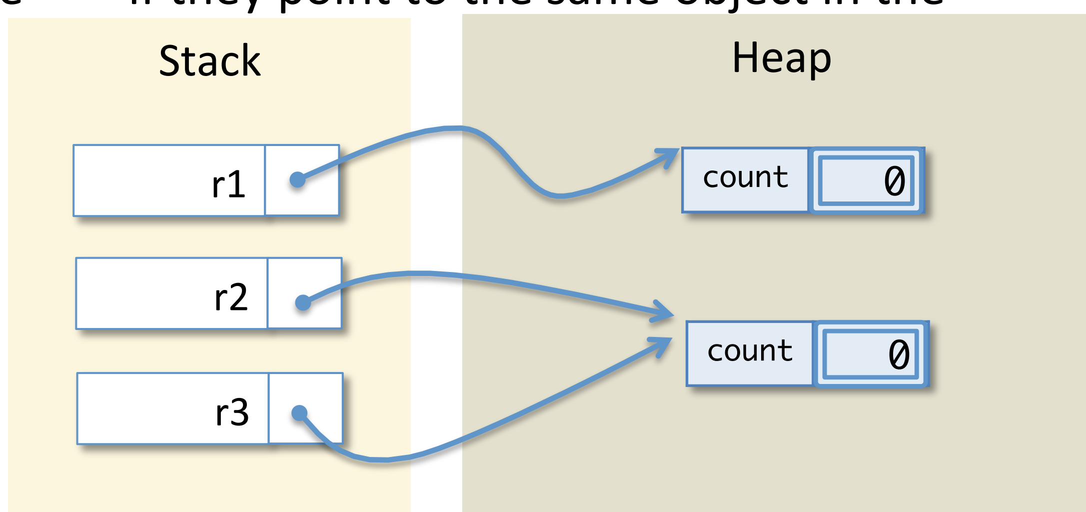
= vs. ==

Reference Equality

- Suppose we have two counters. How do we know whether they share the same internal state?
  - `type counter = { mutable count : int }`
  - We could increment one and see whether the other's value changes.
  - But we could also just test whether the references alias directly.

- Ocaml uses '==' to mean *reference* equality:
  - two reference values are '==' if they point to the same object in the heap; so:

```
r2 == r3

not (r1 == r2)

r1 = r2
```



Structural vs. Reference Equality

- *Structural (in)equality*: `v1 = v2` `v1 <> v2`
  - recursively traverses over the *structure* of the data, comparing the two values' components for structural equality
  - function values are never structurally equivalent to anything
  - structural equality can go into an infinite loop (on cyclic structures)
  - appropriate for comparing *immutable* datatypes

- *Reference (in)equality*: `v1 == v2` `v1 != v2`
  - Only looks at where the two references point in the heap
  - function values are only equal to themselves
  - equates strictly fewer things than structural equality
  - appropriate for comparing *mutable* datatypes

What is the result of evaluating the following expression?

```
let p1 : point = { x = 0; y = 0; } in
let p2 : point = p1 in

p1 = p2
```

1. `true`
2. `false`
3. runtime error
4. compile-time error

Answer: true

What is the result of evaluating the following expression?

```
let p1 : point = { x = 0; y = 0; } in
let p2 : point = p1 in

p1 == p2
```

1. `true`
2. `false`
3. runtime error
4. compile-time error

Answer: true

What is the result of evaluating the following expression?

```
let p1 : point = { x = 0; y = 0; } in
let p2 : point = { x = 0; y = 0; } in

p1 == p2
```

1. `true`
2. `false`
3. runtime error
4. compile-time error

Answer: false

What is the result of evaluating the following expression?

```
let p1 : point = { x = 0; y = 0; } in
let p2 : point = { x = 0; y = 0; } in
let l1 : point list = [p1] in
let l2 : point list = [p2] in

l1 = l2
```

1. `true`
2. `false`
3. runtime error
4. compile-time error

Answer: true

What is the result of evaluating the following expression?

```
let p1 : point = { x = 0; y = 0; } in
let p2 : point = p1 in
let l1 : point list = [p1] in
let l2 : point list = [p2] in

l1 == l2
```

1. true
2. false
3. runtime error
4. compile-time error

Answer: false

Putting State to Work

Mutable Queues

Have you ever implemented the mutable data structure called a **linked list**, in any language?

1. yes
2. no
3. not sure

A design problem

Suppose you are implementing a website to sell tickets to a very popular music event. To be fair, you would like to allow people to select seats first come, first served. How would you do it?

- Understand the problem
  - Some people may visit the website to buy tickets while others are still selecting their seats
  - Need to remember the order in which people purchase tickets
- Define the interface
  - Need a data structure to store ticket purchasers
  - Need to add purchasers to the *end* of the line
  - Need to allow purchasers at the *beginning* of the line to select seats
  - Both kinds of access must be efficient to handle the volume

(Mutable) Queue Interface

```
module type QUEUE =
sig
  (* abstract type *)
  type 'a queue

  (* Make a new, empty queue *)
  val create : unit -> 'a queue

  (* Determine if the queue is empty *)
  val is_empty : 'a queue -> bool


  (* Add a value to the end of the queue *)
  val enq : 'a -> 'a queue -> unit

  (* Remove the first value (if any) and return it *)
  val deq : 'a queue -> 'a

end
```

We can tell, just looking at this interface, that it is for a MUTABLE data structure. How?

Because queues are mutable, we must allocate a new one every time we need one.

Adding an element to the queue returns unit because it modifies the given queue.

Specify the behavior via test cases

```
let test () : bool =
  let q : int queue = create () in
  enq 1 q;
  enq 2 q;
  1 = deq q
;; run_test "queue test 1" test

let test () : bool =
  let q : int queue = create () in
  enq 1 q;
  enq 2 q;
  let _ = deq q in
  2 = deq q
;; run_test "queue test 2" test
```

What value should replace ??? so that the following test passes?

```
let test () : bool =
  let q : int queue = create () in
  enq 1 q;
  let _ = deq q in
  enq 2 q;
  ??? = deq q

;; run_test "enq after deq" test
```

1. 1
2. 2
3. None
4. failwith “empty queue”

Answer: 2

Implementing Linked Queues

Representing links

Data Structure for Mutable Queues

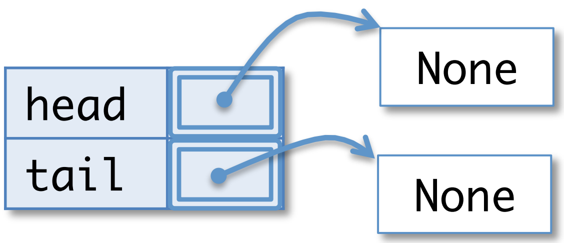
```
type 'a qnode = {
     v: 'a;
     mutable next : 'a qnode option
}

type 'a queue = { mutable head : 'a qnode option;
                  mutable tail : 'a qnode option }
```

There are two parts to a mutable queue:

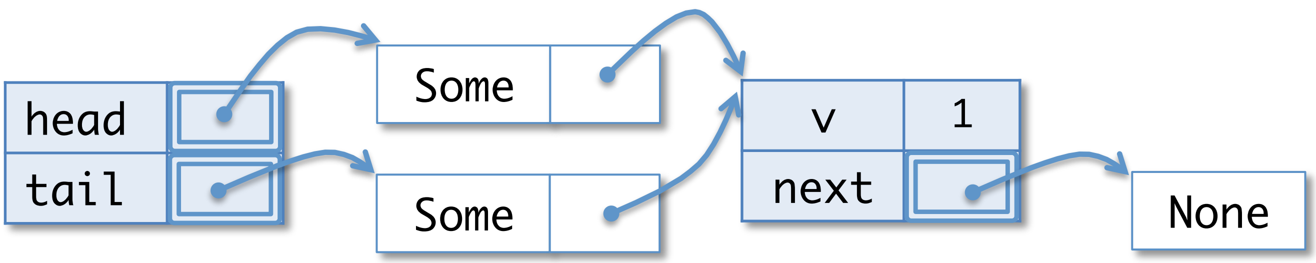
1. the “internal nodes” of the queue, with links from one to the next
2. a record with links to the head and tail nodes

All of the links are *optional* so that the queue can be empty.

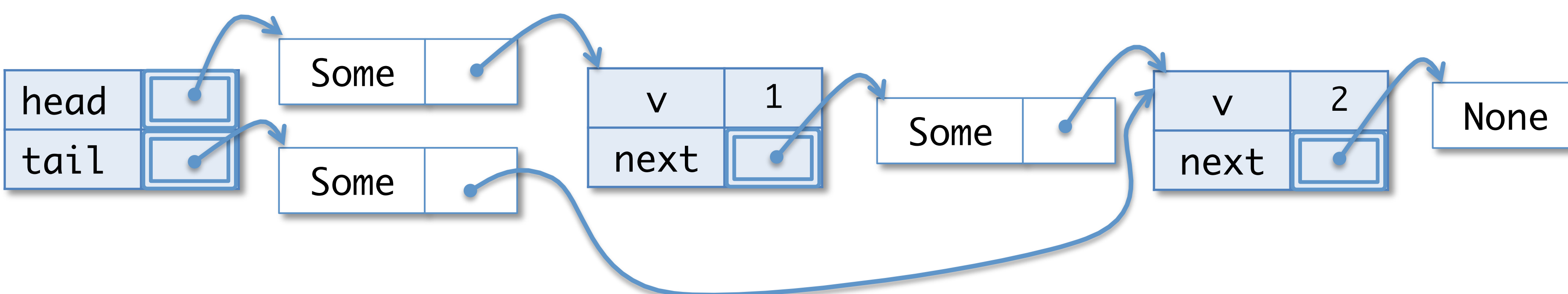
Queues in the Heap



An empty queue

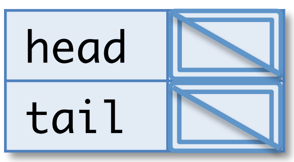


A queue with one element

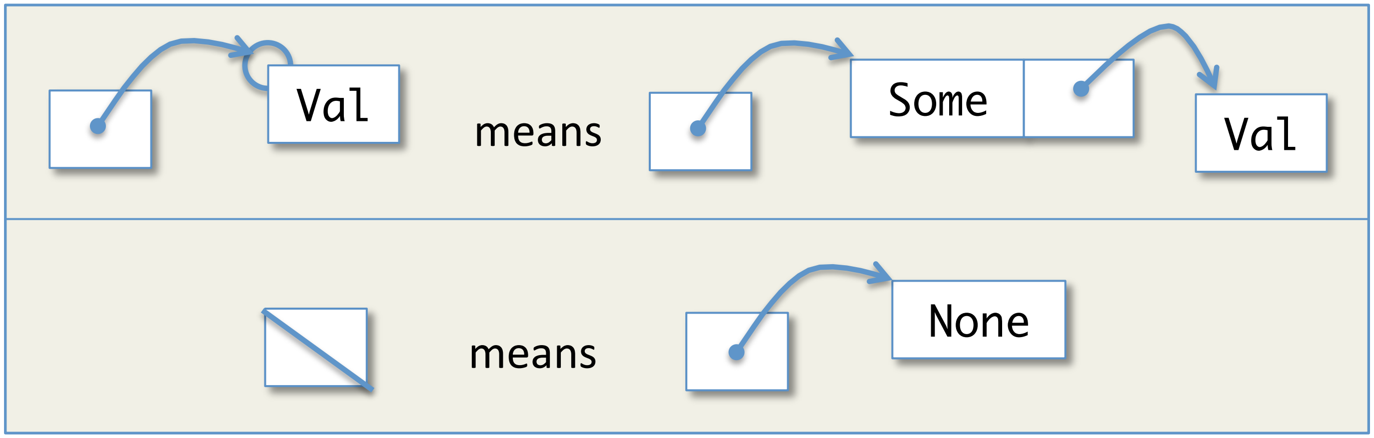


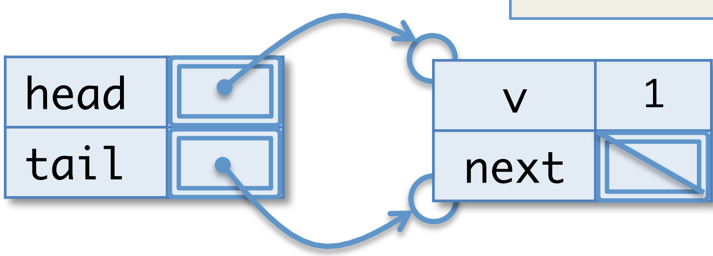
A queue with two elements

Visual Shorthand: Abbreviating Options

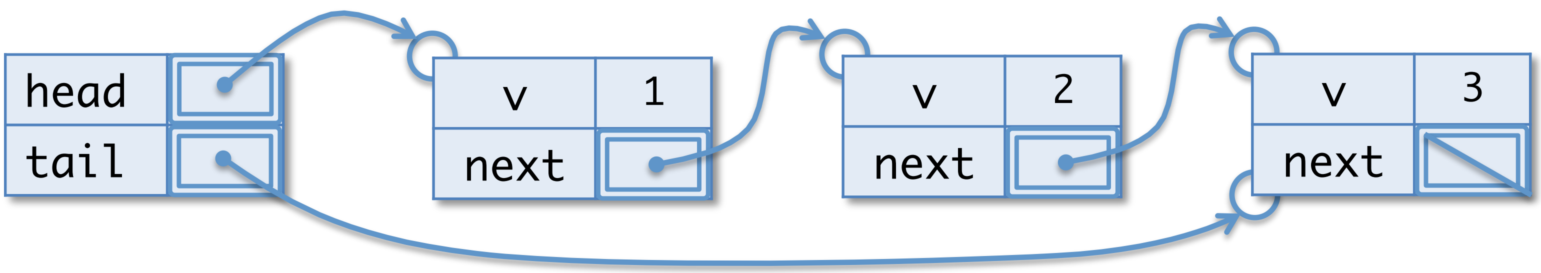


An empty queue



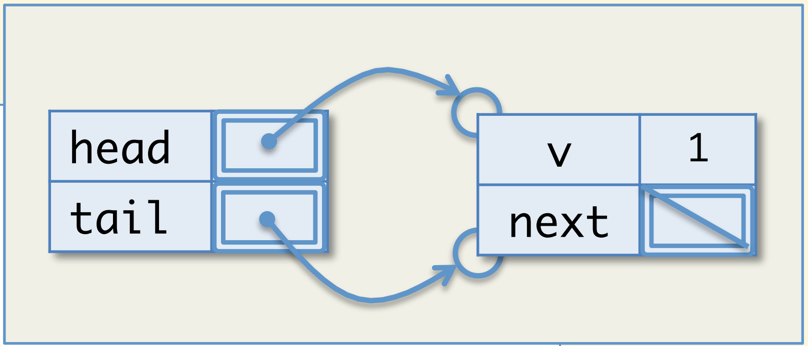


A queue with one element



A queue with three elements

Given the queue datatype shown below, which expression creates a 1-element queue in the heap:




```
type 'a qnode = {
  v: 'a;
  mutable next : 'a qnode option
}

type 'a queue = { mutable head : 'a qnode option;
                  mutable tail : 'a qnode option }
```

1.
```
let q = { head = None; tail = None }
```

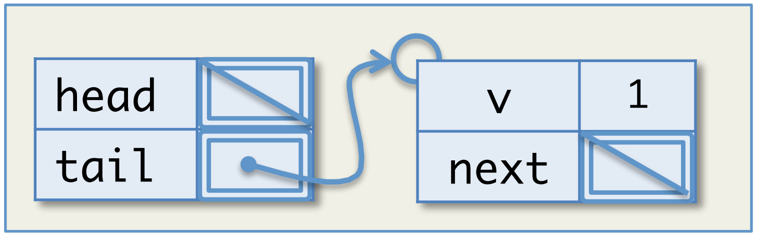
2.
```
let q = { head = 1; tail = None }
```

3.
```
let q = let qn = { v= 1; next = None } in
        { head = qn; tail = None }
```

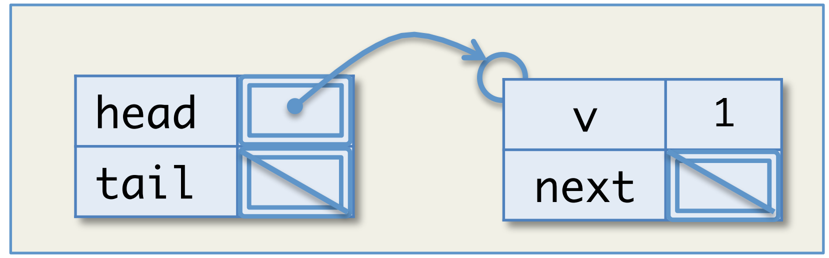
4.
```
let q = let qn = { v= 1; next = None } in
        { head = Some qn; tail = Some qn }
```

Answer: 4

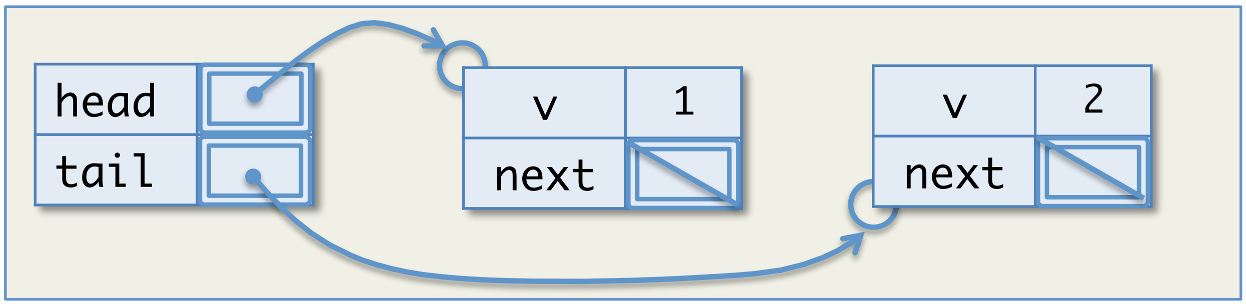
“Bogus” values of type `int queue`



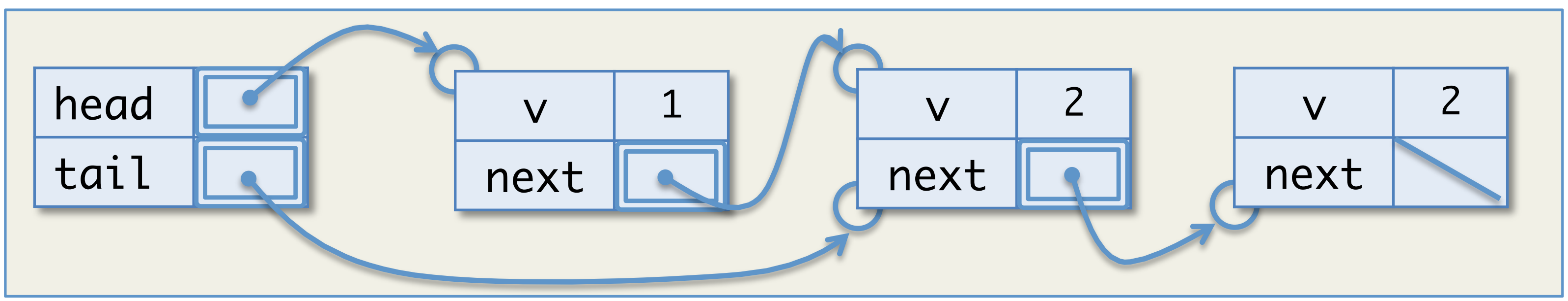
head is None, tail is Some



head is Some, tail is None



tail is not reachable from the head

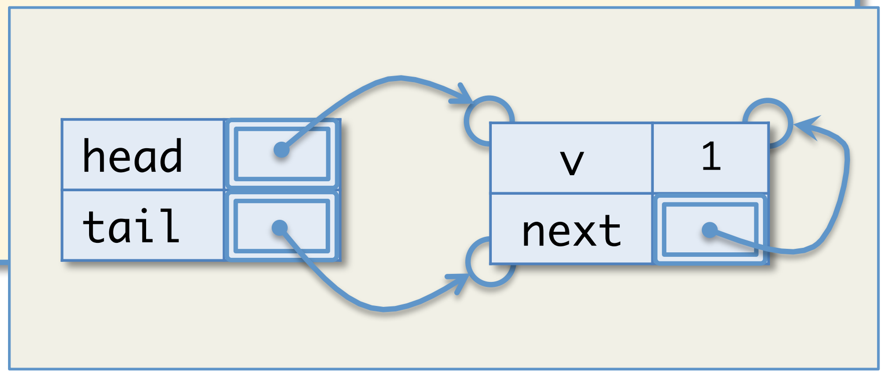


tail doesn’t point to the last element of the queue

Given the queue datatype shown below, is it possible to create a *cycle* of references in the heap. (i.e. a way to get back to the same place by following references.)

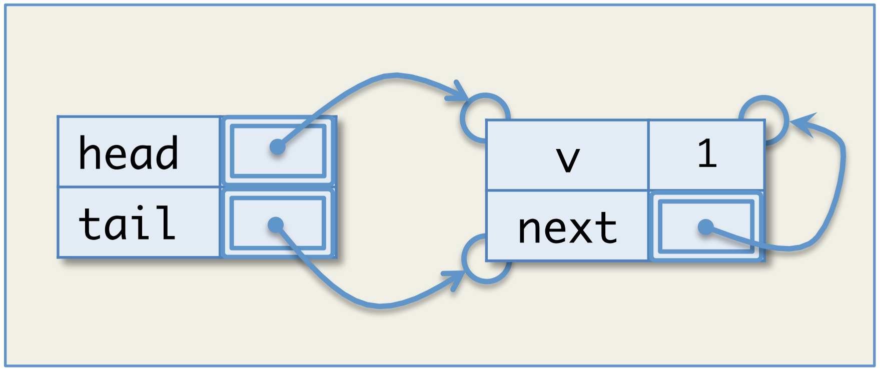
```
type 'a qnode = {
     v: 'a;
     mutable next : 'a qnode option
}

type 'a queue = { mutable head : 'a qnode option;
                  mutable tail : 'a qnode option }
```

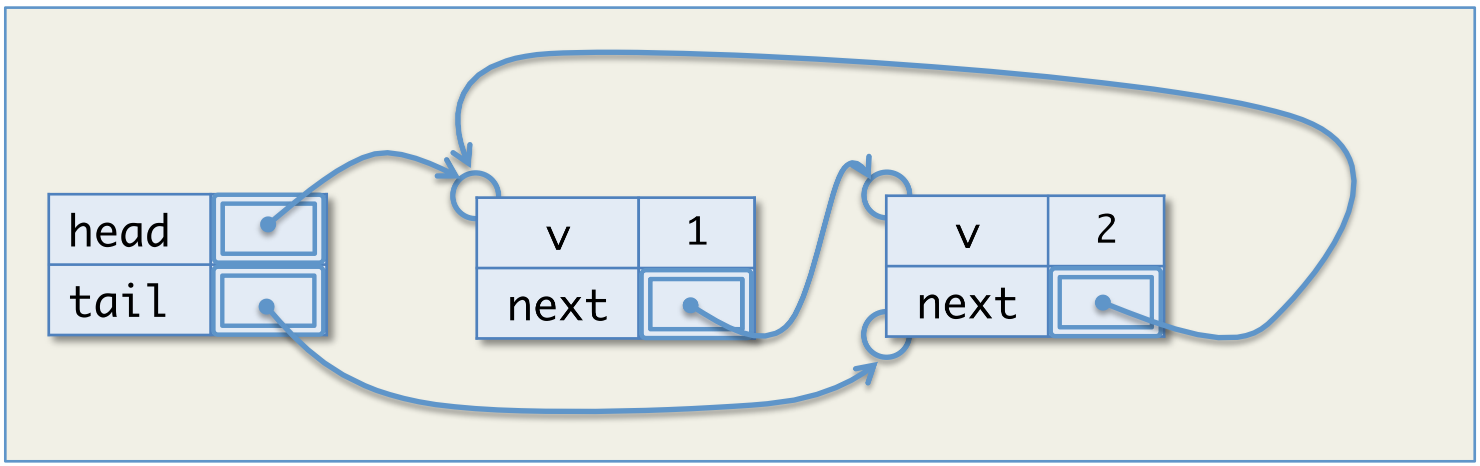
1. yes
2. no
3. not sure



Answer: 1

Cyclic queues





(And infinitely many more…)

Linked Queue Invariants

- Just as we imposed some restrictions on which trees count as legitimate Binary Search Trees, Linked Queues must also satisfy representation *invariants*:

> Either:
> (1) `head` and `tail` are both `None` (i.e. the queue is empty)
> or
> (2) `head` is `Some n1`, `tail` is `Some n2` and
> - `n2` is reachable from `n1` by following 'next' pointers
> - `n2.next` is `None`

- We can check that these properties rule out all of the "bogus" examples.
- Each queue operation may assume that these invariants hold of its inputs, and must ensure that the invariants hold when it's done.

Either:

(1) `head` and `tail` are both None (i.e. the queue is empty)

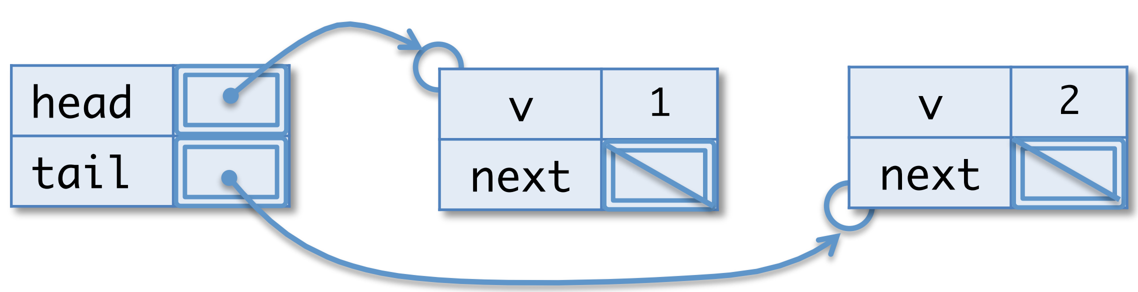
or

(2) `head` is `Some n1`, `tail` is `Some n2` and

- `n2` is reachable from `n1` by following 'next' pointers
- `n2.next` is None

Is this a valid queue?

1. Yes
2. No



Either:

(1) `head` and `tail` are both None (i.e. the queue is empty)

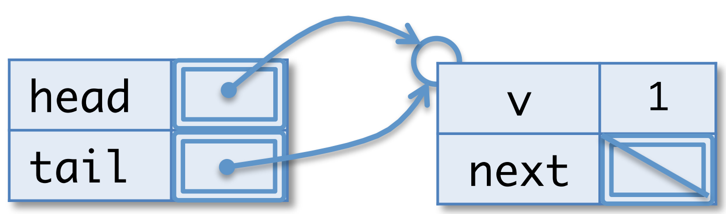
or

(2) `head` is `Some n1`, `tail` is `Some n2` and

- `n2` is reachable from `n1` by following ‘next’ pointers
- `n2.next` is None

Is this a valid queue?

1. Yes
2. No



Implementing Linked Queues

LinkedQ.ml

create and is_empty

```
(* create an empty queue *)
let create () : 'a queue =
    { head = None;
      tail = None }


(* determine whether a queue is empty *)
let is_empty (q:'a queue) : bool =
    q.head = None
```

- `create` *establishes* the queue invariants
  - both head and tail are None
- `is_empty` *assumes* the queue invariants
  - it doesn't have to check that q.tail is None

enq

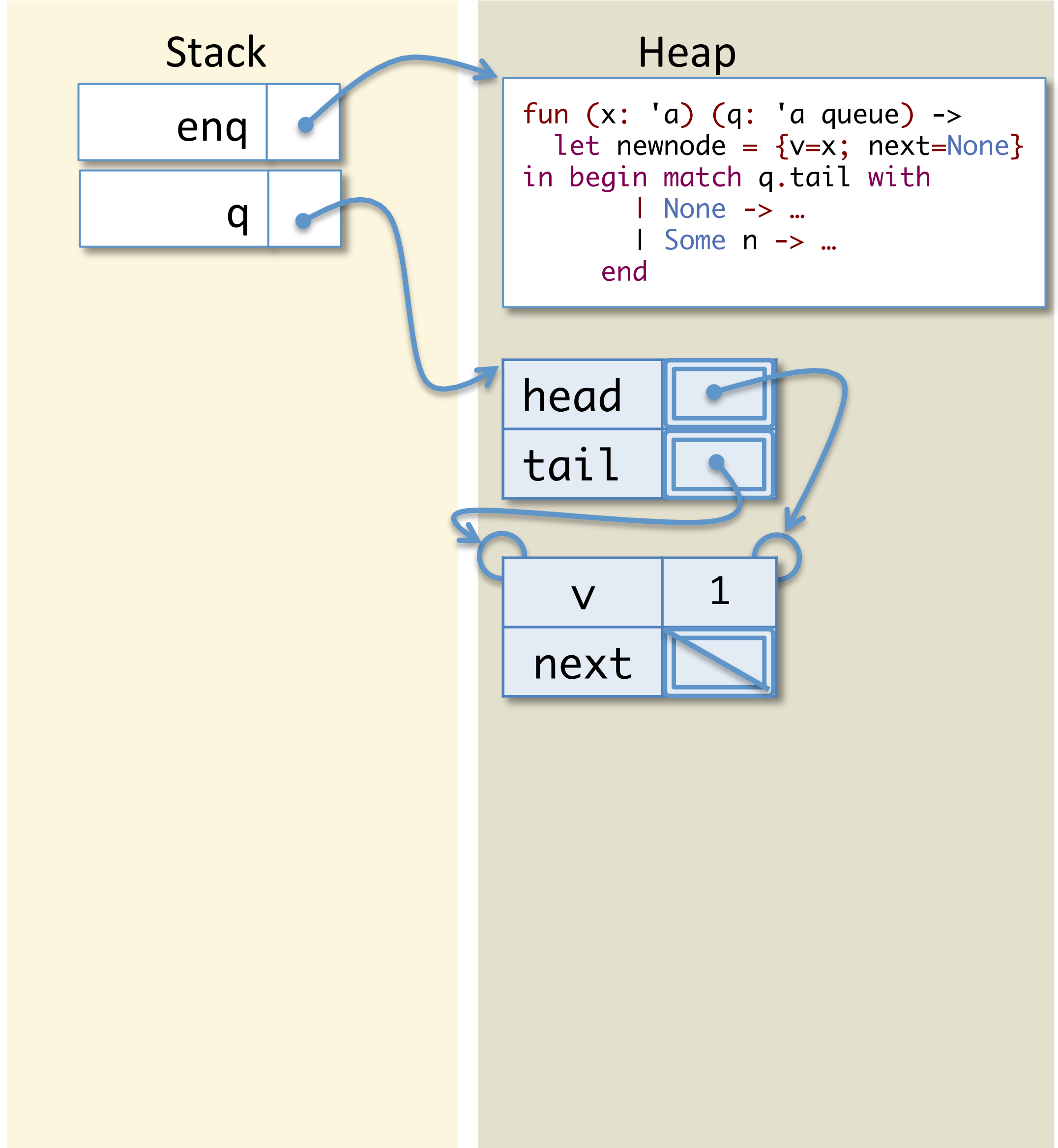
```
(* add an element to the tail of a queue *)
let enq (x: 'a) (q: 'a queue) : unit =
   let newnode = {v=x; next=None} in
   begin match q.tail with
     | None ->
        q.head <- Some newnode;
        q.tail <- Some newnode
     | Some n ->
        n.next <- Some newnode;
        q.tail <- Some newnode
   end
```

- The code for enq is informed by the queue invariant:
  - either the queue is empty, and we just update head and tail, or
  - the queue is non-empty, in which case we have to “patch up” the “next” link of the old tail node to maintain the queue invariant.

Calling Enq on a non-empty queue

Workspace

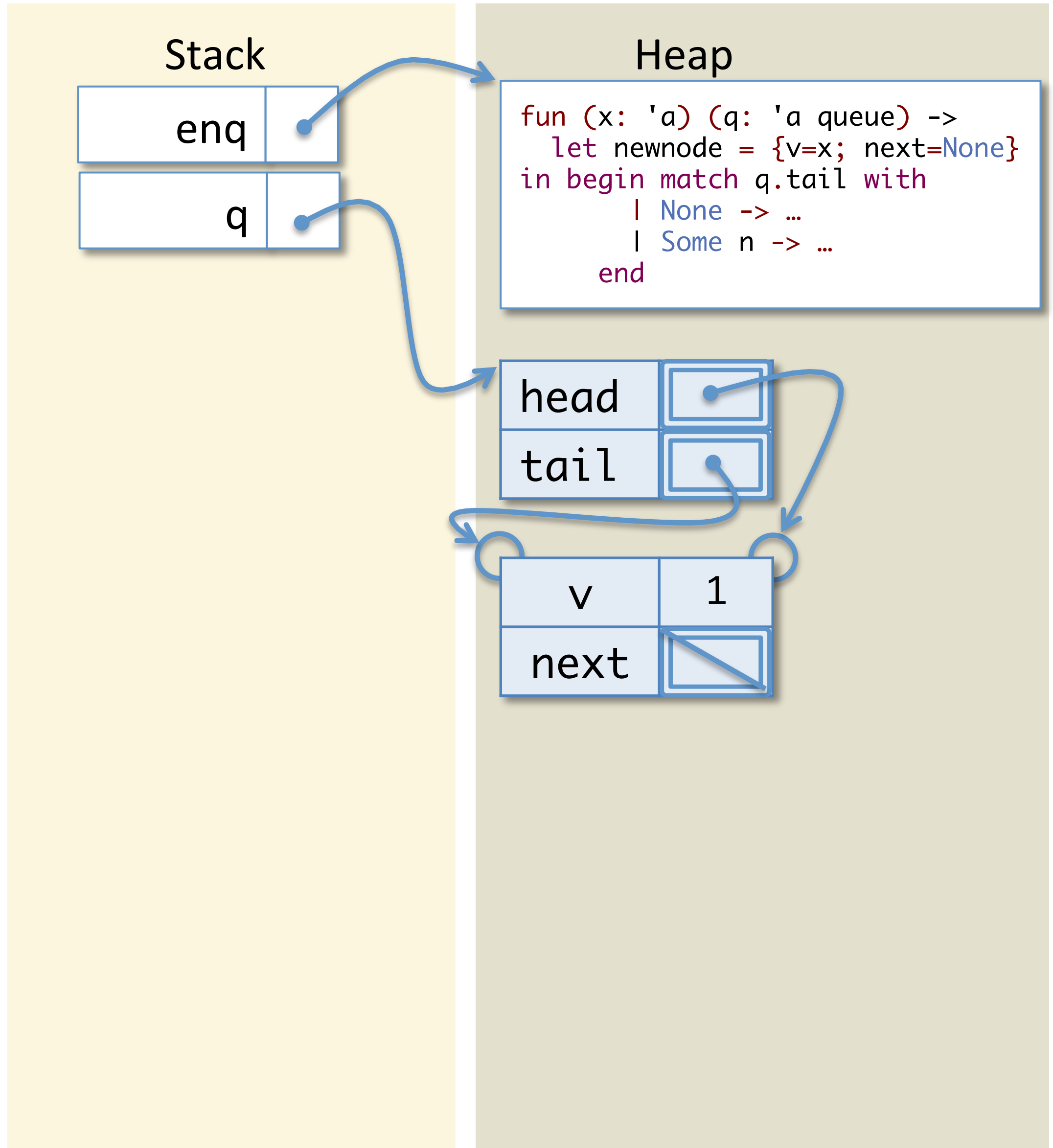
```
enq 2 q
```



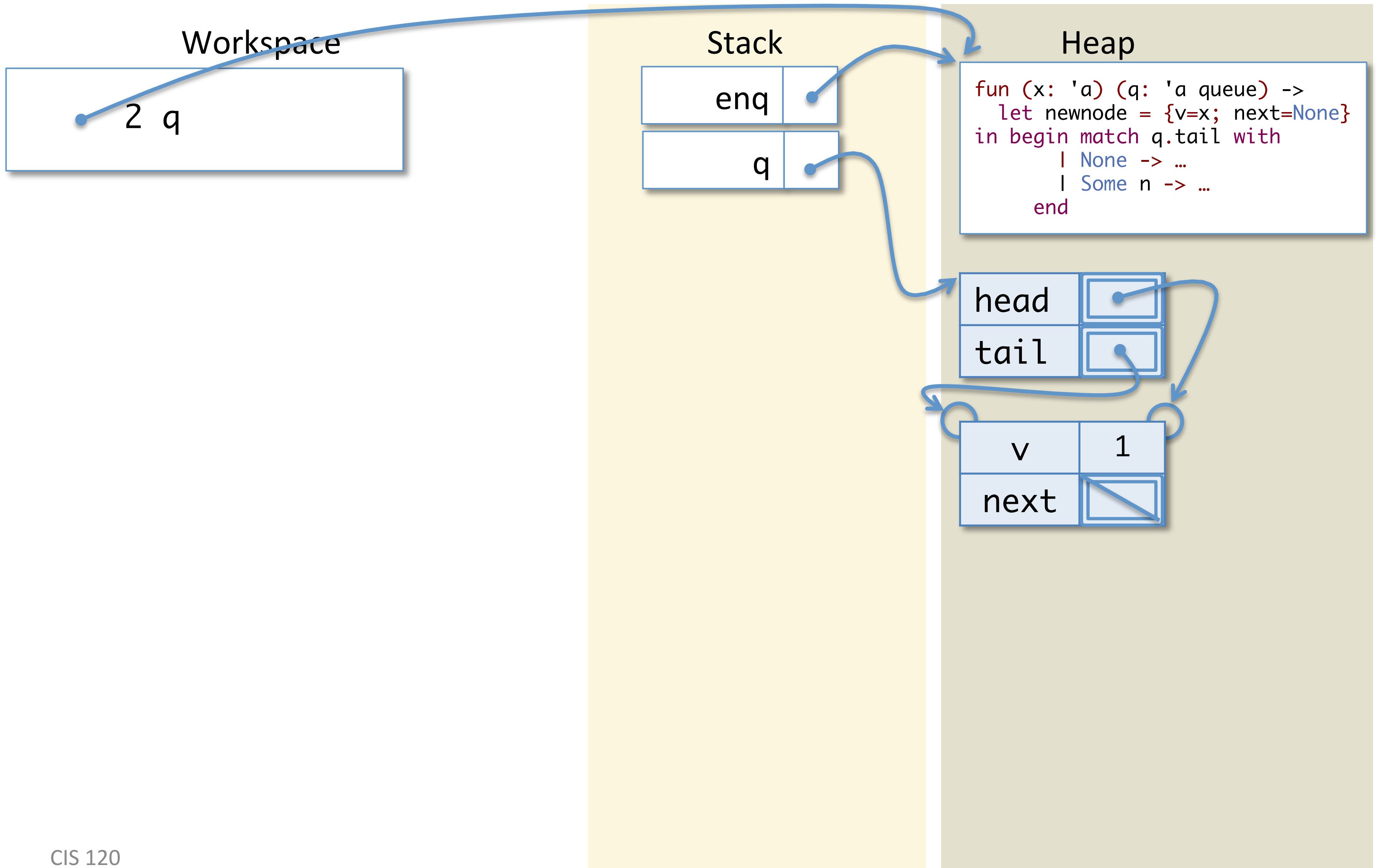
Calling Enq on a non-empty queue

Workspace

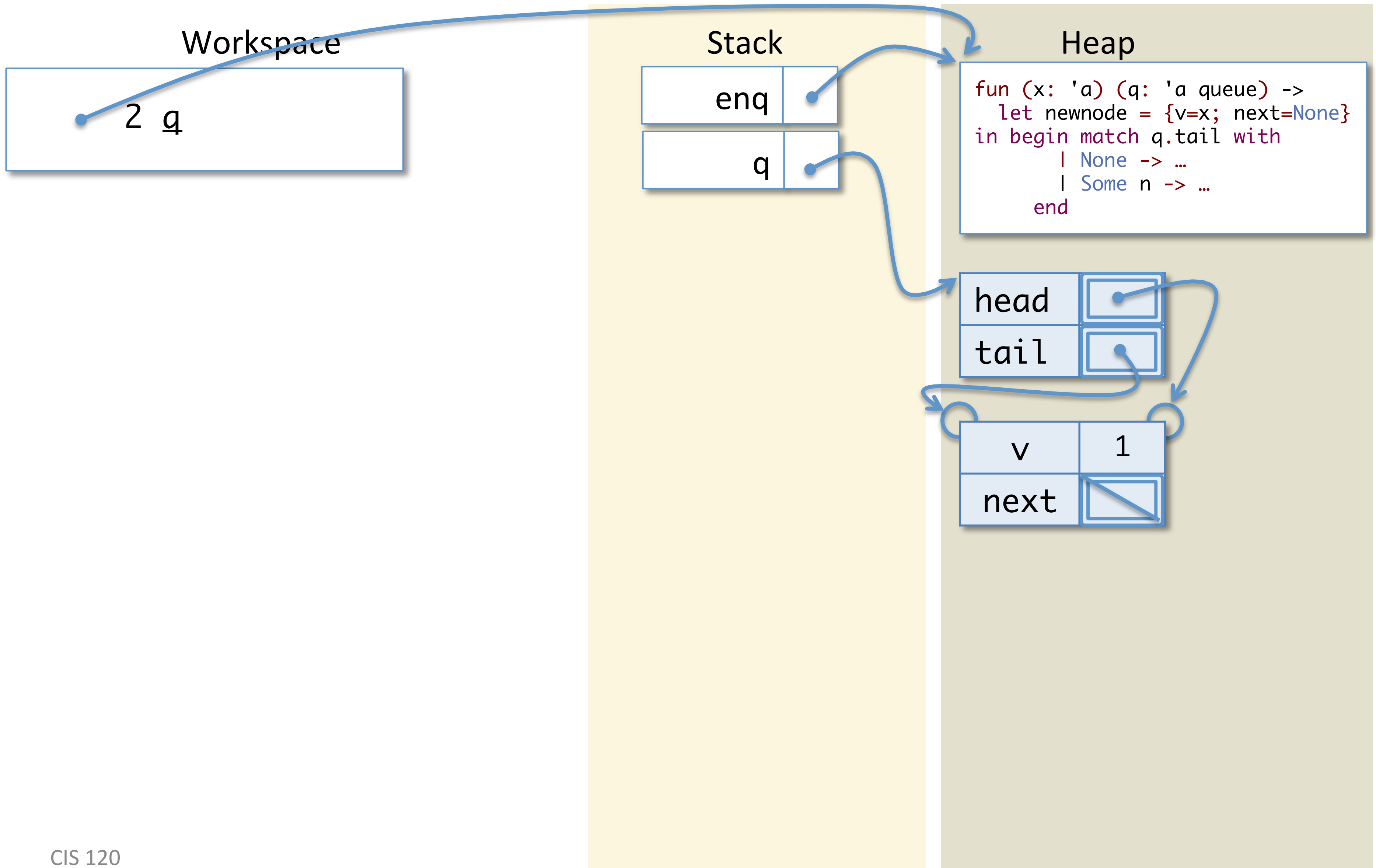
`enq 2 q`



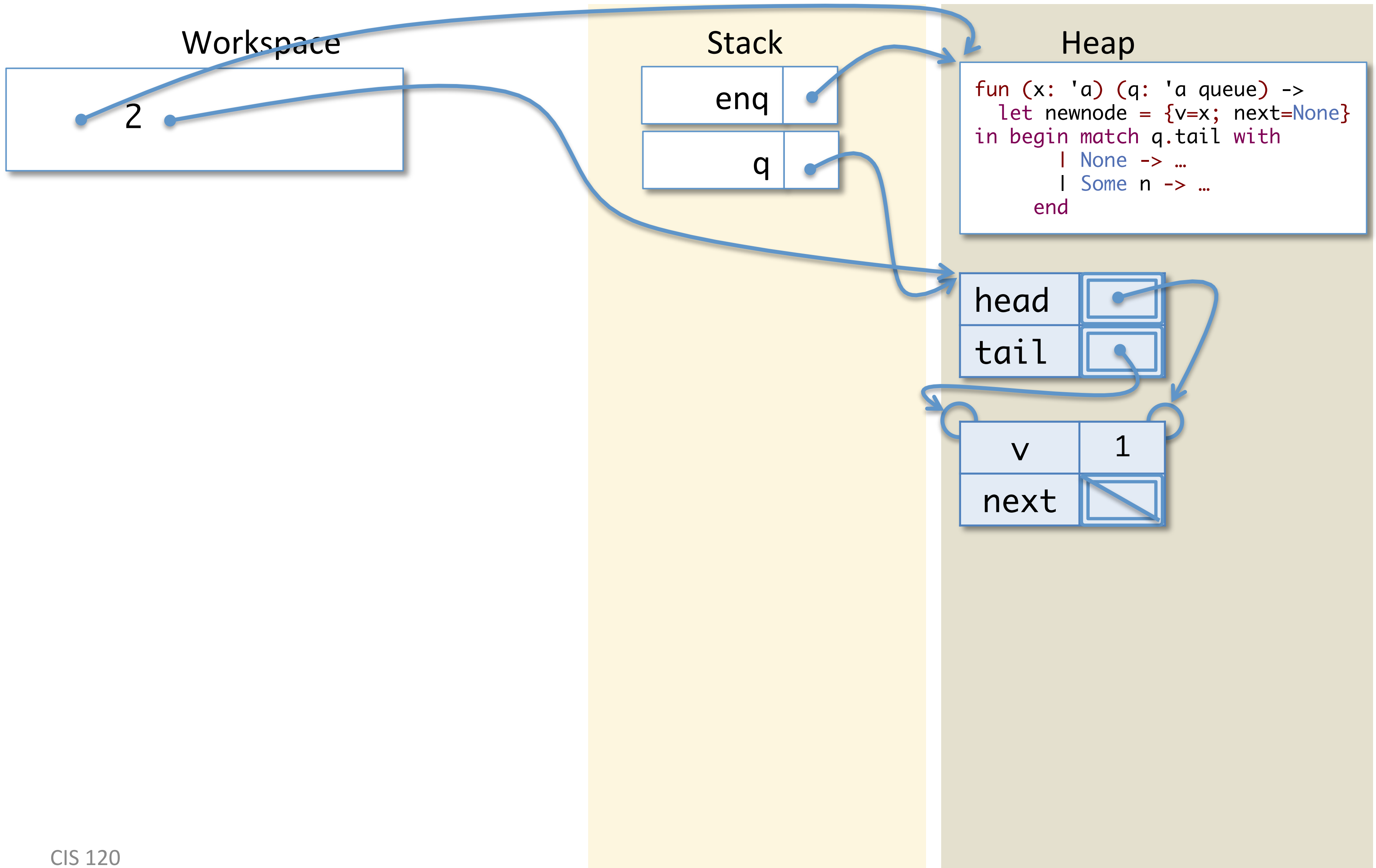
Calling Enq on a non-empty queue



Calling Enq on a non-empty queue



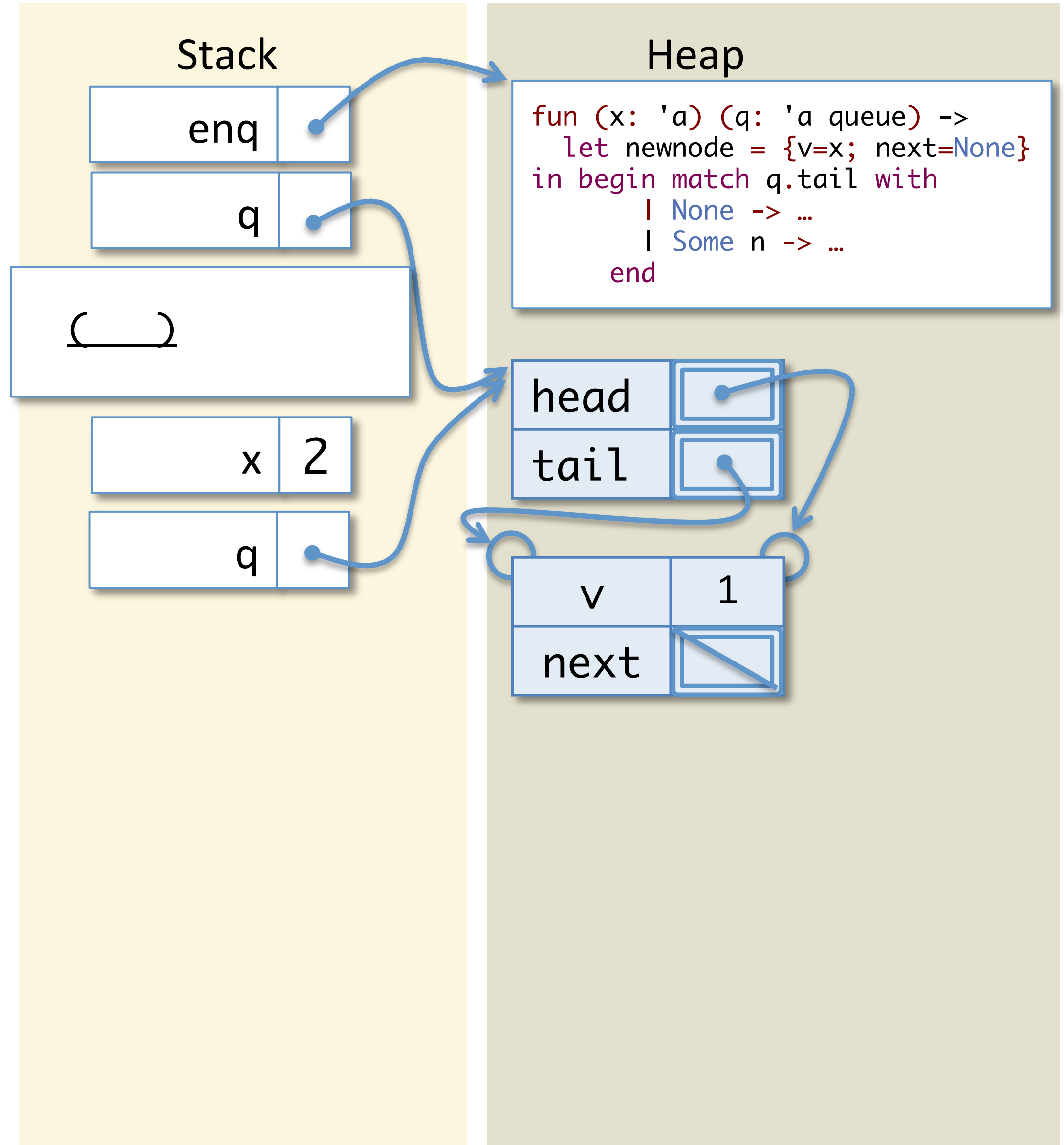
Calling Enq on a non-empty queue



Calling Enq on a non-empty queue

Workspace

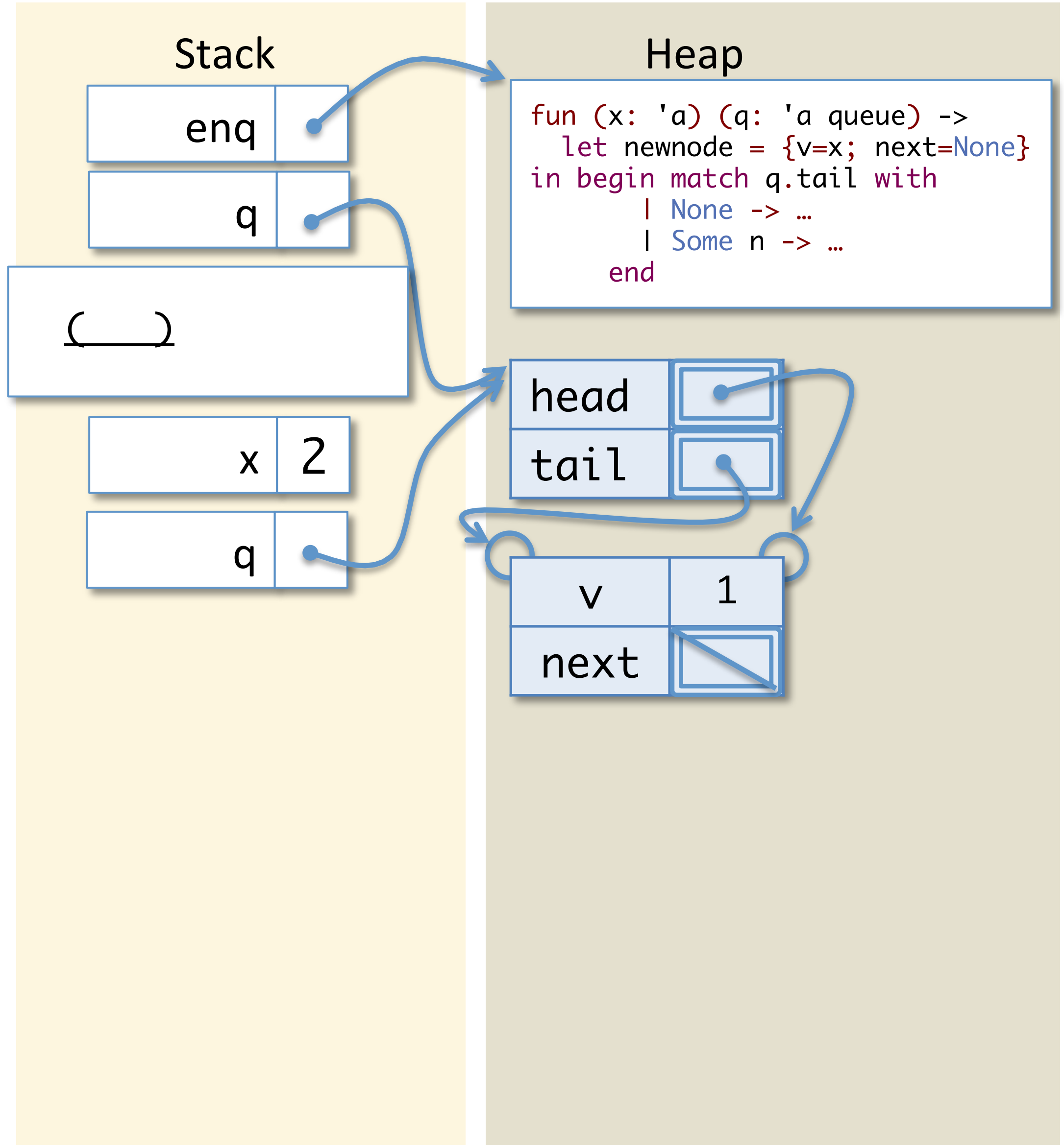
```
let newnode = {v=x; next=None} in
  begin match q.tail with
    | None ->
      q.head <- Some newnode;
      q.tail <- Some newnode
    | Some n ->
      n.next <- Some newnode;
      q.tail <- Some newnode
  end
```



Calling Enq on a non-empty queue

Workspace

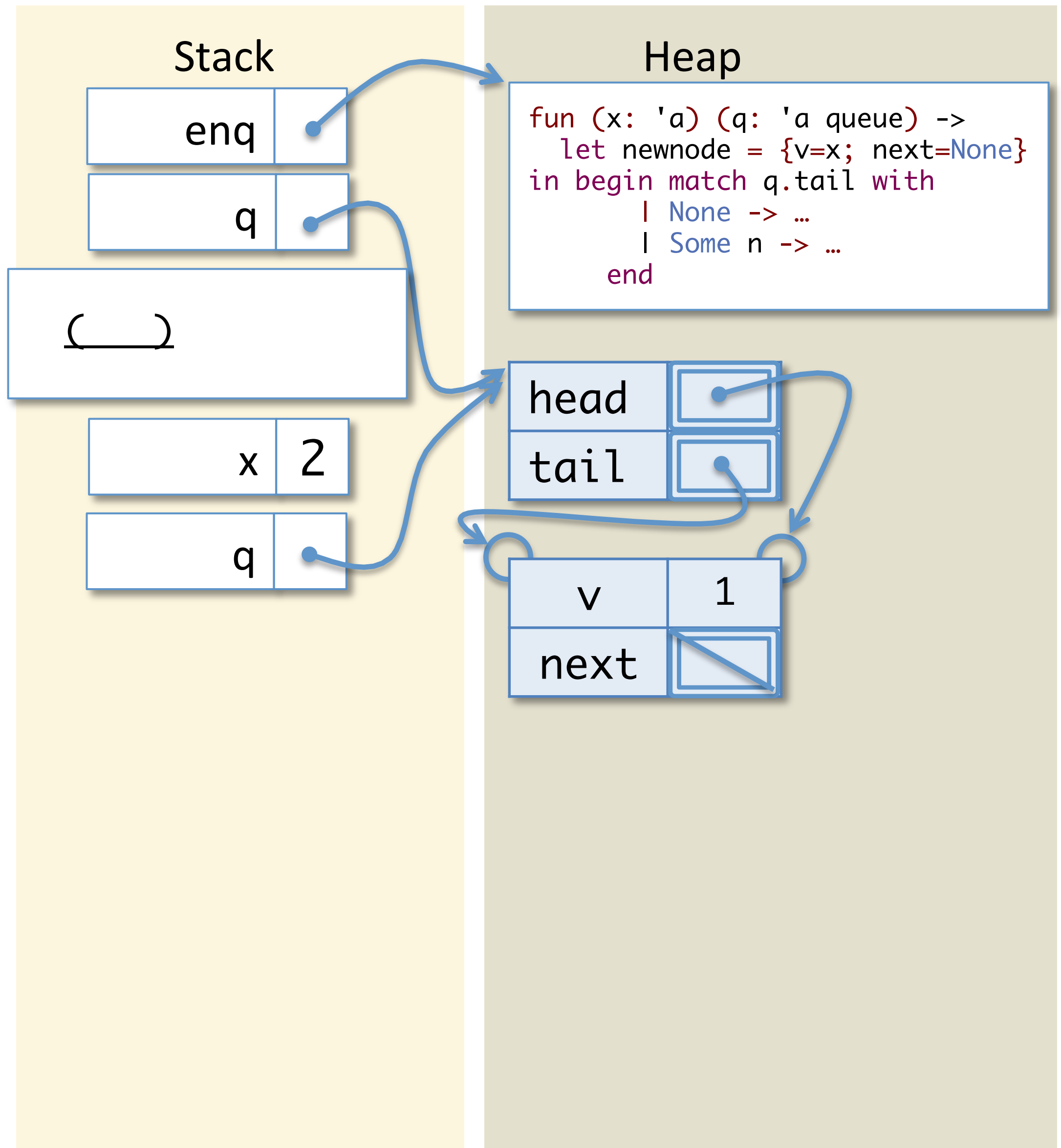
```
let newnode = {v=x; next=None} in
  begin match q.tail with
    | None ->
      q.head <- Some newnode;
      q.tail <- Some newnode
    | Some n ->
      n.next <- Some newnode;
      q.tail <- Some newnode
  end
```



Calling Enq on a non-empty queue

Workspace

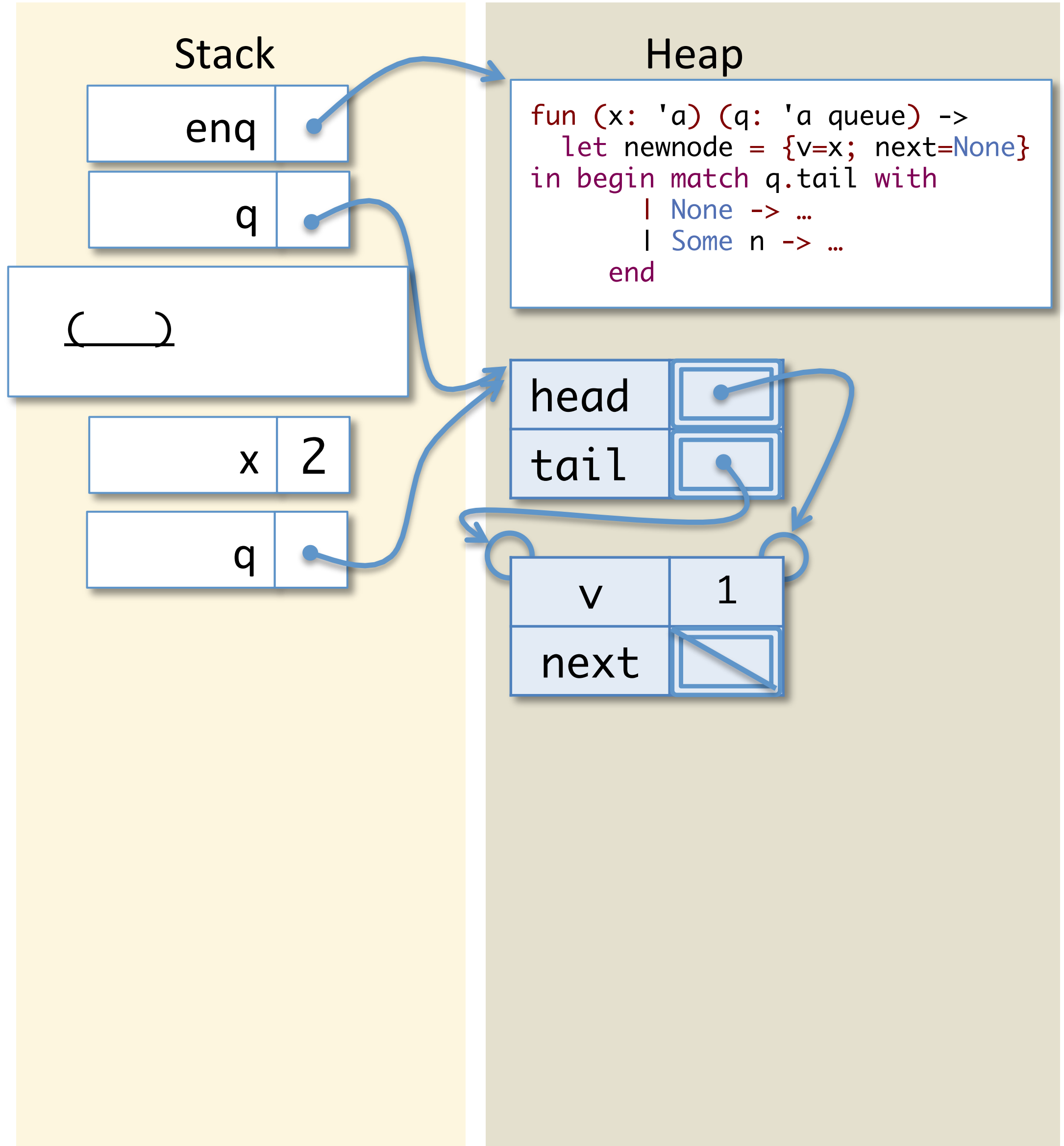
```
let newnode = {v=2; next=None} in
  begin match q.tail with
    | None ->
      q.head <- Some newnode;
      q.tail <- Some newnode
    | Some n ->
      n.next <- Some newnode;
      q.tail <- Some newnode
  end
```



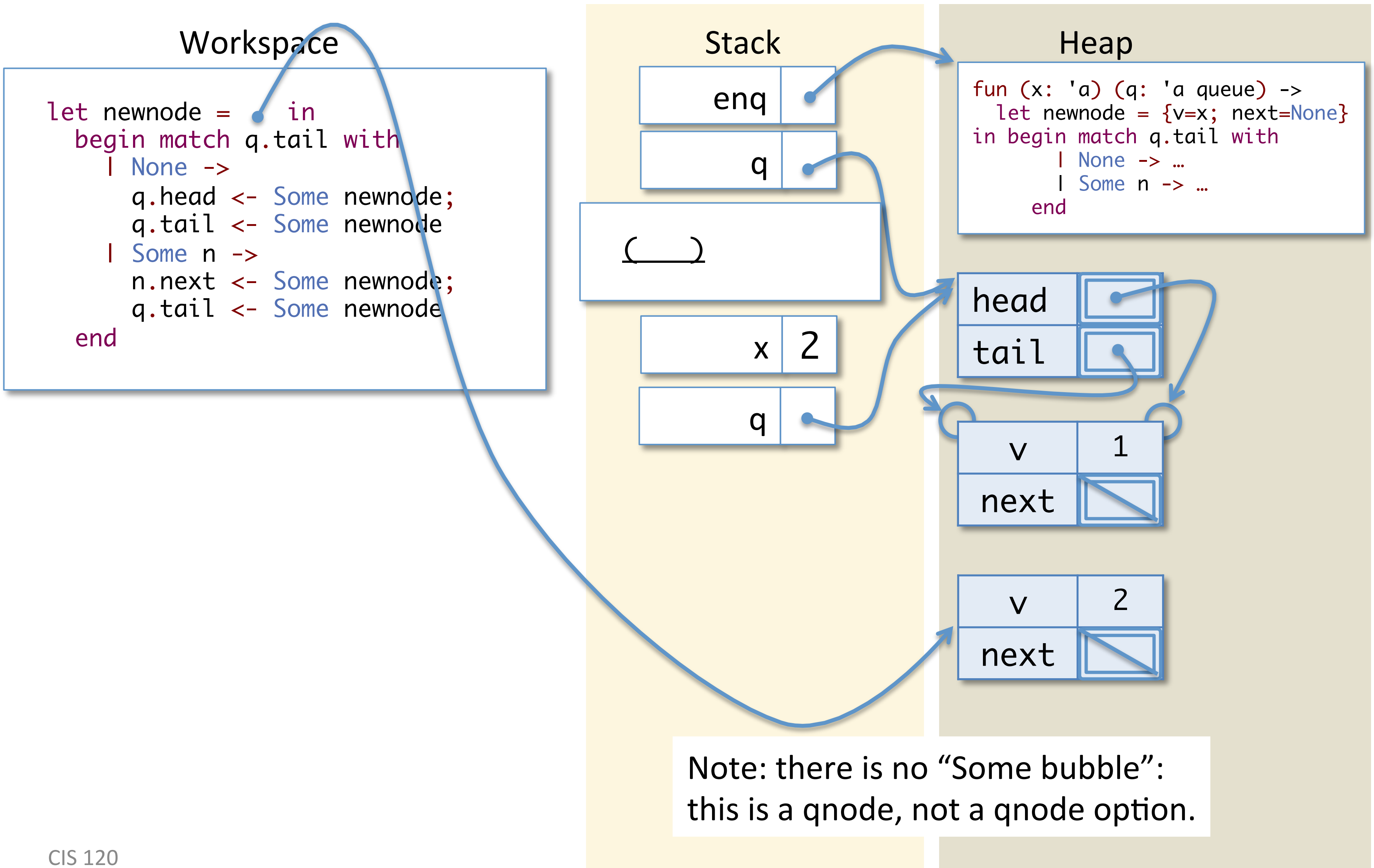
Calling Enq on a non-empty queue

Workspace

```
let newnode = {v=2; next=None} in
  begin match q.tail with
   | None ->
     q.head <- Some newnode;
     q.tail <- Some newnode
   | Some n ->
     n.next <- Some newnode;
     q.tail <- Some newnode
  end
```



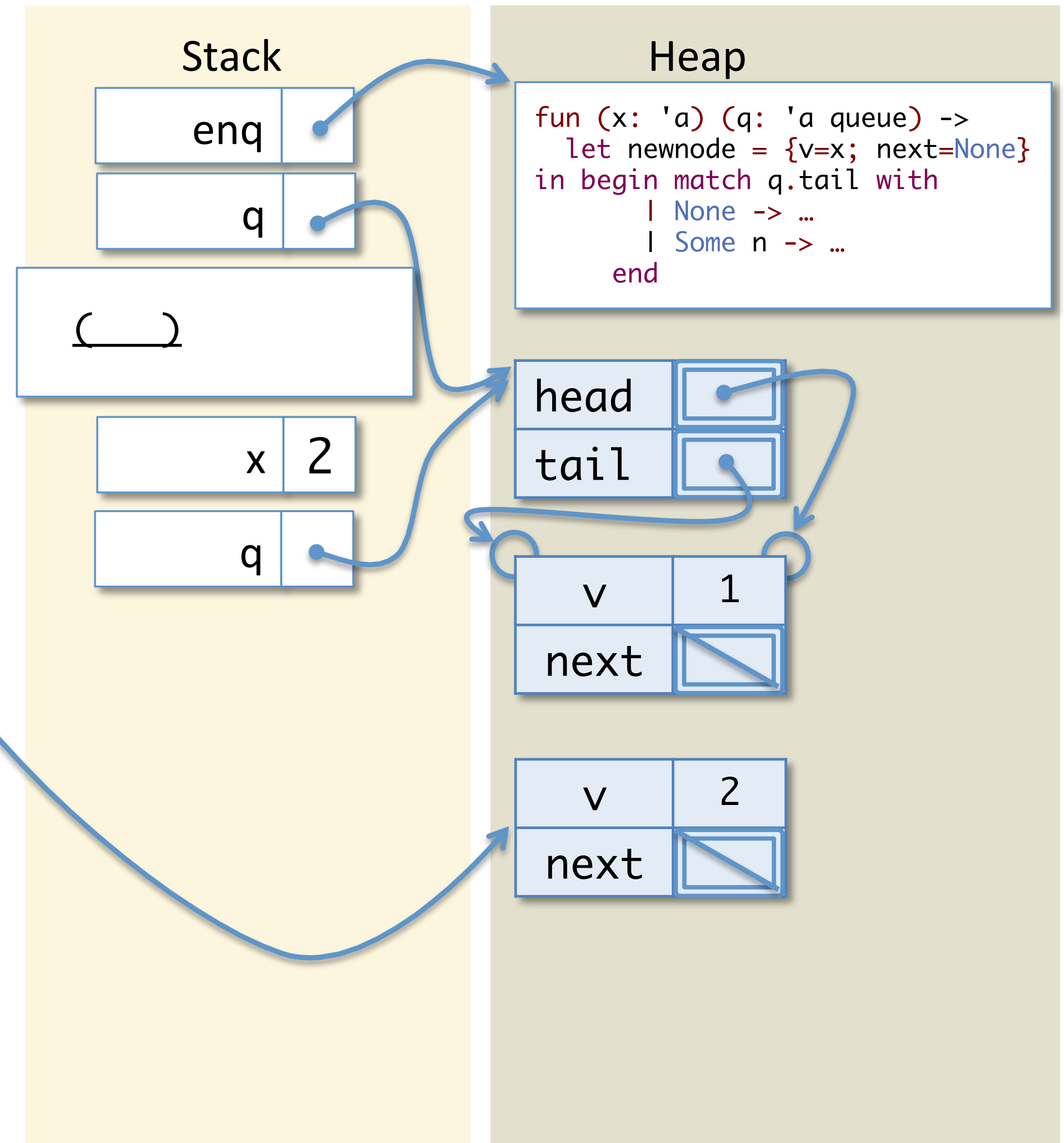
Calling Enq on a non-empty queue



Calling Enq on a non-empty queue

Workspace

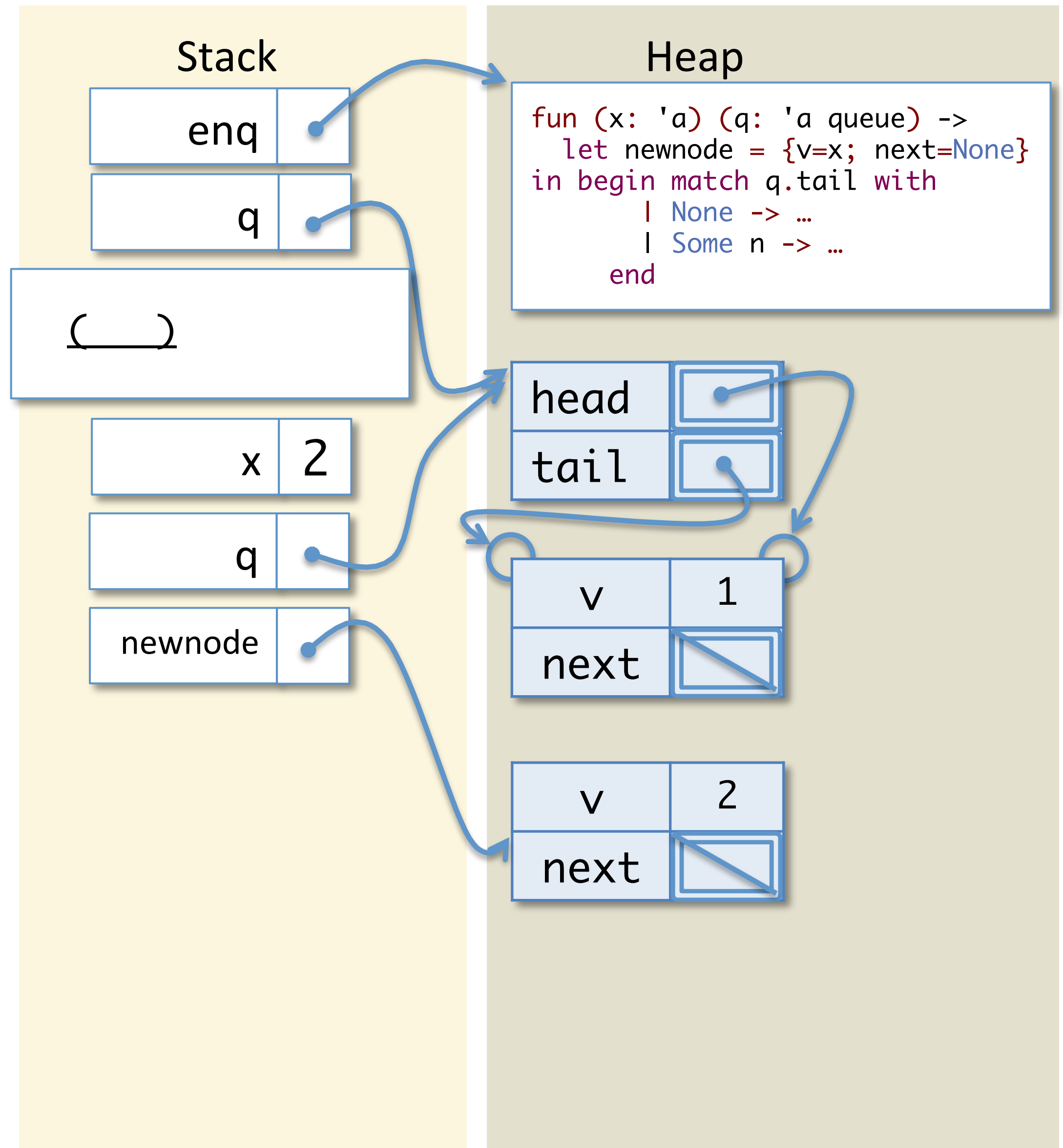
```
let newnode = ___ in
  begin match q.tail with
    | None ->
      q.head <- Some newnode;
      q.tail <- Some newnode
    | Some n ->
      n.next <- Some newnode;
      q.tail <- Some newnode
  end
```



Calling Enq on a non-empty queue

Workspace

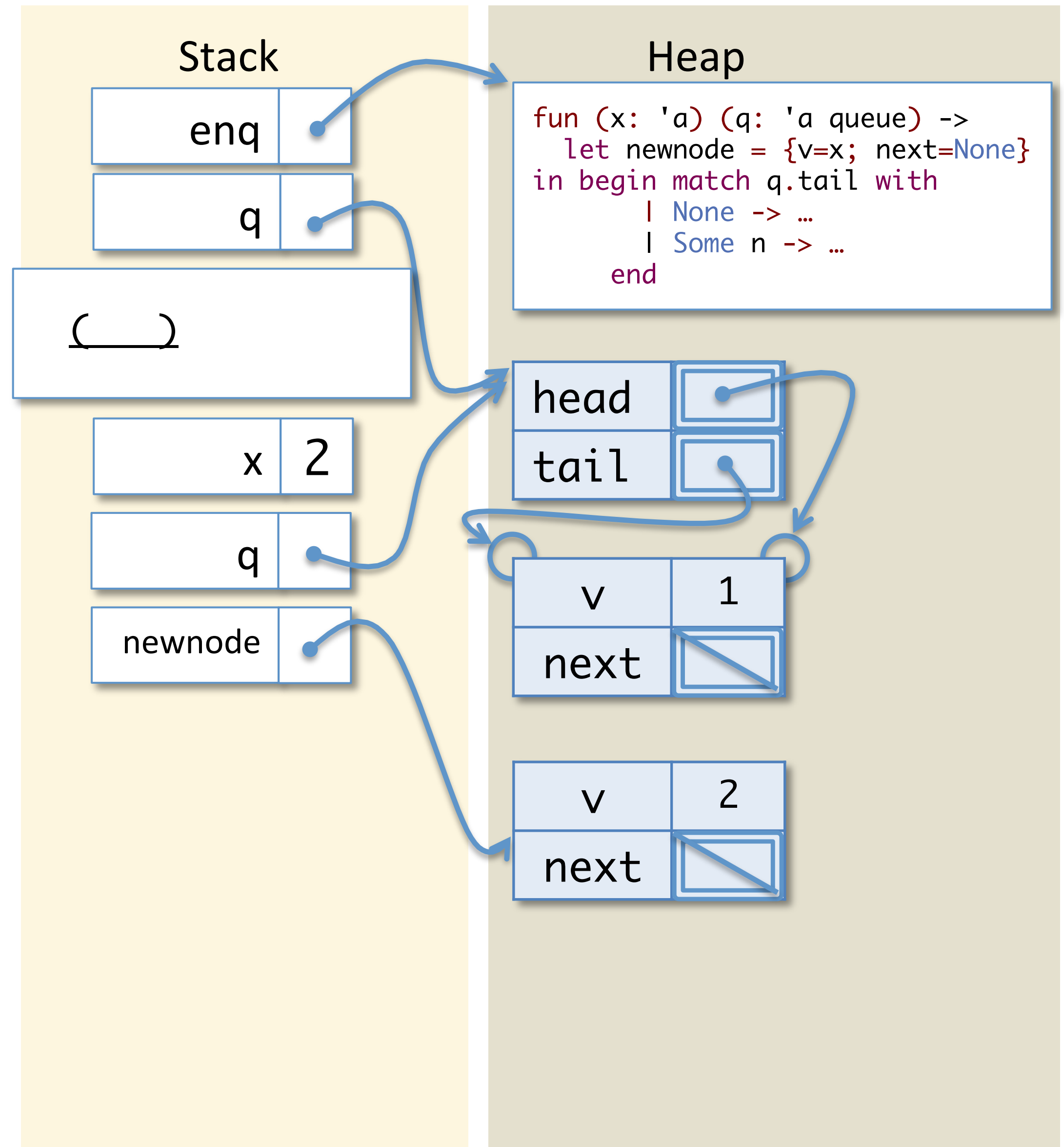
```
begin match q.tail with
  | None ->
    q.head <- Some newnode;
    q.tail <- Some newnode
  | Some n ->
    n.next <- Some newnode;
    q.tail <- Some newnode
end
```



Calling Enq on a non-empty queue

Workspace

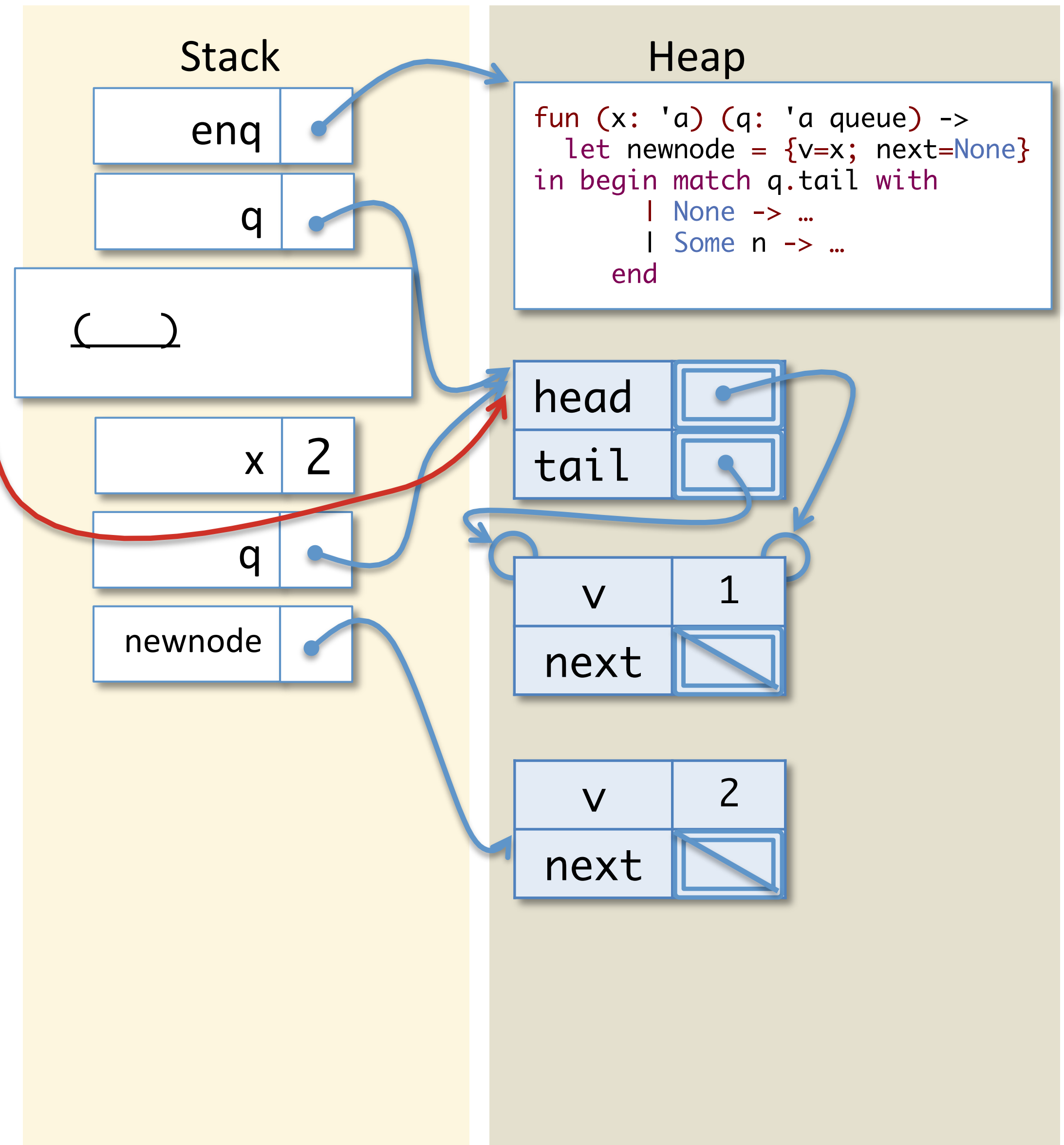
```
begin match q.tail with
  | None ->
    q.head <- Some newnode;
    q.tail <- Some newnode
  | Some n ->
    n.next <- Some newnode;
    q.tail <- Some newnode
end
```



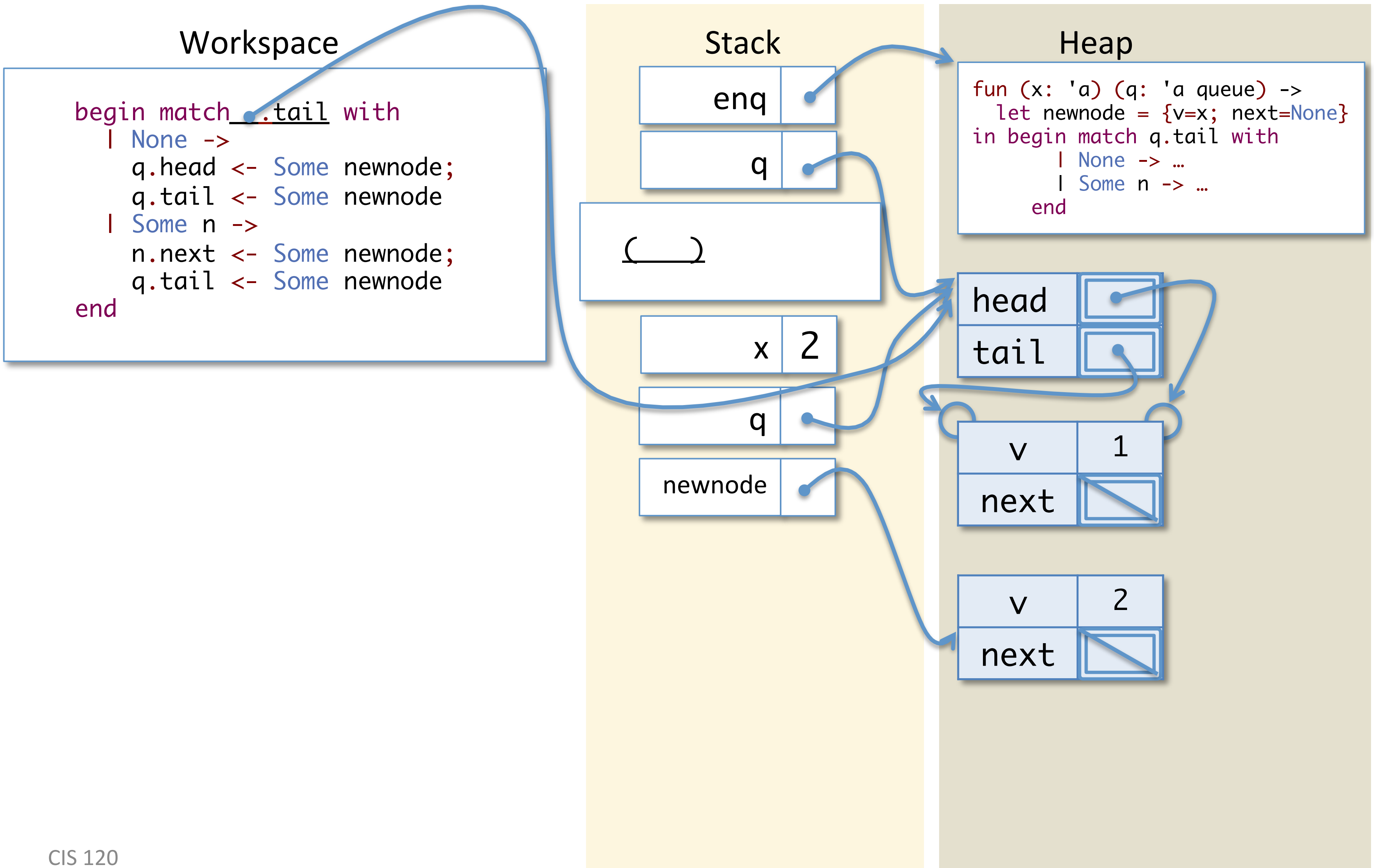
Calling Enq on a non-empty queue

Workspace

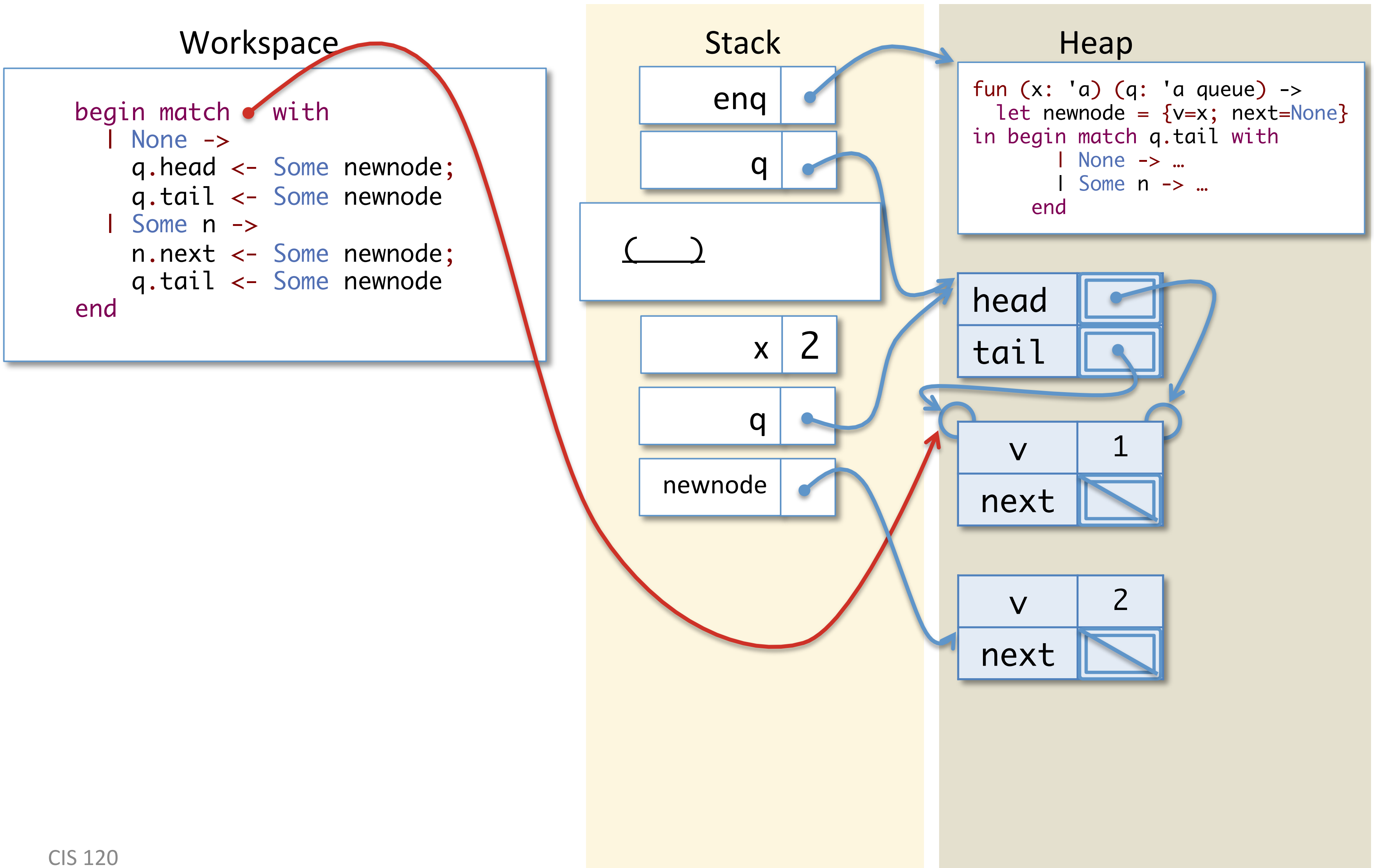
```
begin match  .tail with
  | None ->
    q.head <- Some newnode;
    q.tail <- Some newnode
  | Some n ->
    n.next <- Some newnode;
    q.tail <- Some newnode
end
```



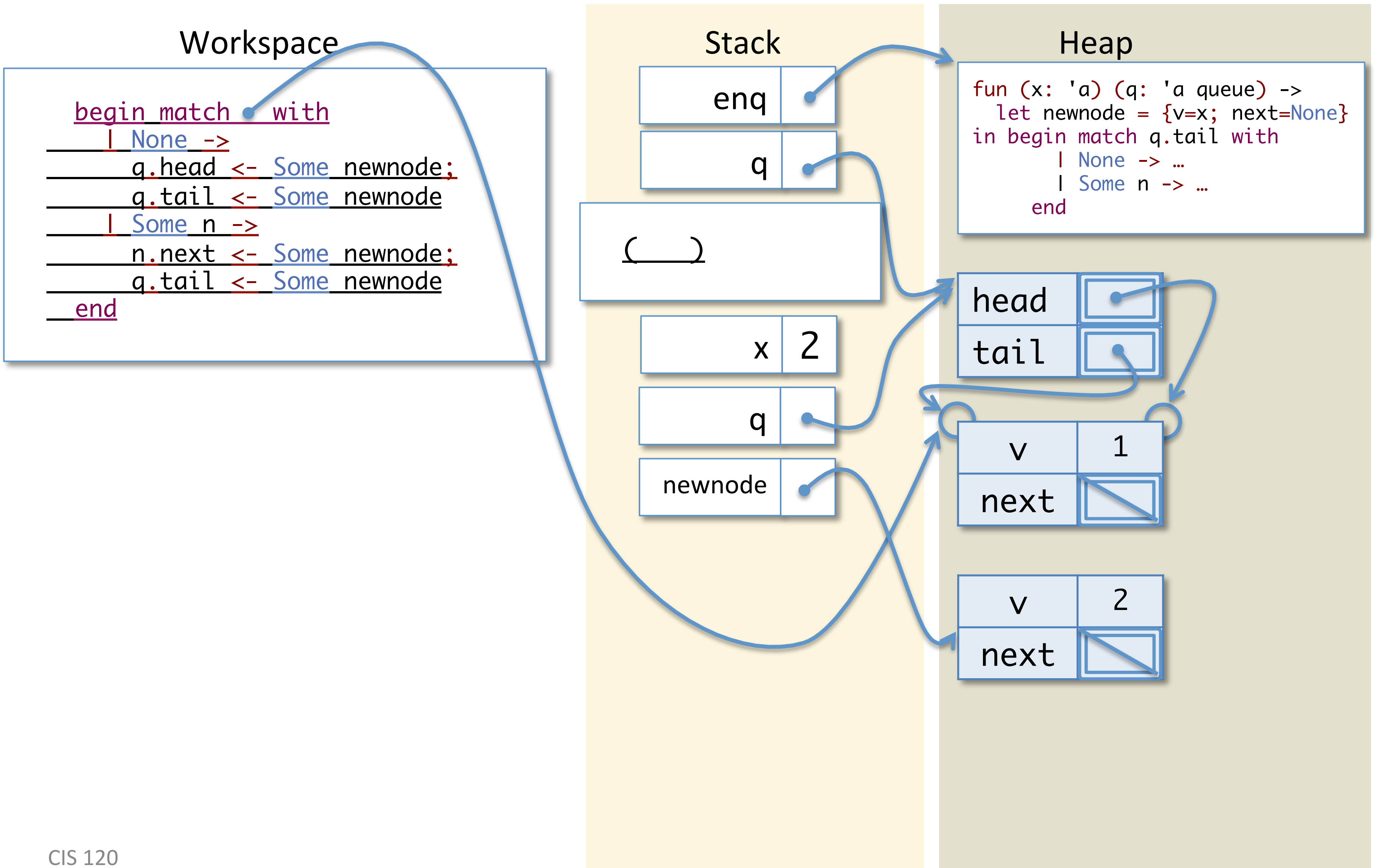
Calling Enq on a non-empty queue



Calling Enq on a non-empty queue



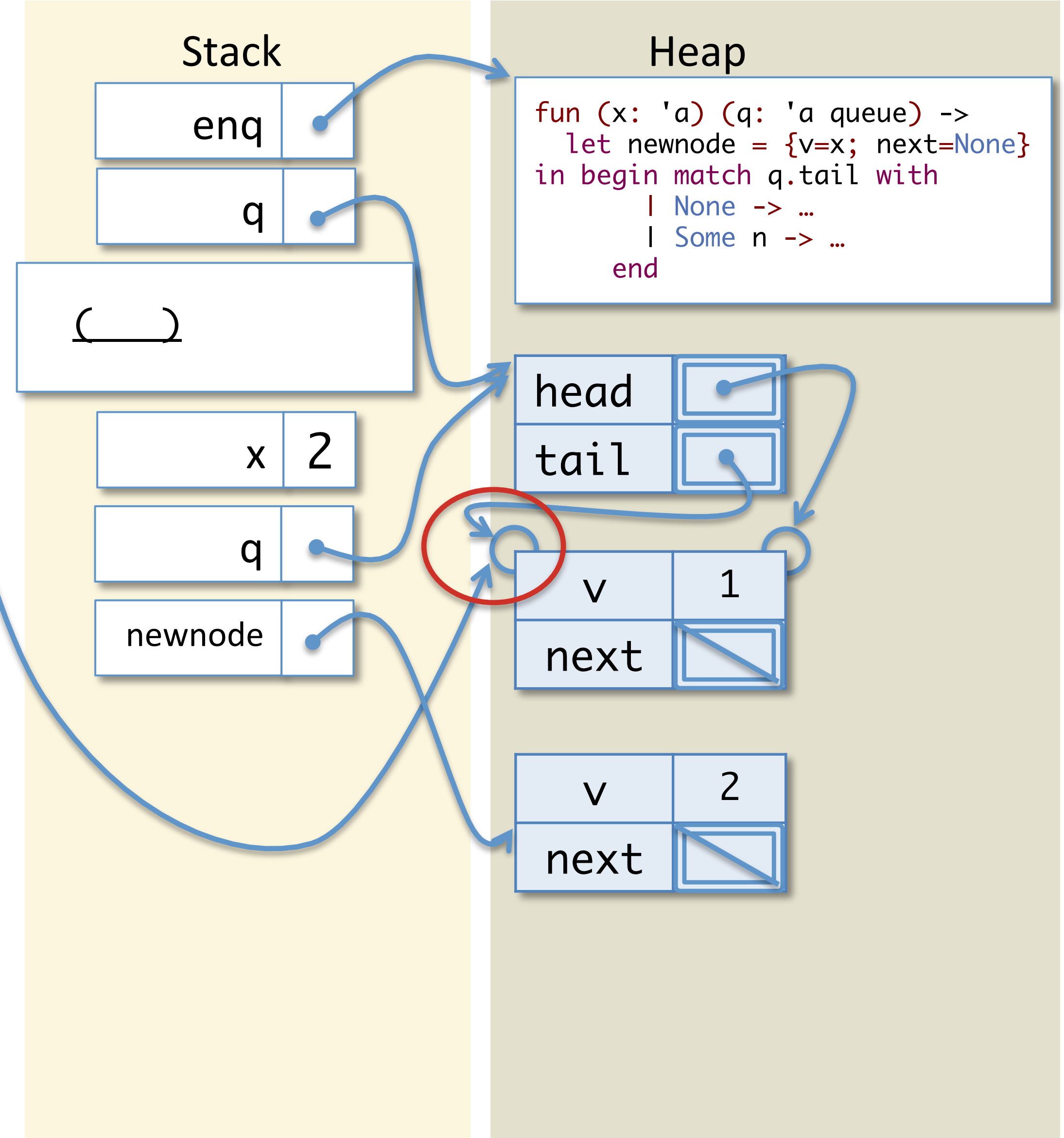
Calling Enq on a non-empty queue



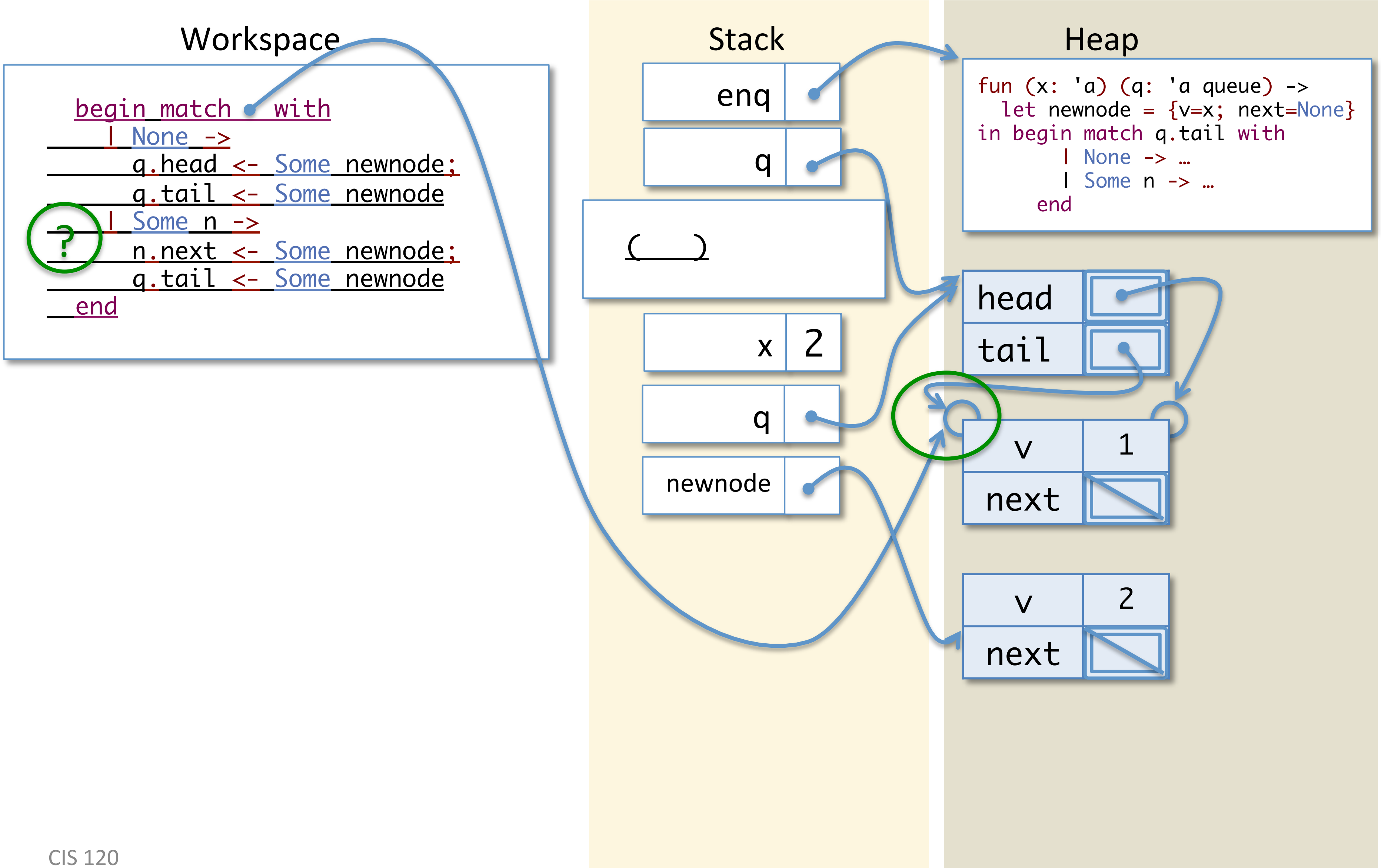
Calling Enq on a non-empty queue

Workspace

```
  begin match   with
?   | None ->
      q.head <- Some newnode;
      q.tail <- Some newnode
    | Some n ->
      n.next <- Some newnode;
      q.tail <- Some newnode
  end
```



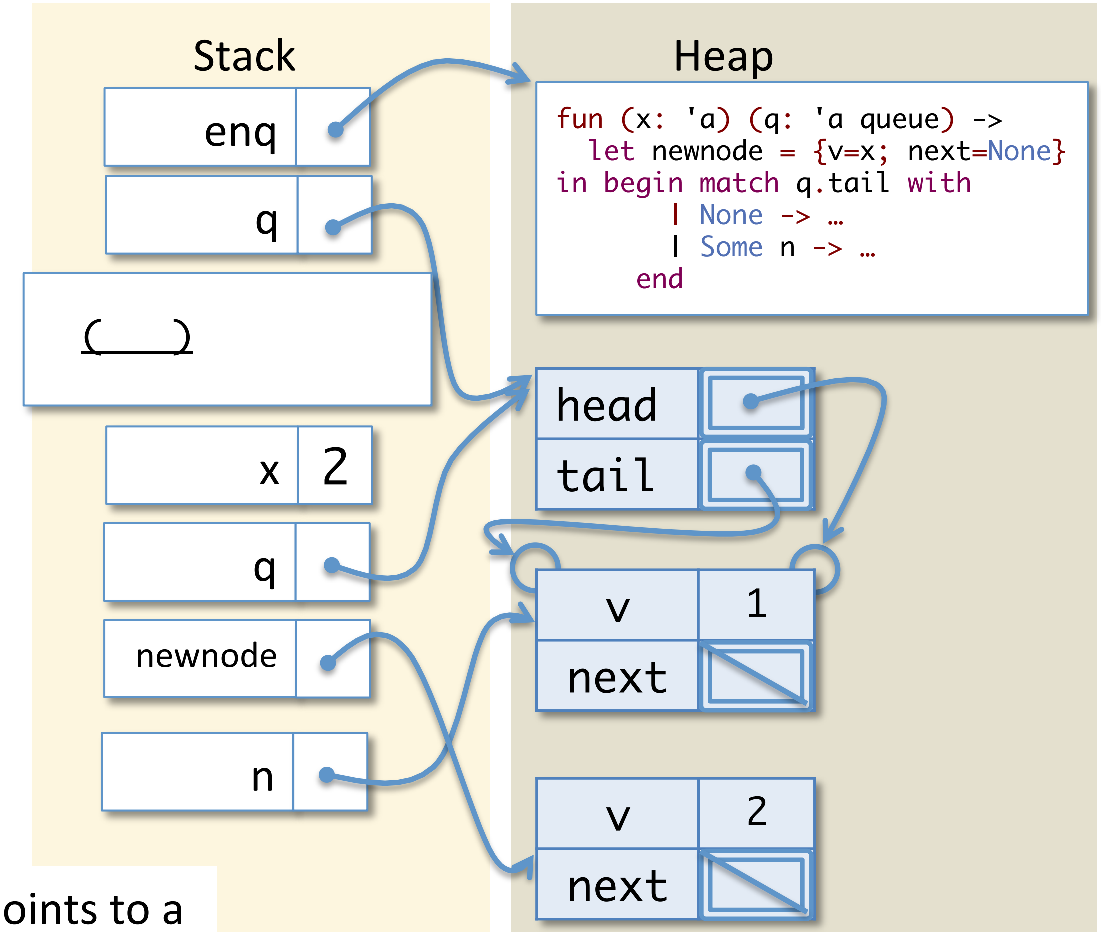
Calling Enq on a non-empty queue



Calling Enq on a non-empty queue

Workspace

```
n.next <- Some newnode;
q.tail <- Some newnode
```

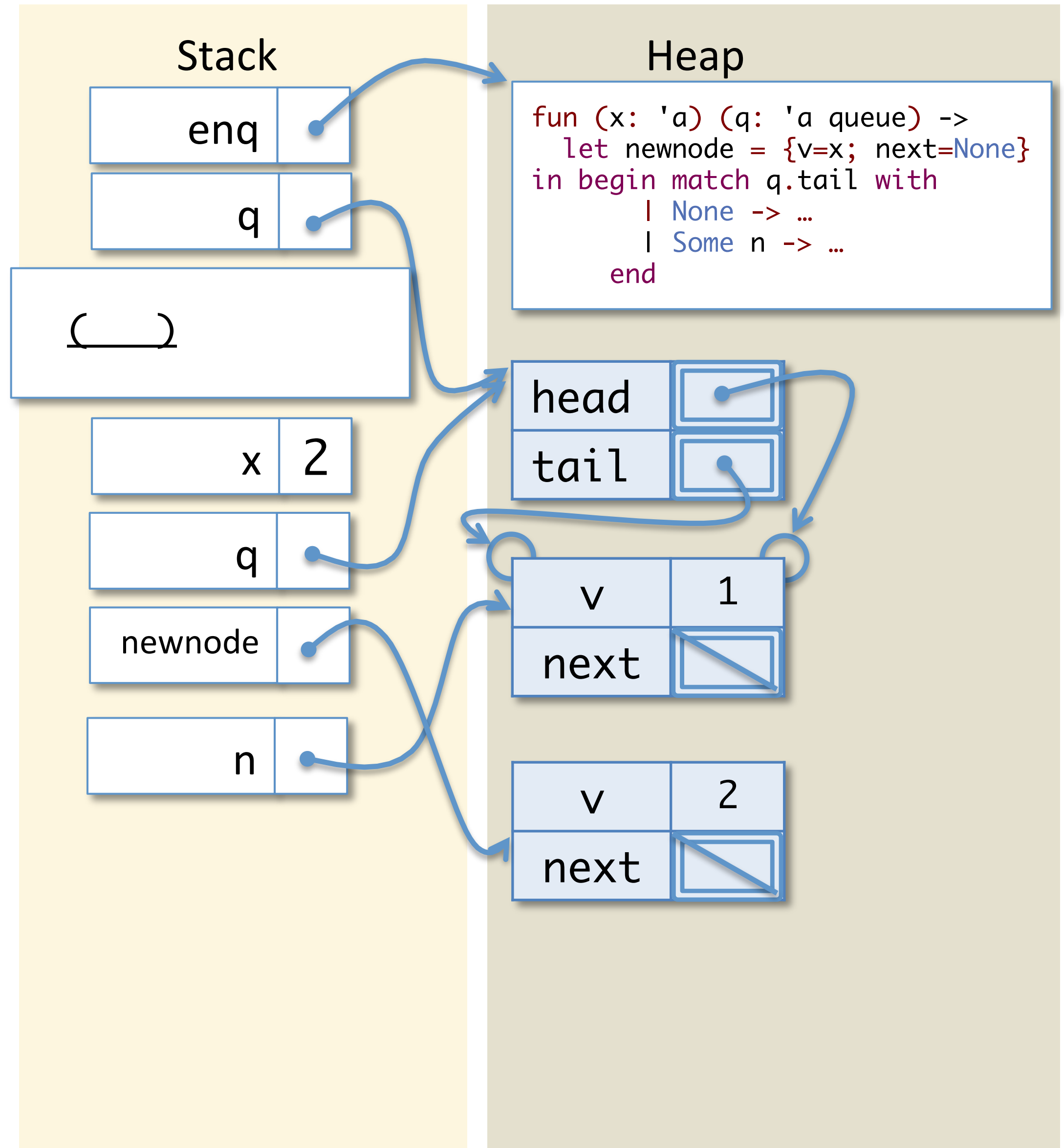


Note: n points to a qnode, not a qnode option.

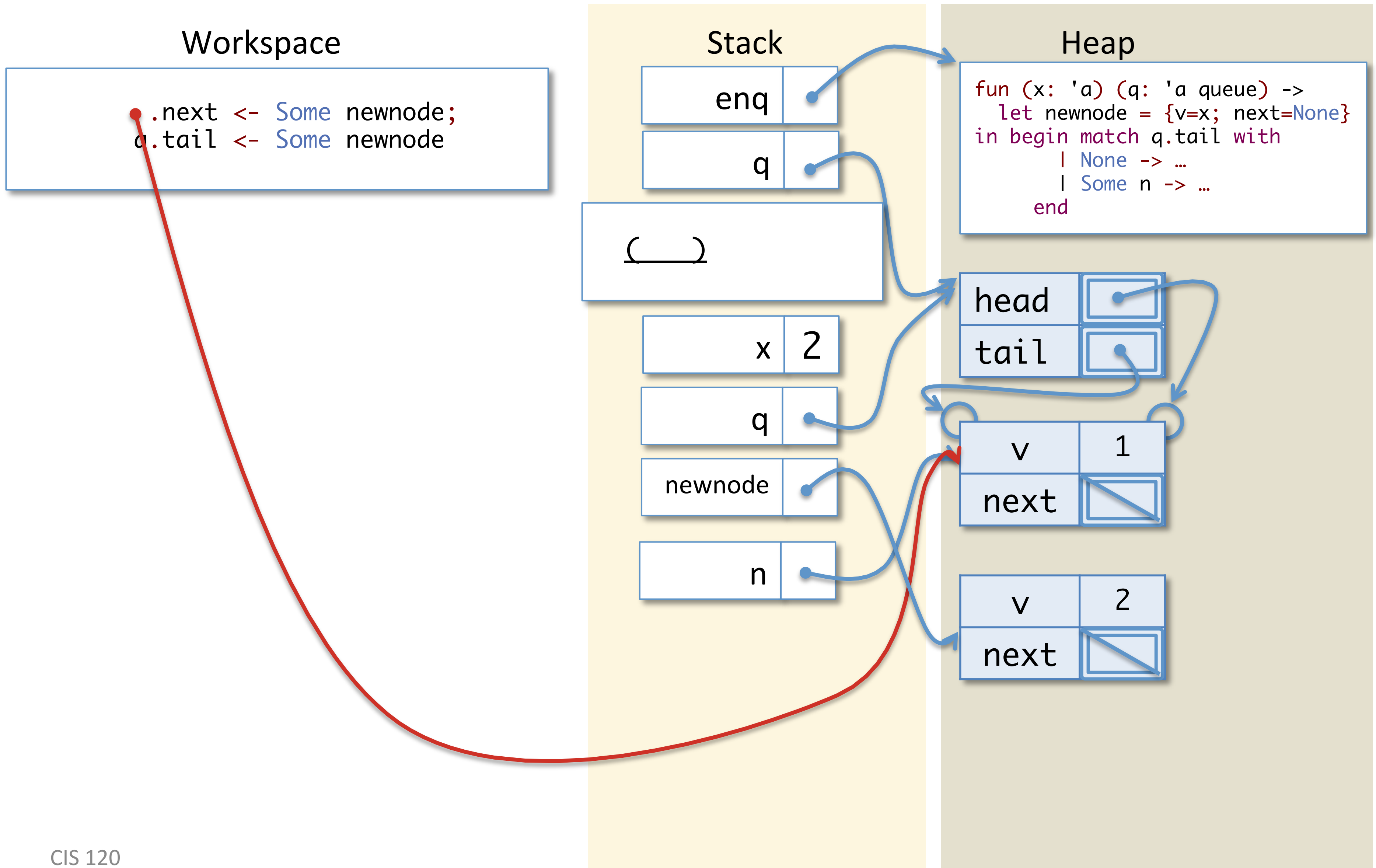
Calling Enq on a non-empty queue

Workspace

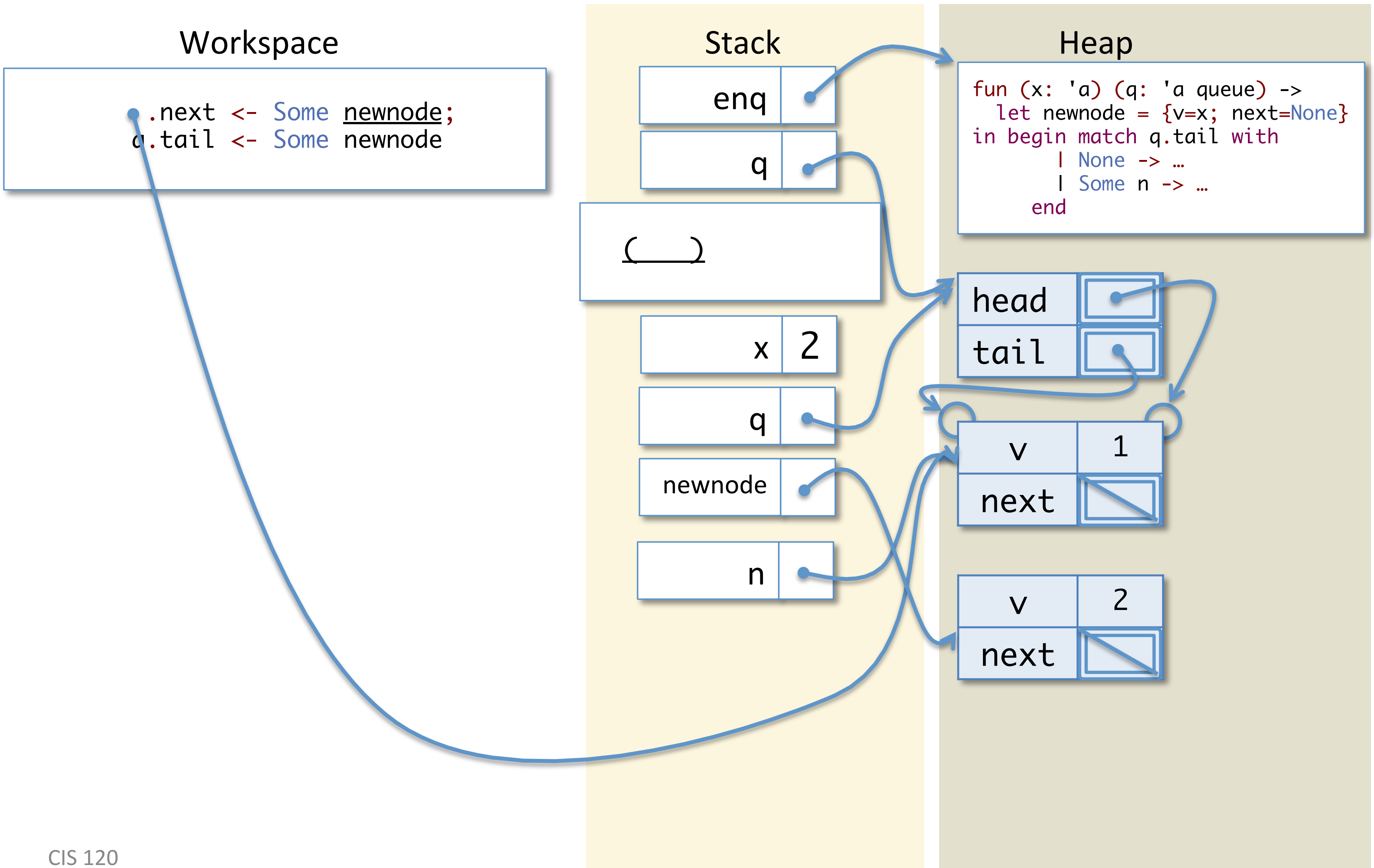
```
n.next <- Some newnode;
q.tail <- Some newnode
```



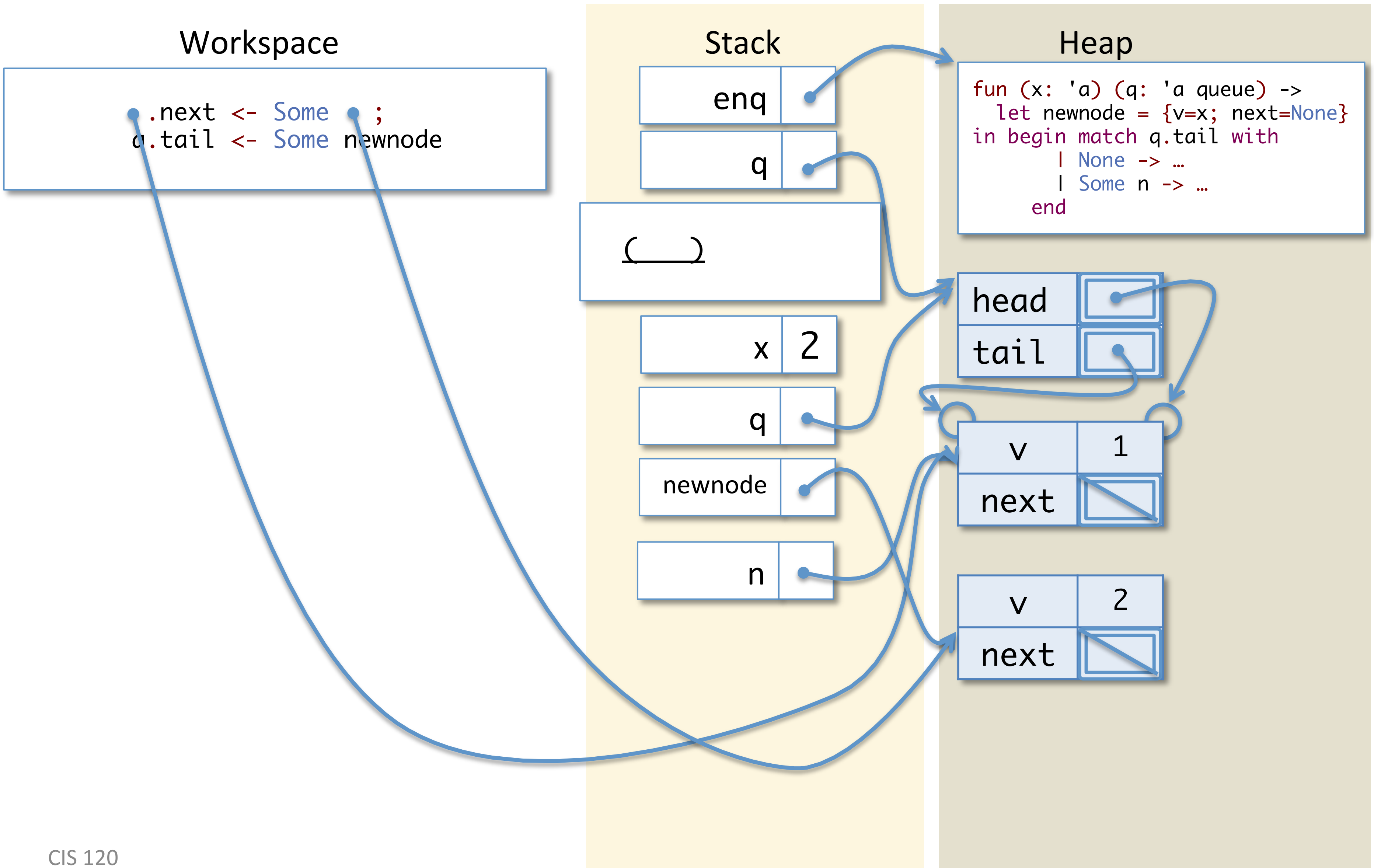
Calling Enq on a non-empty queue



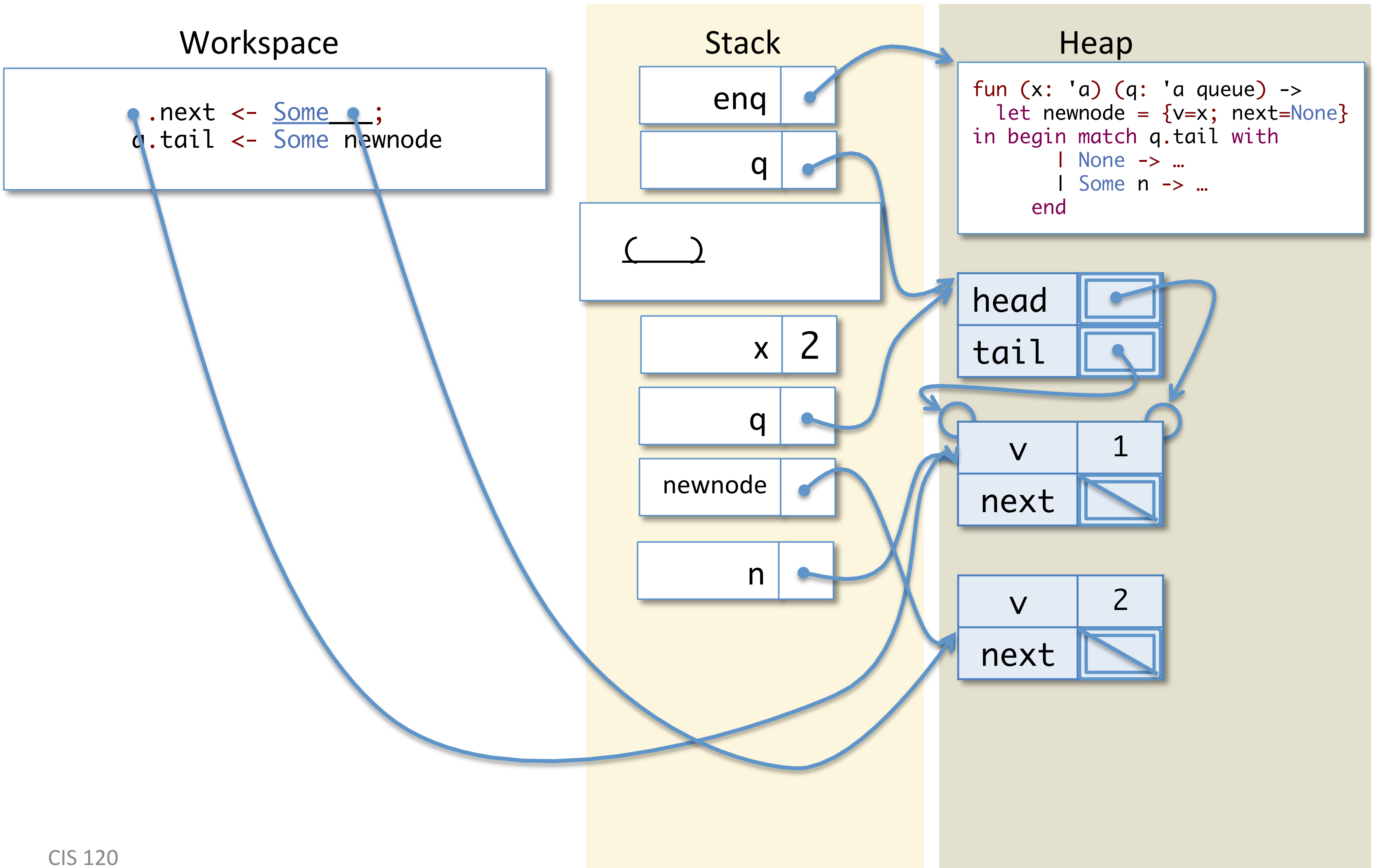
Calling Enq on a non-empty queue



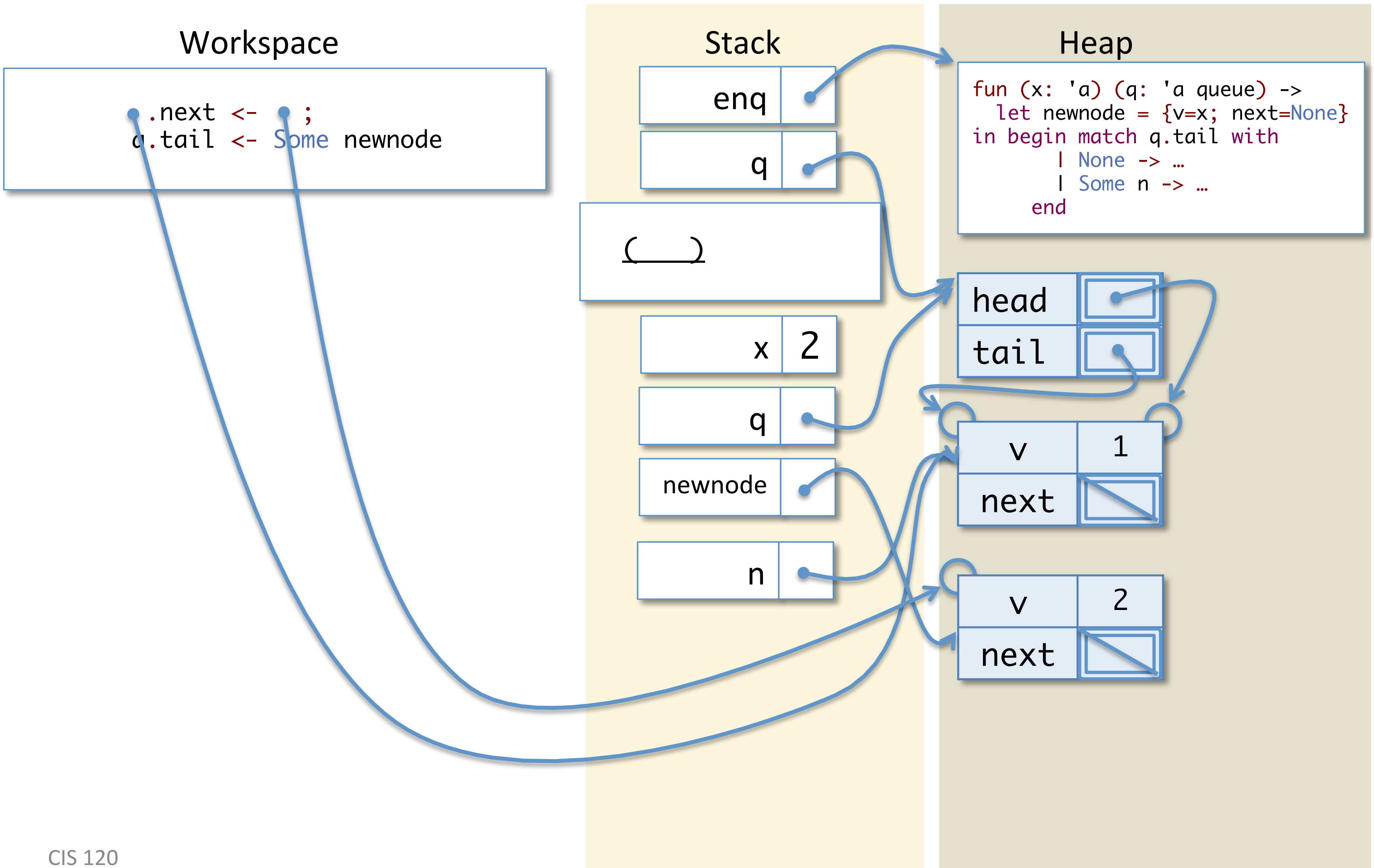
Calling Enq on a non-empty queue



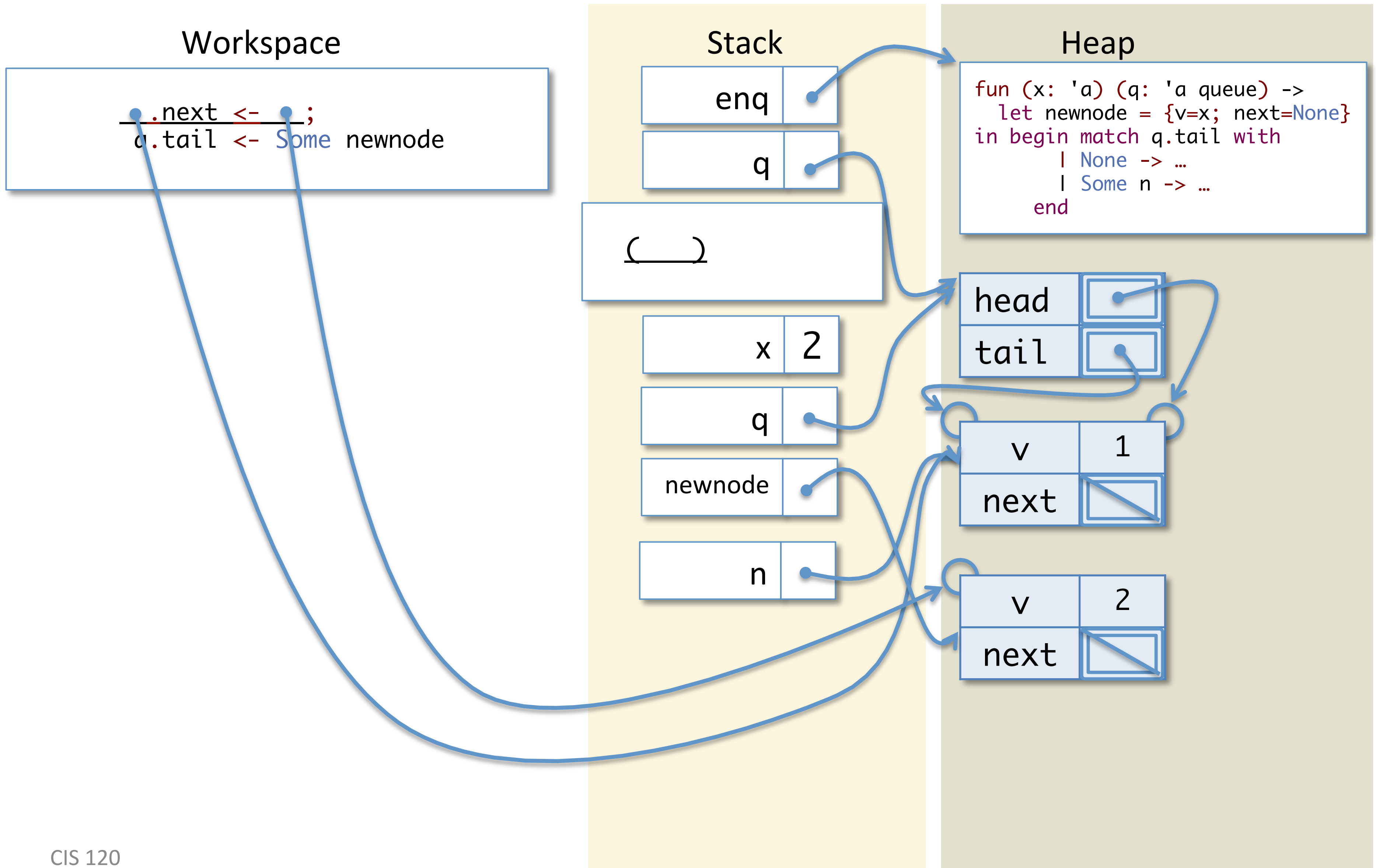
Calling Enq on a non-empty queue



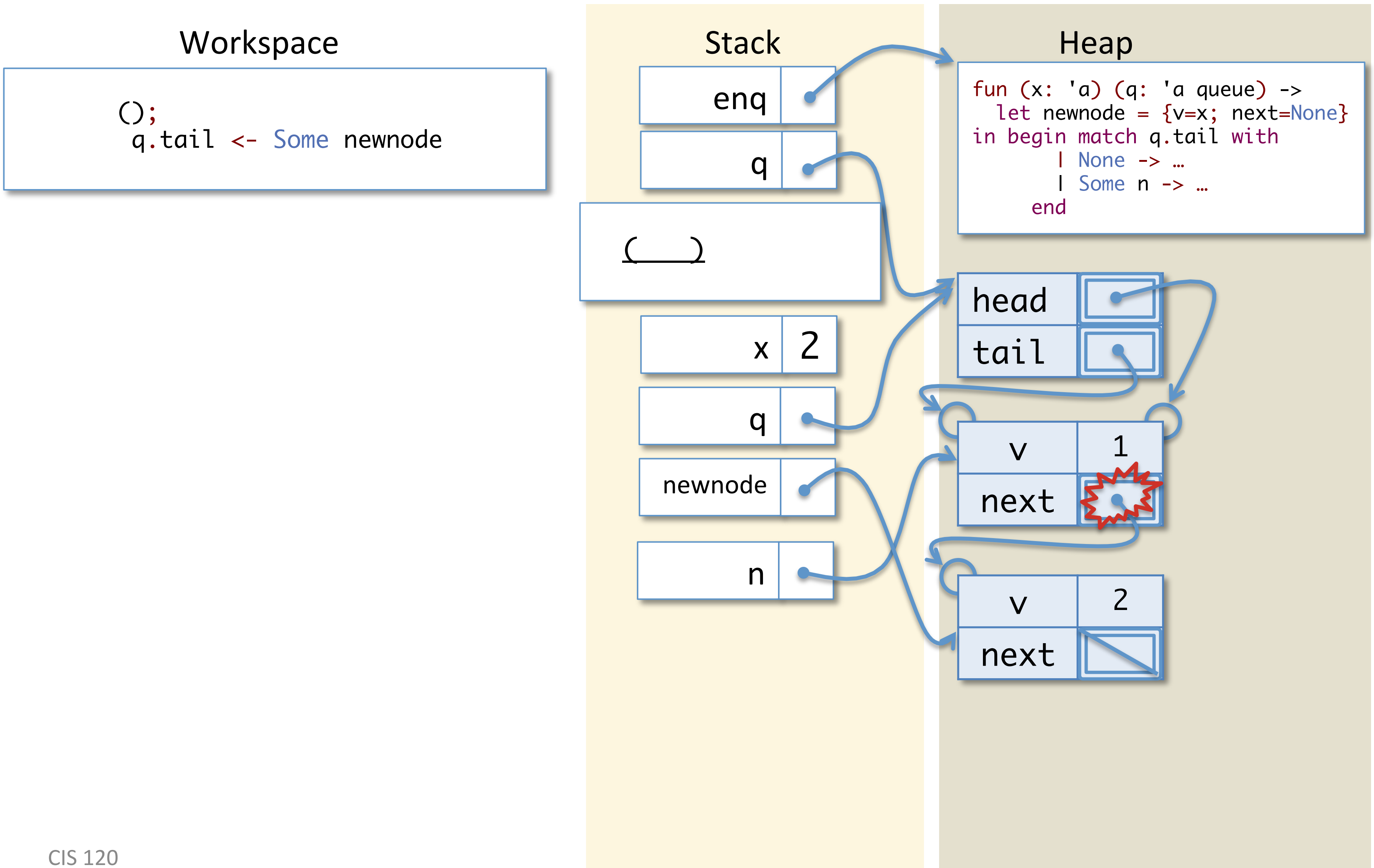
Calling Enq on a non-empty queue



Calling Enq on a non-empty queue



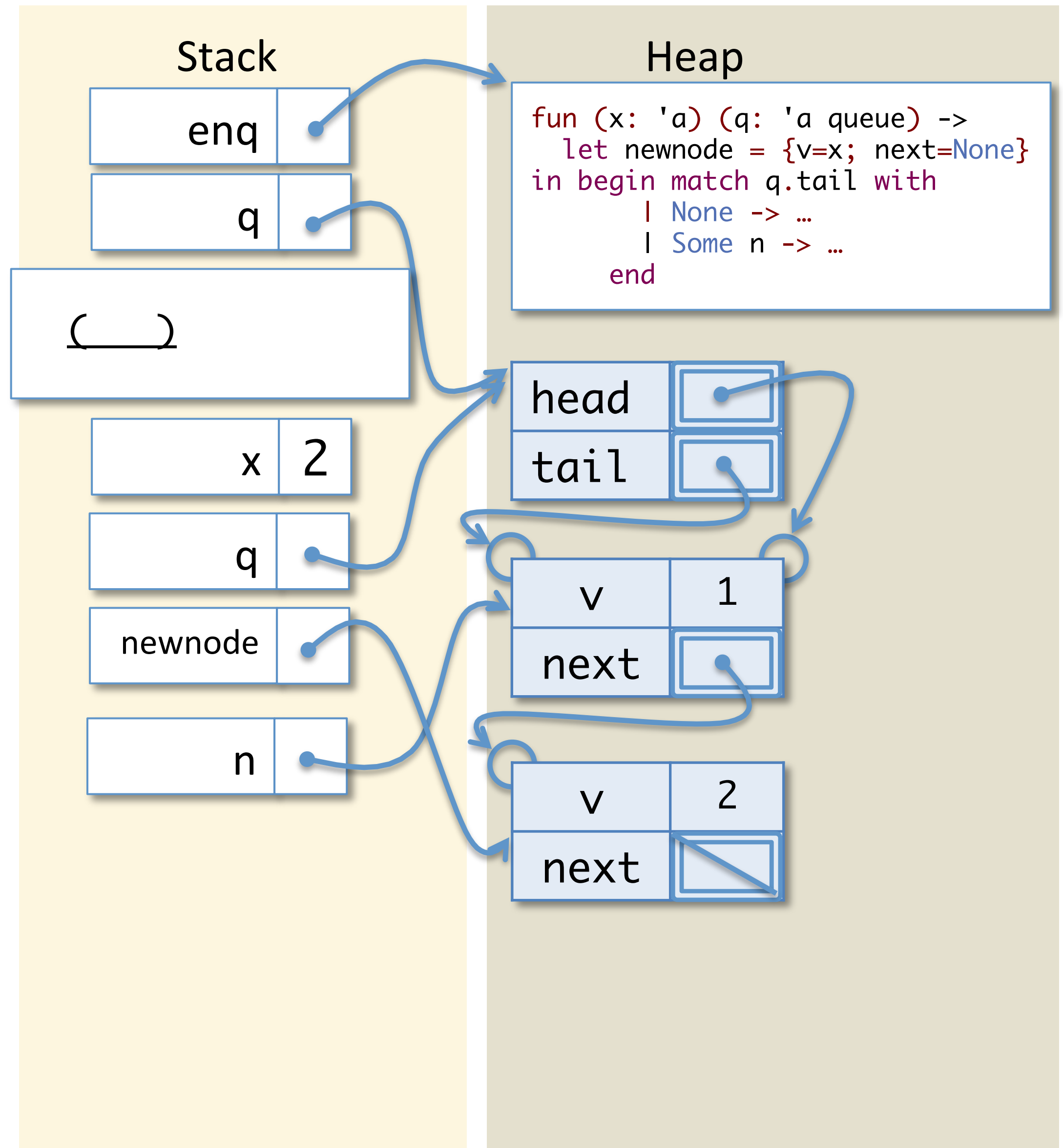
Calling Enq on a non-empty queue



Calling Enq on a non-empty queue

Workspace

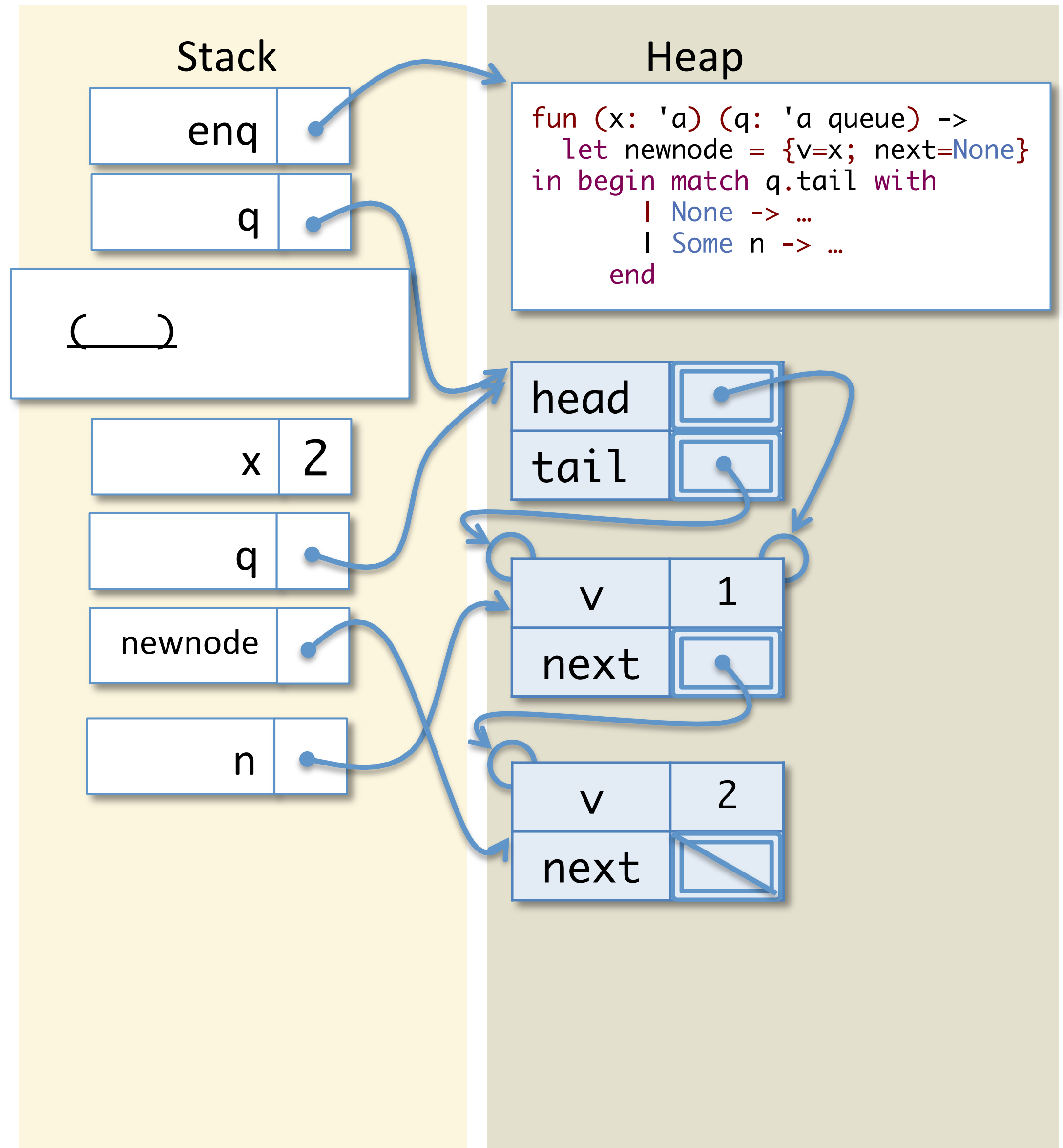
```
();
q.tail <- Some newnode
```



Calling Enq on a non-empty queue

Workspace

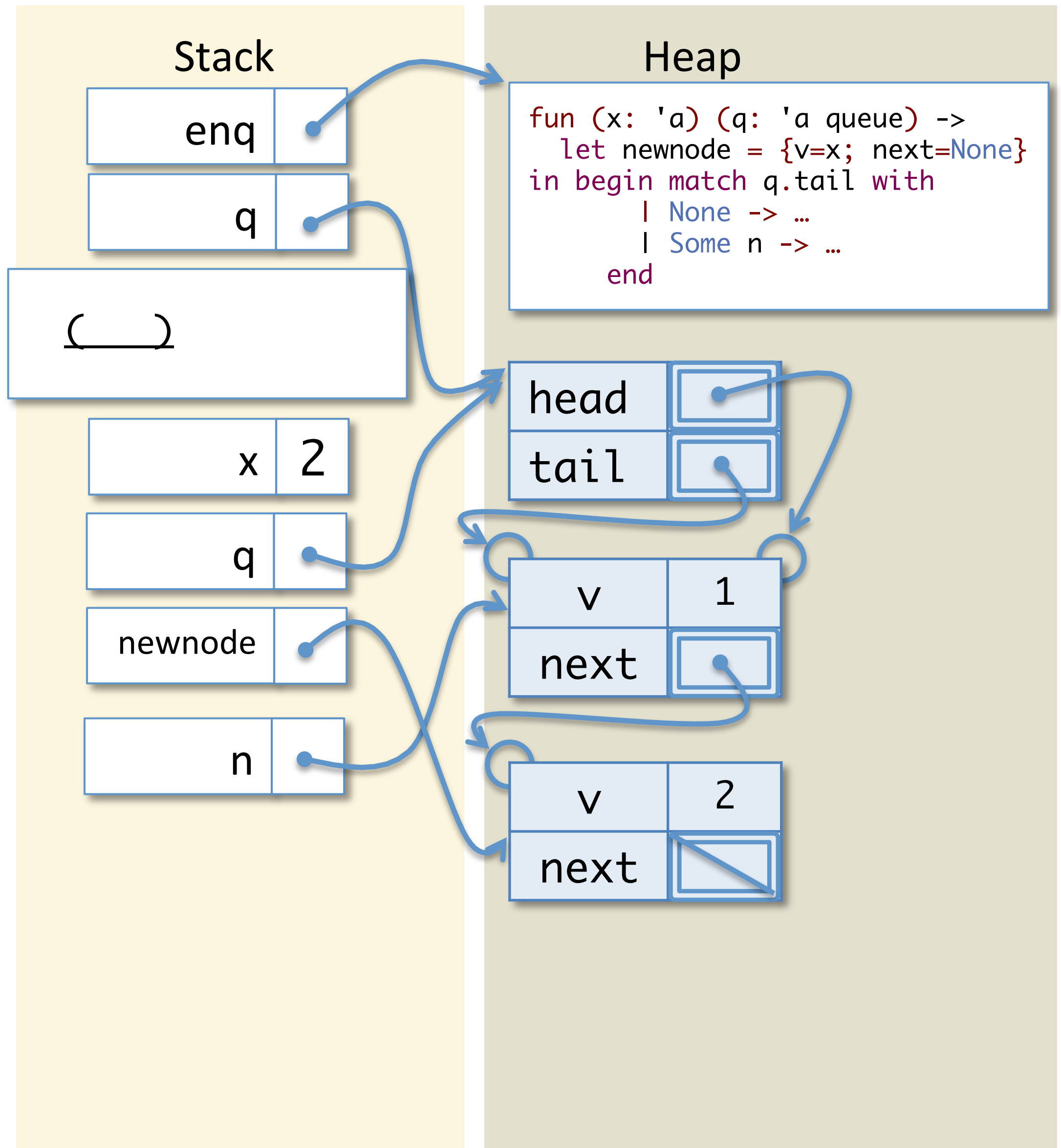
```
q.tail <- Some newnode
```



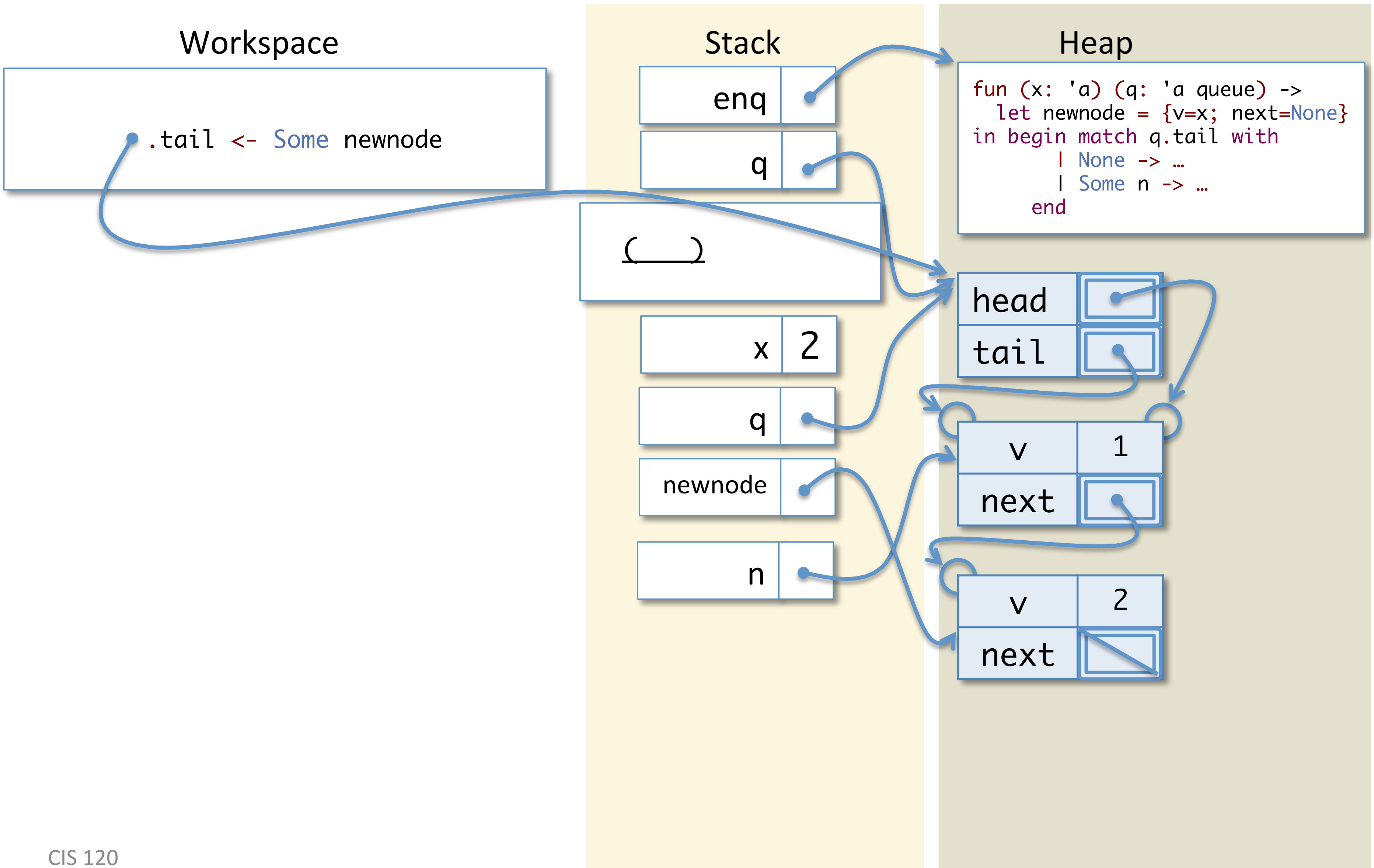
Calling Enq on a non-empty queue

Workspace

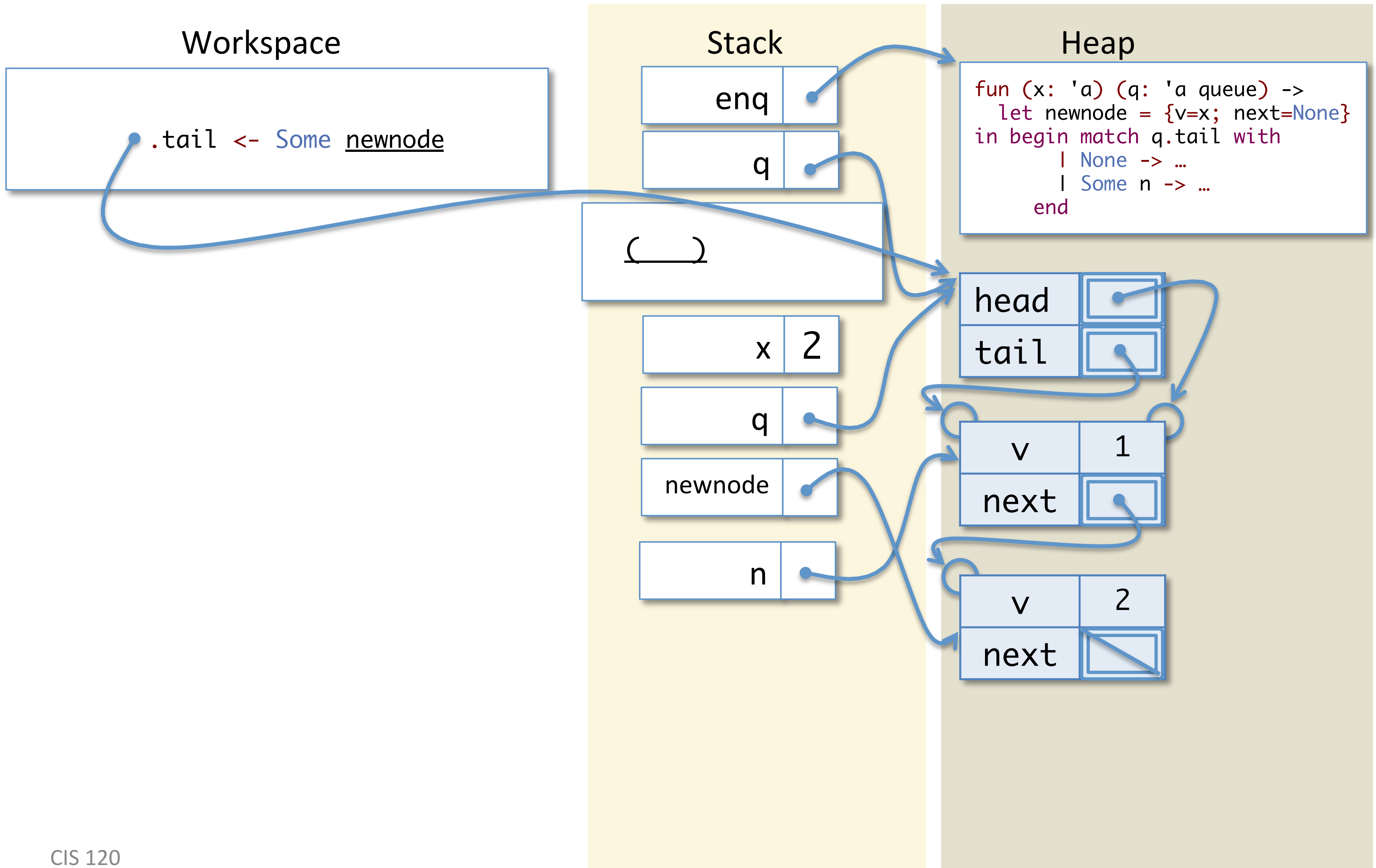
```
q.tail <- Some newnode
```



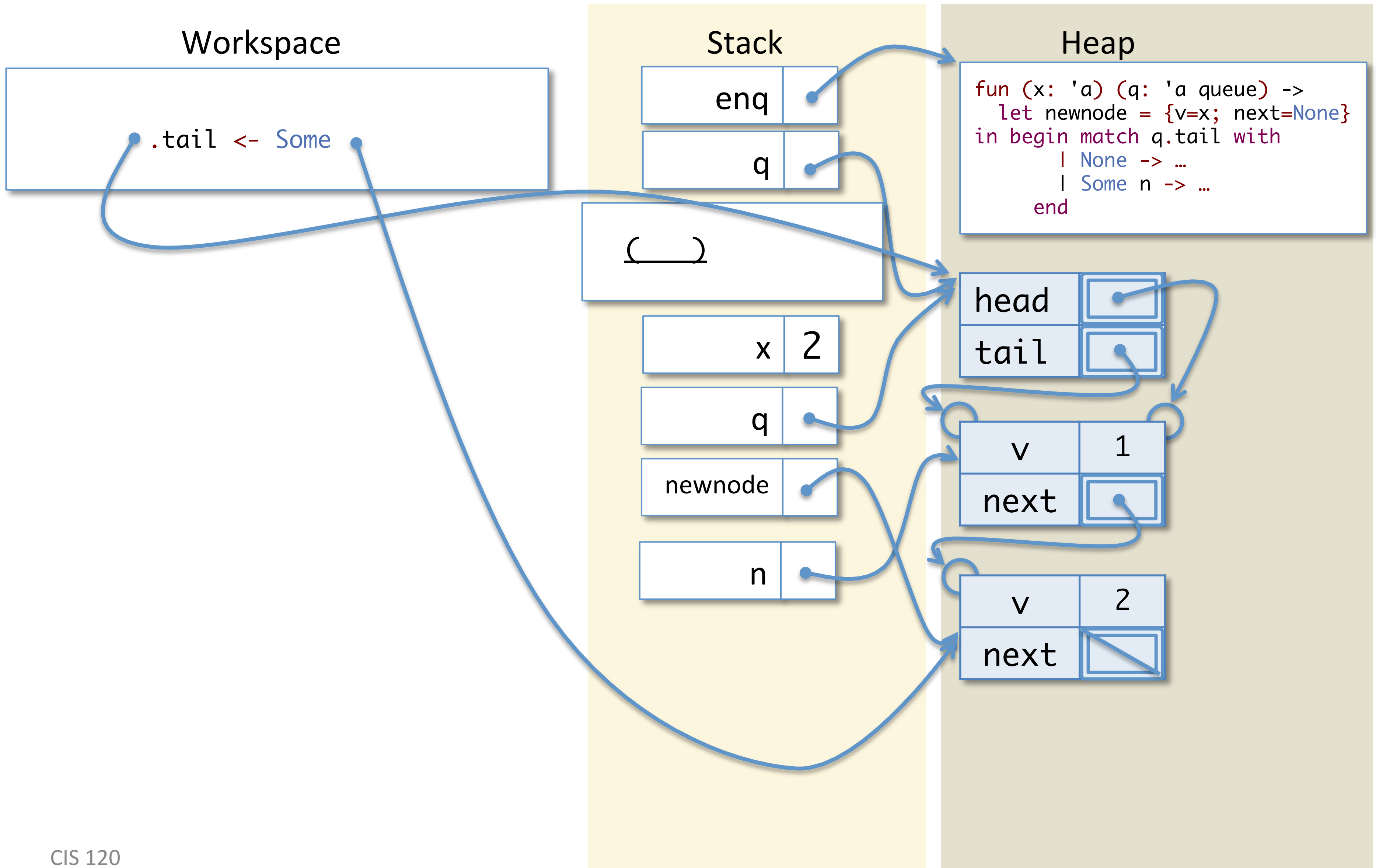
Calling Enq on a non-empty queue



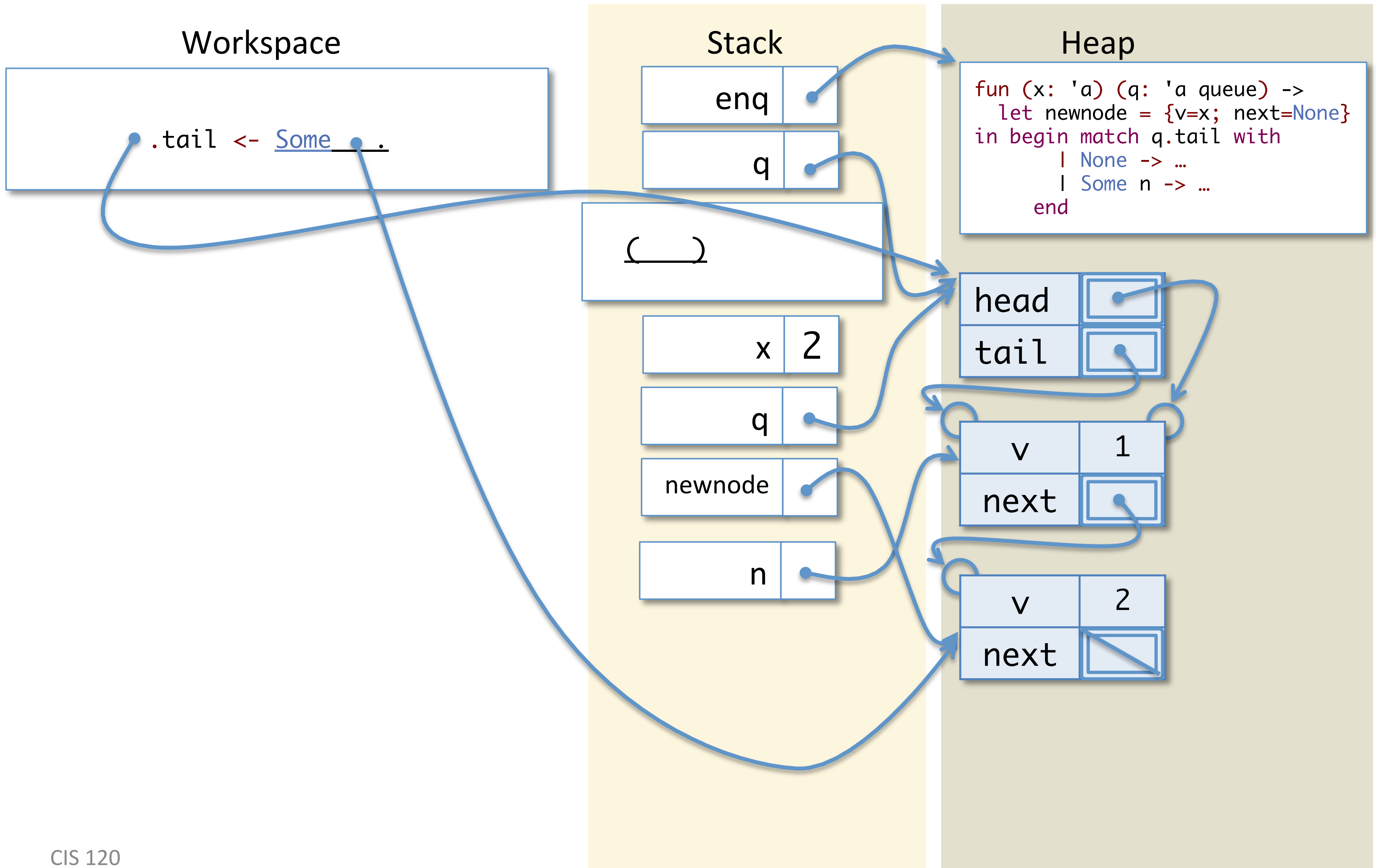
Calling Enq on a non-empty queue



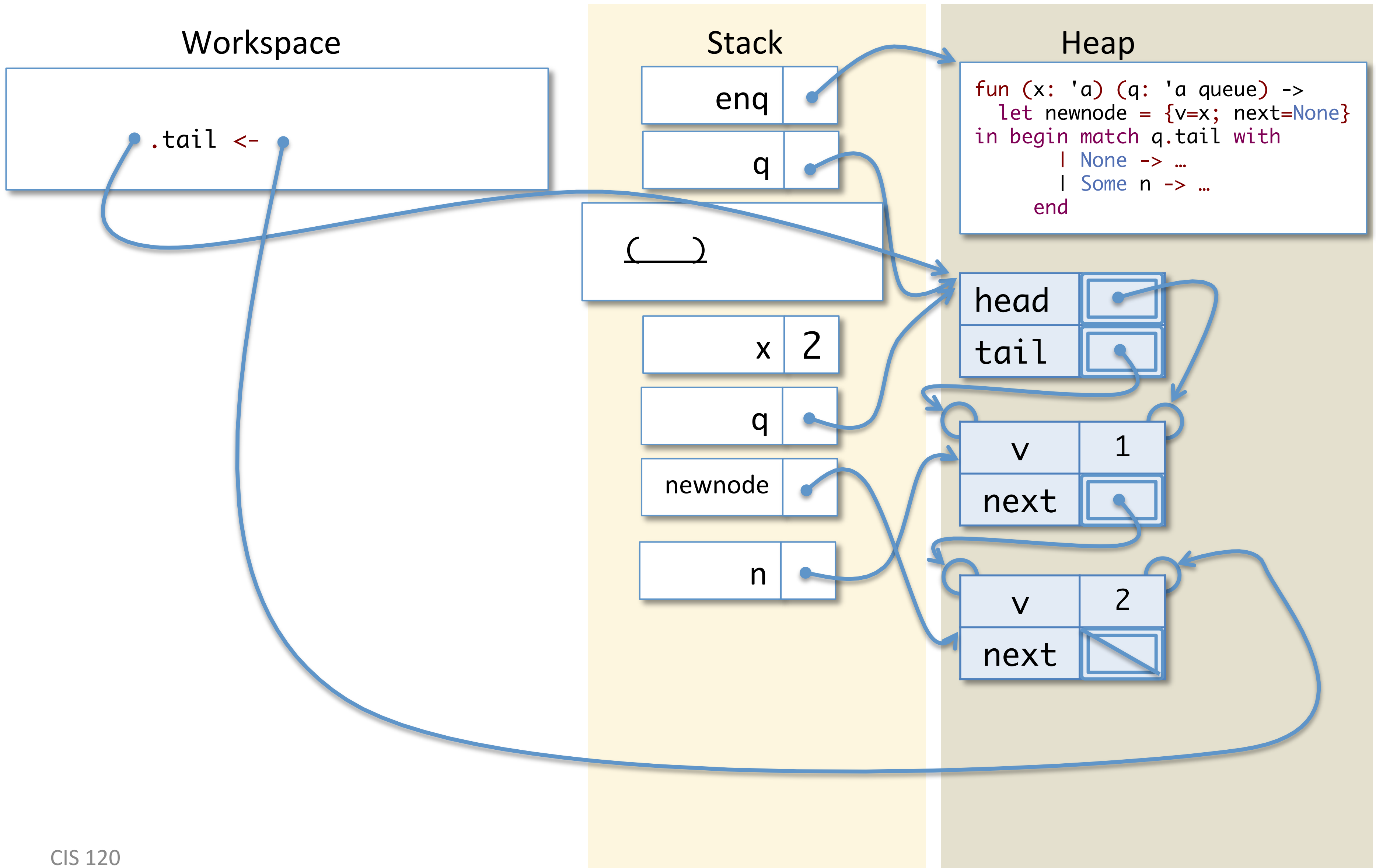
Calling Enq on a non-empty queue



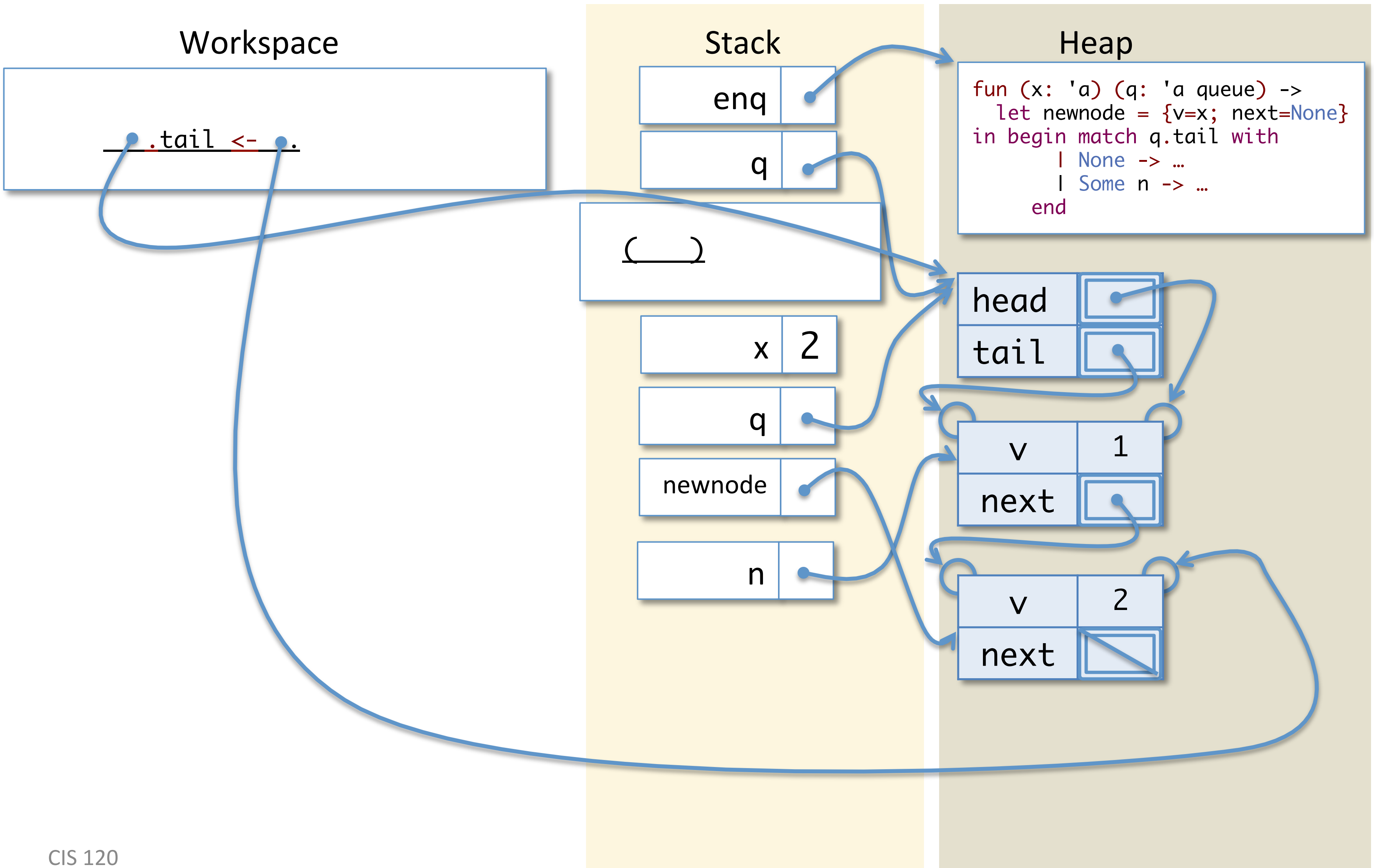
Calling Enq on a non-empty queue



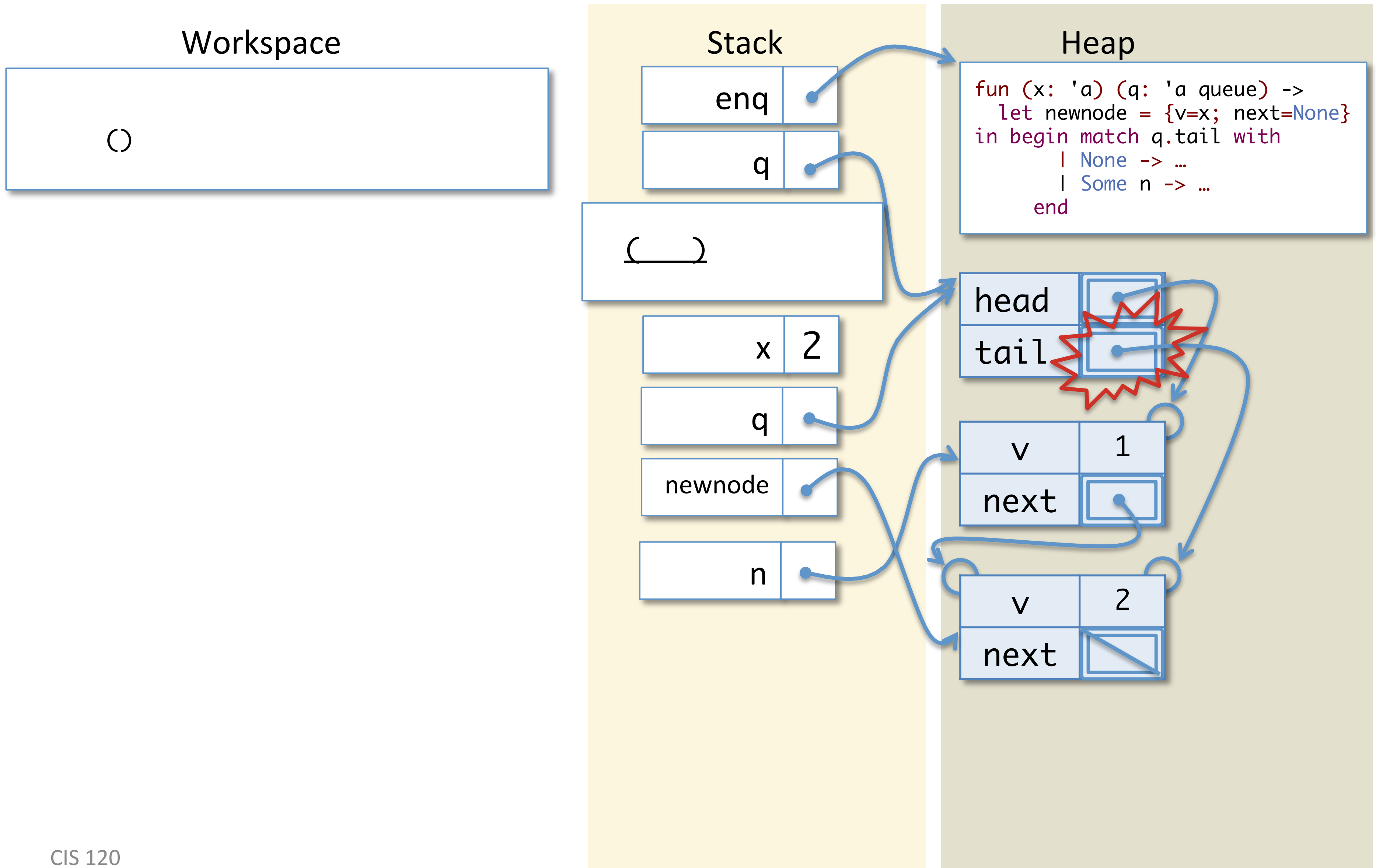
Calling Enq on a non-empty queue



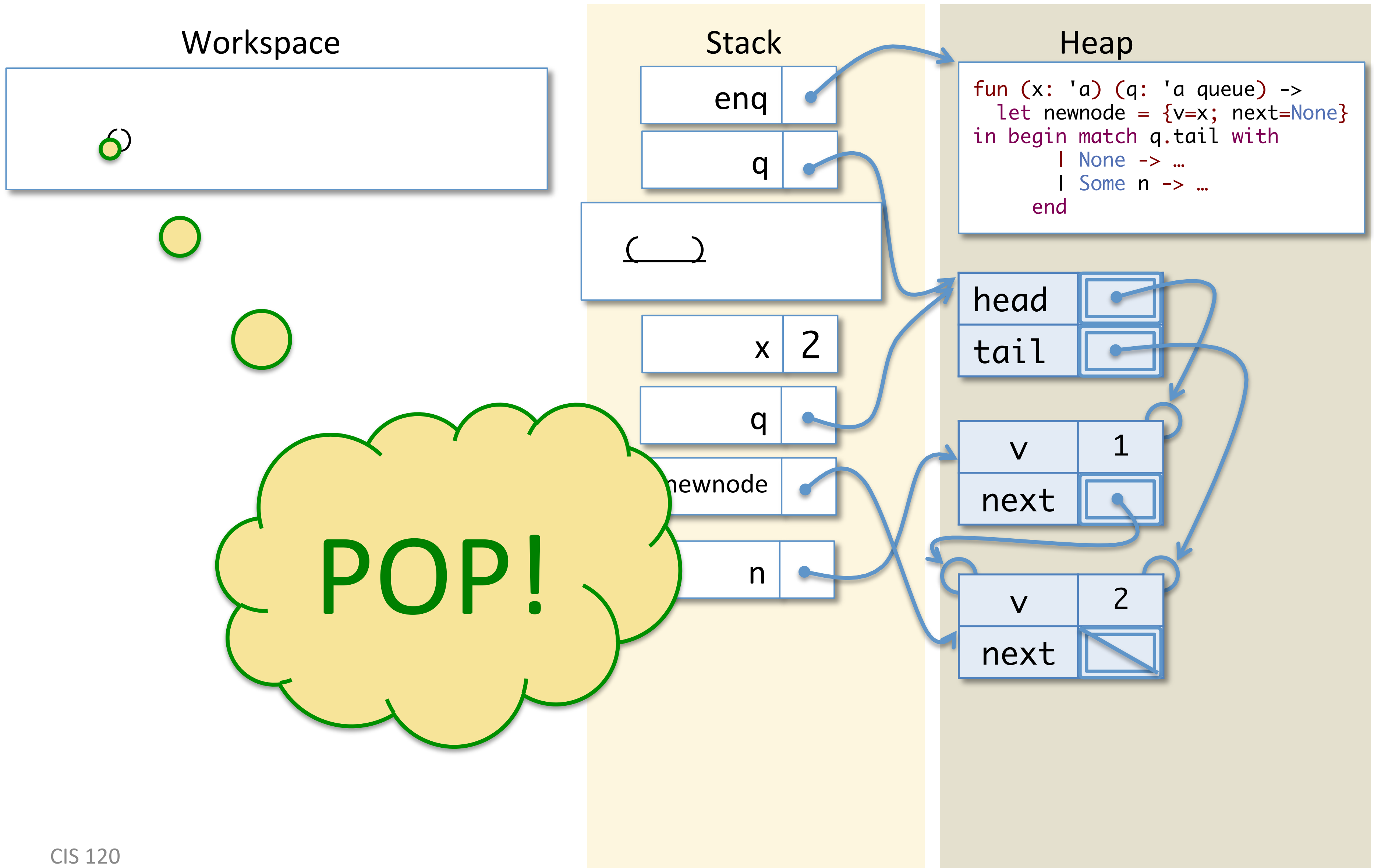
Calling Enq on a non-empty queue



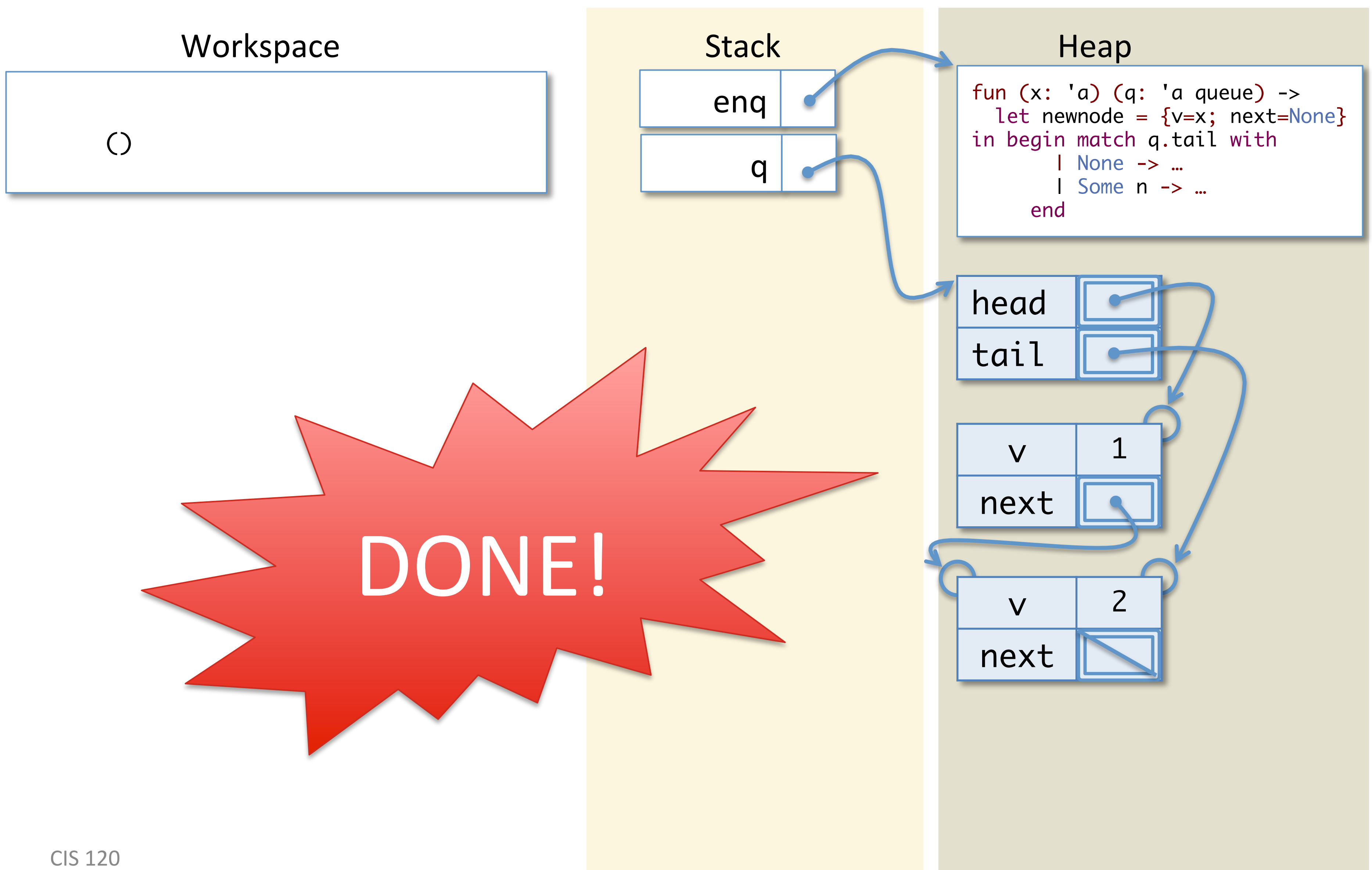
Calling Enq on a non-empty queue



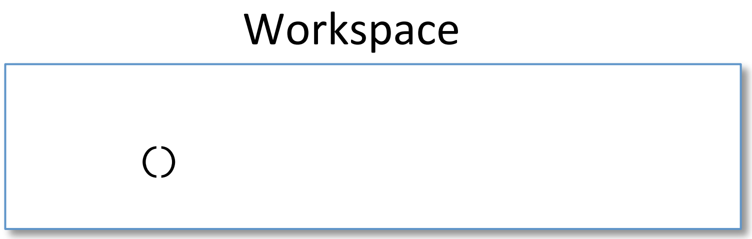
Calling Enq on a non-empty queue

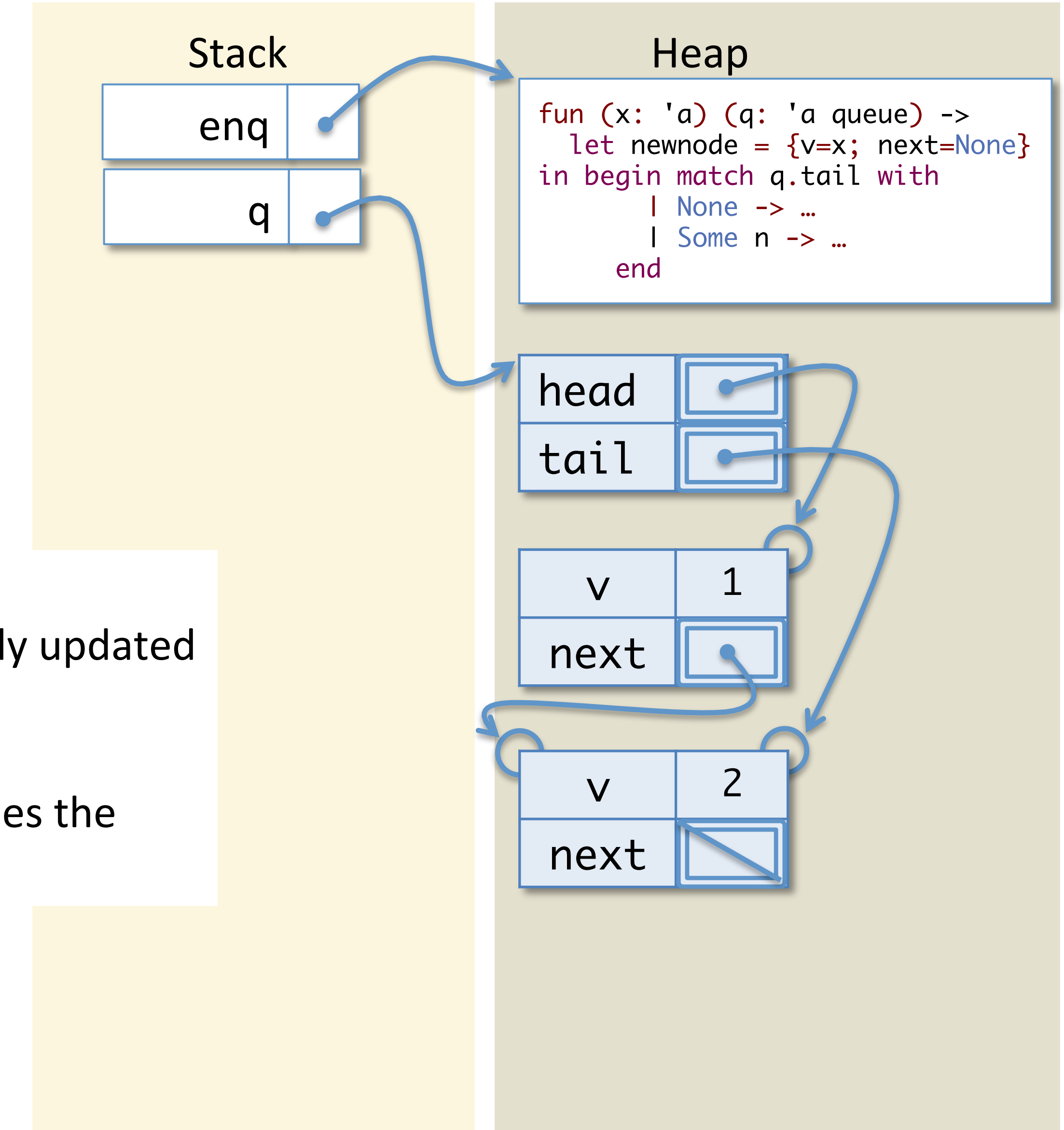


Calling Enq on a non-empty queue



Calling Enq on a non-empty queue





Notes:
- the enq function imperatively updated the structure of q

- the new structure still satisfies the queue invariants

deq

```
(* remove an element from the head of the queue *)
 let deq (q: 'a queue) : 'a =
   begin match q.head with
     | None ->
         failwith "deq called on empty queue"
     | Some n ->
         q.head <- n.next;
         if n.next = None then q.tail <- None;
         n.v
   end
```

- The code for deq must also “patch pointers” to maintain the queue invariant:
  - The head pointer is always updated to the next element in the queue.
  - If the removed node was the last one in the queue, the tail pointer must be updated to None