Programming Languages and Techniques (CIS120)

Lecture 15

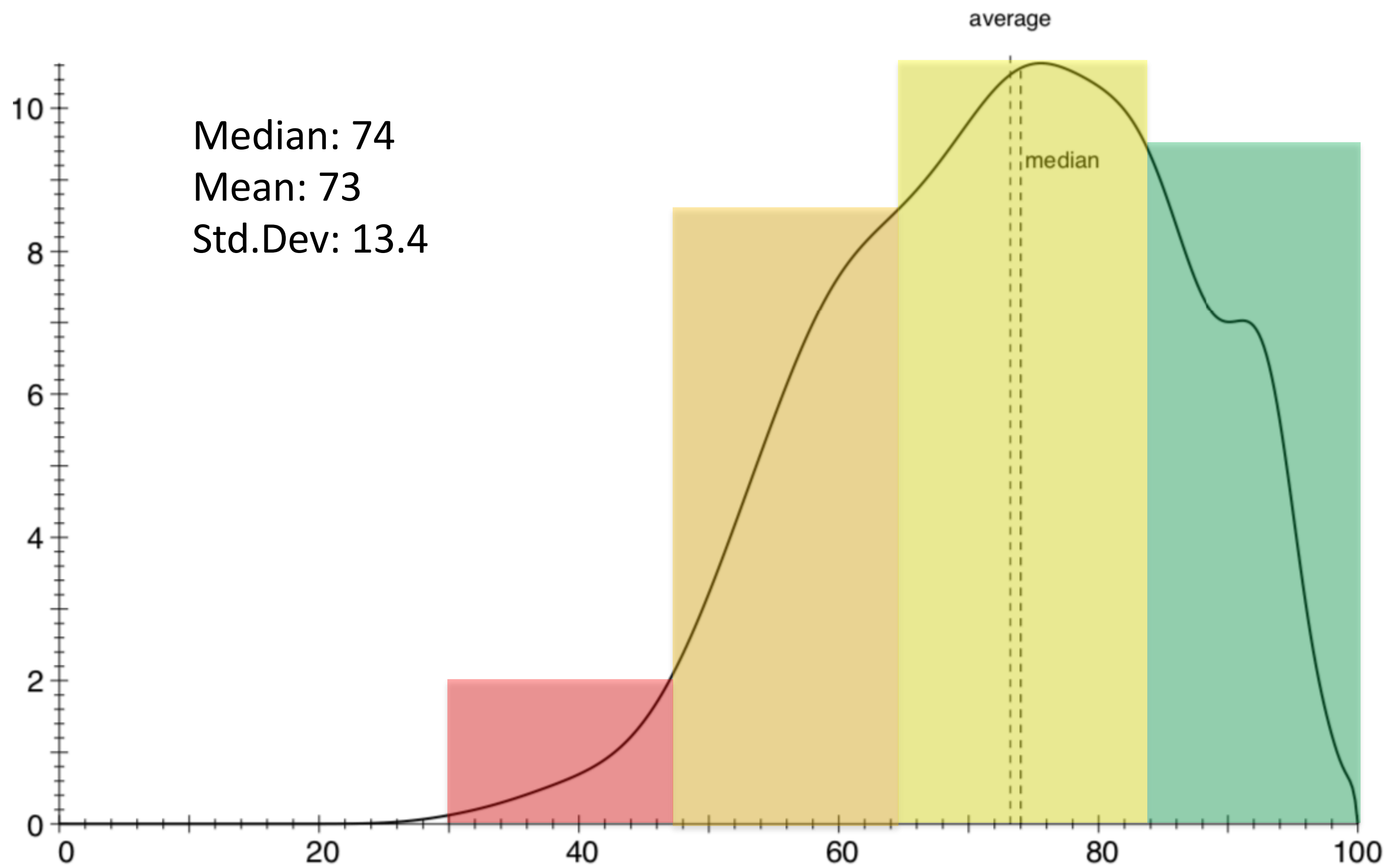
Queues

Lecture notes: Chapter 16

Announcements

- Homework 4
  - due Tuesday, October 8th at 11:59 pm
- No Recitations Sections Next Week
  - Fall Break!
- Midterm 1 is graded
  - Solutions available on the course web site (see the schedule)
  - Submit regrade requests via Gradescope before Thursday, October 10th.

Midterm 1 Analysis



Median: 74
Mean: 73
Std.Dev: 13.4

Putting State to Work: Mutable Queues

A design problem

Suppose you are implementing a website for constituents to submit questions to their political representatives. To be fair, you would like to deal with questions in first-come, first-served order. How would you do it?

- Understand the problem
  - Need to keep track of pending questions, in the order in which they were submitted
- Define the interface
  - Need a data structure to store questions
  - Need to add questions to the *end* of the queue
  - Need to allow responders to retrieve questions from the *beginning* of the queue
  - Both kinds of access must be efficient to handle large 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 a queue is empty *)
  val is_empty : 'a queue -> bool

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

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

end
```

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

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

A: Adding an element to a 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
```

Implementing Linked Queues

Representing links

Data Structure for Mutable Queues

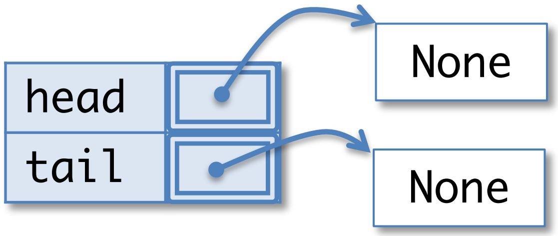
```
type 'a qnode = {
     v: 'a;
     mutable next : 'a qnode option
}

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

There are two parts to a mutable queue:

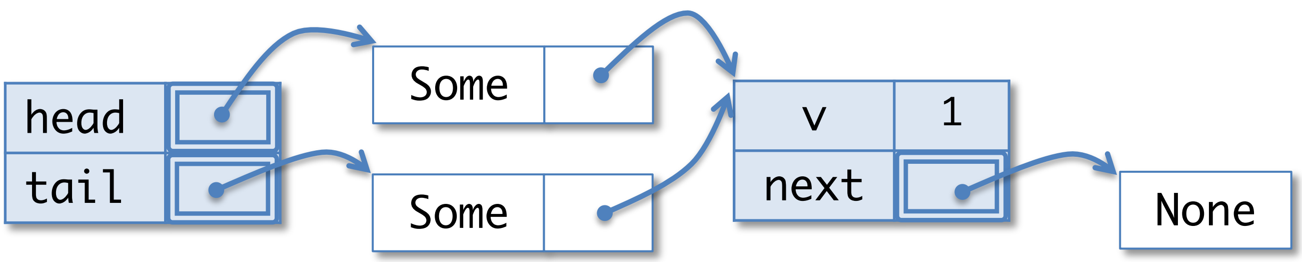
1. the “internal nodes” of the queue, with links from one to the next
2. a record with links to the head and tail nodes

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

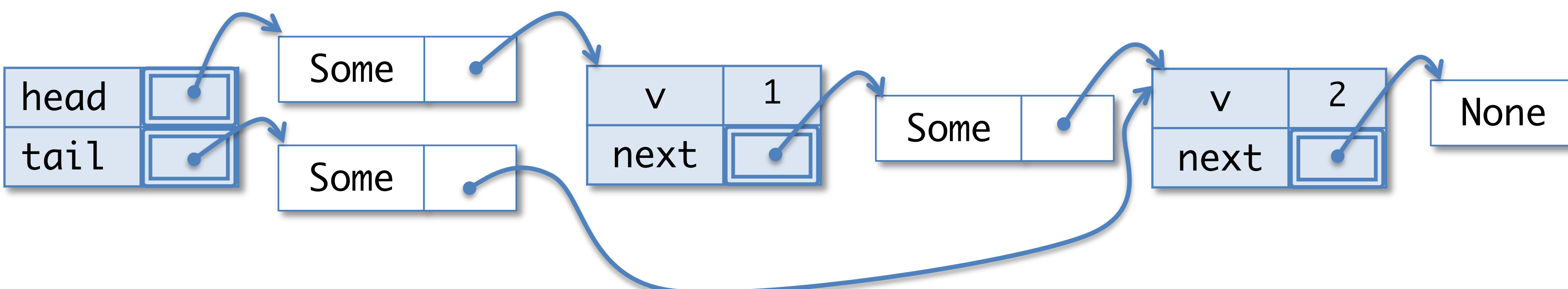
Queues in the Heap



An empty queue

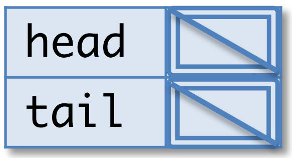


A queue with one element

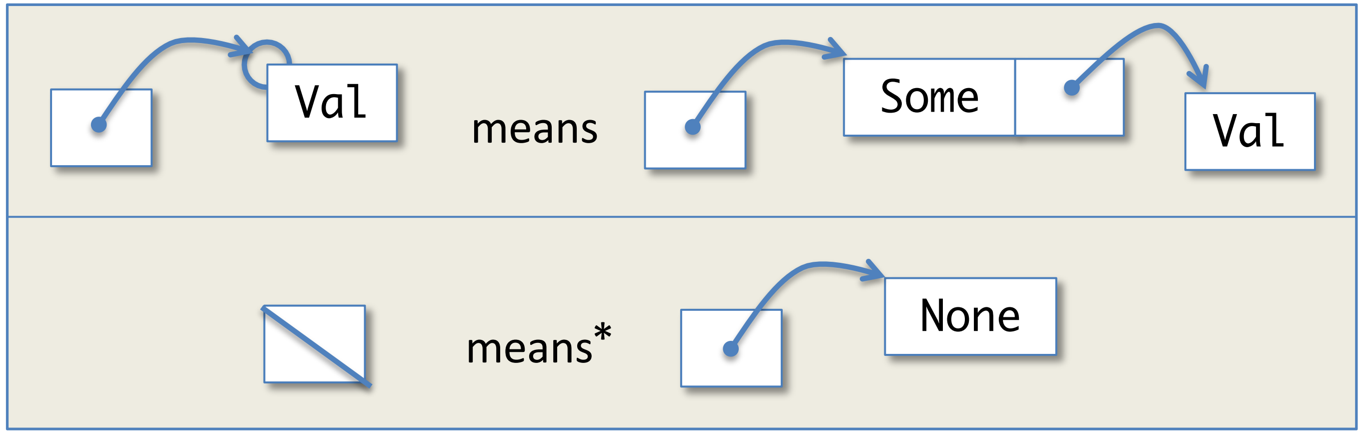


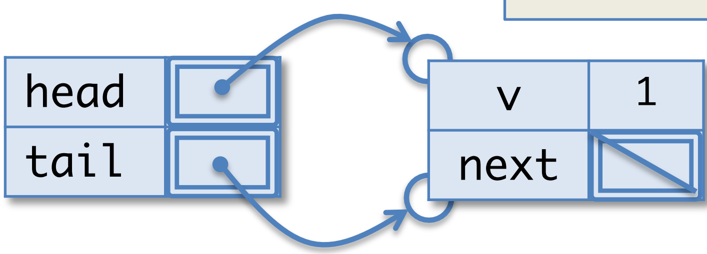
A queue with two elements

Visual Shorthand: Abbreviating Options



An empty queue





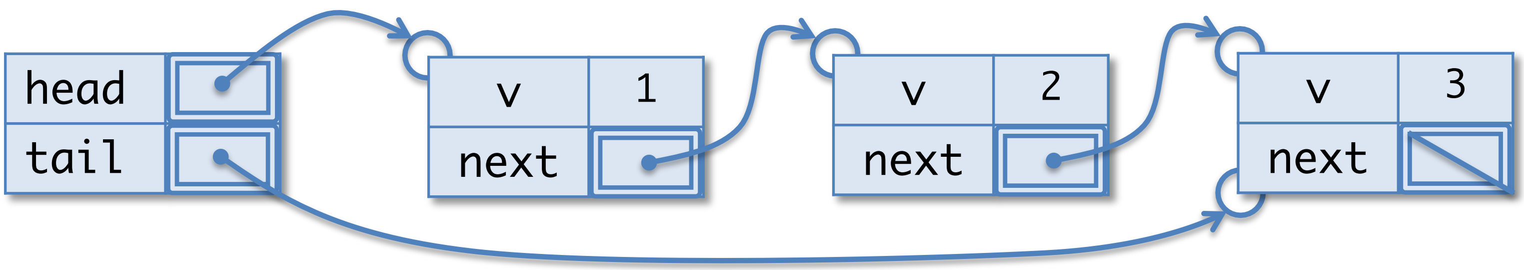
A queue with one element

*Note: Ocaml can optimize "nullary" constructors like Nil, None, Empty so that they aren't allocated in the heap. This is why

`None == None`

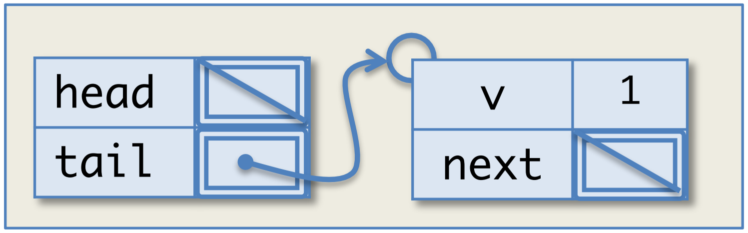
even though `not ((Some x) == (Some x))`.

Be careful with equality and options.

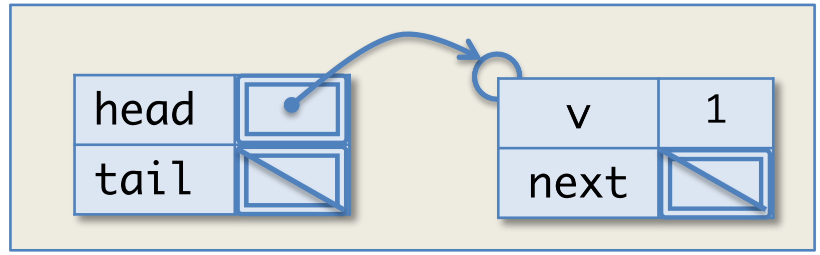


A queue with three elements

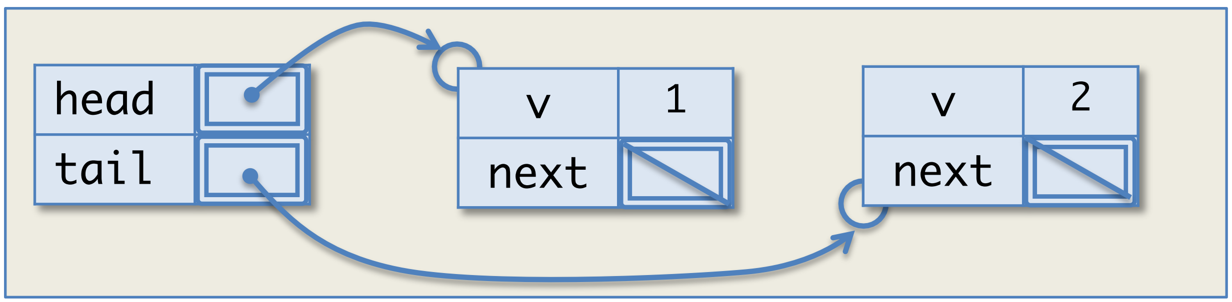
“Bogus” values of type `int queue`



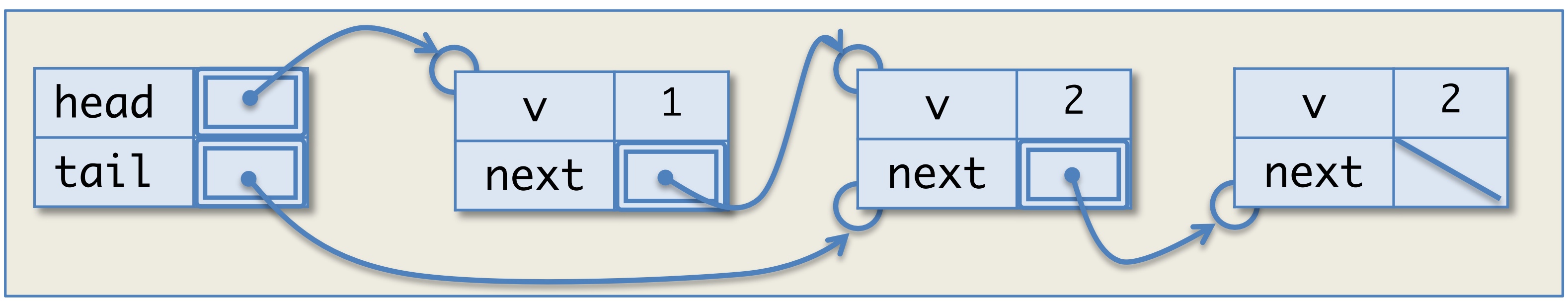
head is None, tail is Some



head is Some, tail is None



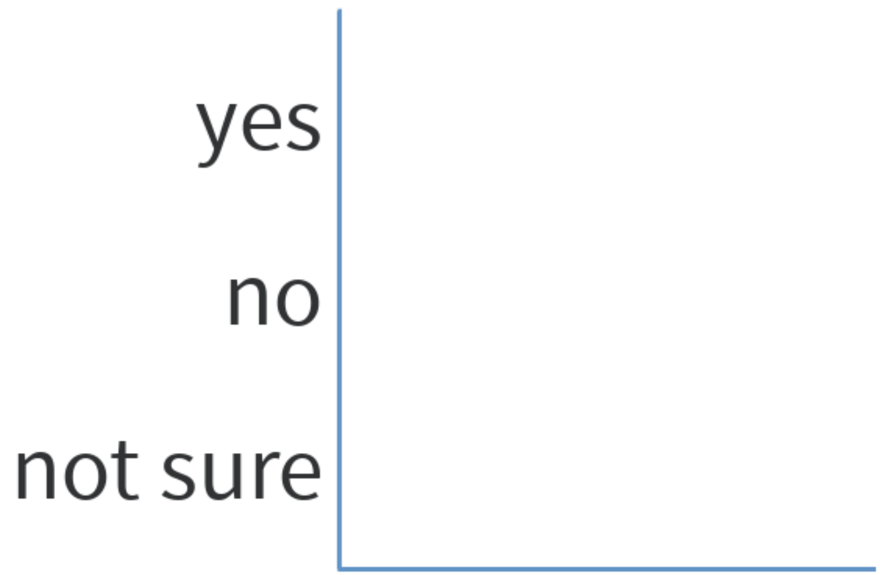
tail is not reachable from the head



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.)

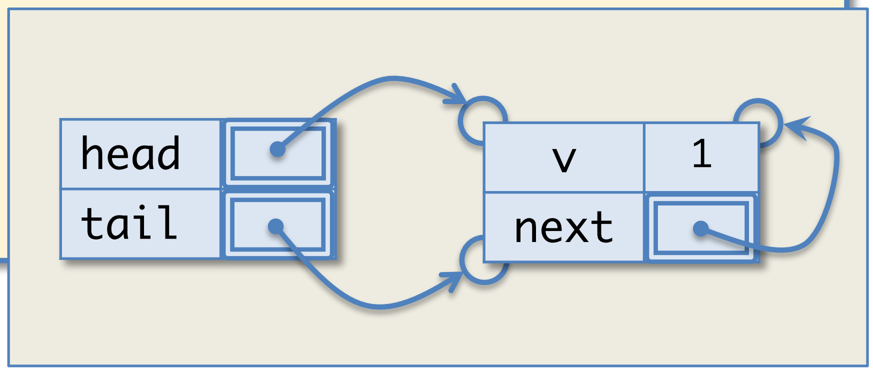
```
type 'a qnode = {
     v: 'a;
     mutable next : 'a qnode option
}

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



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.)

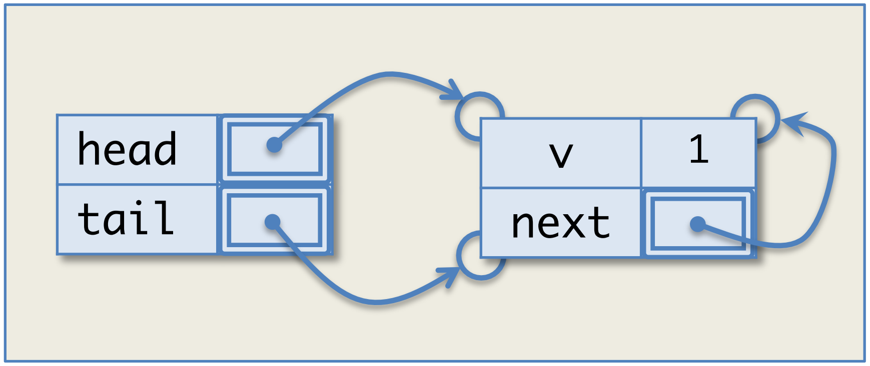
```
type 'a qnode = {
     v: 'a;
     mutable next : 'a qnode option
}

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

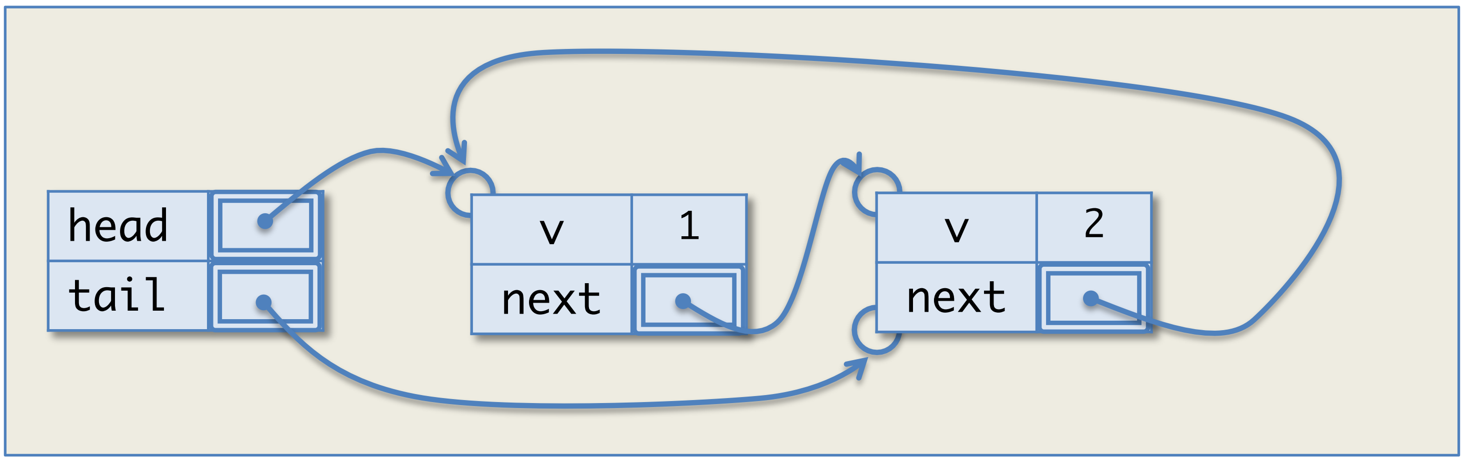
1. yes
2. no
3. not sure



Answer: 1

Cyclic `int` queue values





(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 prove that these properties suffice to 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.

Is this a valid queue?

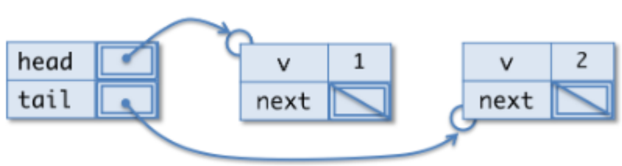
Either:

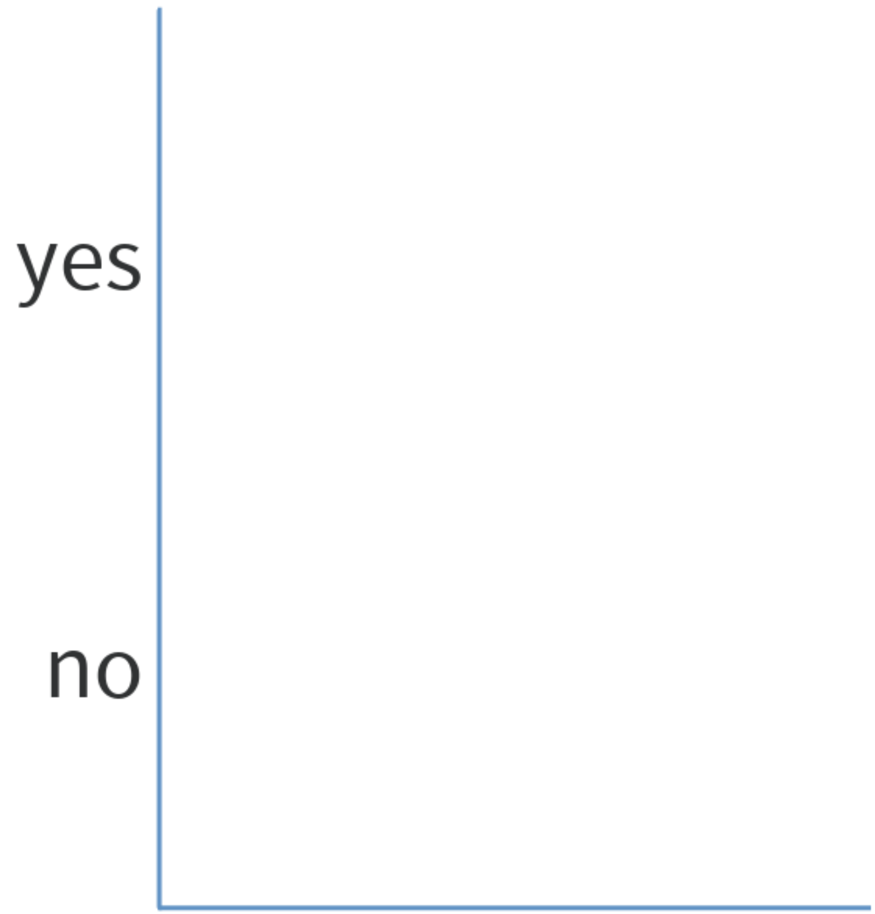
(1) `head` and `tail` are both `None` (i.e. the queue is empty)

or

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

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

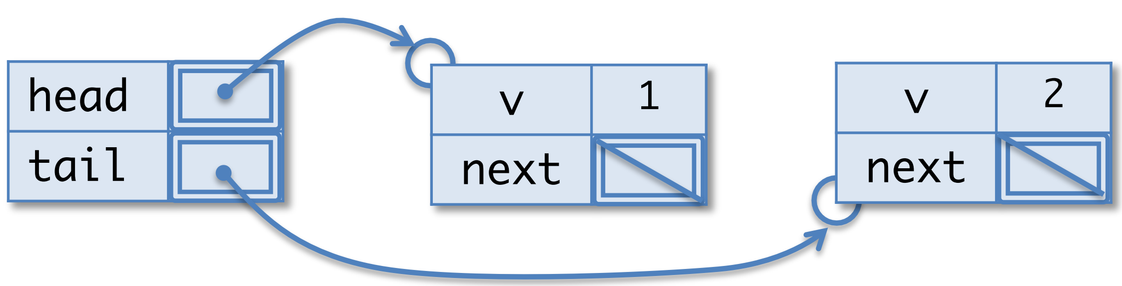




Either:
(1) head and tail are both None (i.e. the queue is empty)
or
(2) head is Some n1, tail is Some n2 and
- n2 is reachable from n1 by following 'next' pointers
- n2.next is None

Is this a valid queue?

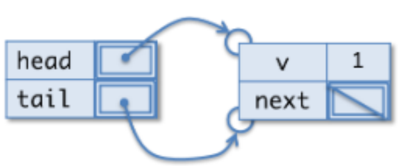
1. Yes
2. No

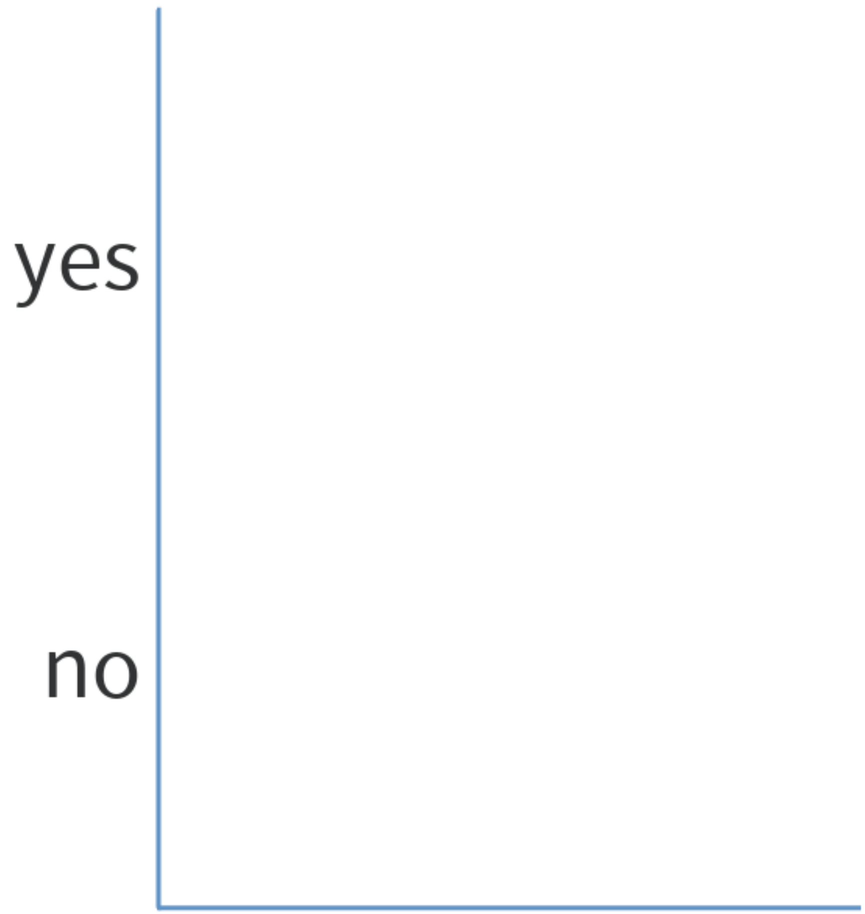


ANSWER: No

Is this a valid queue?

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





Either:

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

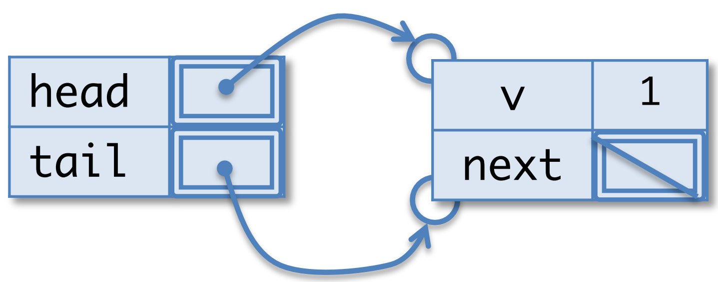
or

(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?

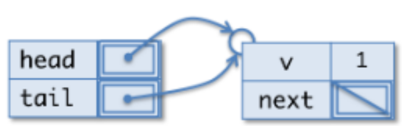
1. Yes
2. No

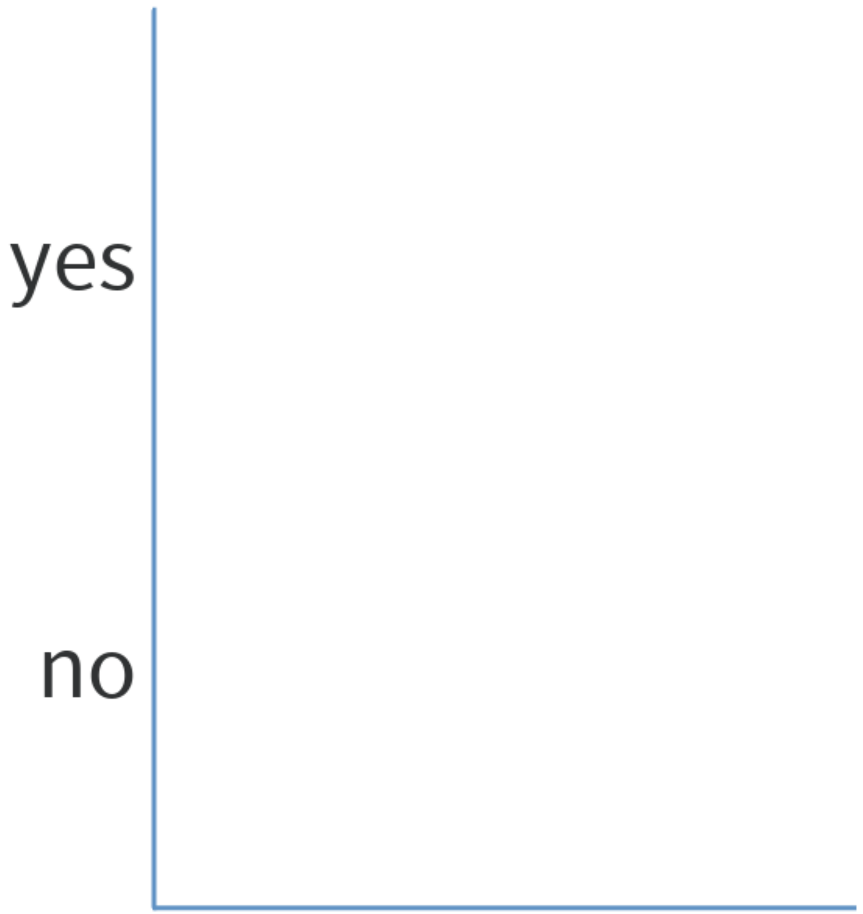


ANSWER: Yes

Is this a valid queue?

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`





Either:

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

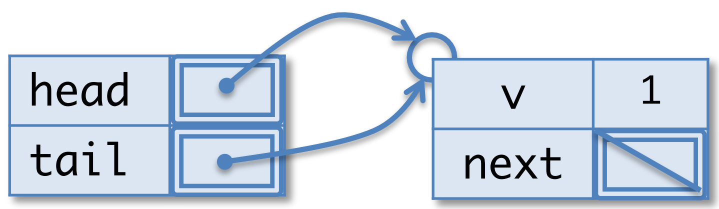
or

(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



ANSWER: Yes

Implementing Linked Queues

q.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
```

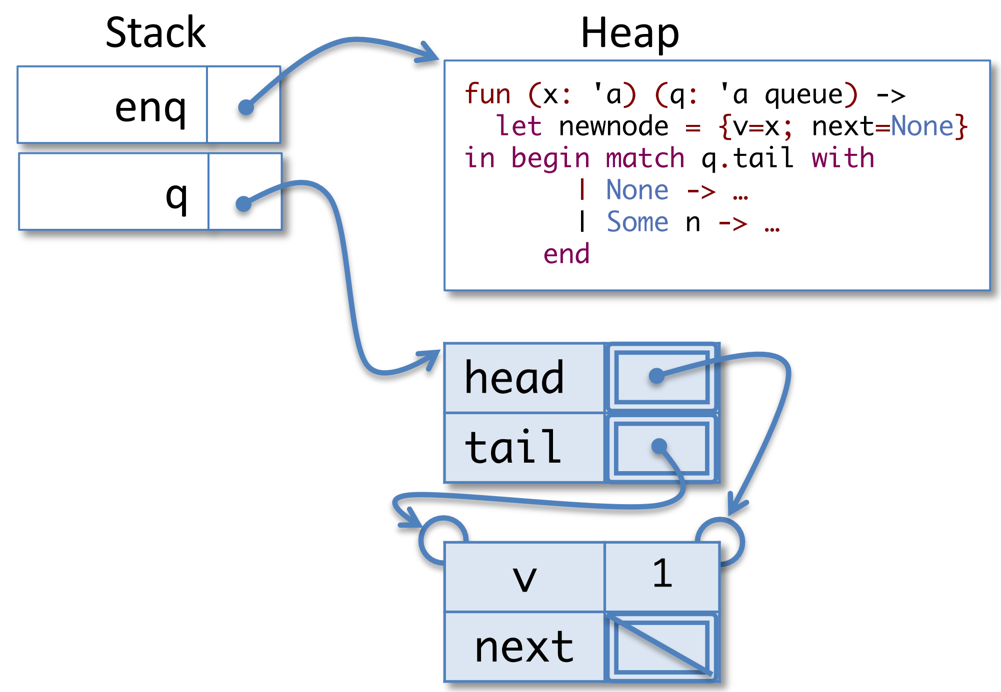
- The code for enq is informed by the queue invariant:
  - either the queue is empty, and we just update head and tail, or
  - the queue is non-empty, in which case we have to “patch up” the “next” link of the old tail node to maintain the queue invariant.

What is your current level of comfort with the Abstract Stack Machine?

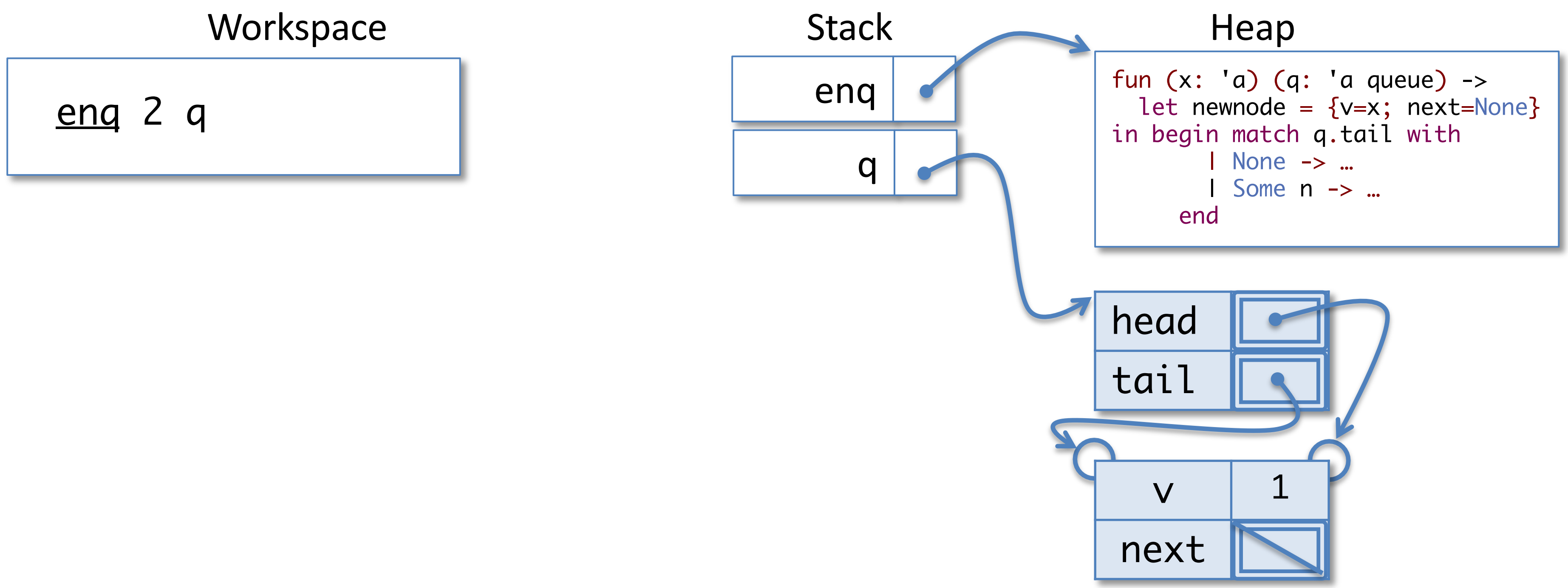
Calling Enq on a non-empty queue

Workspace

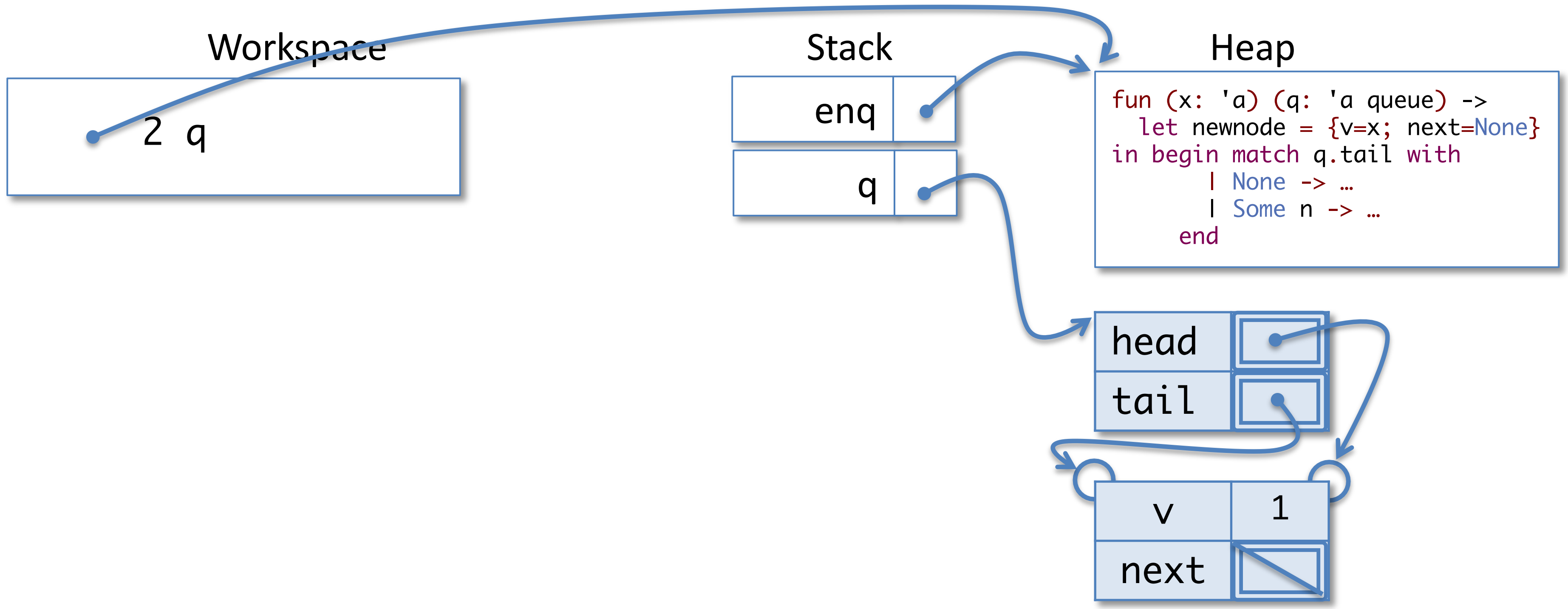
```
enq 2 q
```



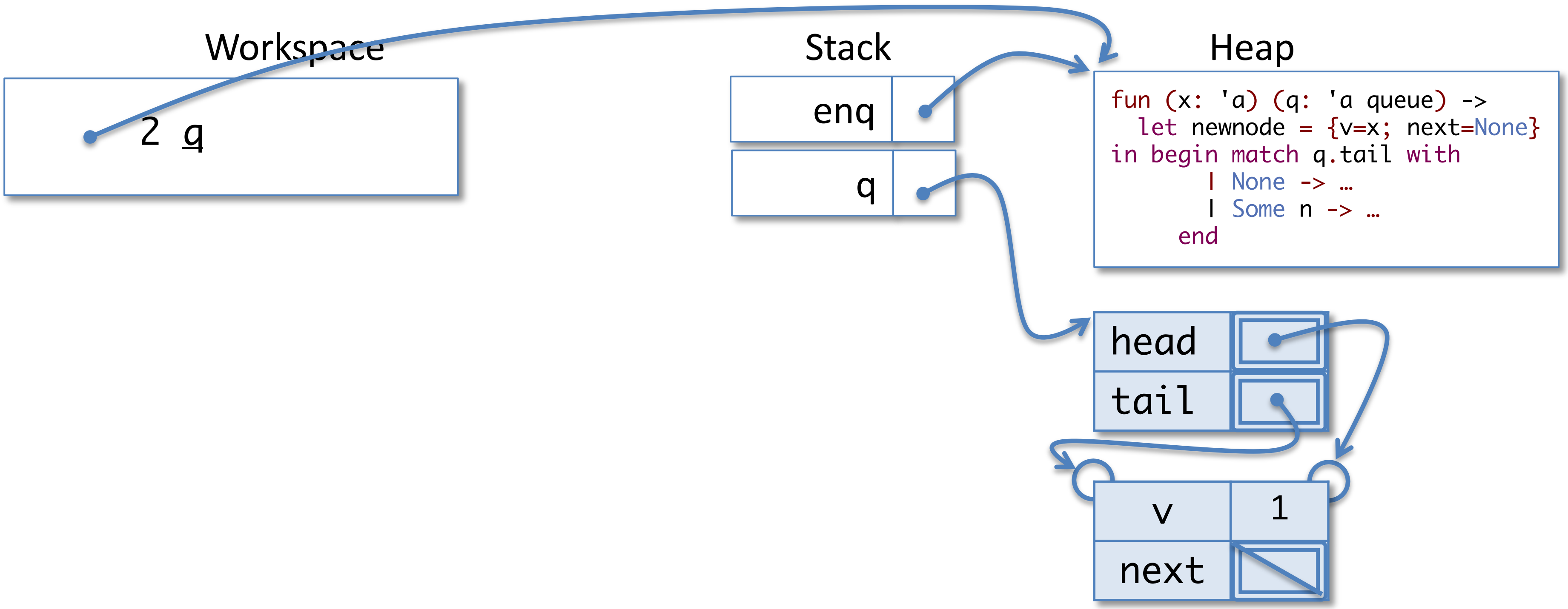
Calling Enq on a non-empty queue



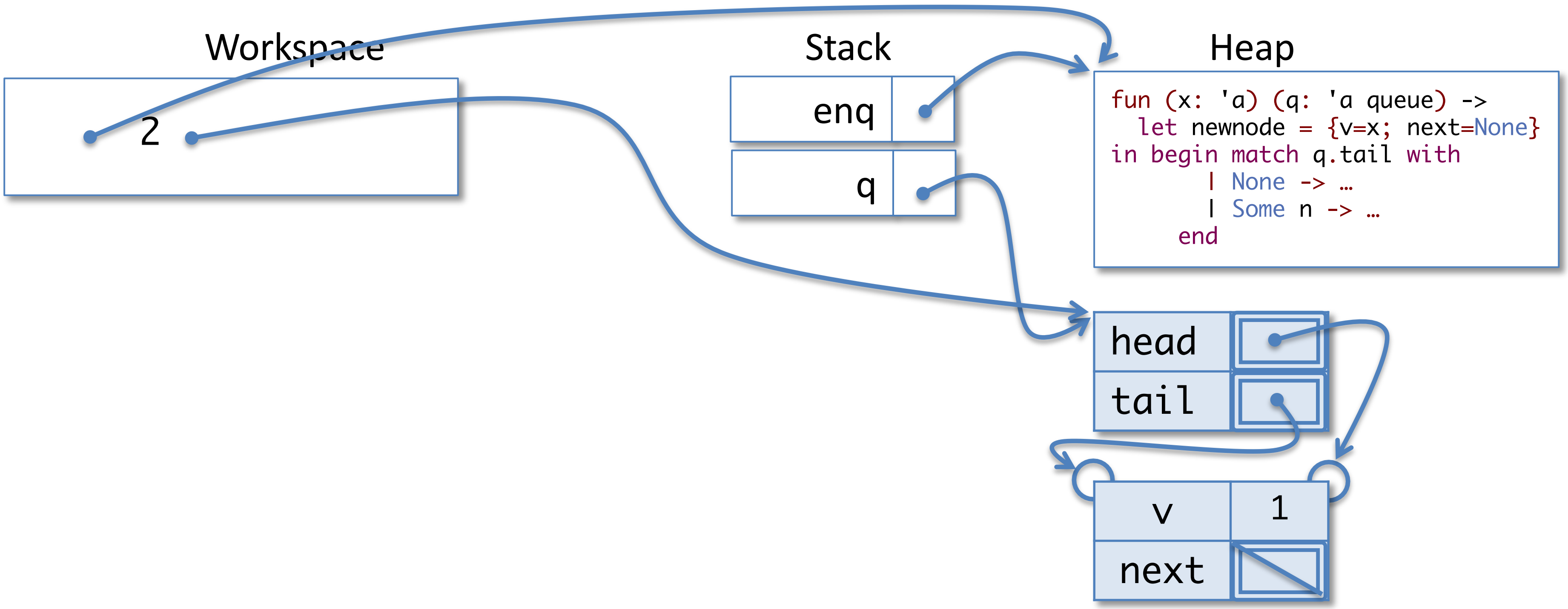
Calling Enq on a non-empty queue



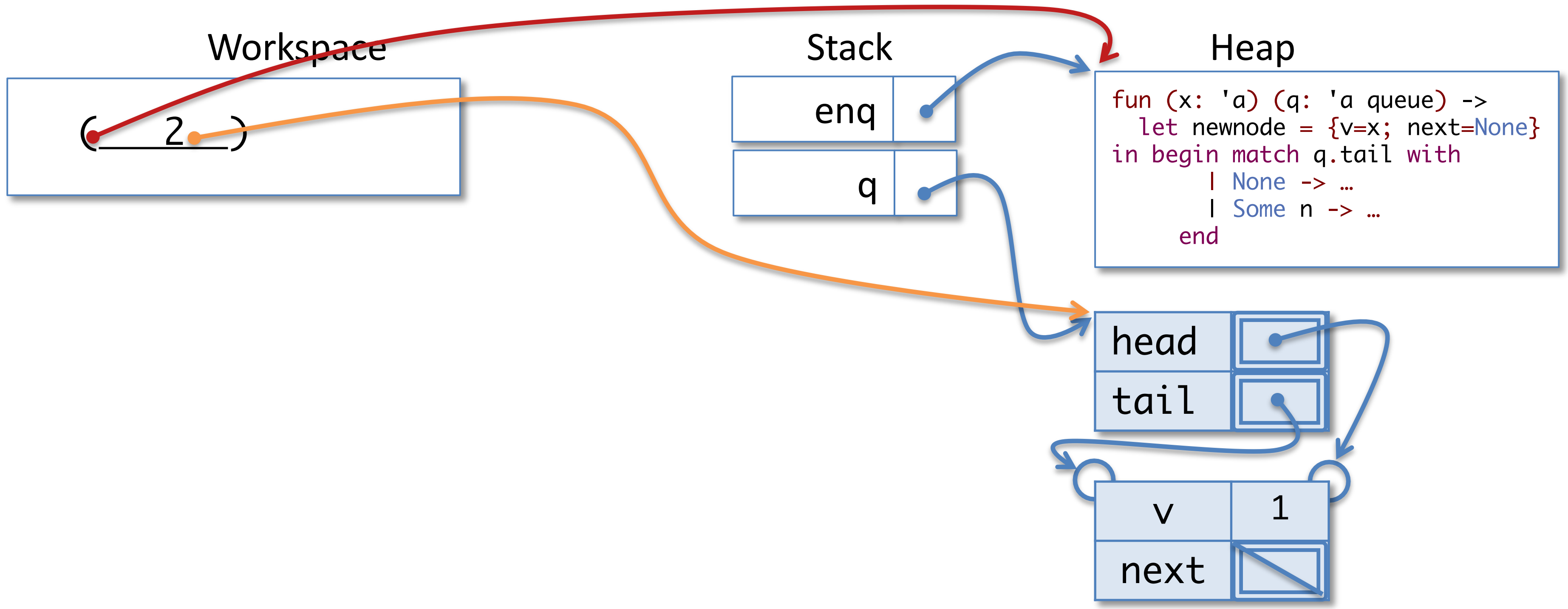
Calling Enq on a non-empty queue



Calling Enq on a non-empty queue



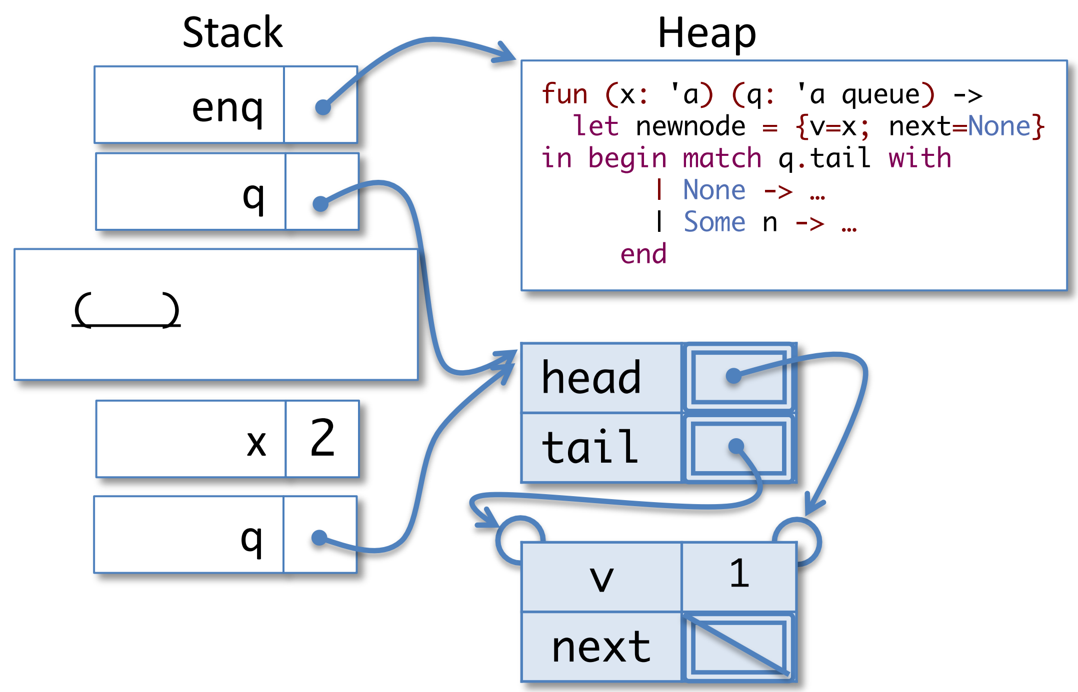
Calling Enq on a non-empty queue



Calling Enq on a non-empty queue

Workspace

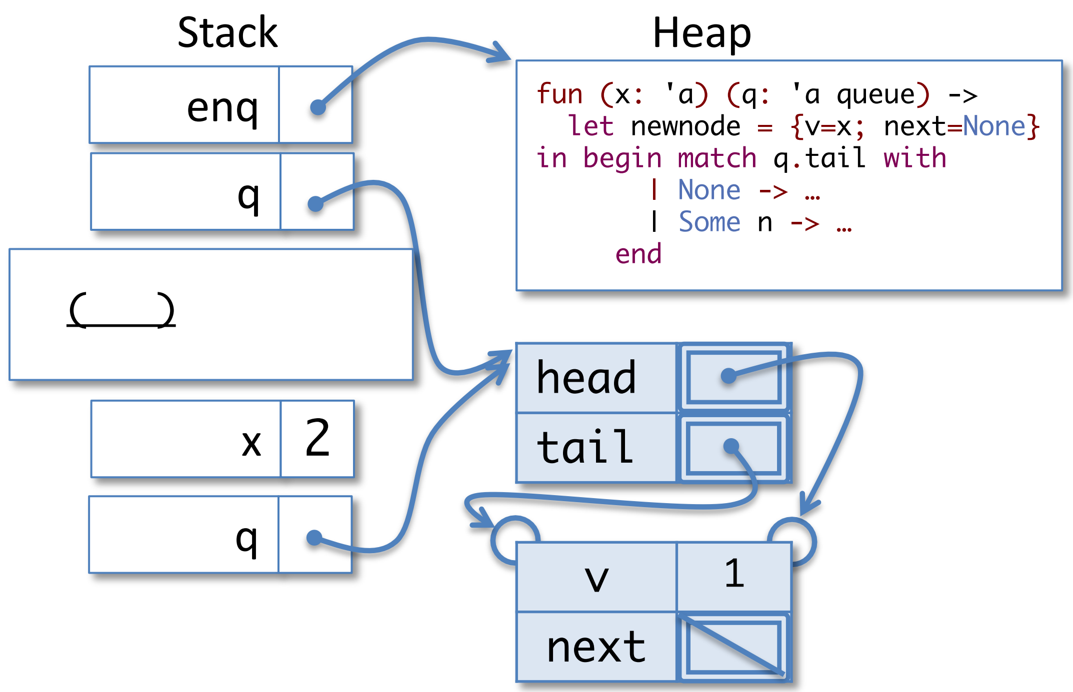
```
let newnode = {v=x; next=None} in
  begin match q.tail with
    | None ->
      q.head <- Some newnode;
      q.tail <- Some newnode
    | Some n ->
      n.next <- Some newnode;
      q.tail <- Some newnode
  end
```



Calling Enq on a non-empty queue

Workspace

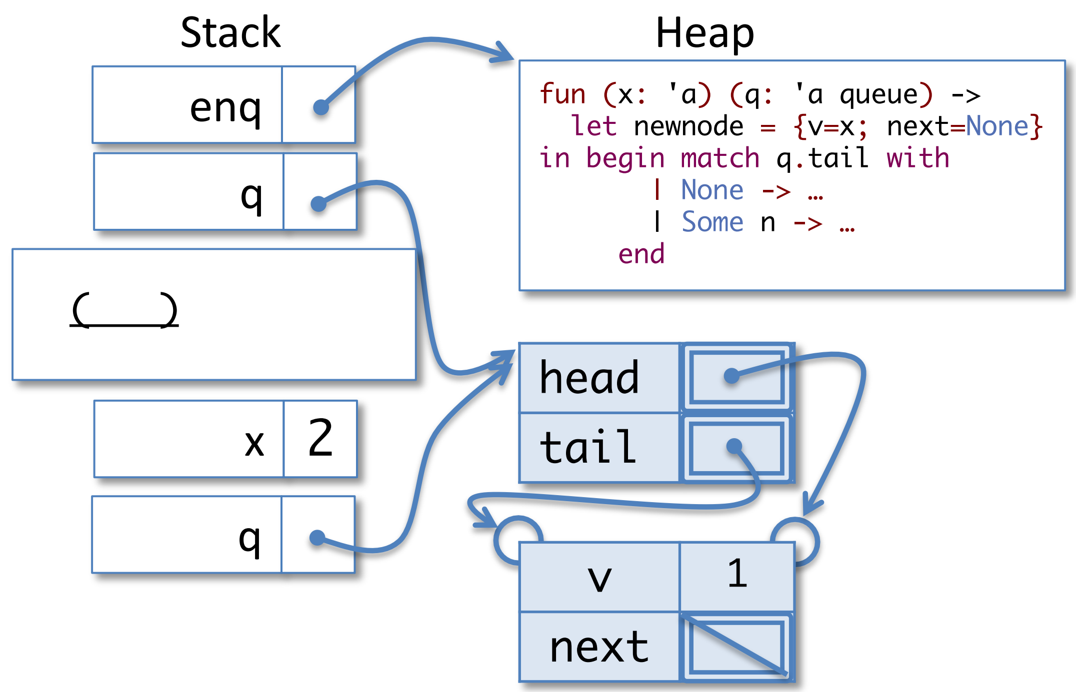
```
let newnode = {v=x; next=None} in
  begin match q.tail with
    | None ->
      q.head <- Some newnode;
      q.tail <- Some newnode
    | Some n ->
      n.next <- Some newnode;
      q.tail <- Some newnode
  end
```



Calling Enq on a non-empty queue

Workspace

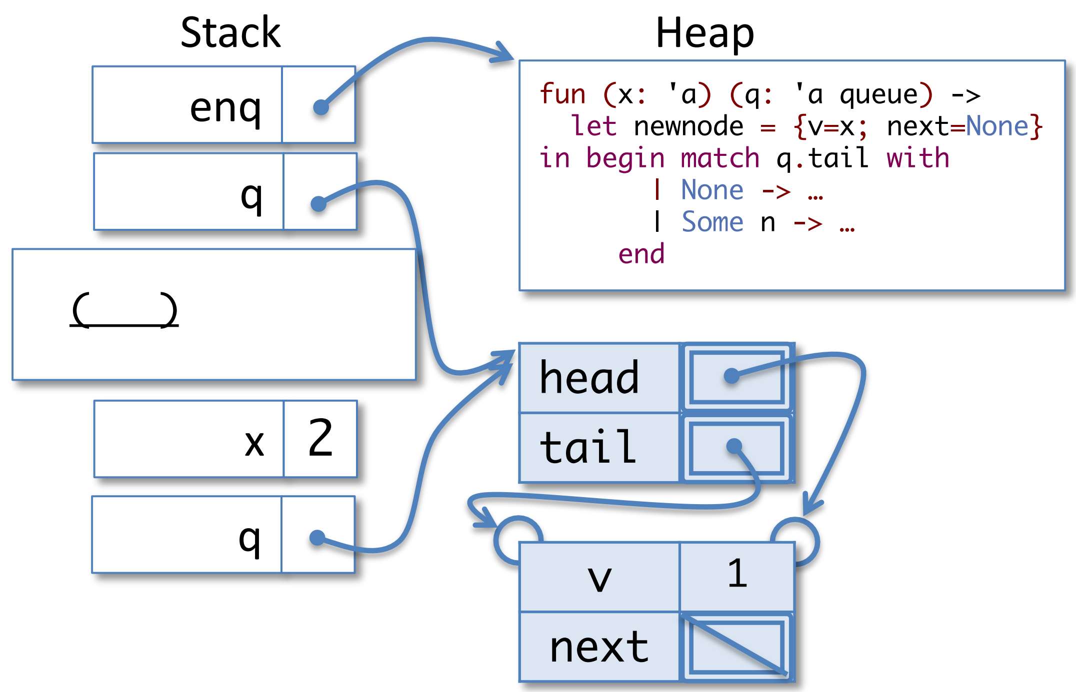
```
let newnode = {v=2; next=None} in
  begin match q.tail with
    | None ->
      q.head <- Some newnode;
      q.tail <- Some newnode
    | Some n ->
      n.next <- Some newnode;
      q.tail <- Some newnode
  end
```



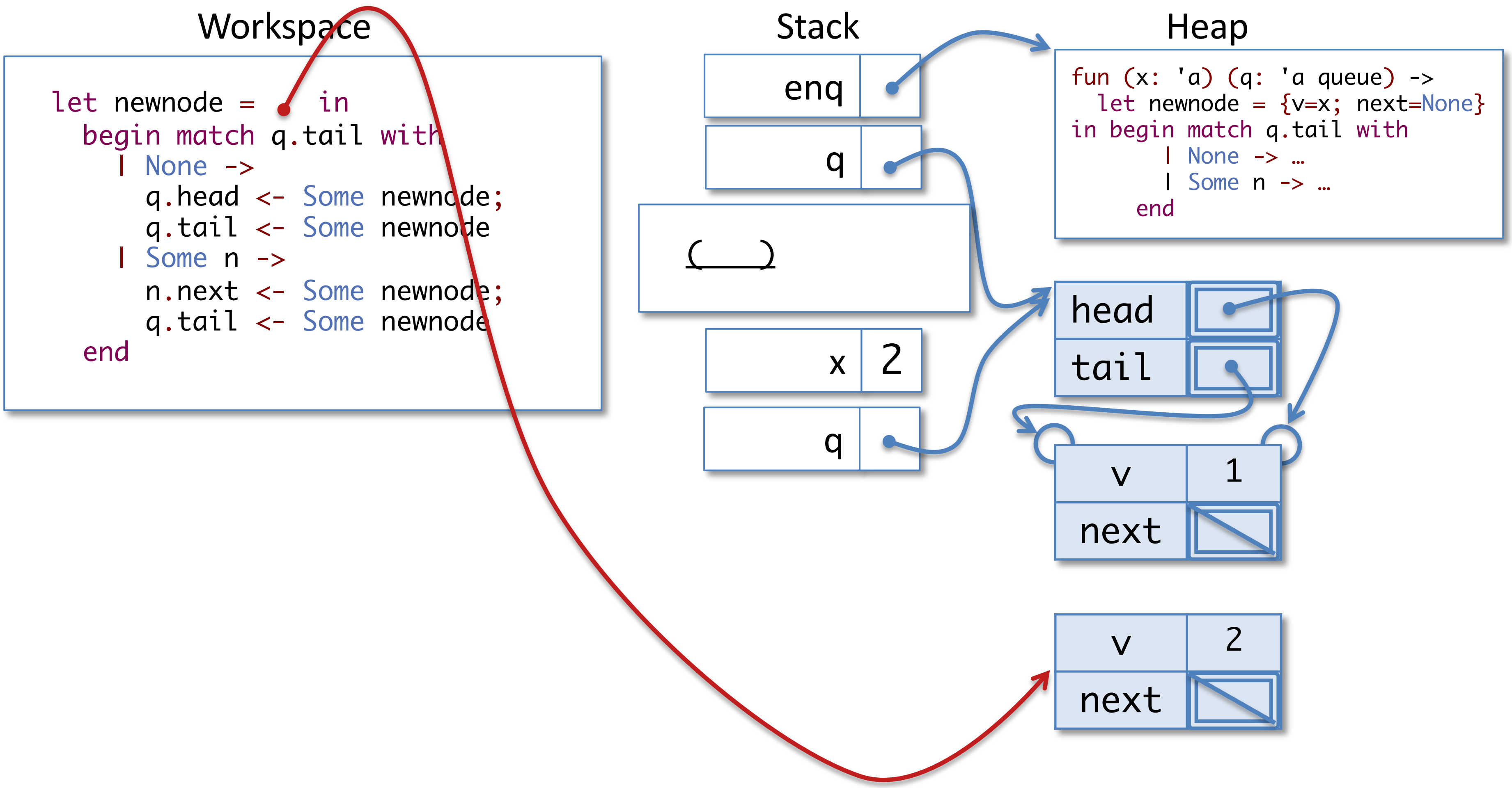
Calling Enq on a non-empty queue

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
```

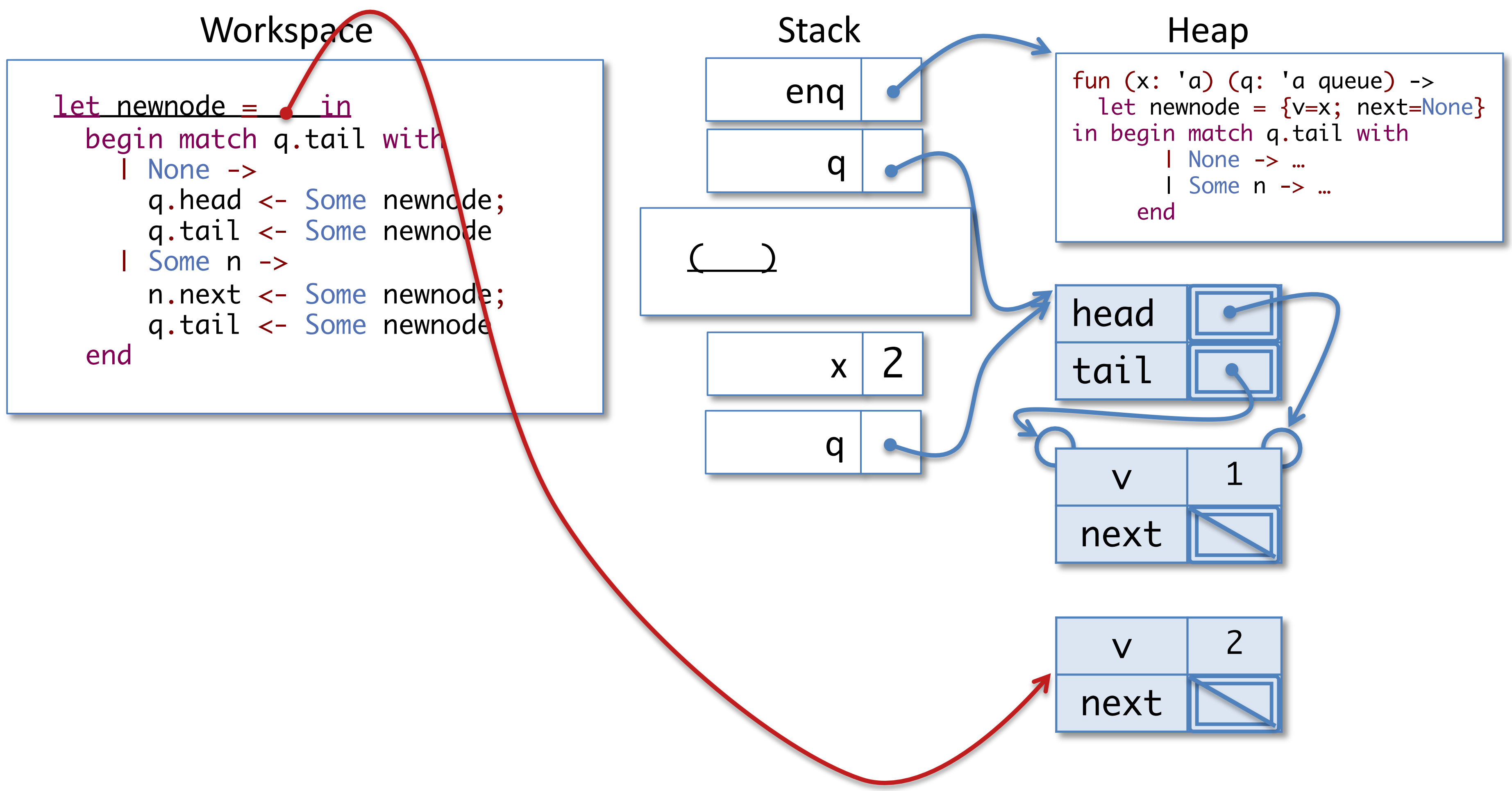


Calling Enq on a non-empty queue



Note: there is no "Some bubble": this is a qnode, not a qnode option.

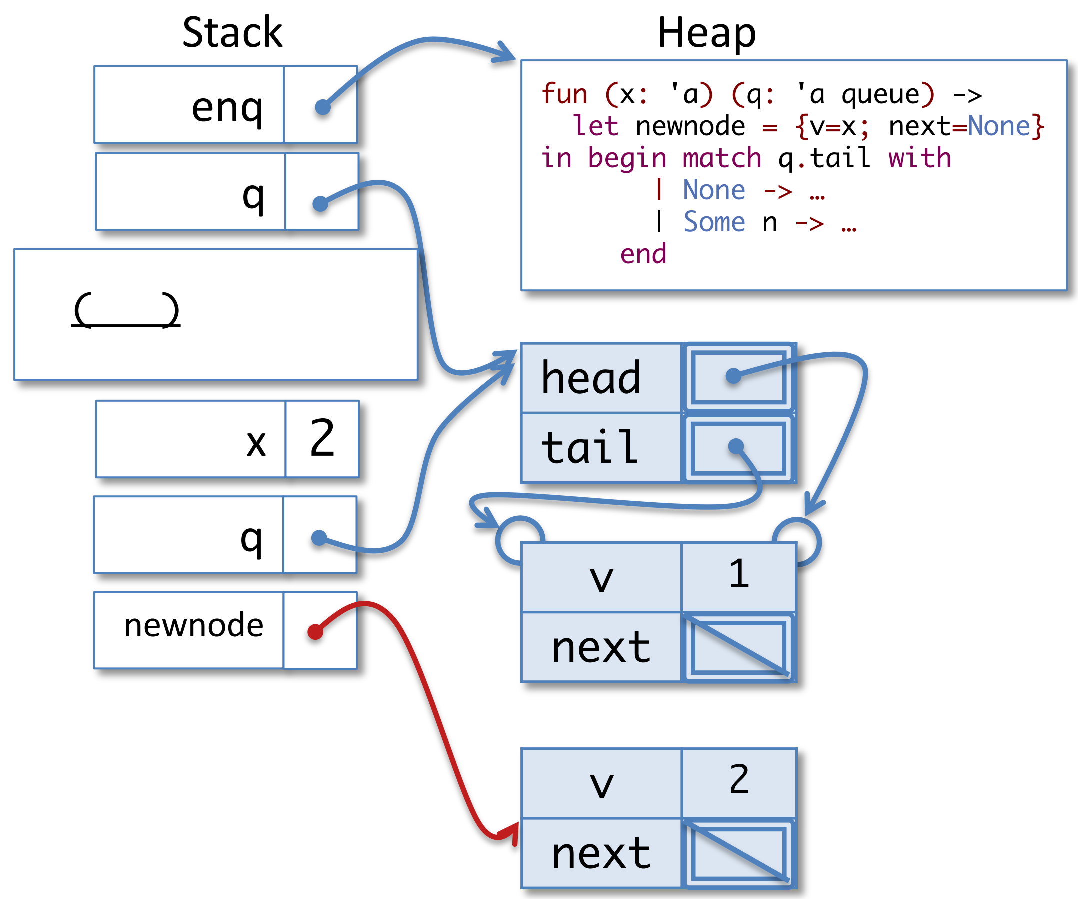
Calling Enq on a non-empty queue



Calling Enq on a non-empty queue

Workspace

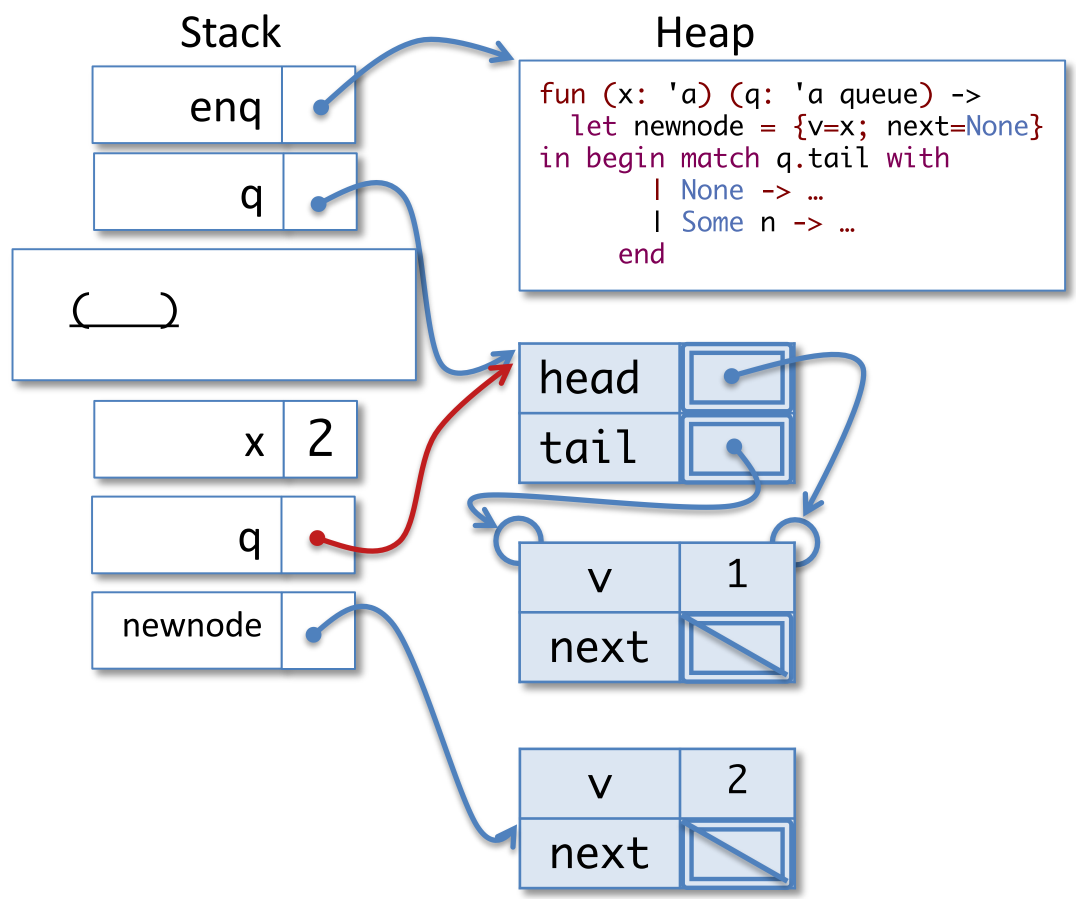
```
begin match q.tail with
  | None ->
    q.head <- Some newnode;
    q.tail <- Some newnode
  | Some n ->
    n.next <- Some newnode;
    q.tail <- Some newnode
end
```



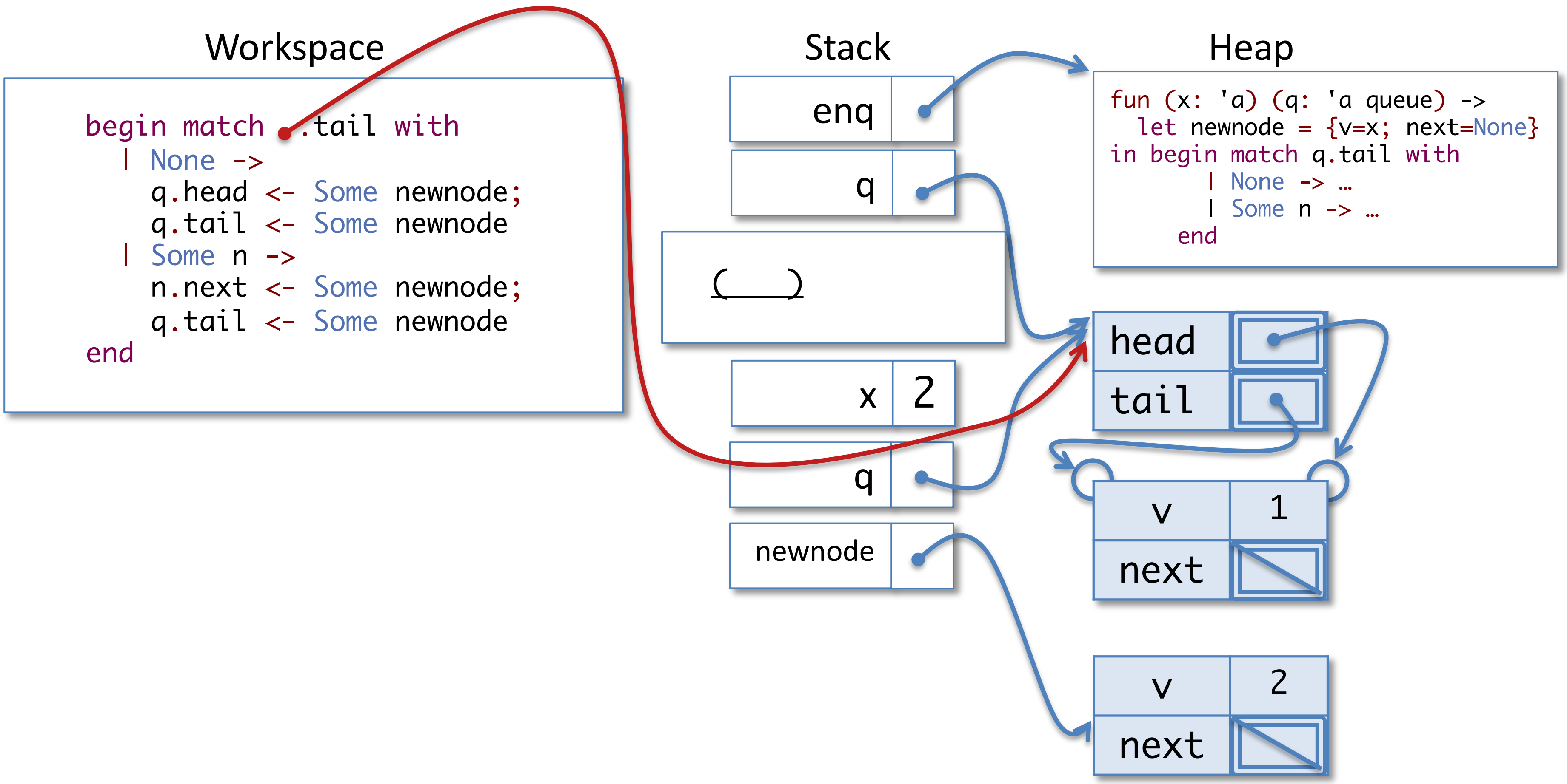
Calling Enq on a non-empty queue

Workspace

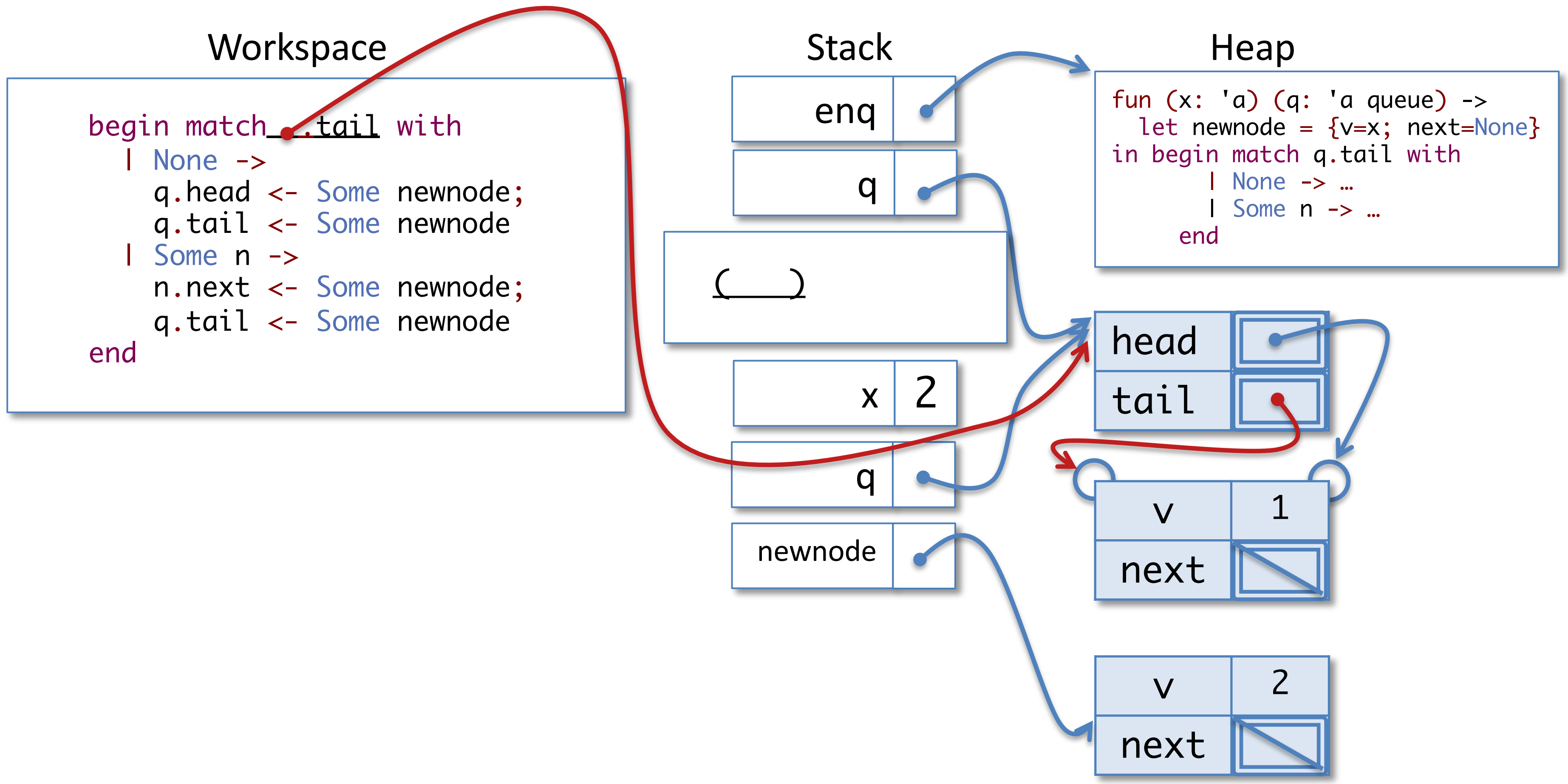
```
begin match q.tail with
  | None ->
    q.head <- Some newnode;
    q.tail <- Some newnode
  | Some n ->
    n.next <- Some newnode;
    q.tail <- Some newnode
end
```



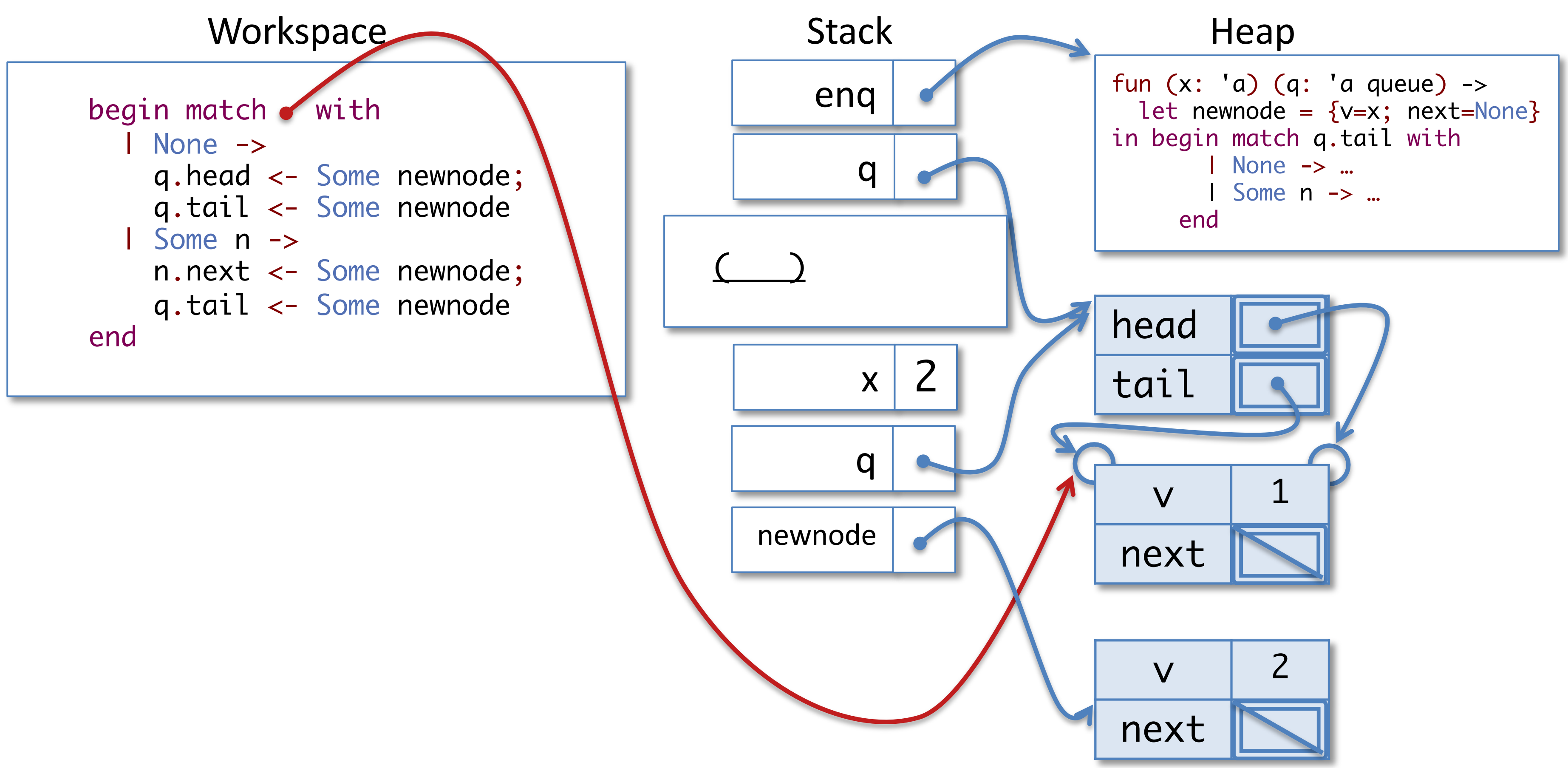
Calling Enq on a non-empty queue



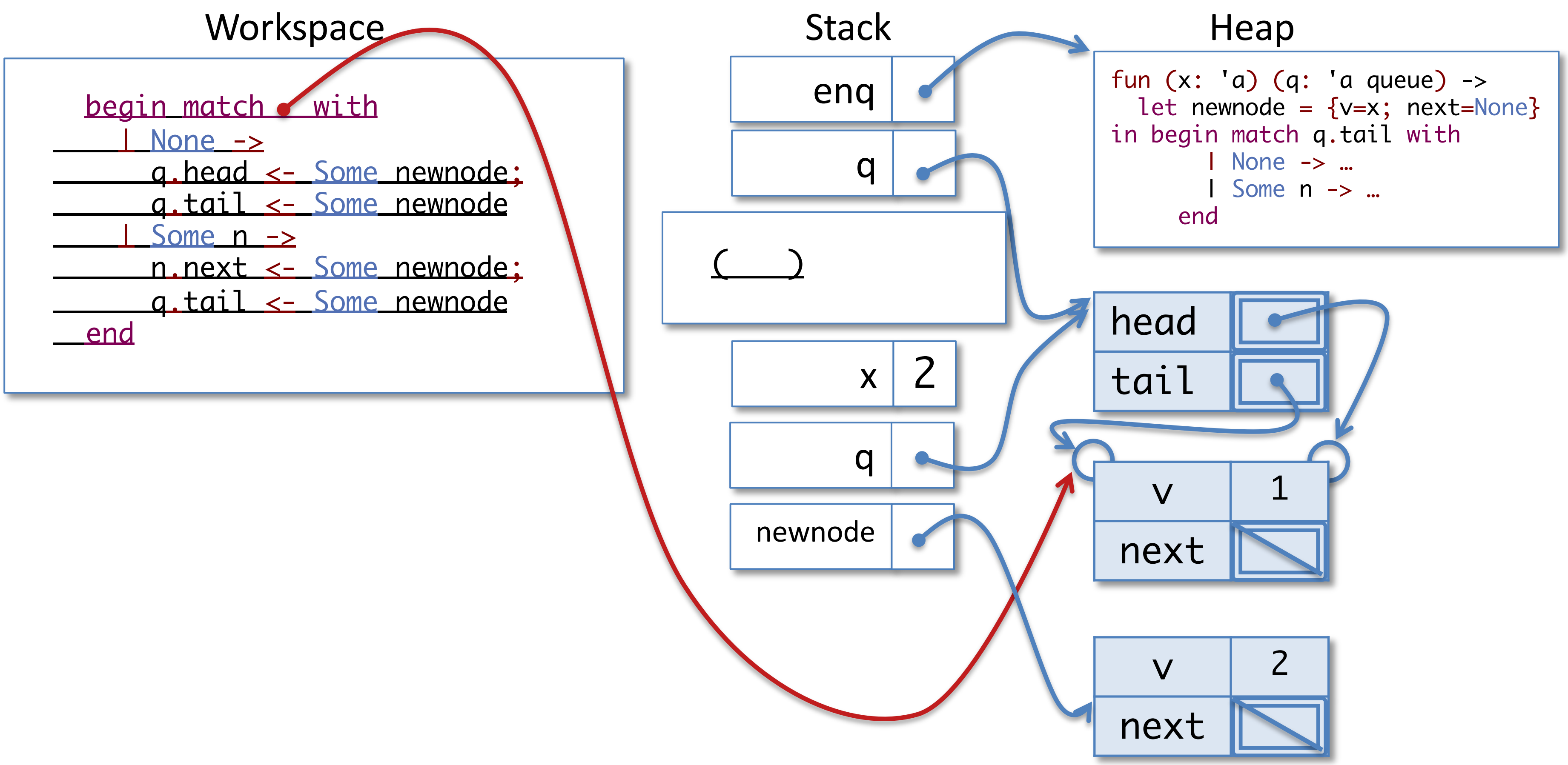
Calling Enq on a non-empty queue



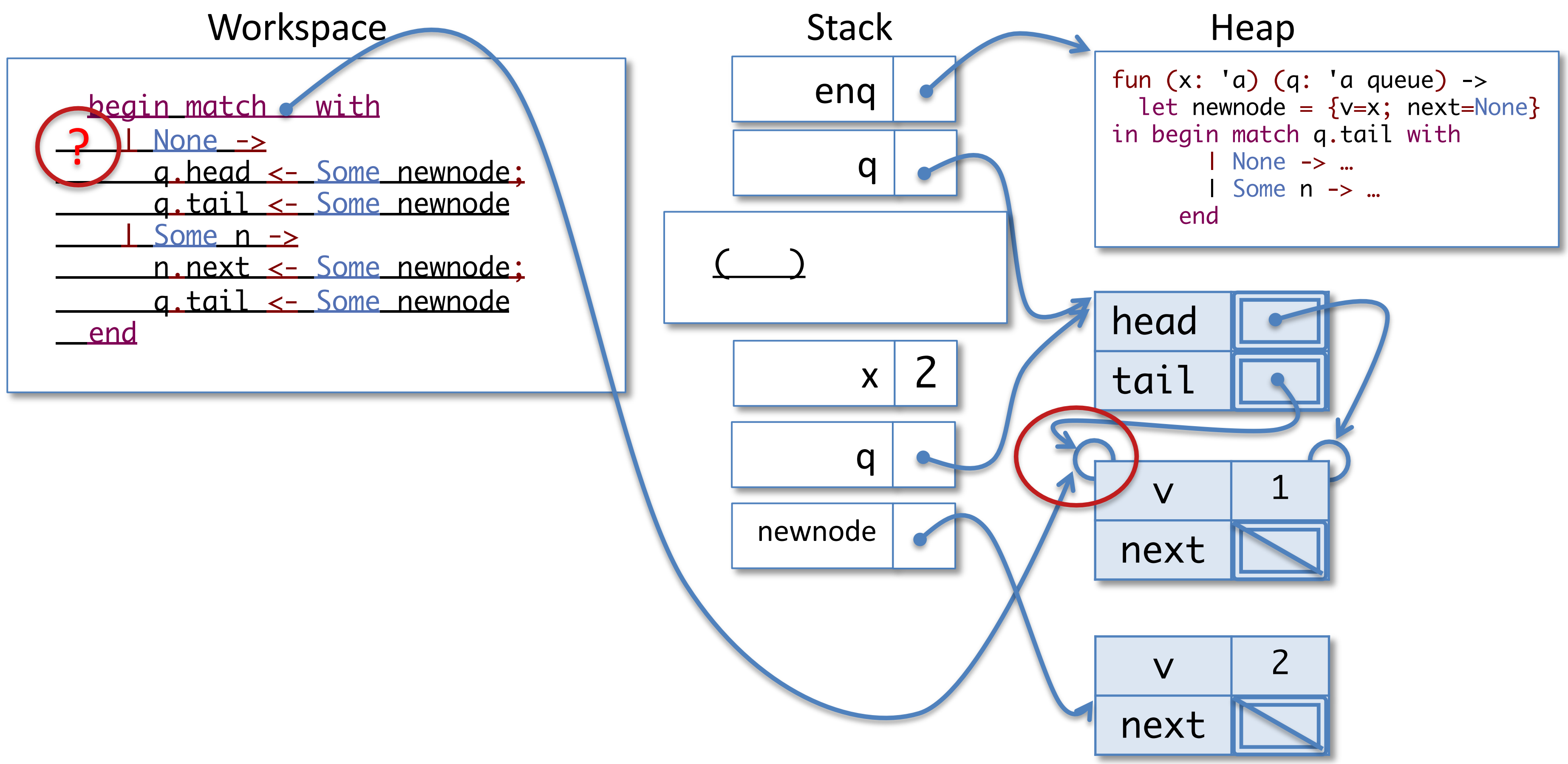
Calling Enq on a non-empty queue



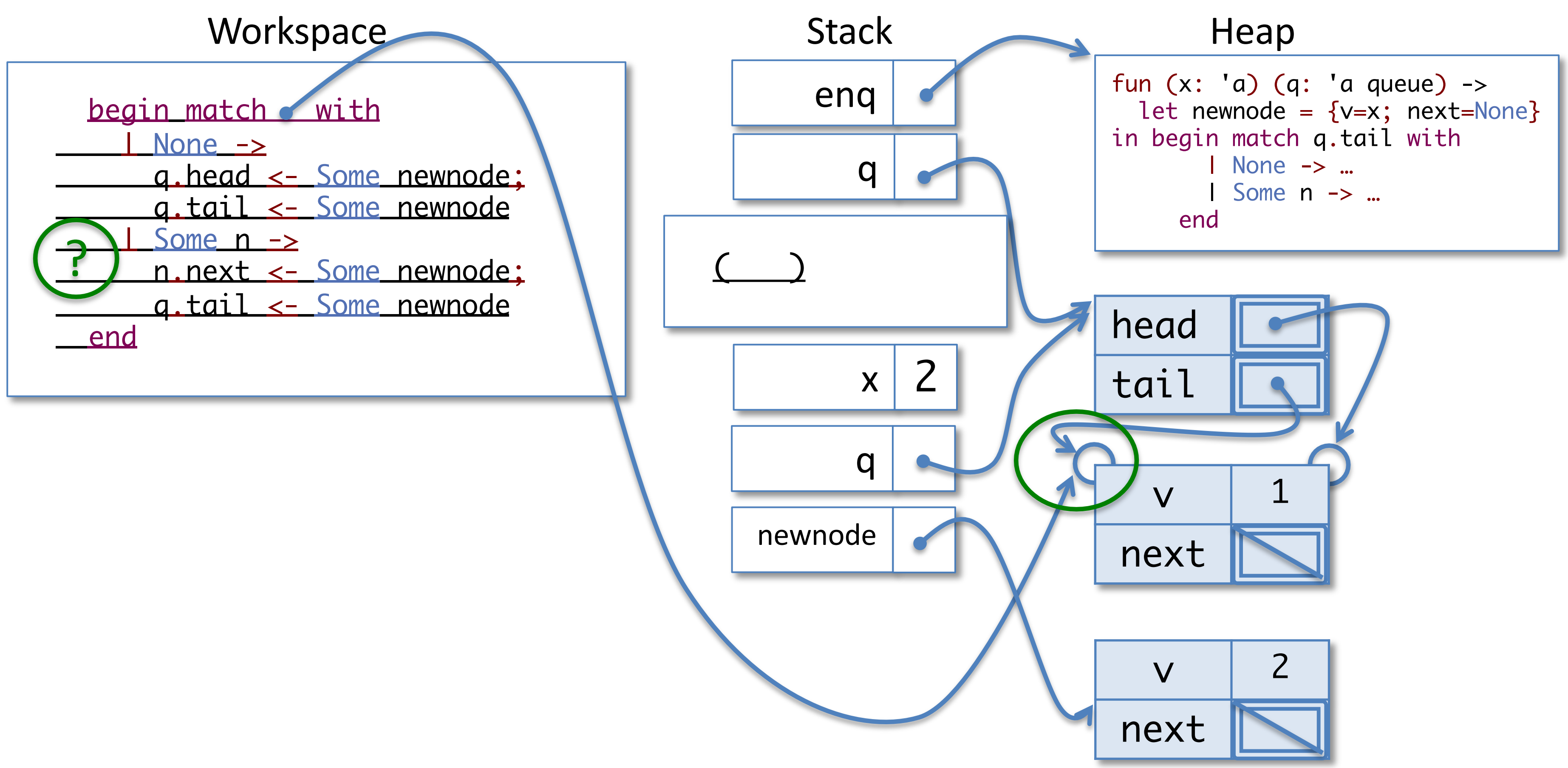
Calling Enq on a non-empty queue



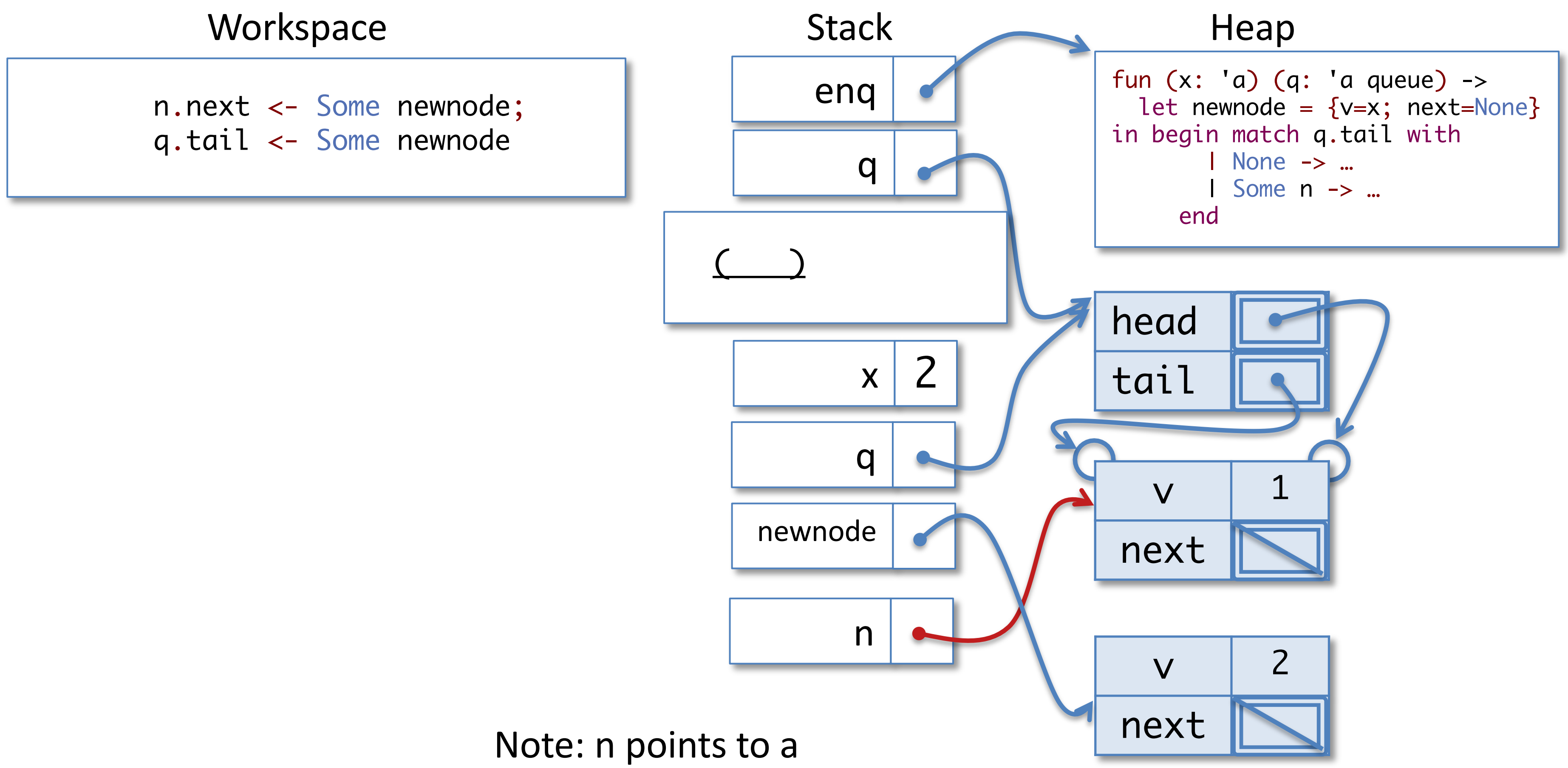
Calling Enq on a non-empty queue



Calling Enq on a non-empty queue

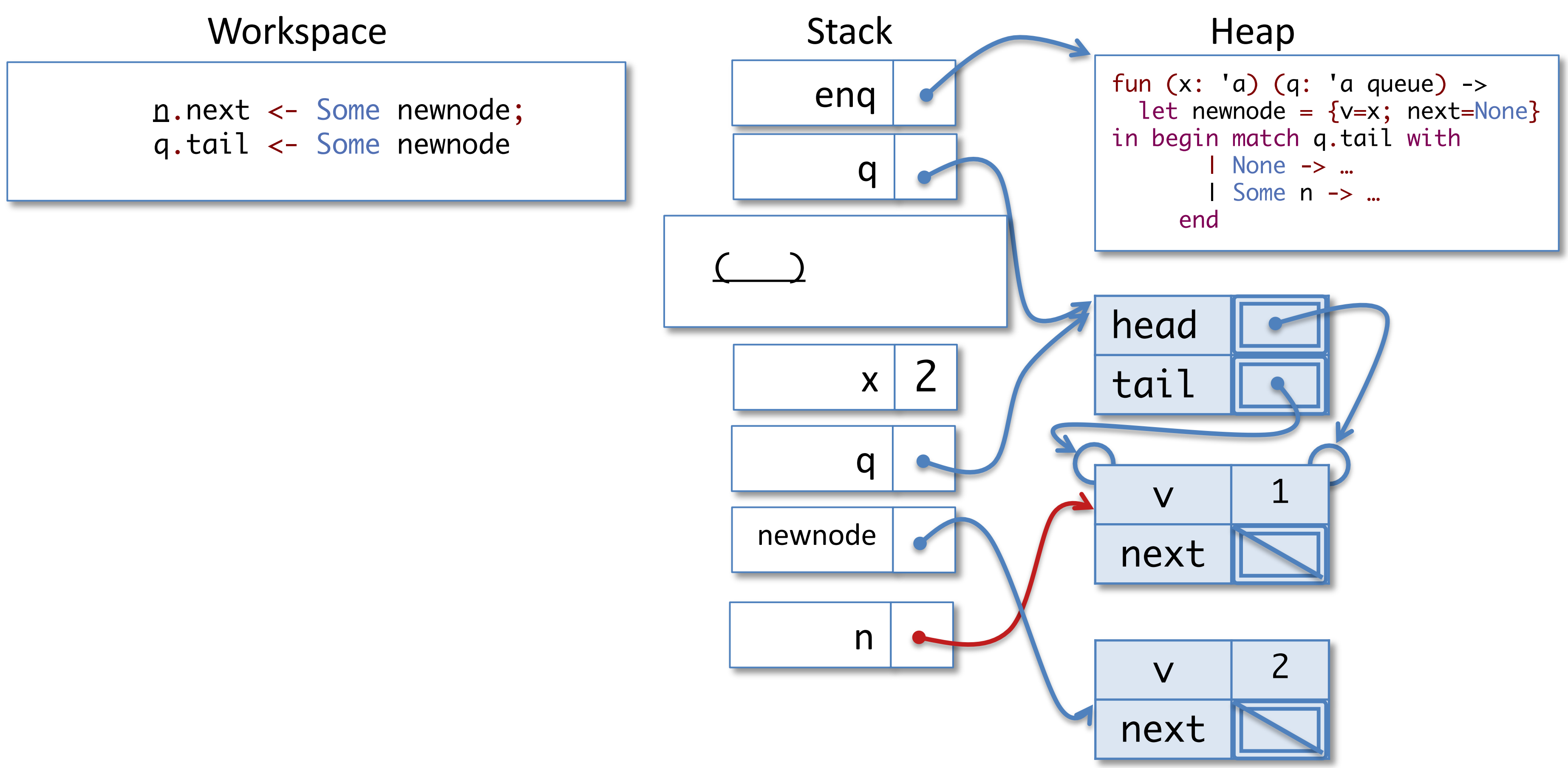


Calling Enq on a non-empty queue

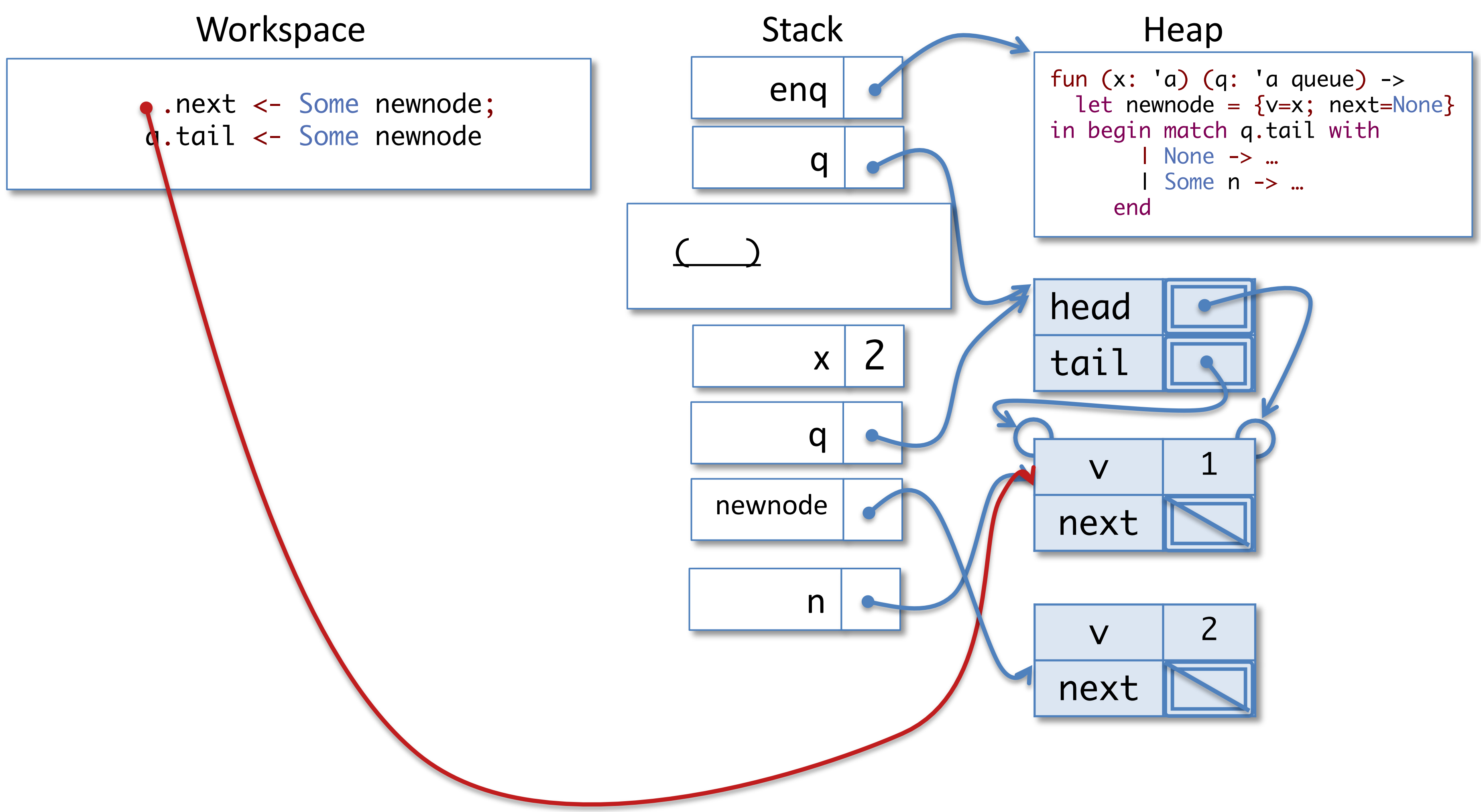


Note: n points to a qnode, not a qnode option.

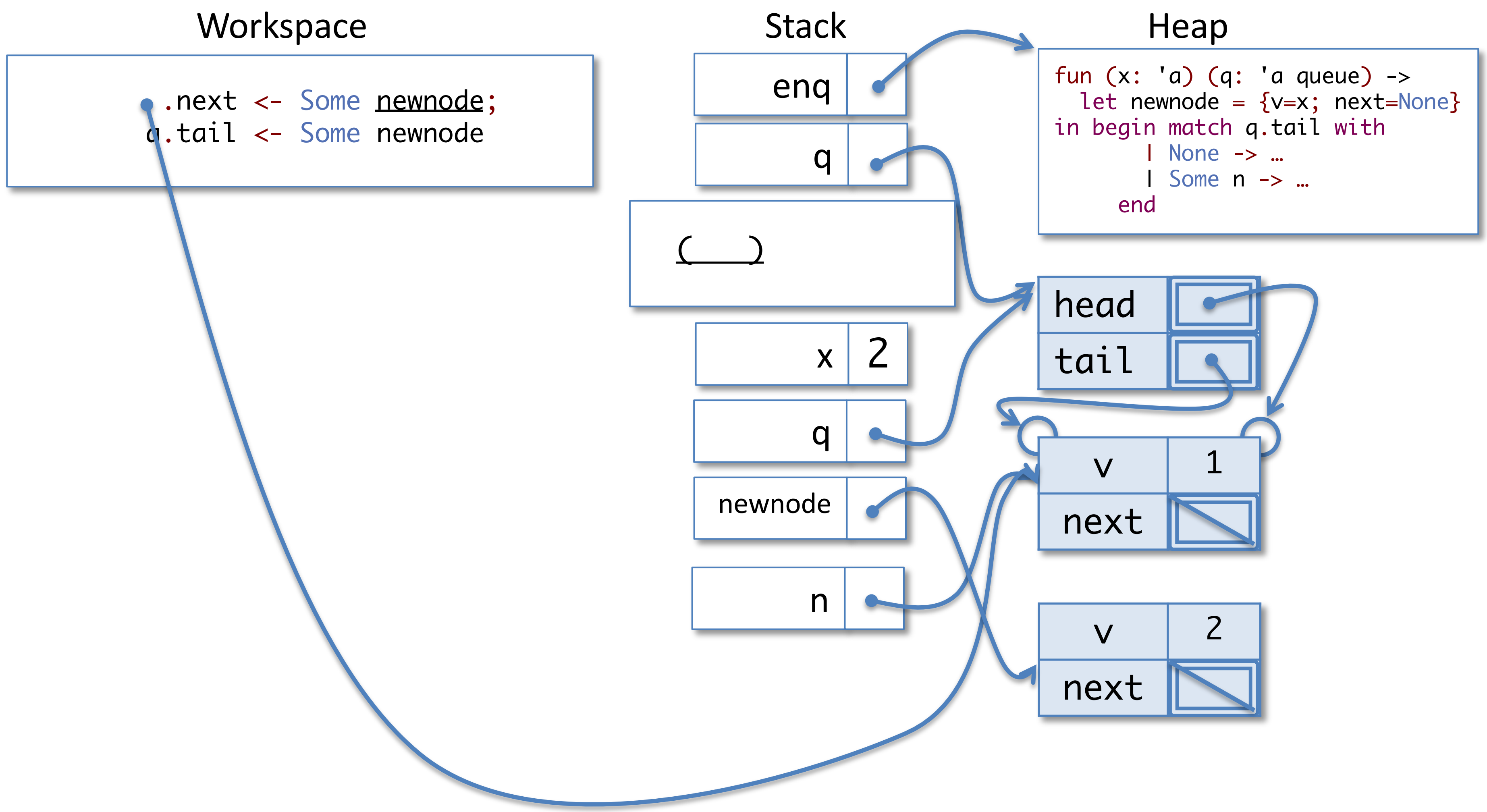
Calling Enq on a non-empty queue



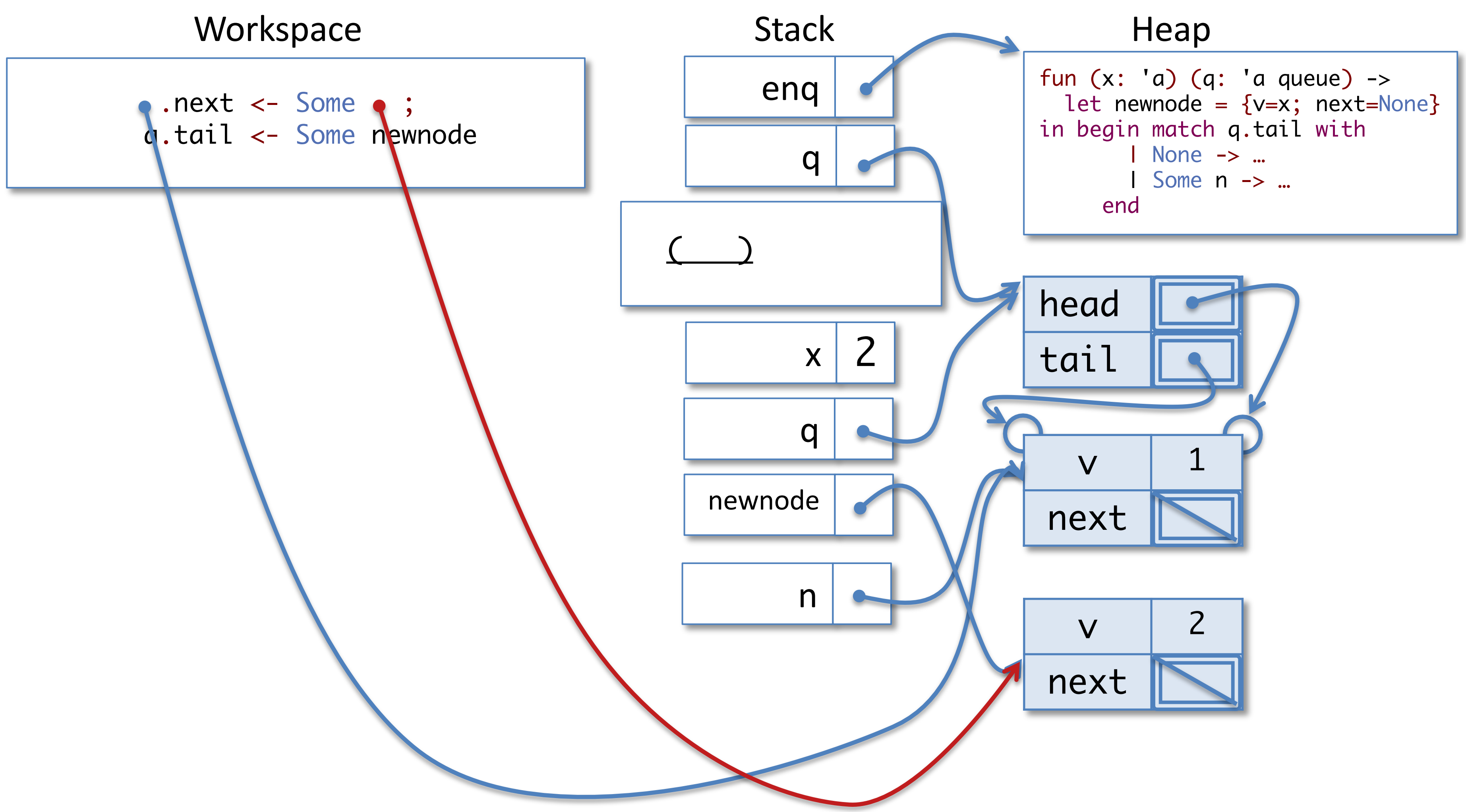
Calling Enq on a non-empty queue



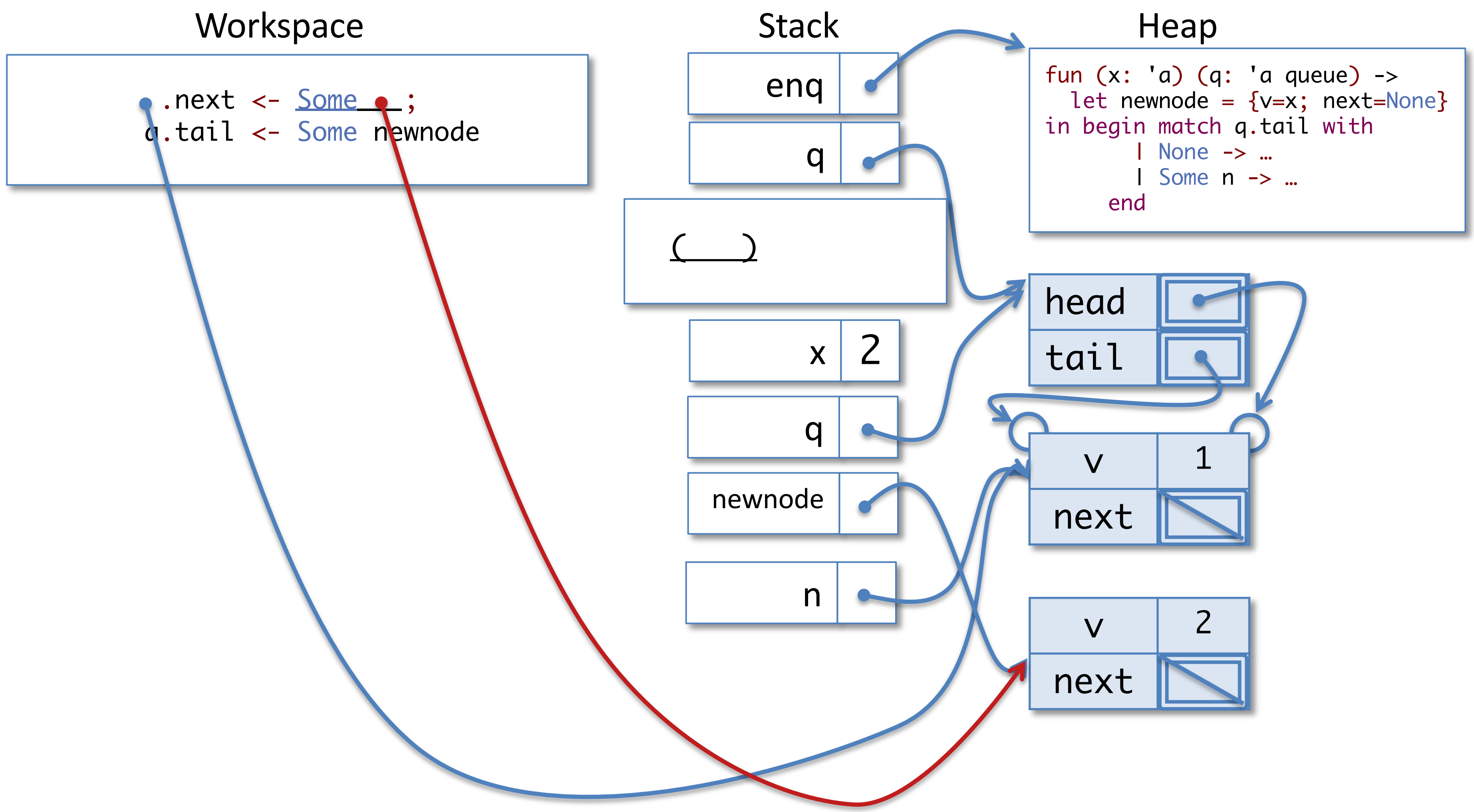
Calling Enq on a non-empty queue



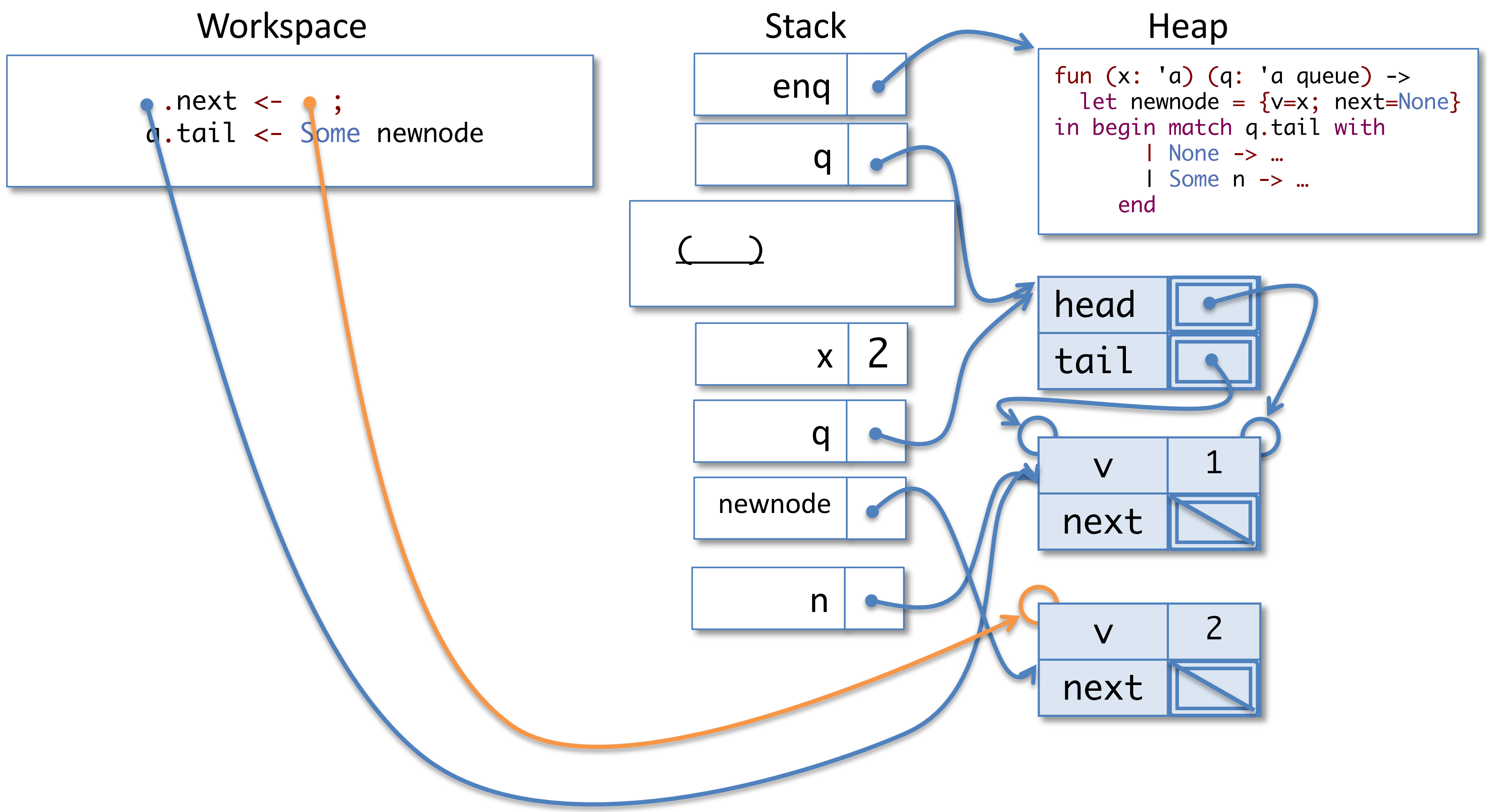
Calling Enq on a non-empty queue



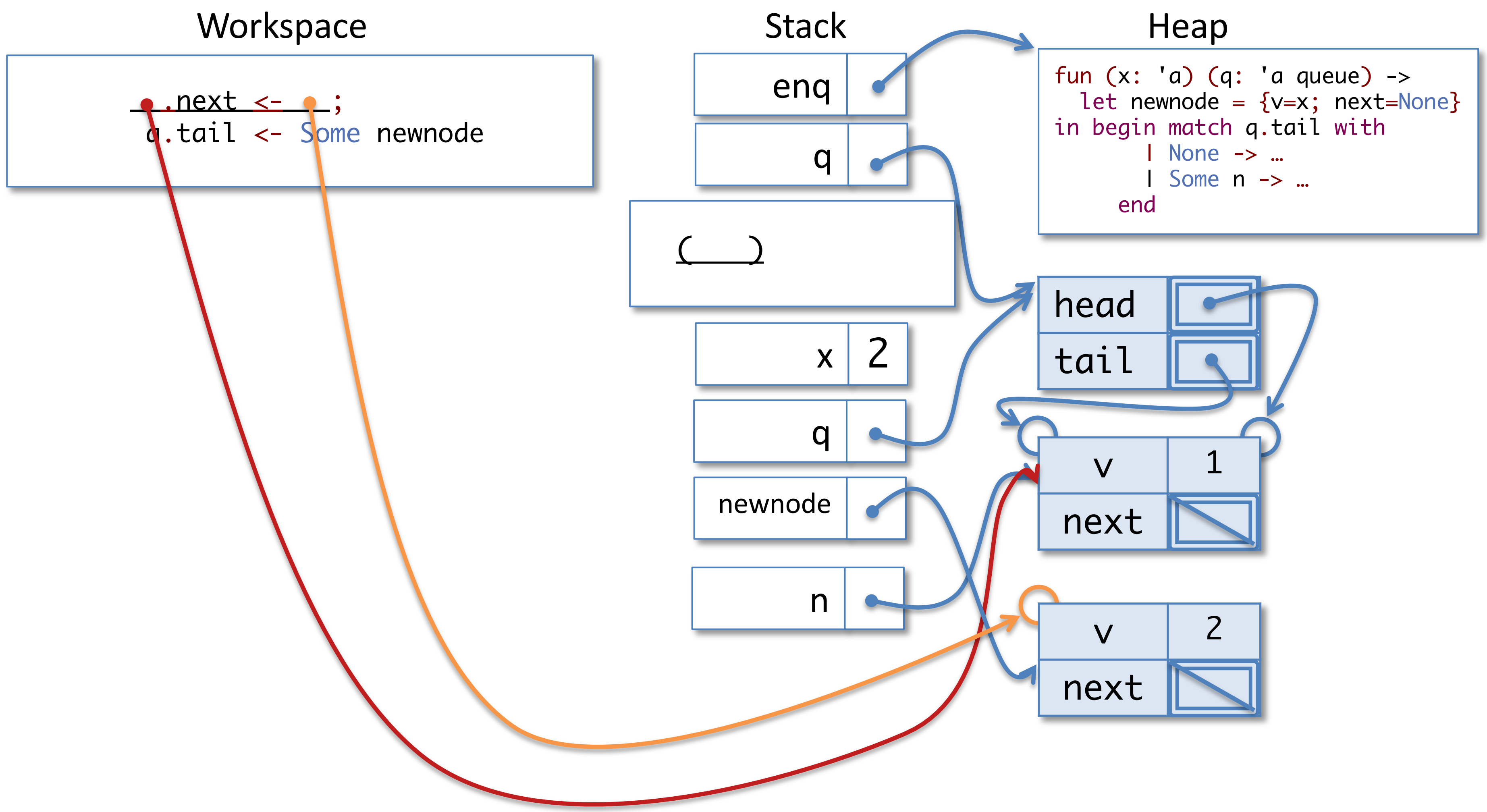
Calling Enq on a non-empty queue



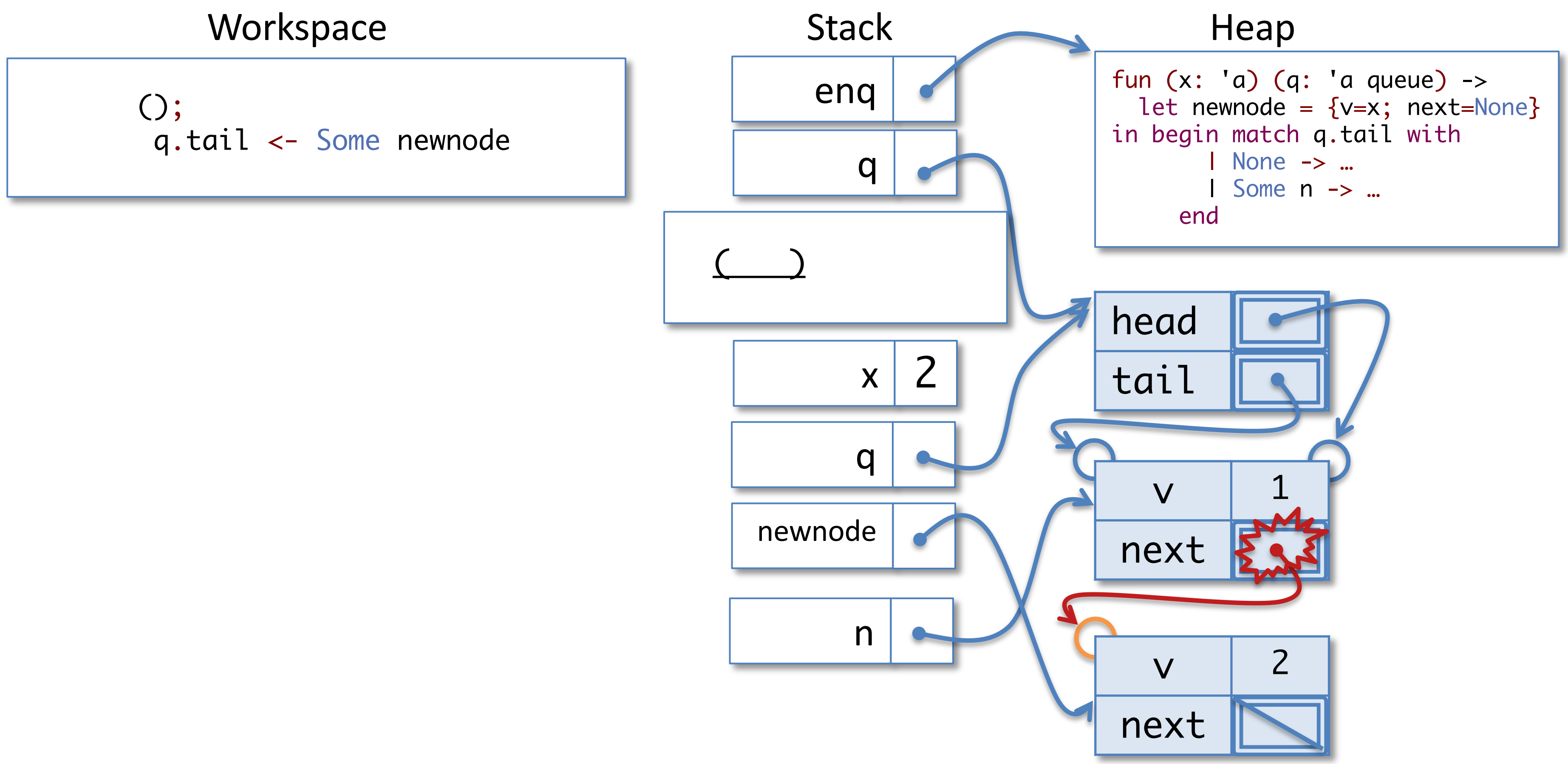
Calling Enq on a non-empty queue



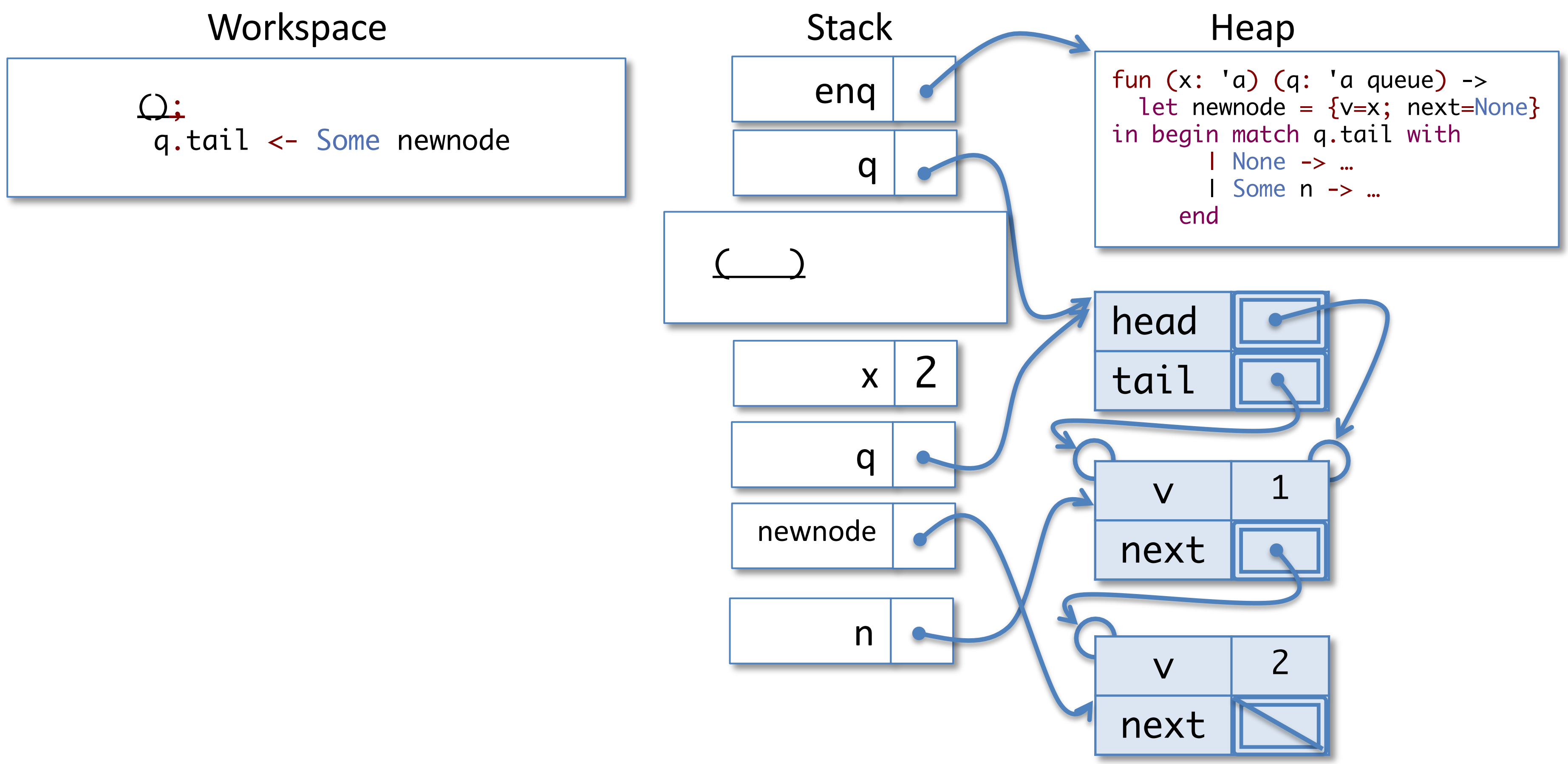
Calling Enq on a non-empty queue



Calling Enq on a non-empty queue



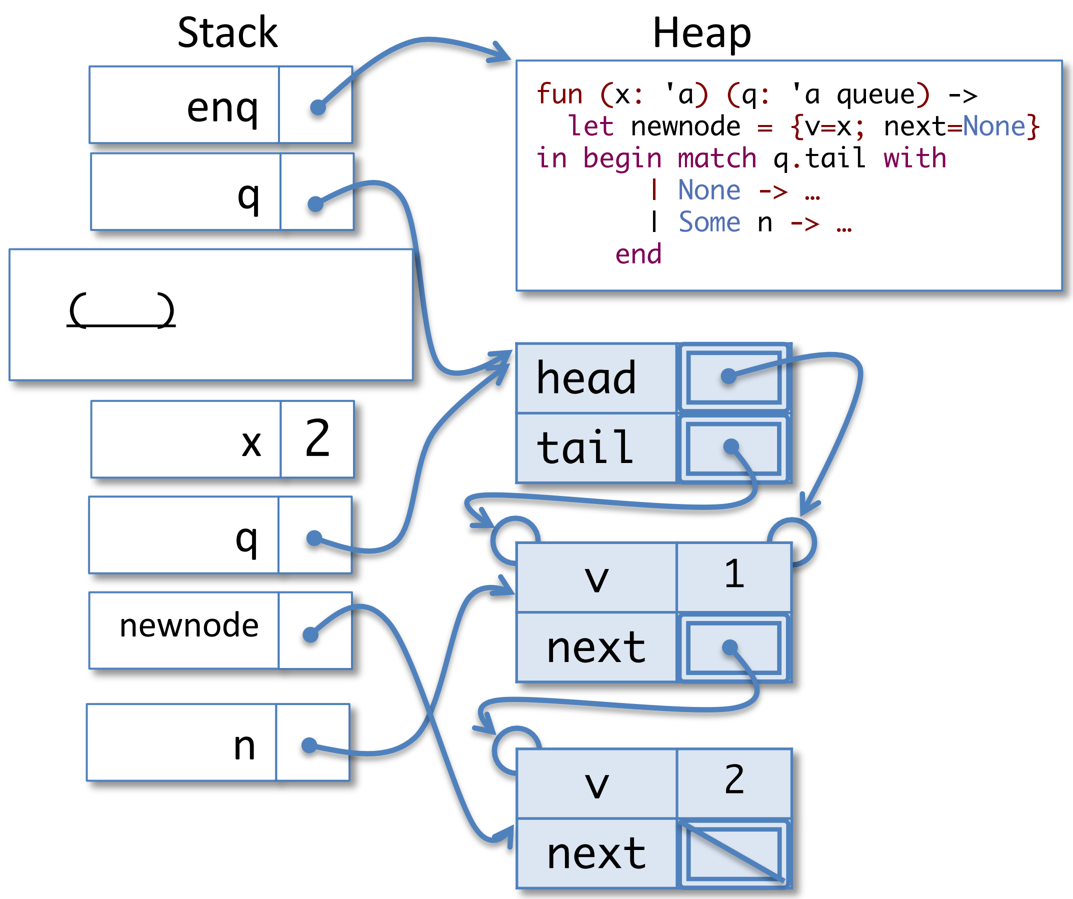
Calling Enq on a non-empty queue



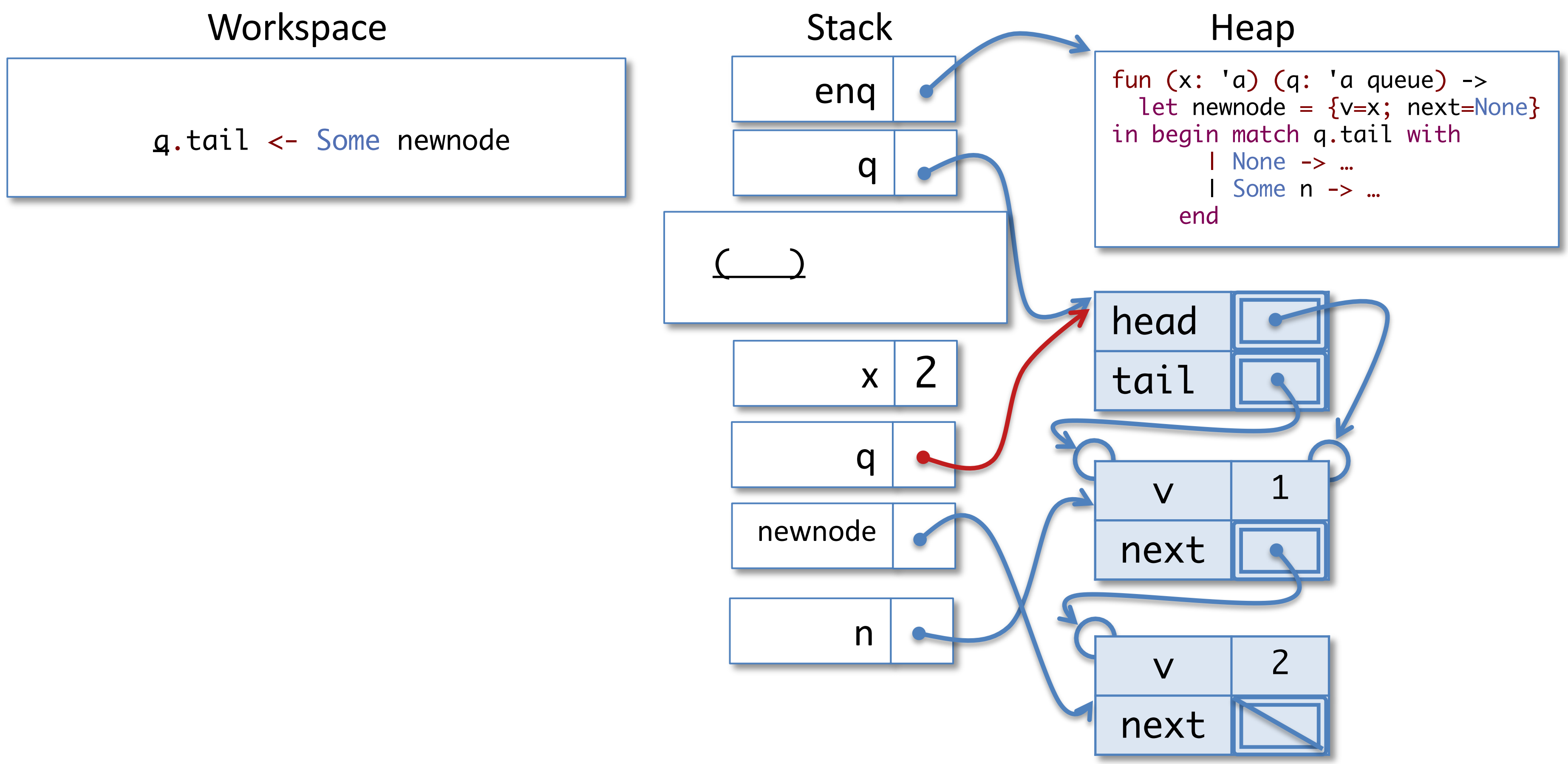
Calling Enq on a non-empty queue

Workspace

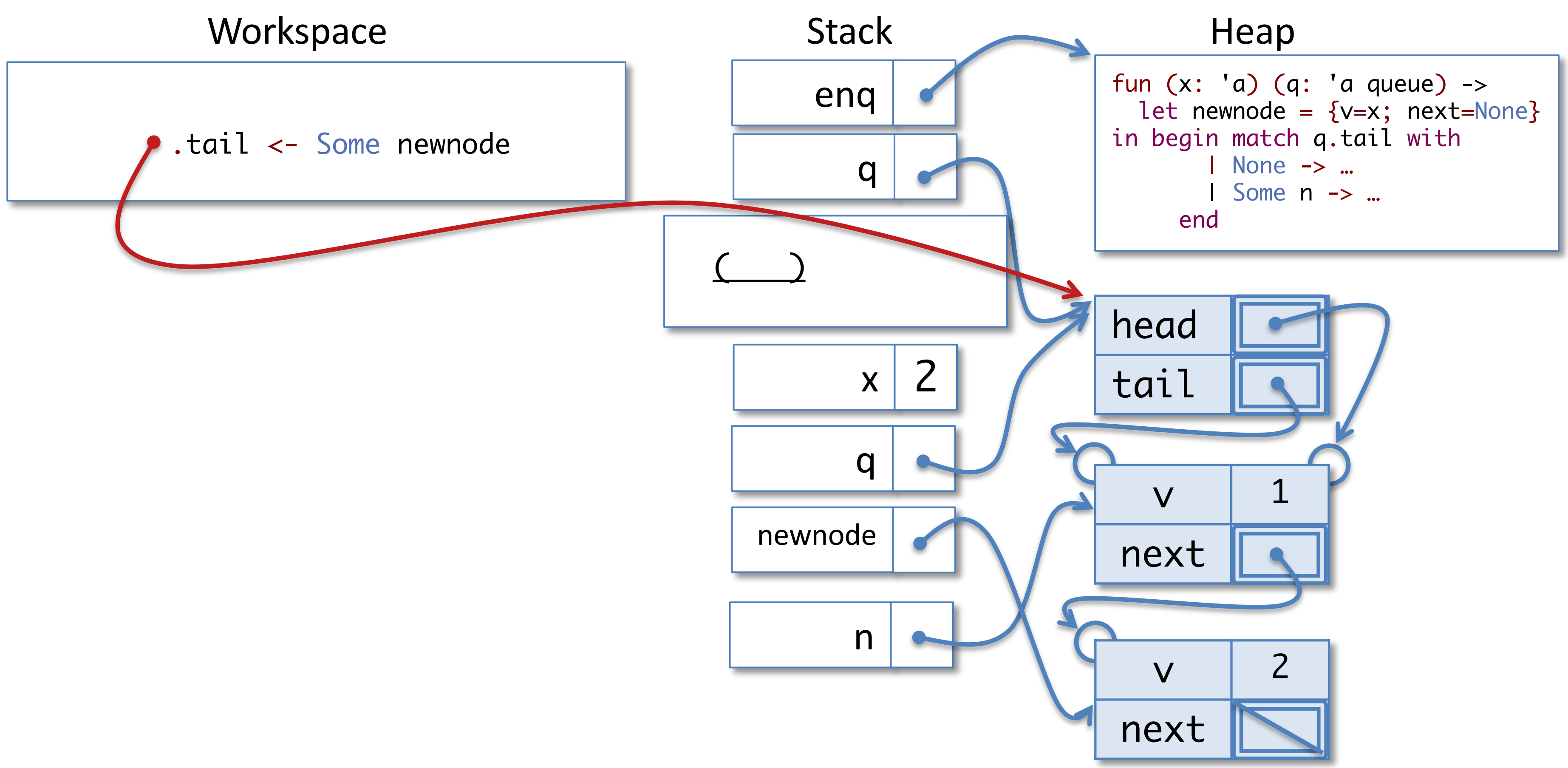
```
q.tail <- Some newnode
```



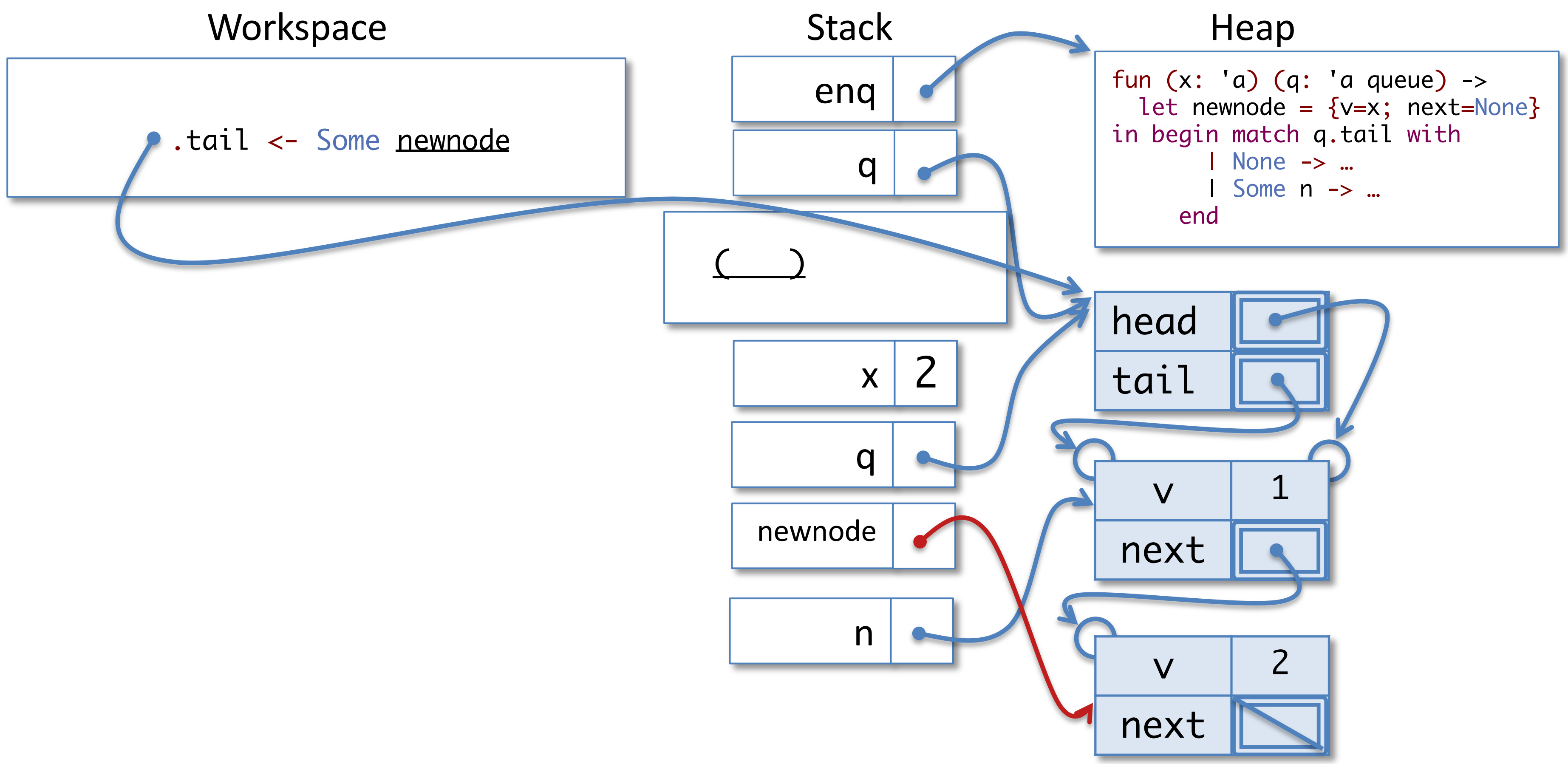
Calling Enq on a non-empty queue



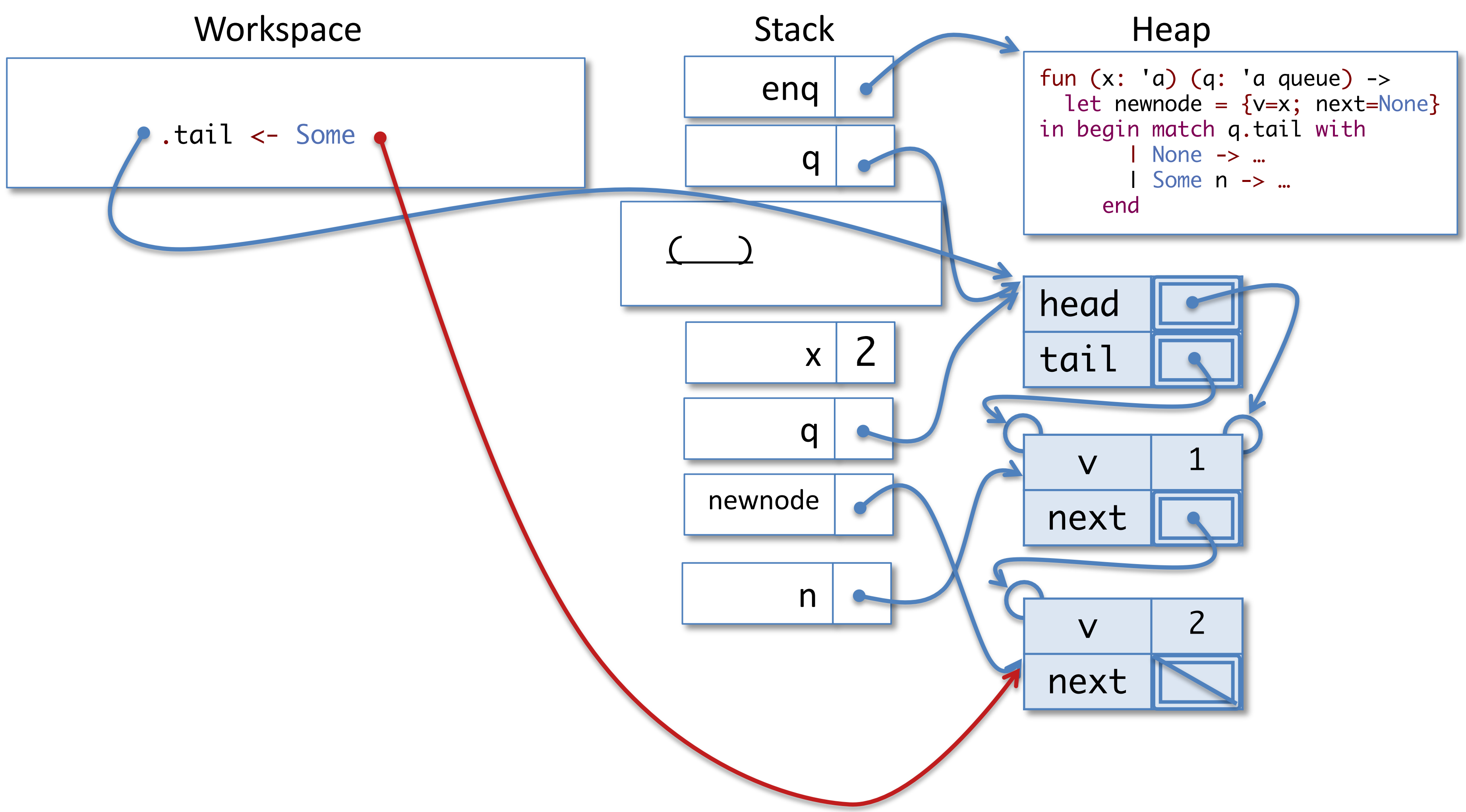
Calling Enq on a non-empty queue



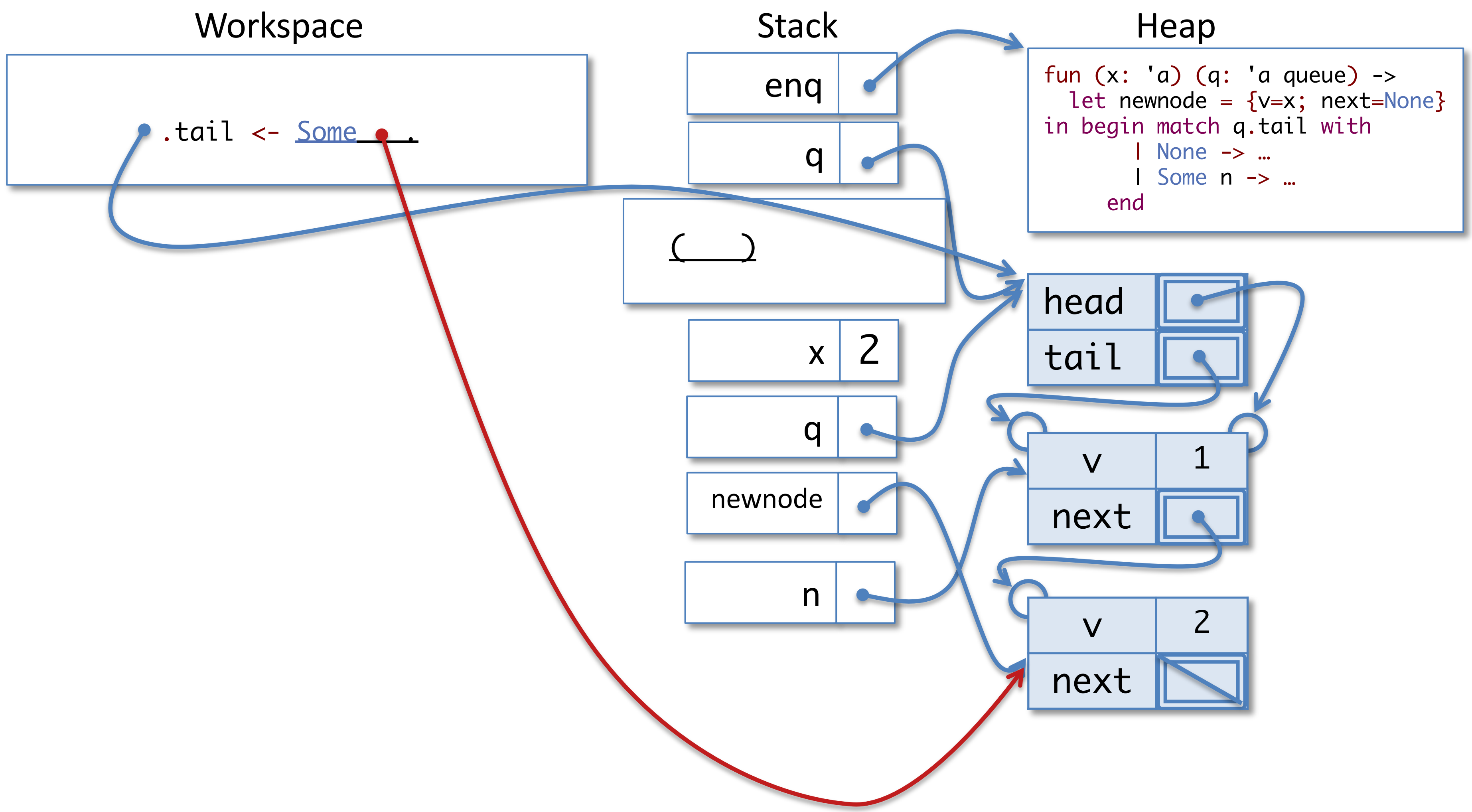
Calling Enq on a non-empty queue



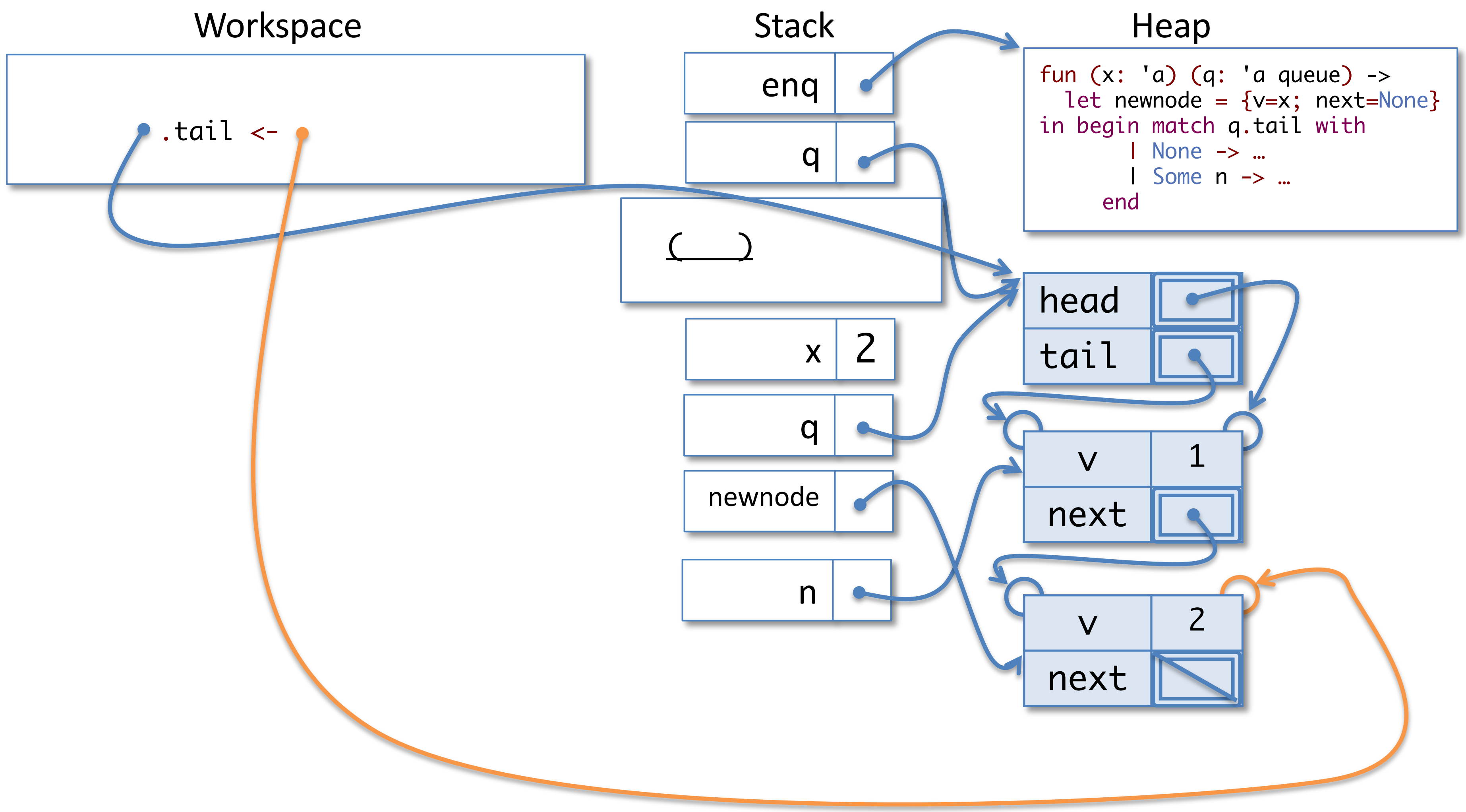
Calling Enq on a non-empty queue



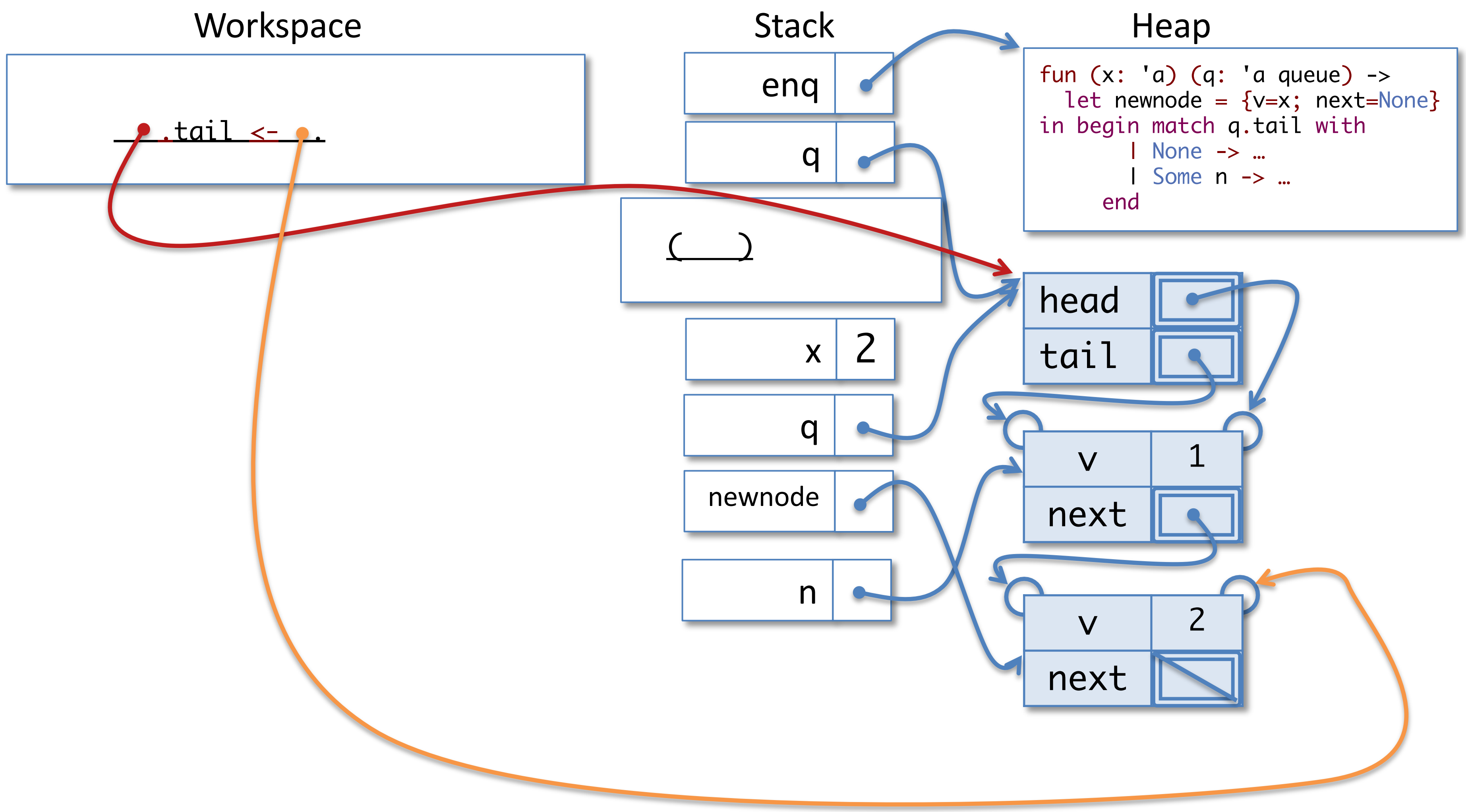
Calling Enq on a non-empty queue



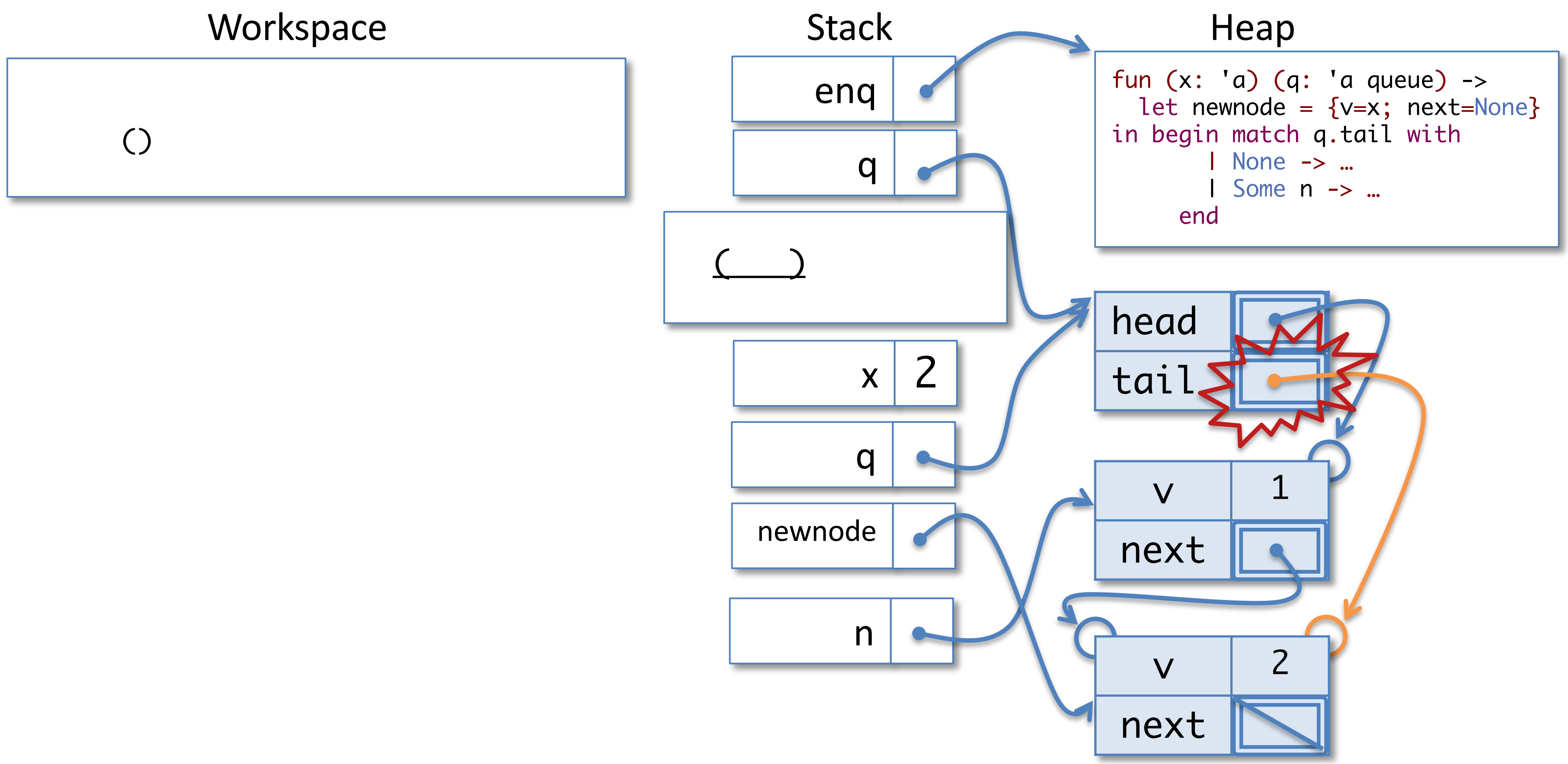
Calling Enq on a non-empty queue



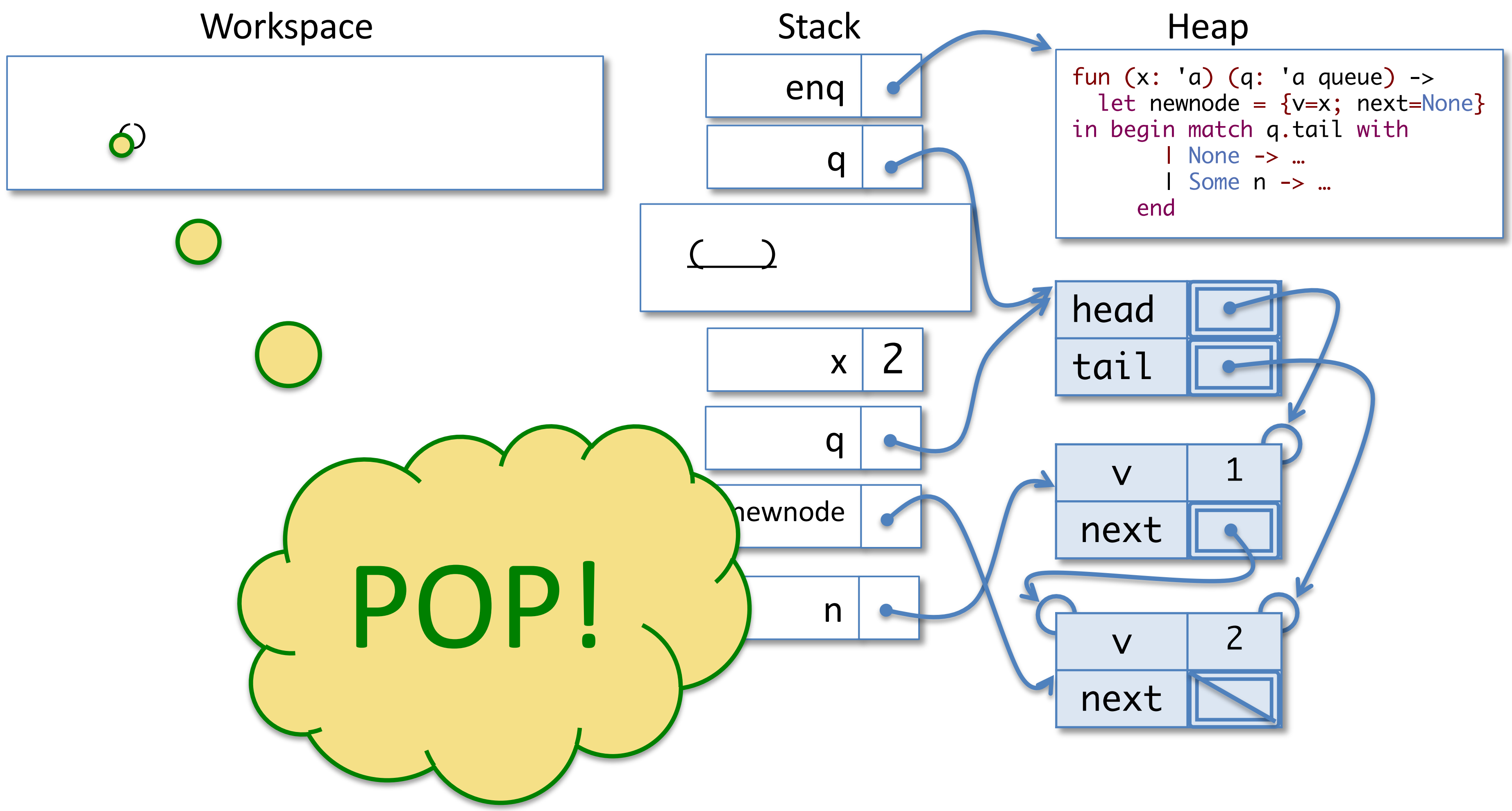
Calling Enq on a non-empty queue



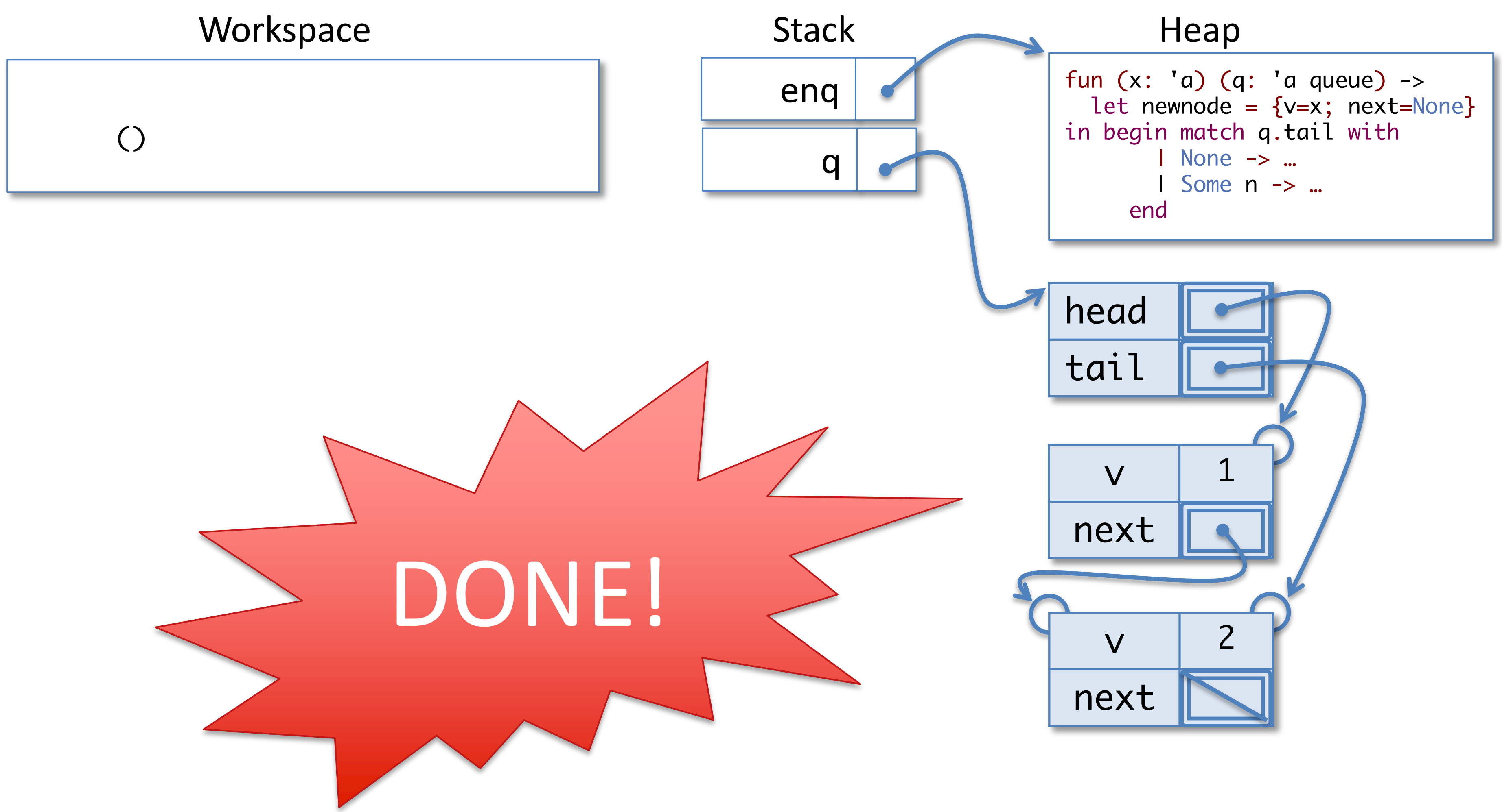
Calling Enq on a non-empty queue



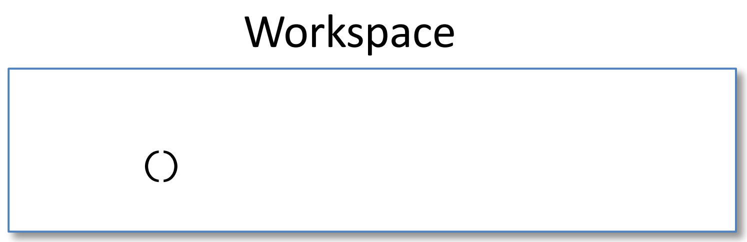
Calling Enq on a non-empty queue

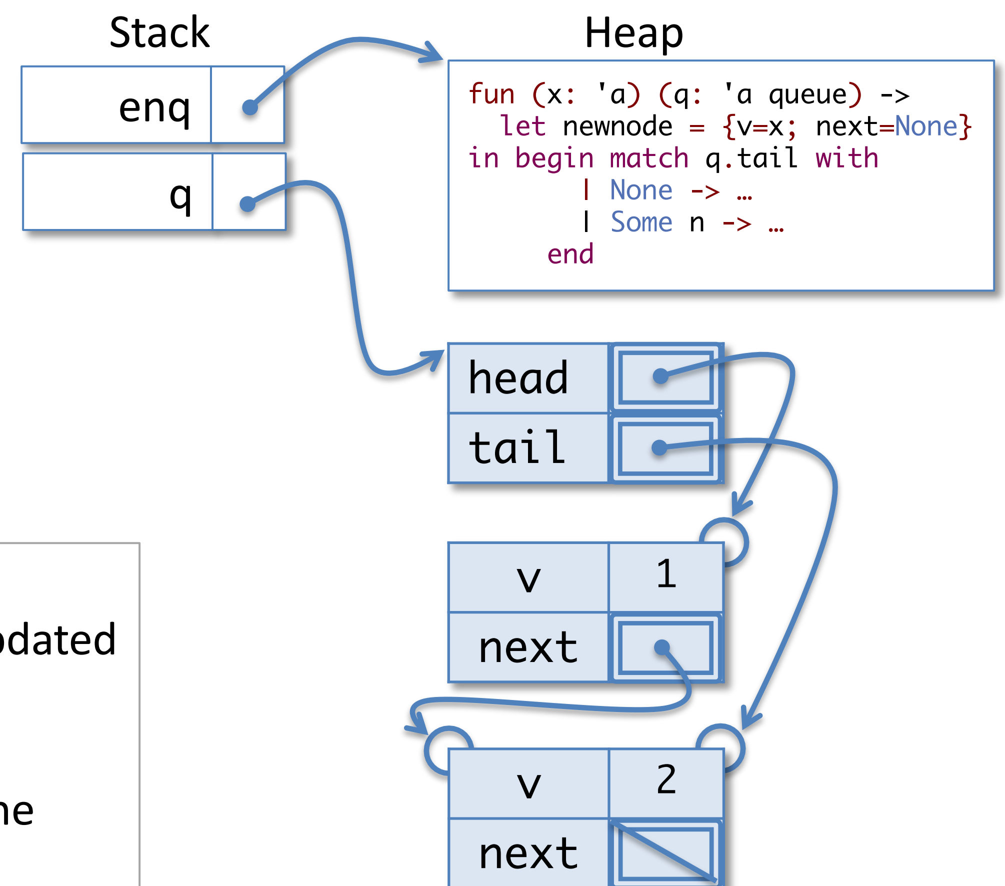


Calling Enq on a non-empty queue



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

Challenge problem - buggy deq

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

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

```
(* remove element at the head of queue and return it  *)
 let deq (q: 'a queue) : 'a =
   begin match q.head with
     | None ->
         failwith "empty queue"
     | Some n ->
         q.head <- n.next;
         n.v
   end
```

Which test case shows the bug?

1.
```
let q = create () in
enq 1 q;
1 = deq q
```

2.
```
let q = create () in
enq 1 q;
enq 2 q;
ignore (deq q);
2 = deq q
```

3.
```
let q = create () in
enq 1 q;
ignore (deq q);
enq 2 q;
2 = deq q
```

ANSWER: 3

4. All of them

deq

```
(* remove an element from the head of the queue *)
 let deq (q: 'a queue) : 'a =
   begin match q.head with
     | None ->
         failwith "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