Programming Languages and Techniques (CIS120)

Lecture 14

Feb 13, 2012

Imperative Queues

Announcements

- Homework 4 due at midnight

- Homework 5 (queues) will be available on the web after the exam
  - It is due Thurs Feb 23rd at 11:59:59pm

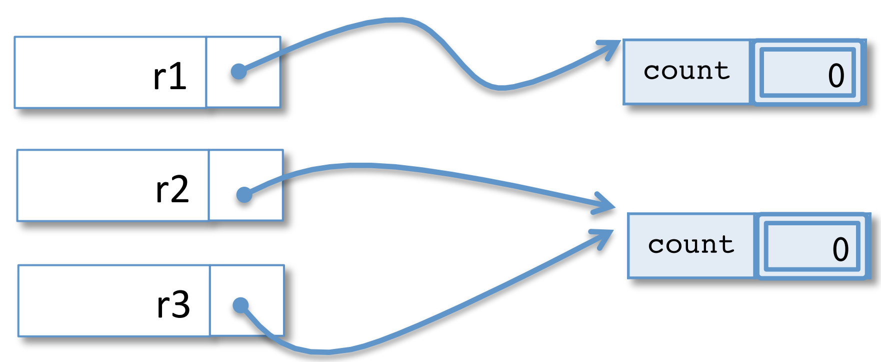
- Midterm 1 will be in class on Wednesday, February 12th
  - ROOM: LLAB 10
  - TIME: 11:00a.m. sharp
  - Covers up to Feb 8th (no mutable records)

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 data in the heap:

```
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)
  - use this for comparing *immutable* datatypes

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

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 datastructure 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
  - Needs to be mutable so the state can be shared across web sessions

Queue Interface

```
module type QUEUE =
sig
  (* type of the data structure *)
  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 tail of the queue *)
  val enq : 'a -> 'a queue -> unit

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

end
```

Define test cases

```
(* Some test cases for the queue *)
;; open ListQ

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
```

Implement the behavior

```
module ListQ : QUEUE  = struct

  type 'a queue = { mutable contents : ‘a list }

  let create () : 'a queue =
    { contents = [] }

  let is_empty (q:'a queue) : bool =
    q.contents = []

  let enq (x:'a) (q:'a queue) : unit =
    q.contents <- (q.contents @ [x])

  let deq (q:'a queue) : 'a =
    begin match q.contents with
      | [] -> failwith "deq called on empty queue"
      | x::tl -> q.contents <- tl; x
    end
 end
```

Here we are using type abstraction to protect the state. Outside of the module, no one knows that queues are implemented with references. So, only these functions can modify the state of a queue.

A Better Implementation

- Implementation is slow because of append:
  - `q.contents @ [x]` copies the entire list each time
  - As the queue gets longer, it takes longer to add data
  - Only has a *single* reference to the beginning of the list

- Let's do it again with TWO references, one to the beginning (head) and one to the end (tail).
  - Dequeue by updating the head reference (as before)
  - Enqueue by updating the tail of the list
- The list itself must be mutable
  - because we add to one end and remove from the other
- Step 1: *Understand the problem*

Data Structure for Mutable Queues

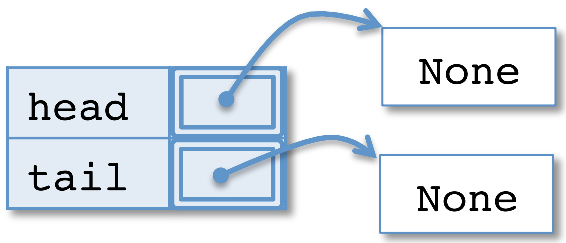
```
type 'a qnode = {
     v: 'a;
     mutable next : 'a qnode option
}

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

There are two parts to a mutable queue:

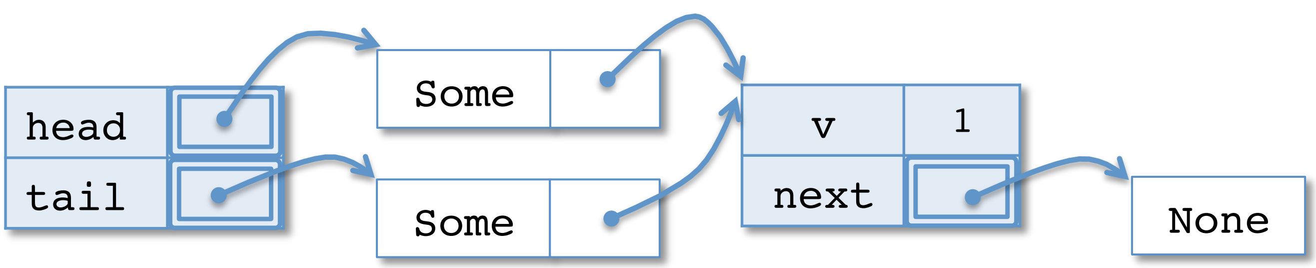
- the “internal nodes” of the queue with links from one to the next
- the head and tail references themselves

All of the references are options so that the queue can be empty.

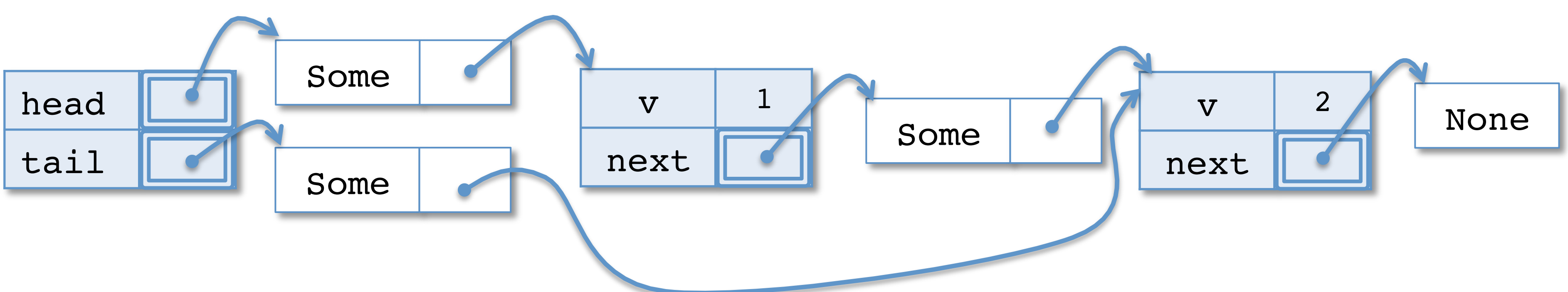
Queues in the Heap



An empty queue

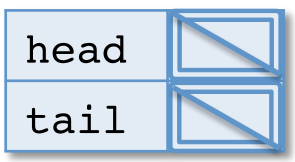


A queue with one element

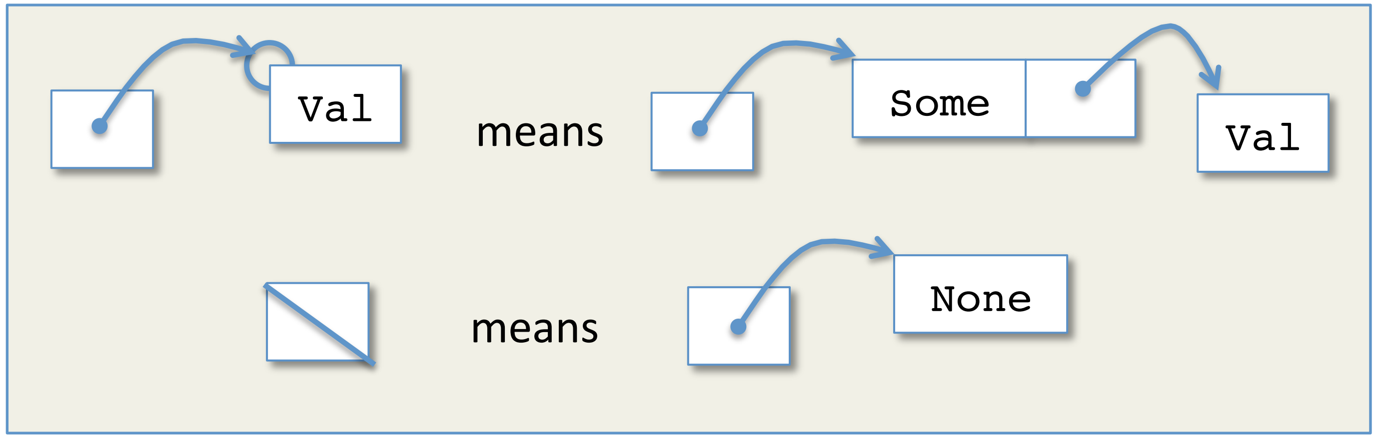


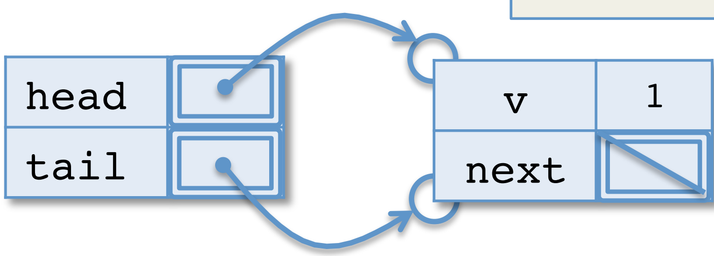
A queue with two elements

Visual Shorthand: Abbreviating Options

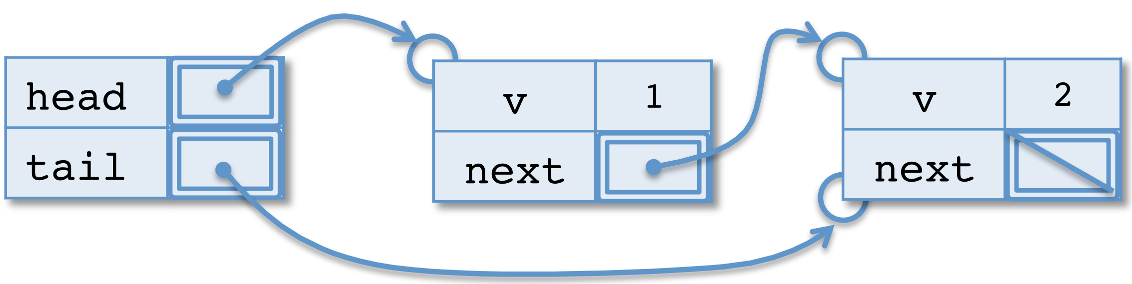


An empty queue



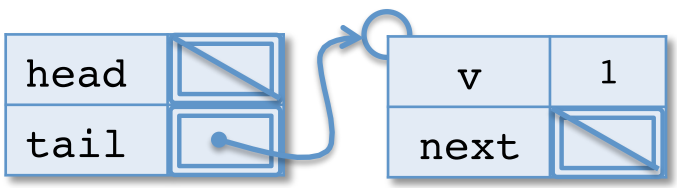


A queue with one element

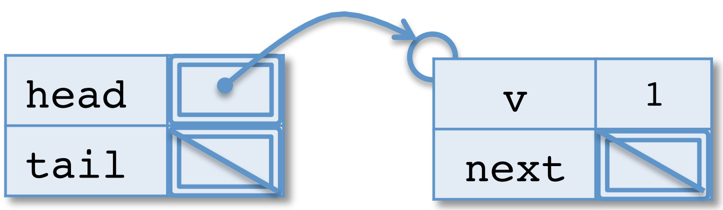


A queue with two elements

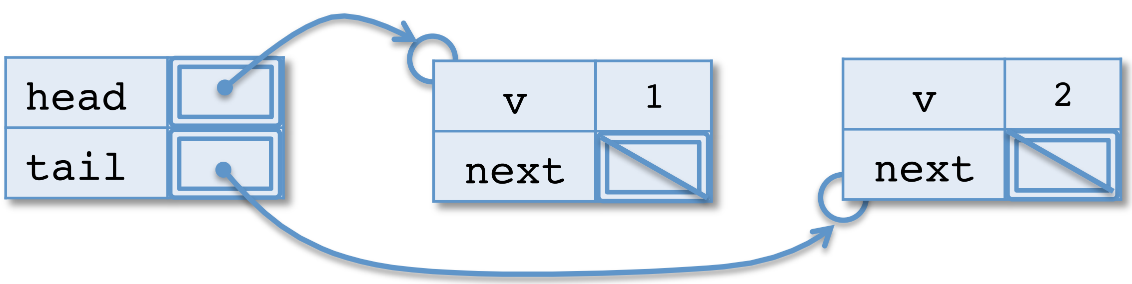
“Bogus” values of type `int queue`



head is None, tail is Some n

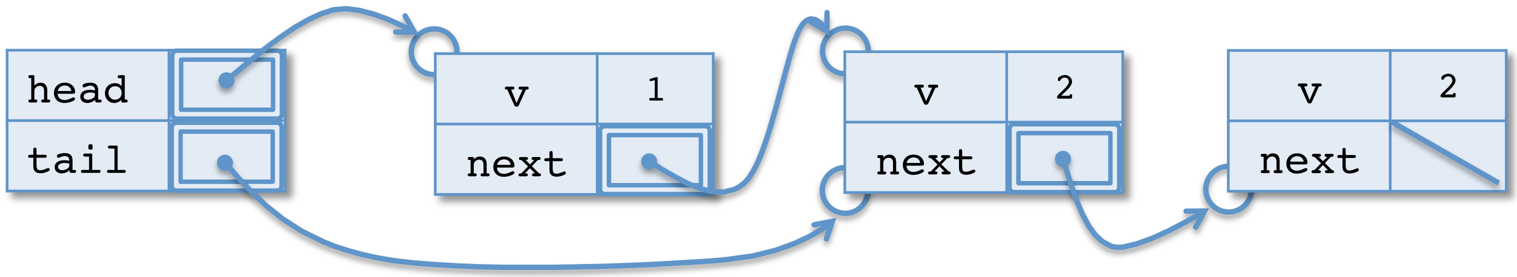


head is Some, tail is None

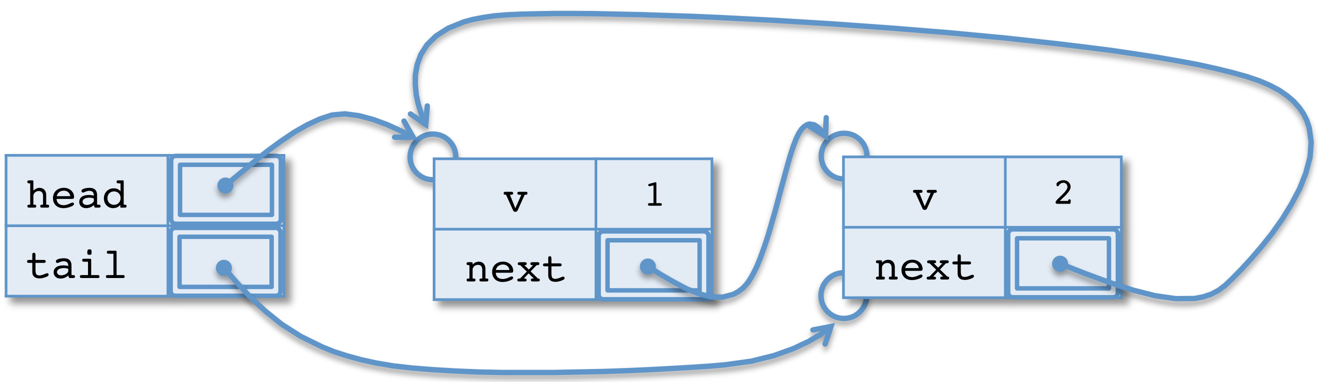


tail is not reachable from the head

More bogus `int` queues



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



the links contain a *cycle*!

Queue Invariants

- Just as we imposed some restrictions on which trees are legitimate Binary Search Trees, Queues must also satisfy some *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.
- A queue operation may assume that these queue invariants hold of its inputs, so long as it ensures that the invariants hold when it's done.

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

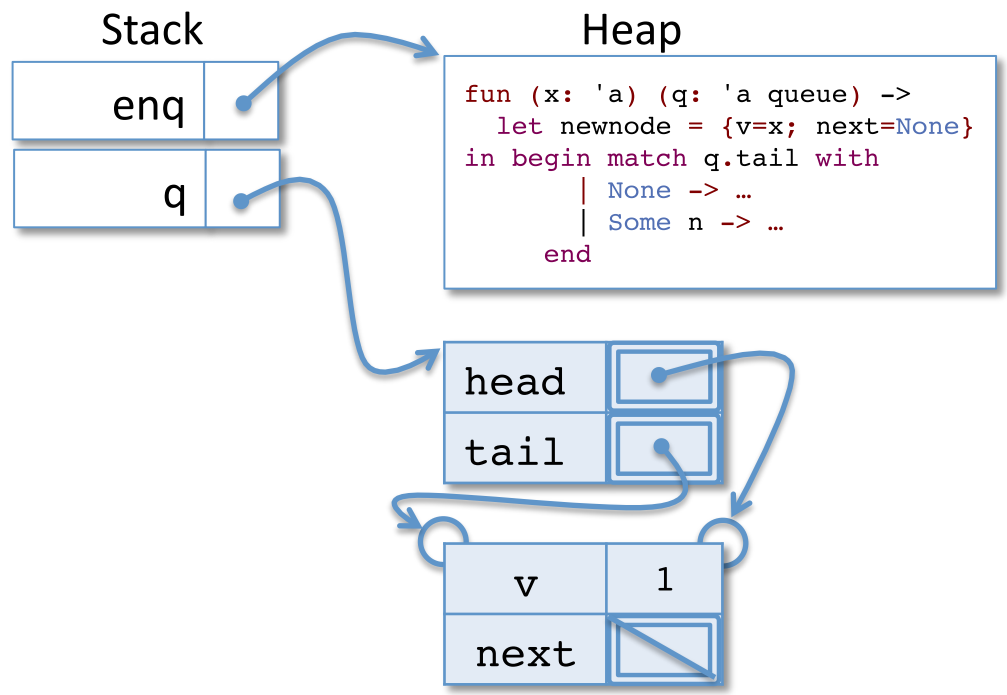
```
(* add an element to the tail of a queue *)
let enq (x: 'a) (q: 'a queue) : unit =
   let newnode = {v=x; next=None} in
   begin match q.tail with
     | None ->
         (* Note that the invariant tells us
           that q.head is also None *)
         q.head <- Some newnode;
         q.tail <-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

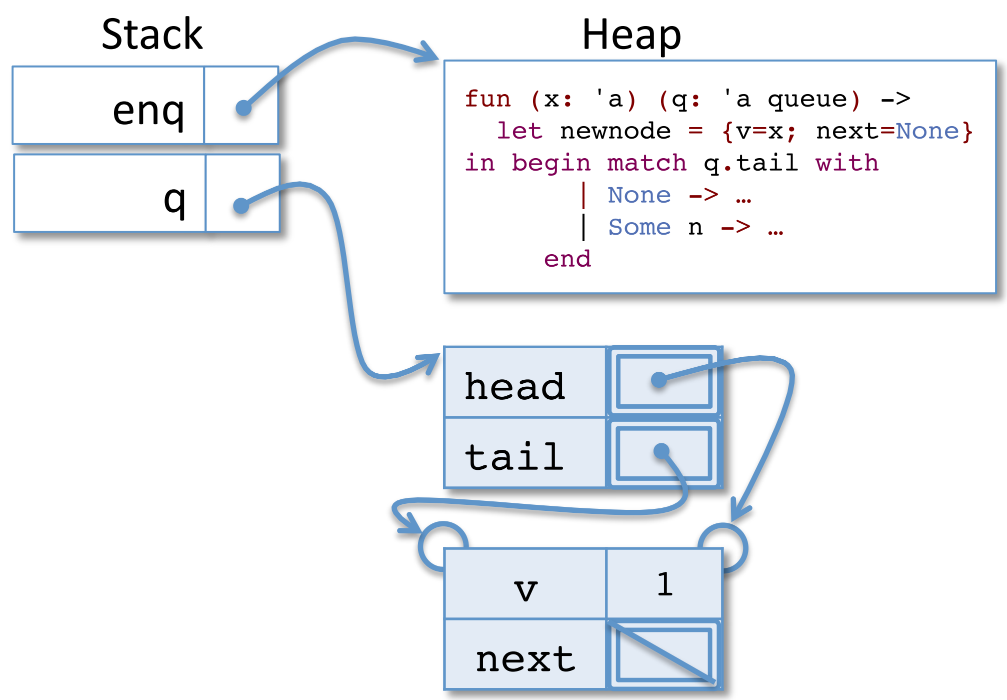
```
enq 2 q
```



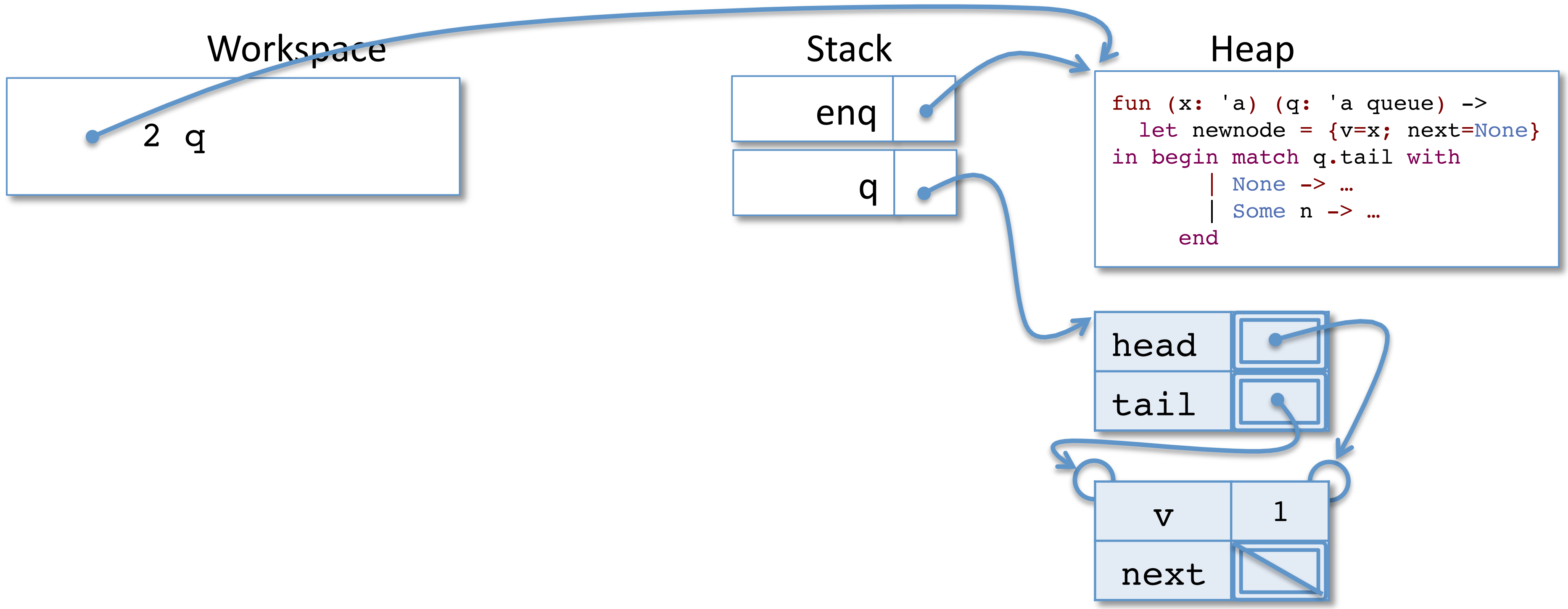
Calling Enq on a non-empty queue

Workspace

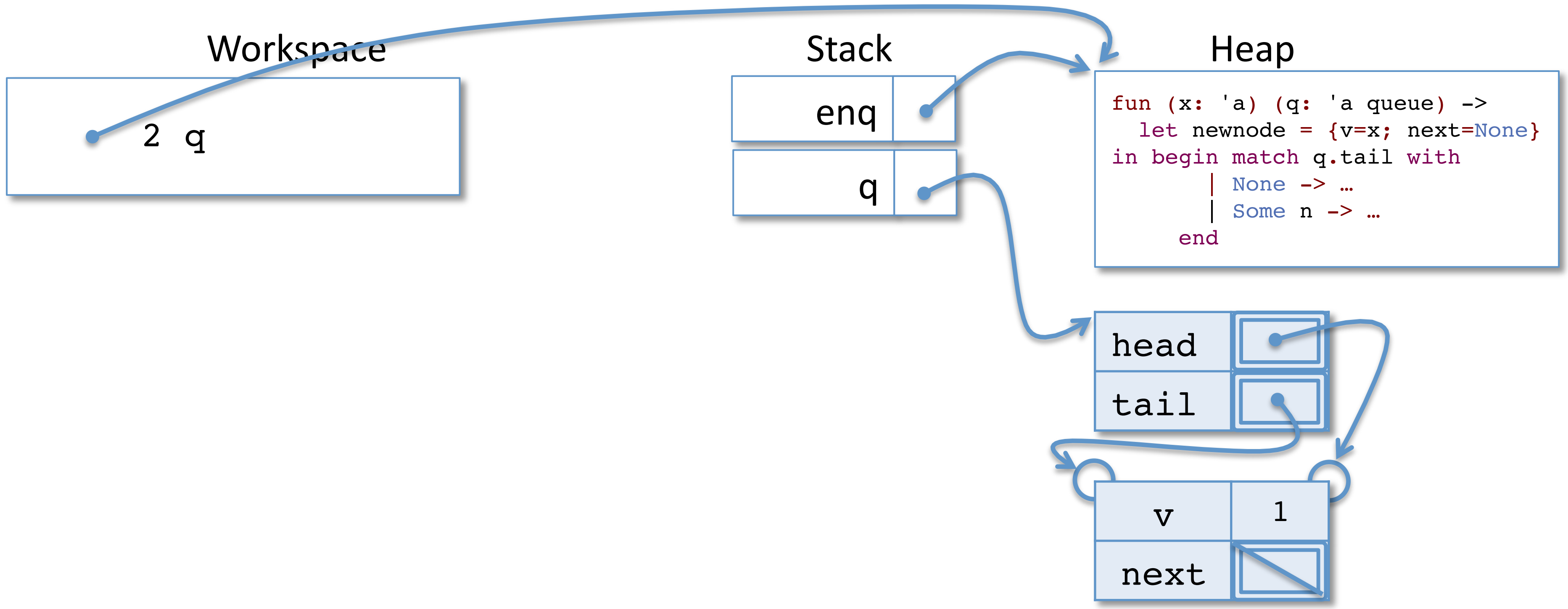
```
enq 2 q
```



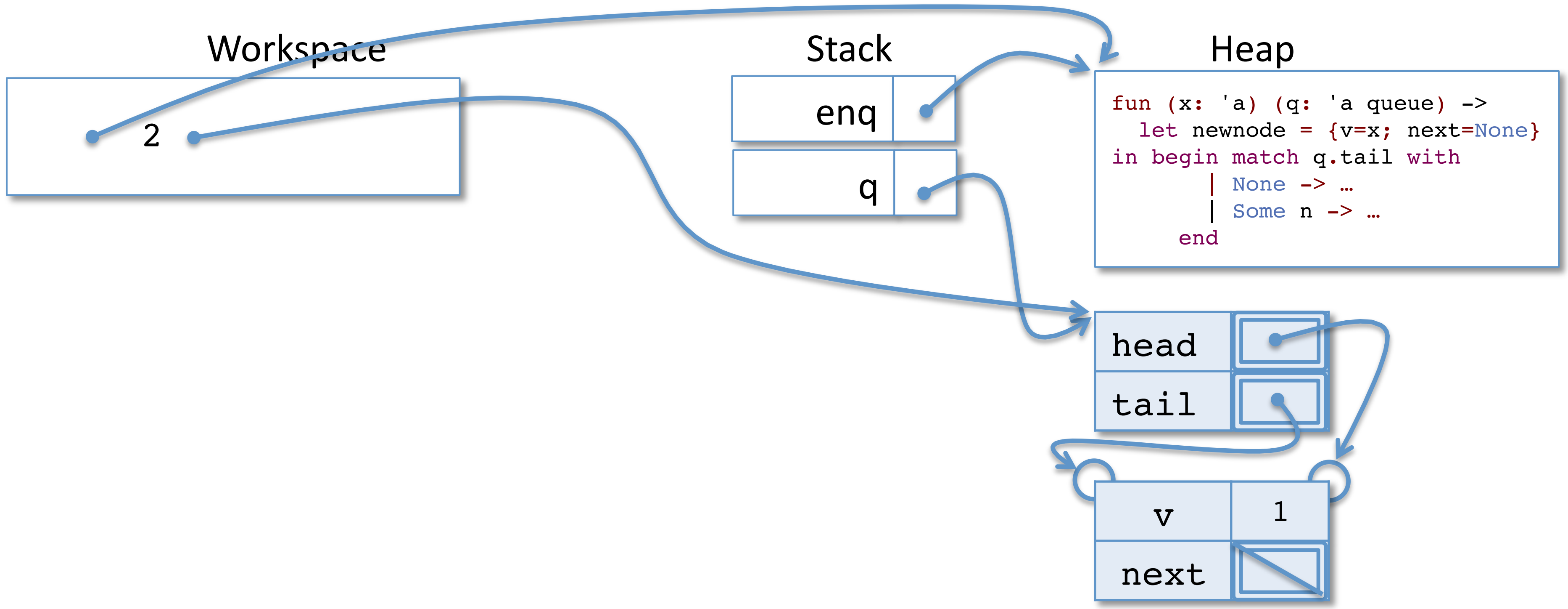
Calling Enq on a non-empty queue



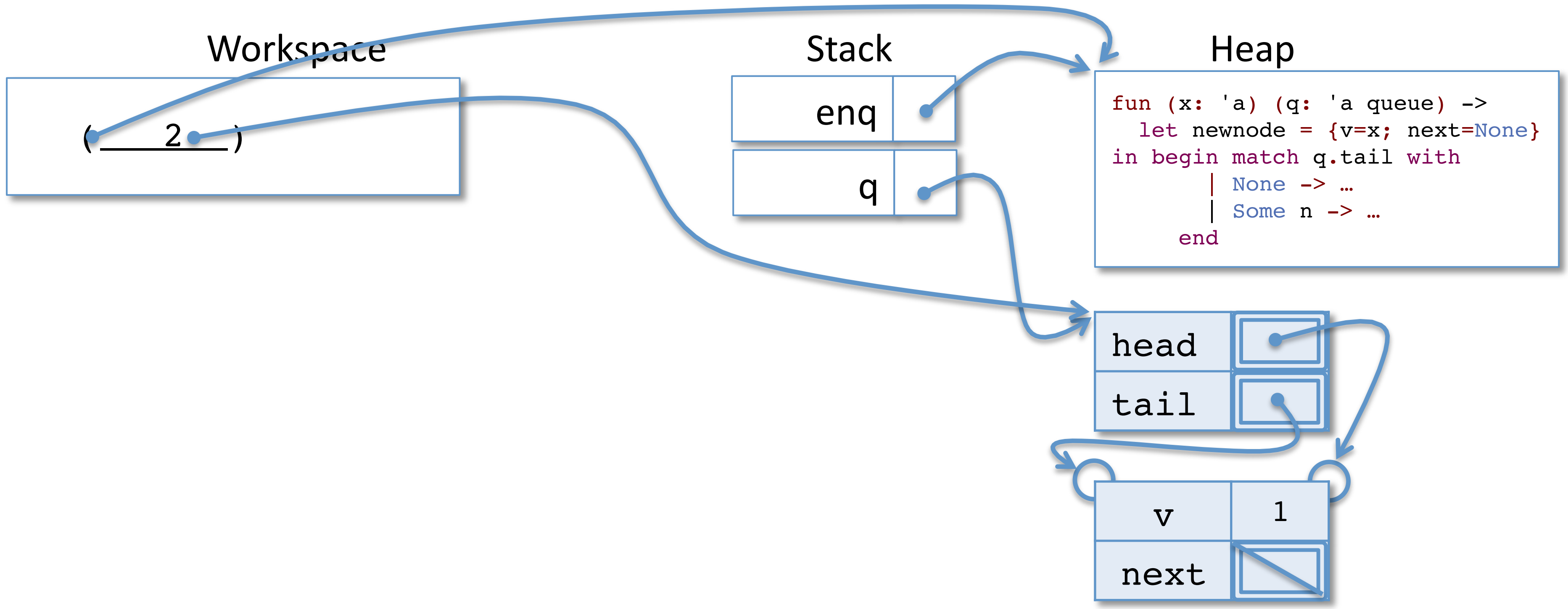
Calling Enq on a non-empty queue



Calling Enq on a non-empty queue



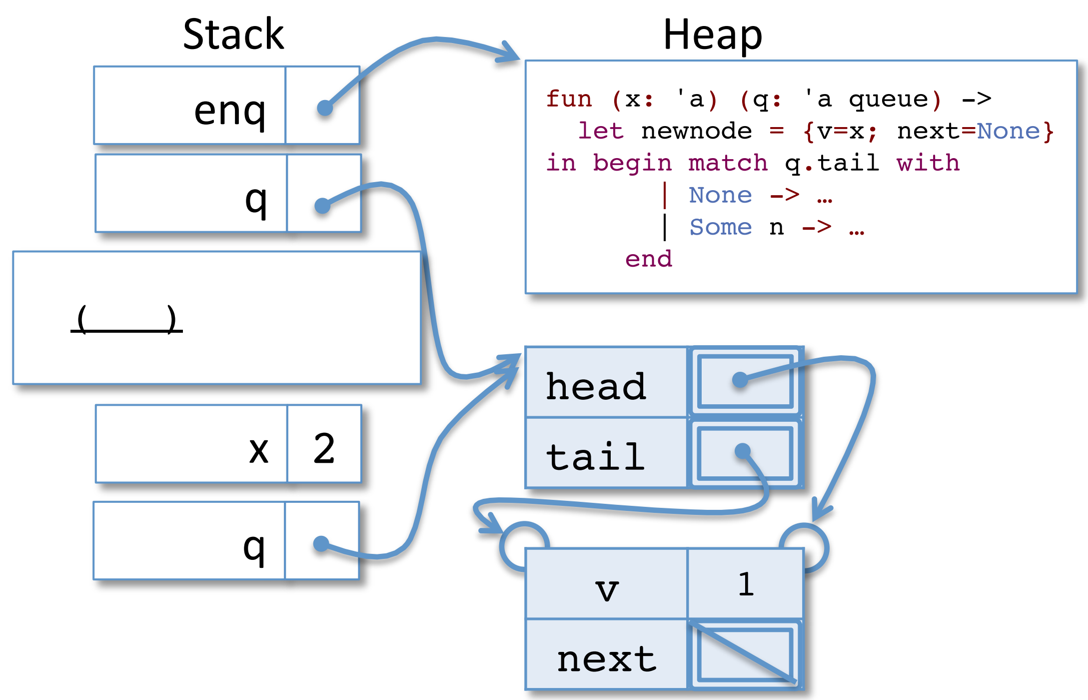
Calling Enq on a non-empty queue



Calling Enq on a non-empty queue

Workspace

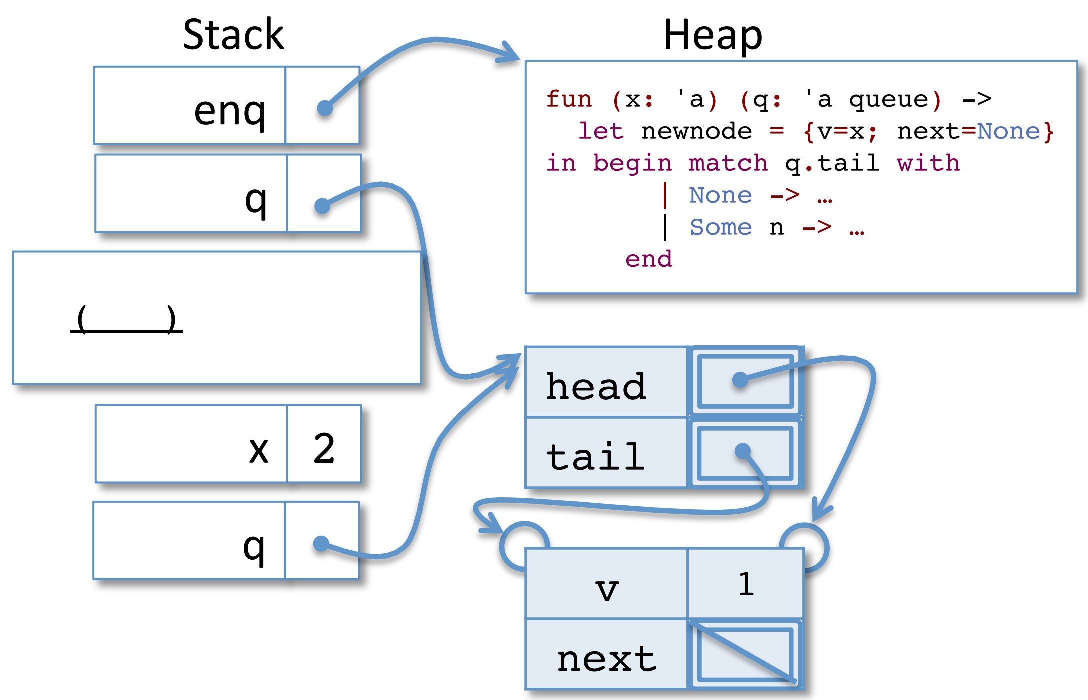
```
let newnode = {v=x; next=None} in
  begin match q.tail with
    | None ->
      q.head <- Some newnode;
      q.tail <- Some newnode
    | Some n ->
      n.next <- Some newnode;
      q.tail <- Some newnode
  end
```



Calling Enq on a non-empty queue

Workspace

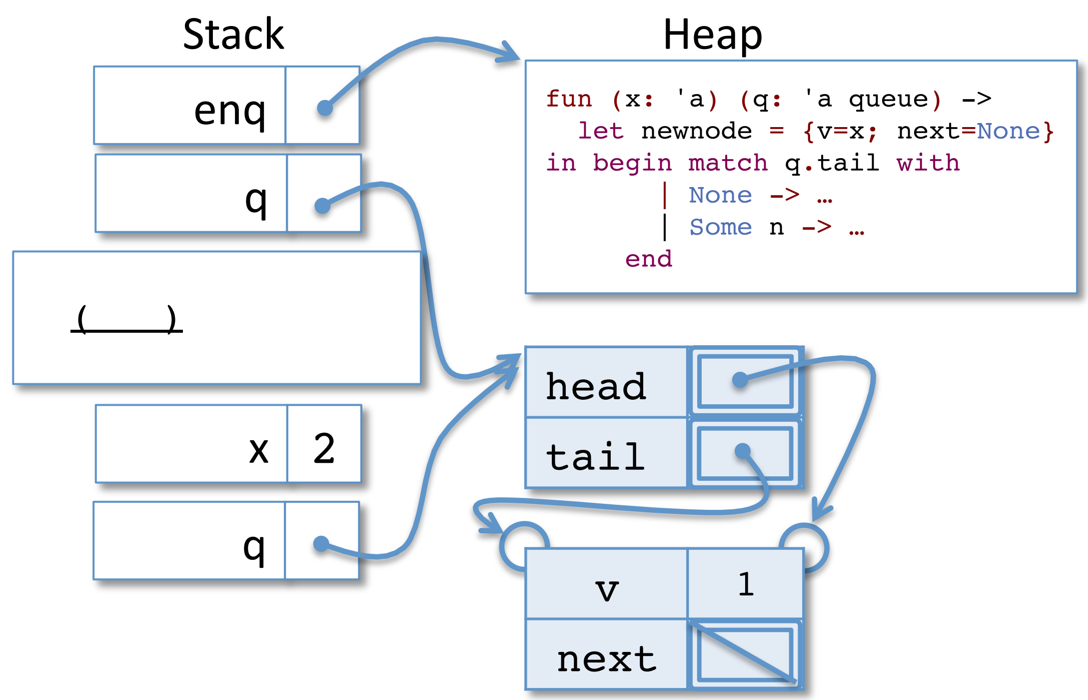
```
let newnode = {v=x; next=None} in
  begin match q.tail with
    | None ->
      q.head <- Some newnode;
      q.tail <- Some newnode
    | Some n ->
      n.next <- Some newnode;
      q.tail <- Some newnode
  end
```



Calling Enq on a non-empty queue

Workspace

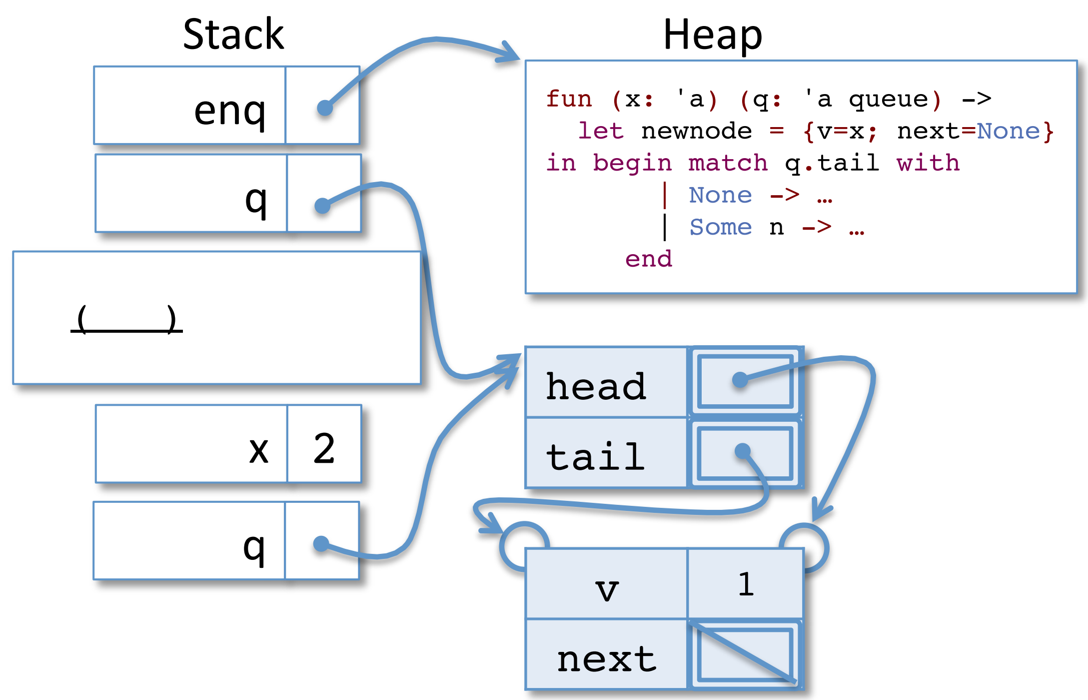
```
let newnode = {v=2; next=None} in
  begin match q.tail with
    | None ->
      q.head <- Some newnode;
      q.tail <- Some newnode
    | Some n ->
      n.next <- Some newnode;
      q.tail <- Some newnode
  end
```



Calling Enq on a non-empty queue

Workspace

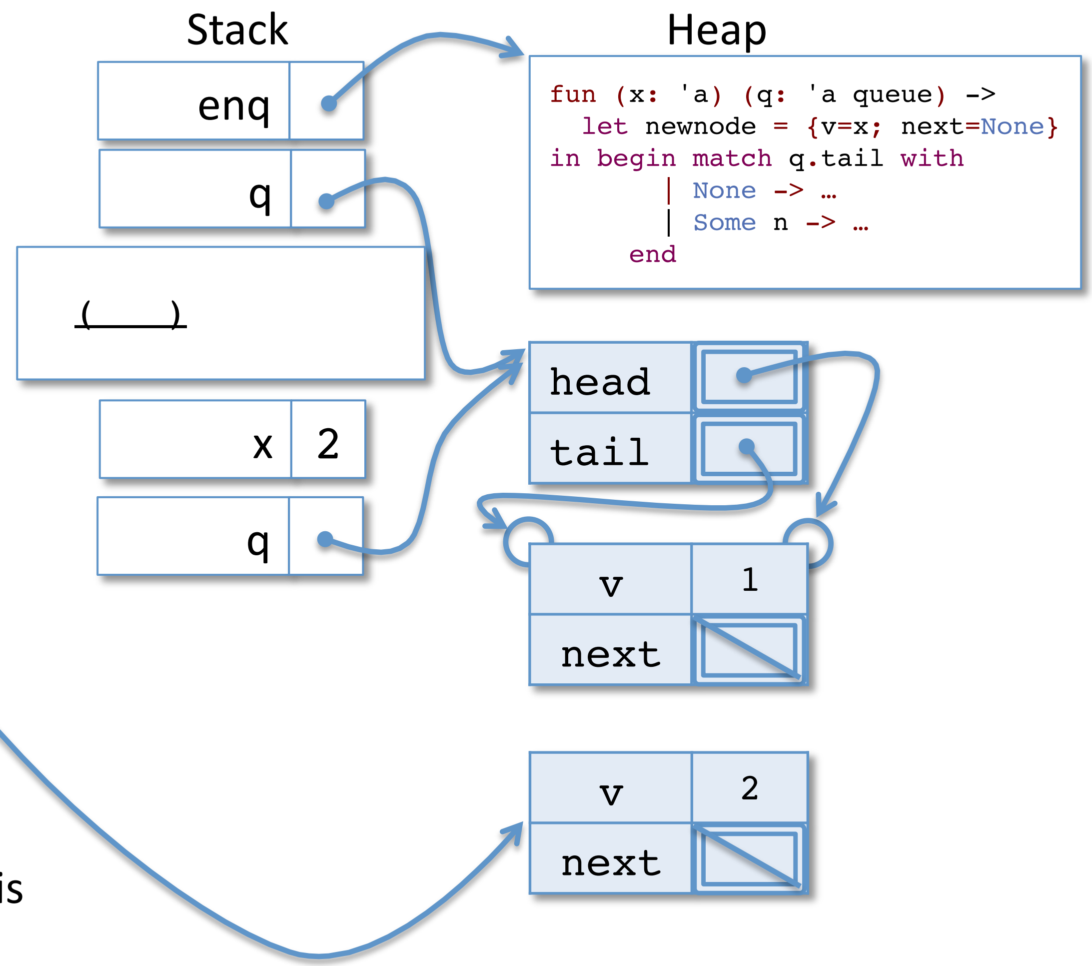
```
let newnode = {v=2; next=None} in
  begin match q.tail with
    | None ->
      q.head <- Some newnode;
      q.tail <- Some newnode
    | Some n ->
      n.next <- Some newnode;
      q.tail <- Some newnode
  end
```



Calling Enq on a non-empty queue

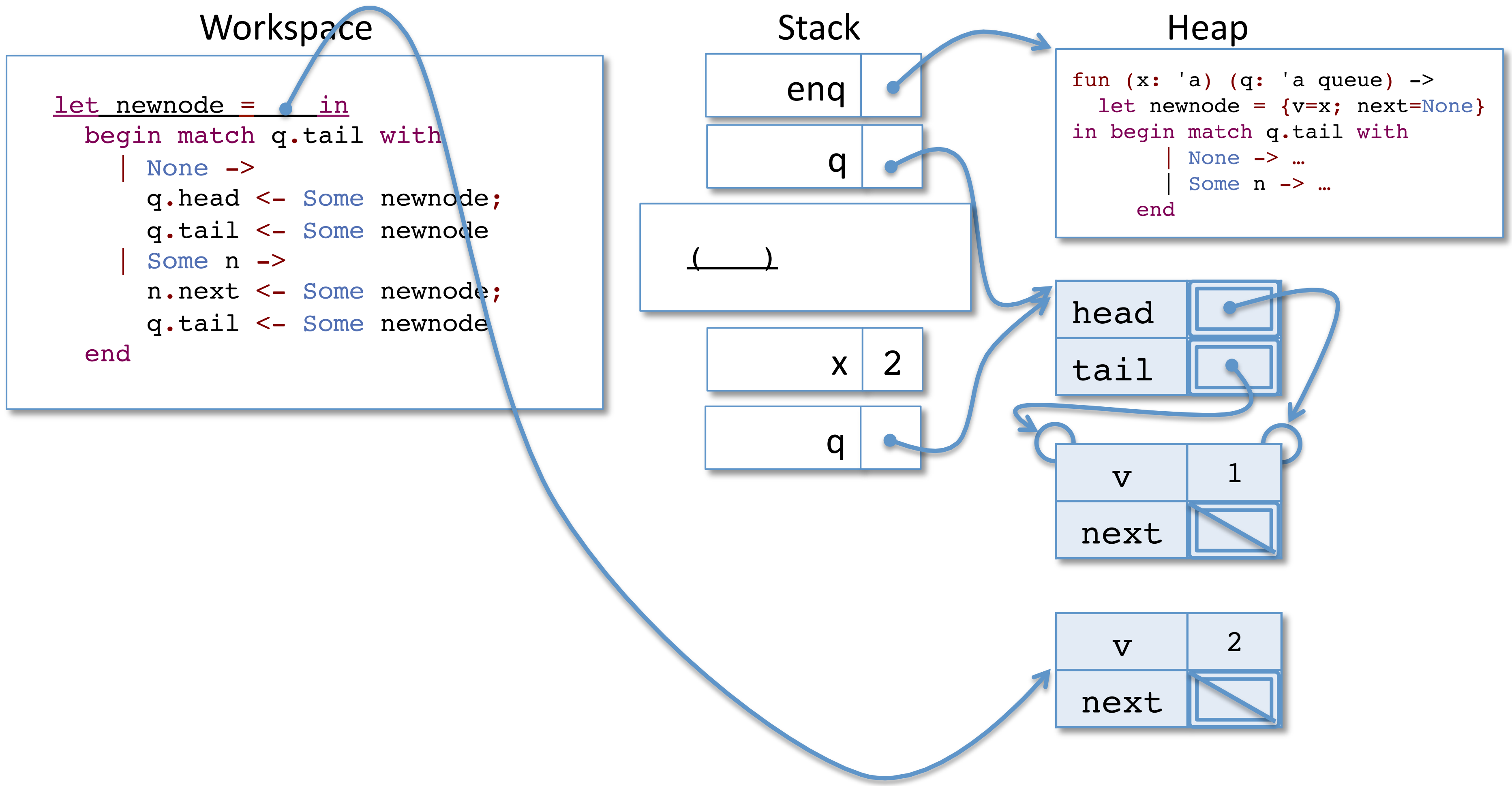
Workspace

```
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
```



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

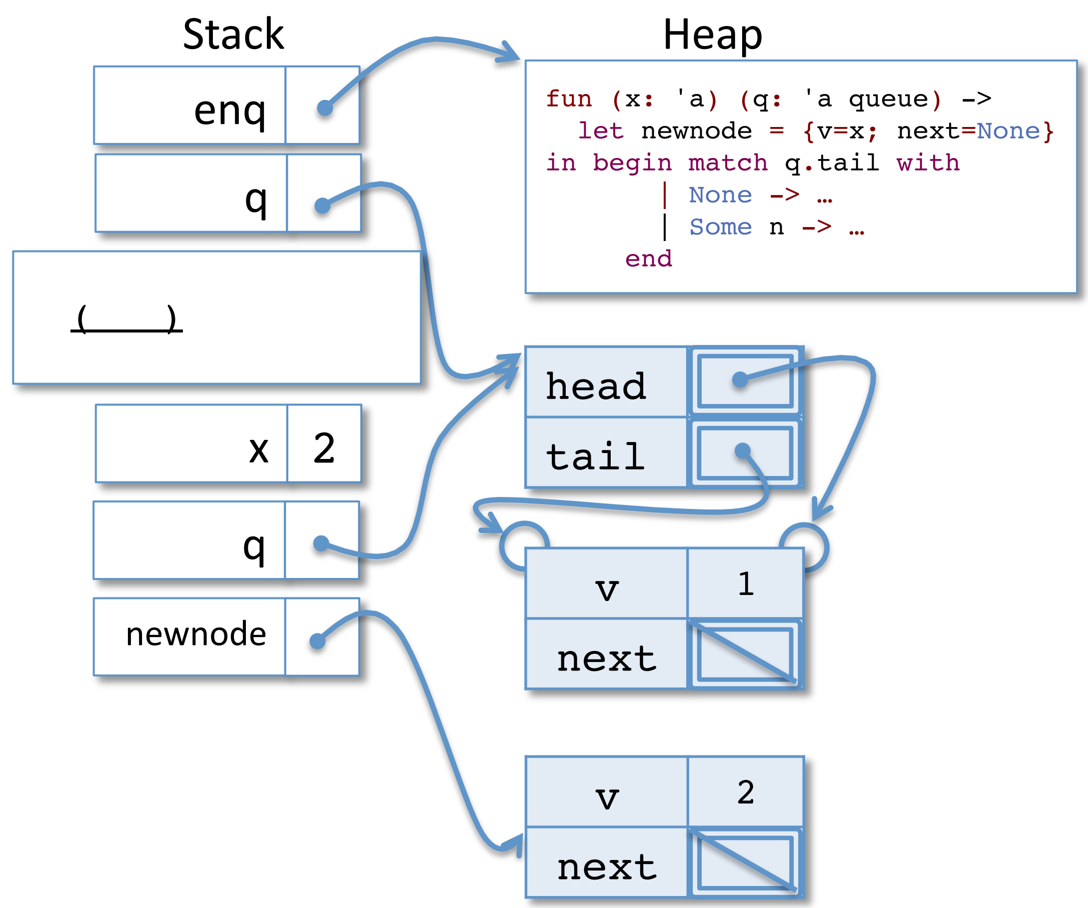
Calling Enq on a non-empty queue



Calling Enq on a non-empty queue

Workspace

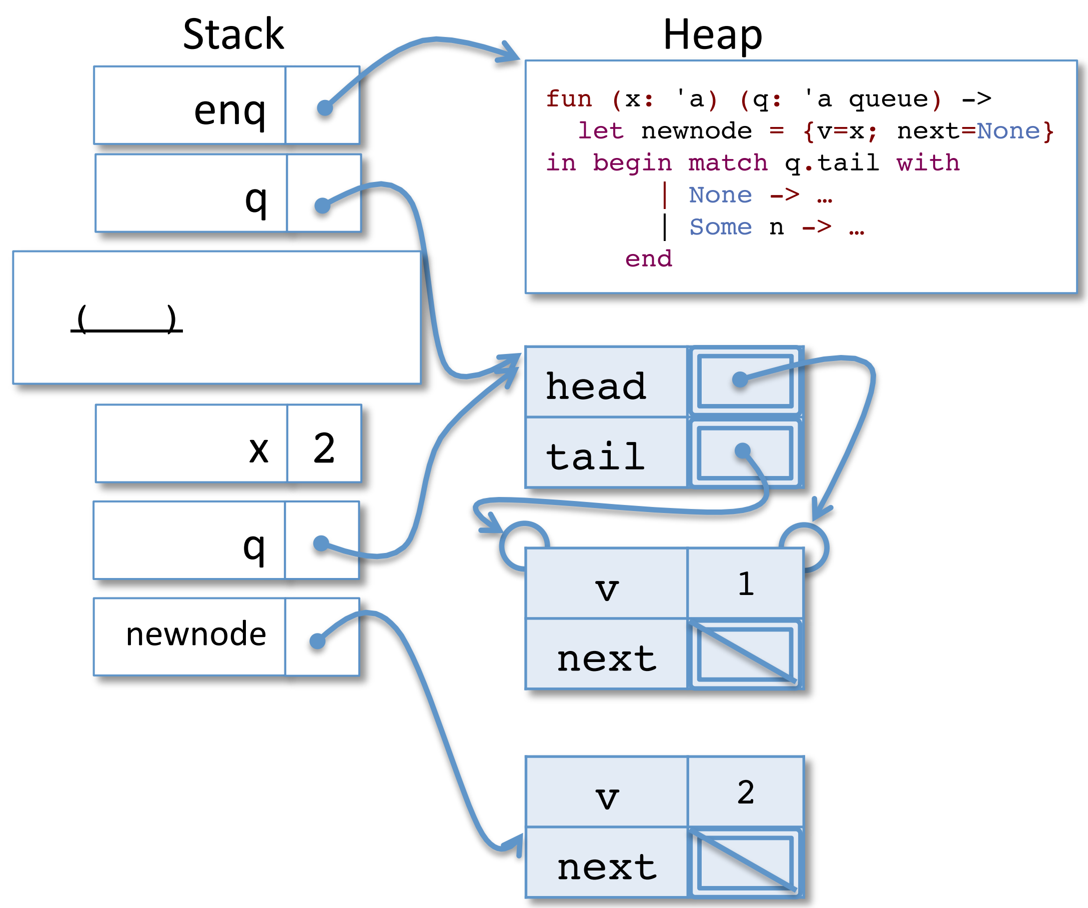
```
begin match q.tail with
  | None ->
    q.head <- Some newnode;
    q.tail <- Some newnode
  | Some n ->
    n.next <- Some newnode;
    q.tail <- Some newnode
end
```



Calling Enq on a non-empty queue

Workspace

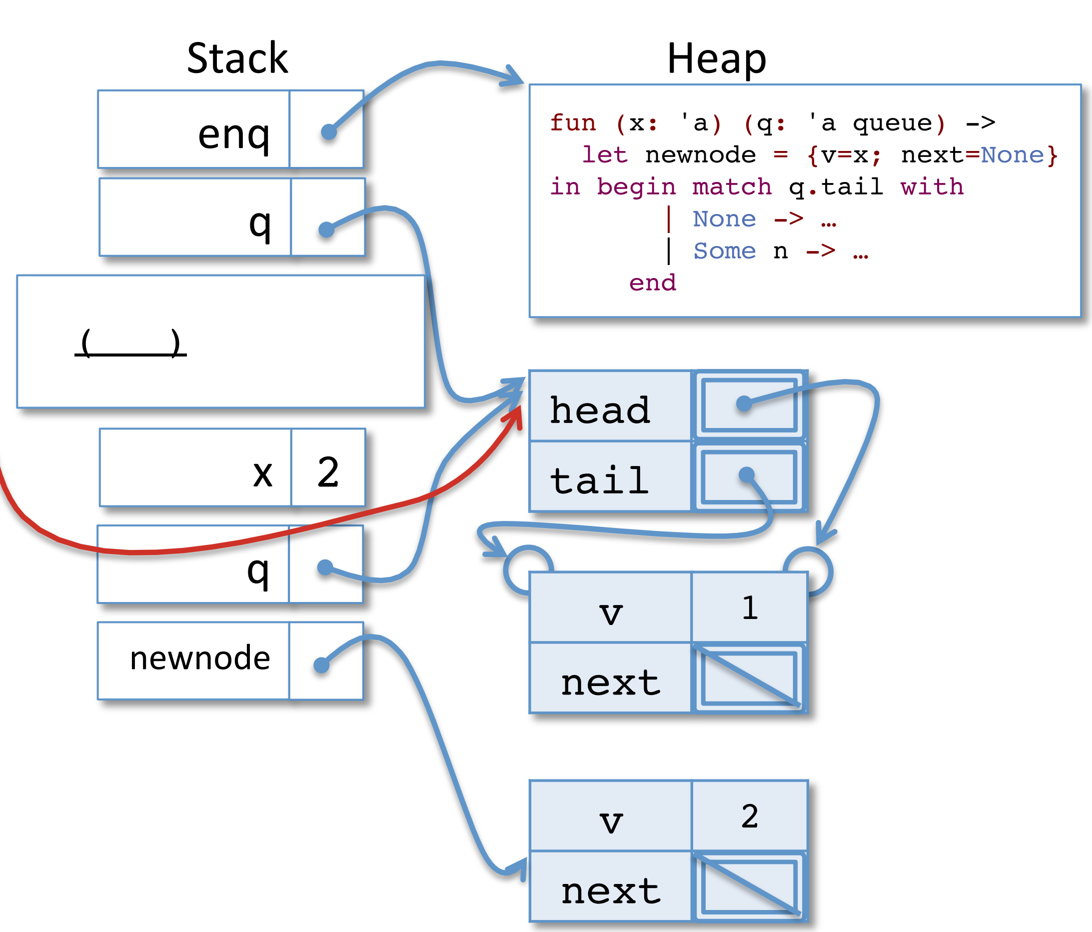
```
begin match q.tail with
   | None ->
     q.head <- Some newnode;
     q.tail <- Some newnode
   | Some n ->
     n.next <- Some newnode;
     q.tail <- Some newnode
end
```



Calling Enq on a non-empty queue

Workspace

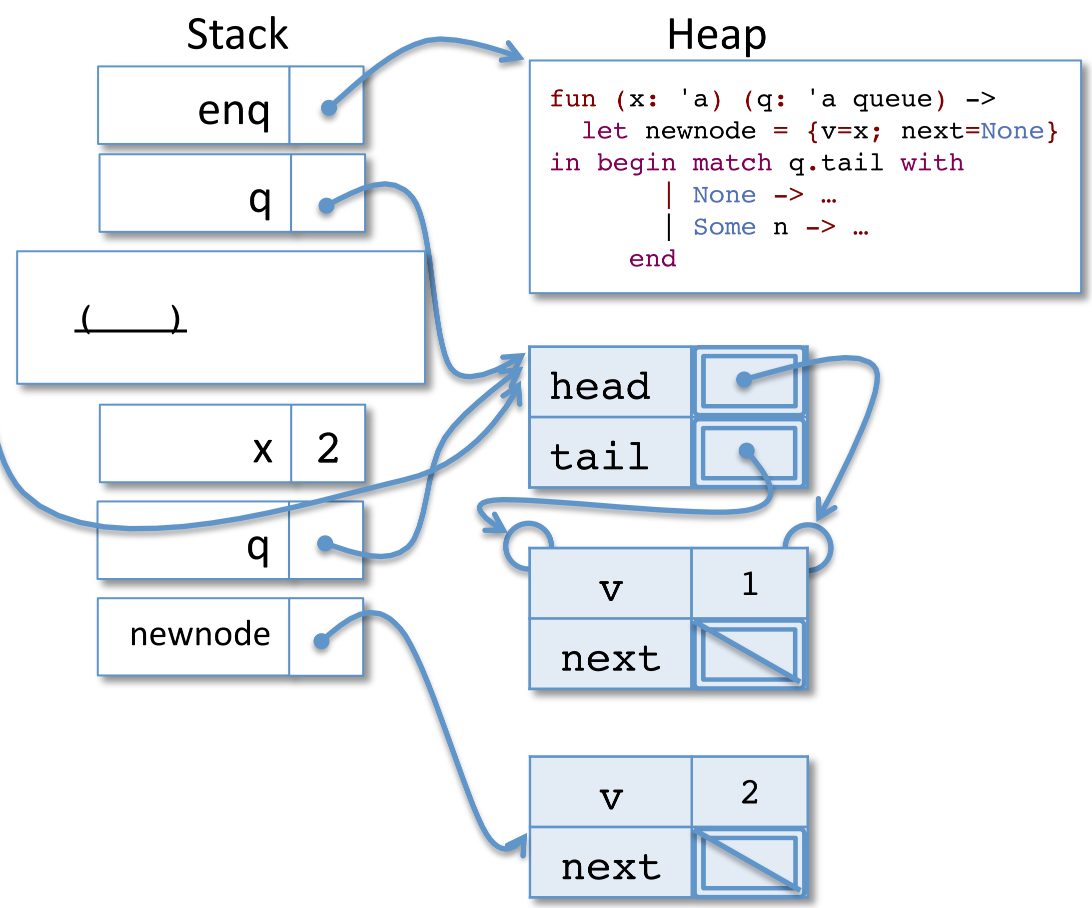
```
begin match  .tail with
   | None ->
     q.head <- Some newnode;
     q.tail <- Some newnode
   | Some n ->
     n.next <- Some newnode;
     q.tail <- Some newnode
end
```



Calling Enq on a non-empty queue

Workspace

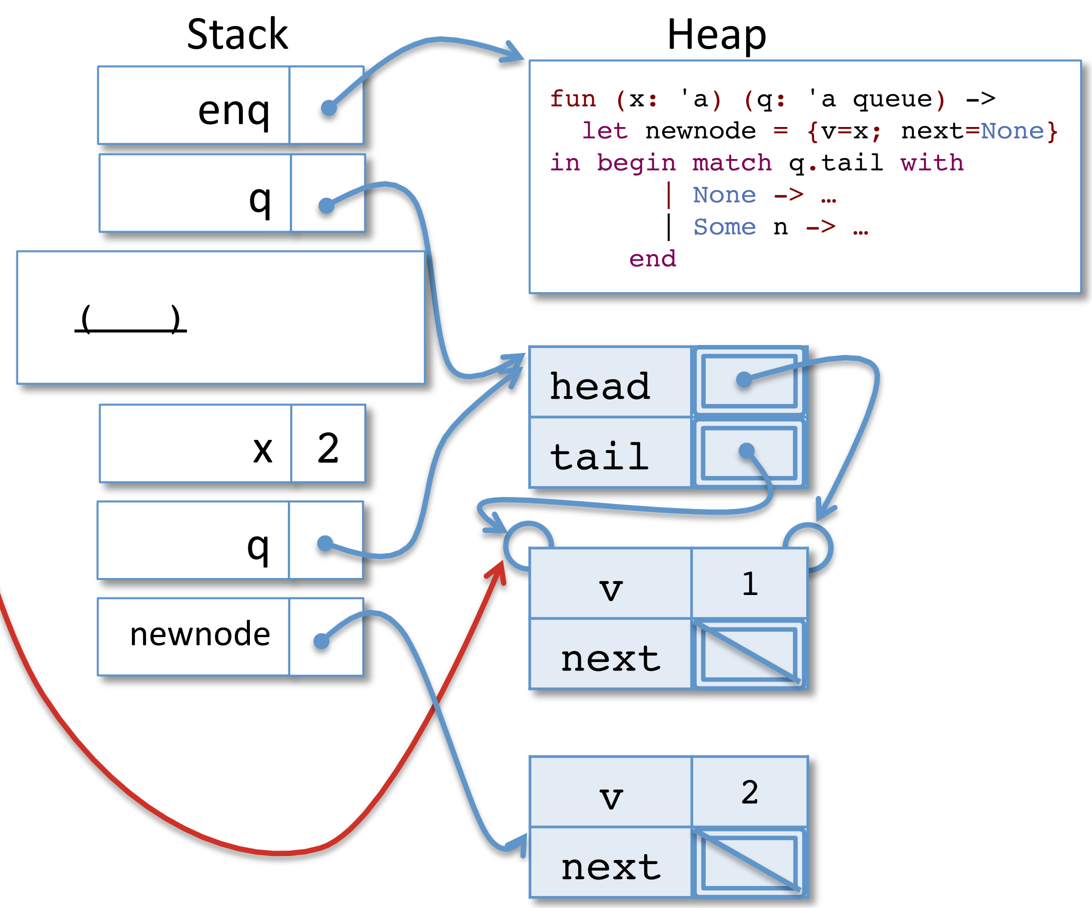
```
begin match ___.tail with
   | None ->
     q.head <- Some newnode;
     q.tail <- Some newnode
   | Some n ->
     n.next <- Some newnode;
     q.tail <- Some newnode
end
```



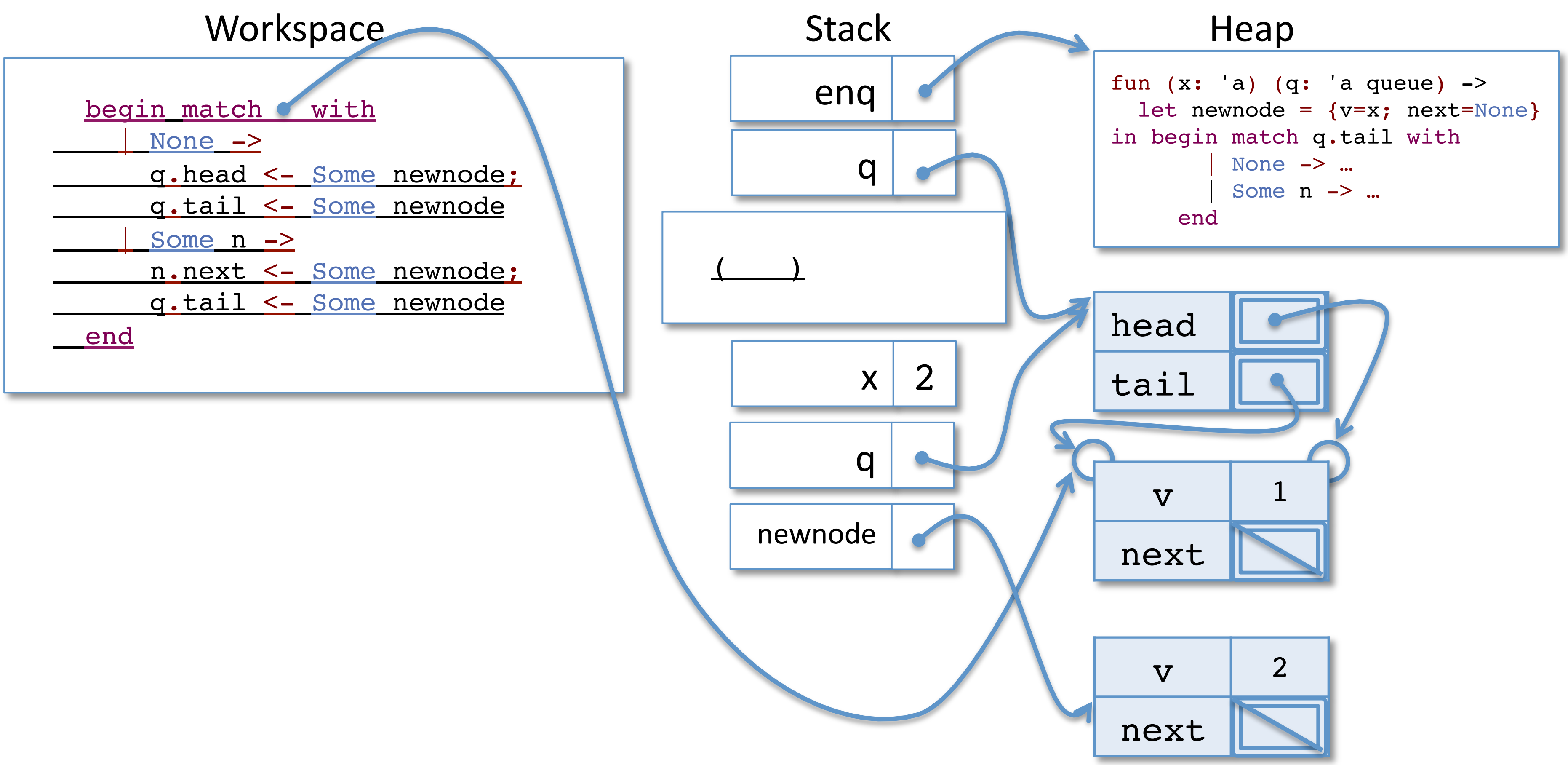
Calling Enq on a non-empty queue

Workspace

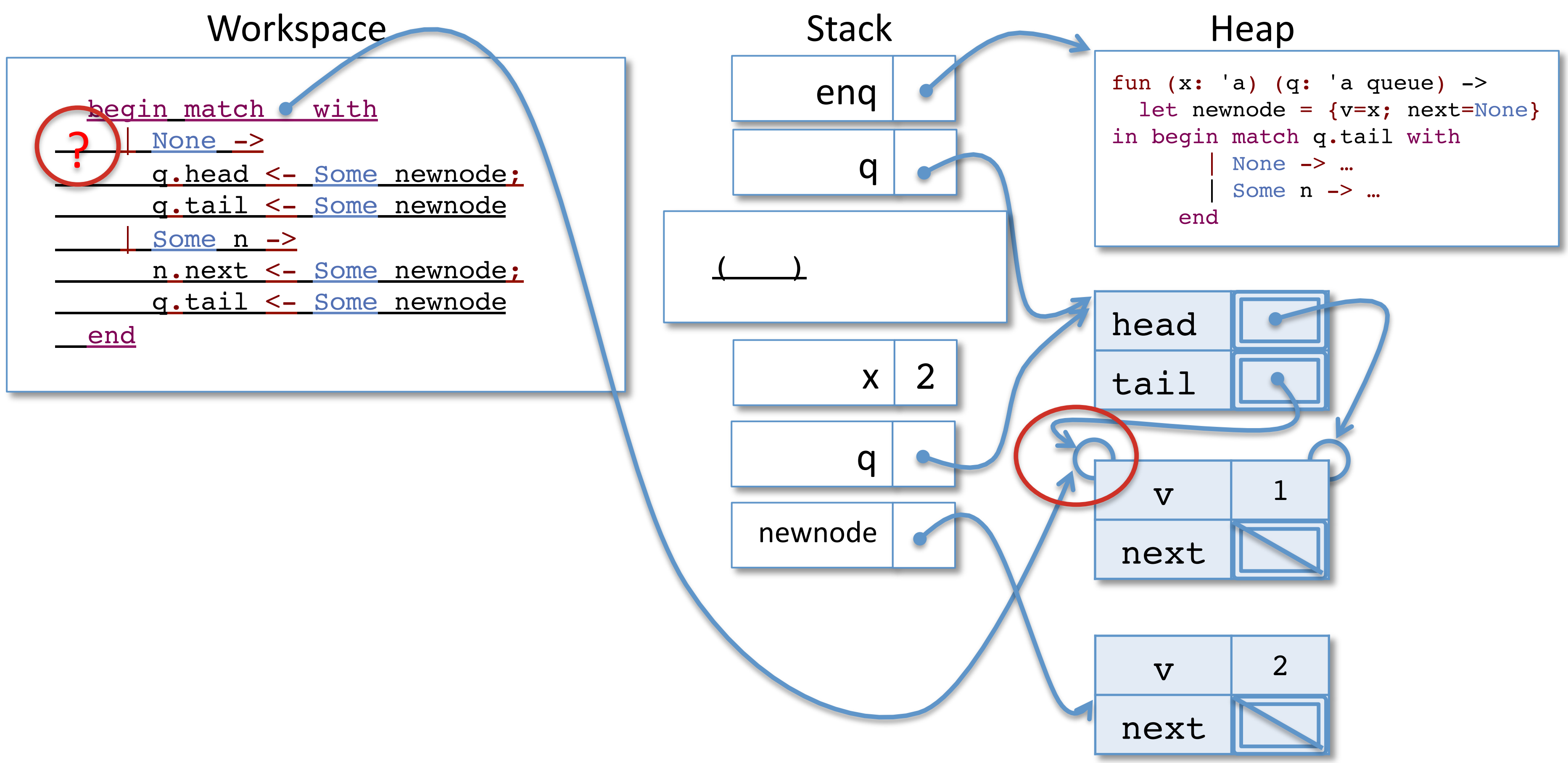
```
begin match ● with
  | None ->
    q.head <- Some newnode;
    q.tail <- Some newnode
  | Some n ->
    n.next <- Some newnode;
    q.tail <- Some newnode
end
```



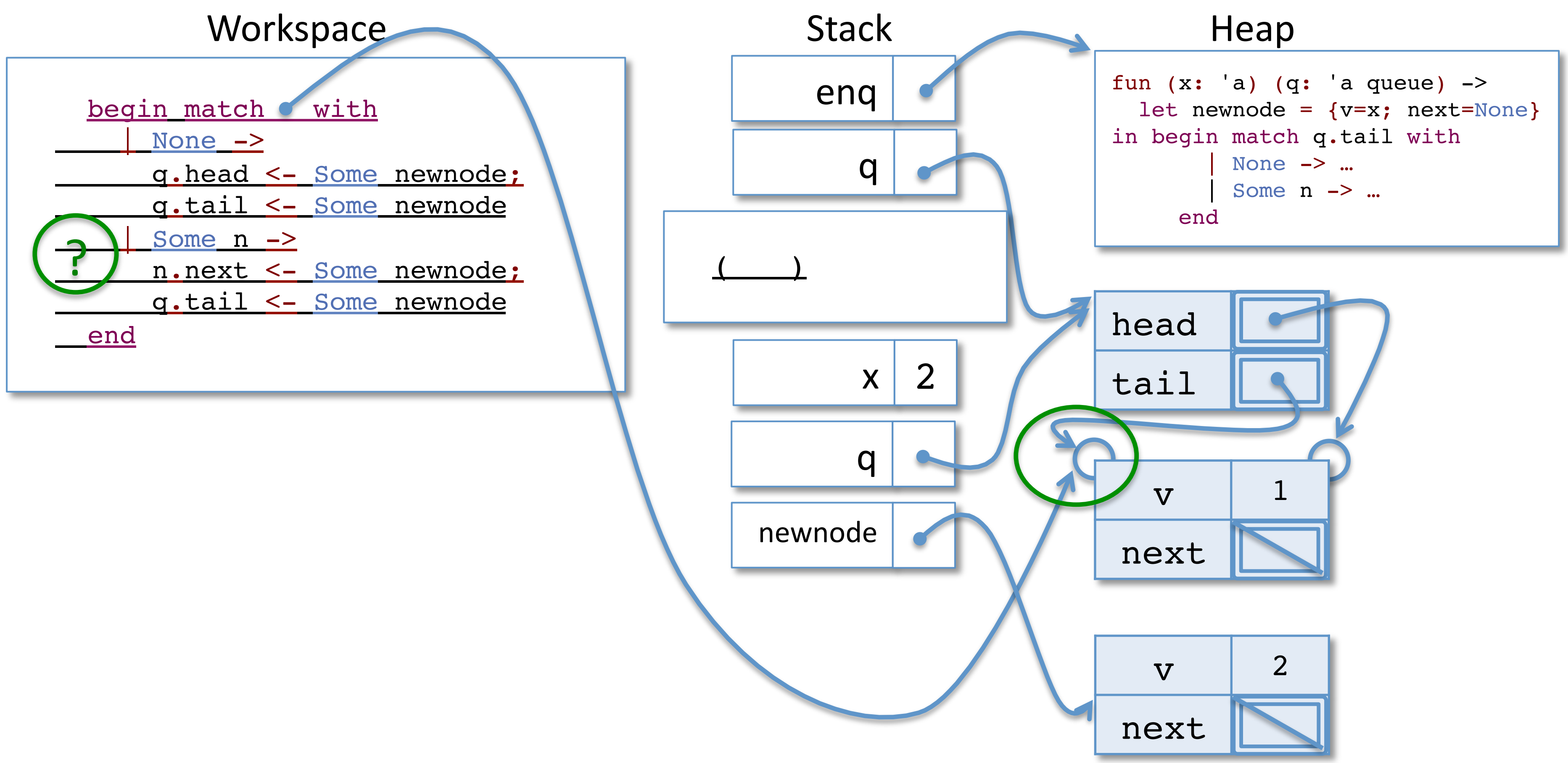
Calling Enq on a non-empty queue



Calling Enq on a non-empty queue



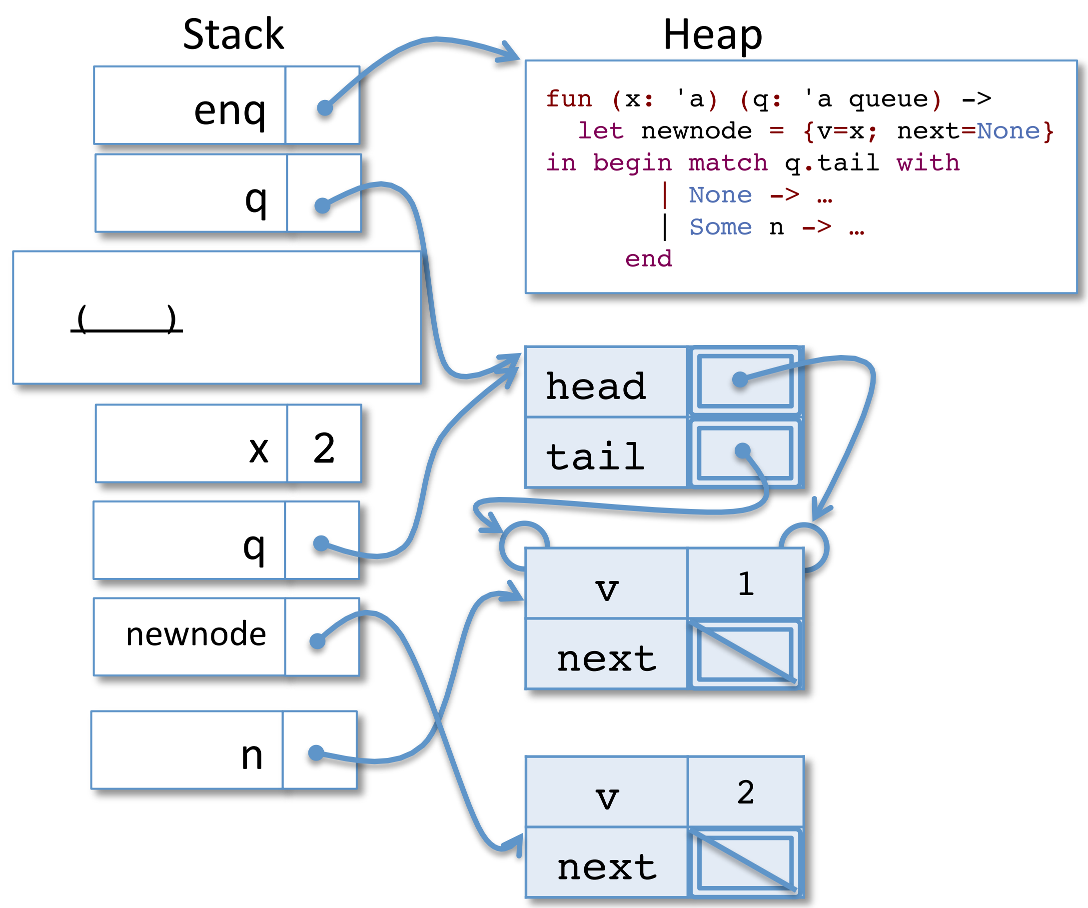
Calling Enq on a non-empty queue



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.

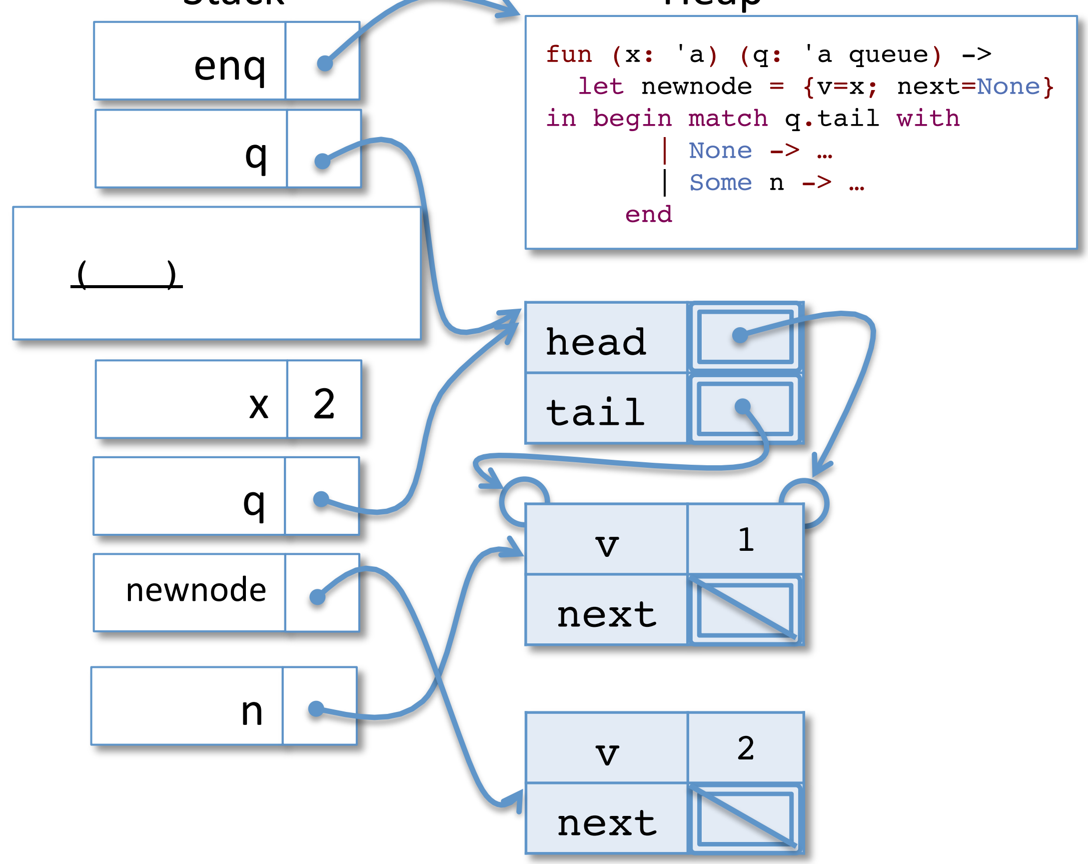
Calling Enq on a non-empty queue

Workspace

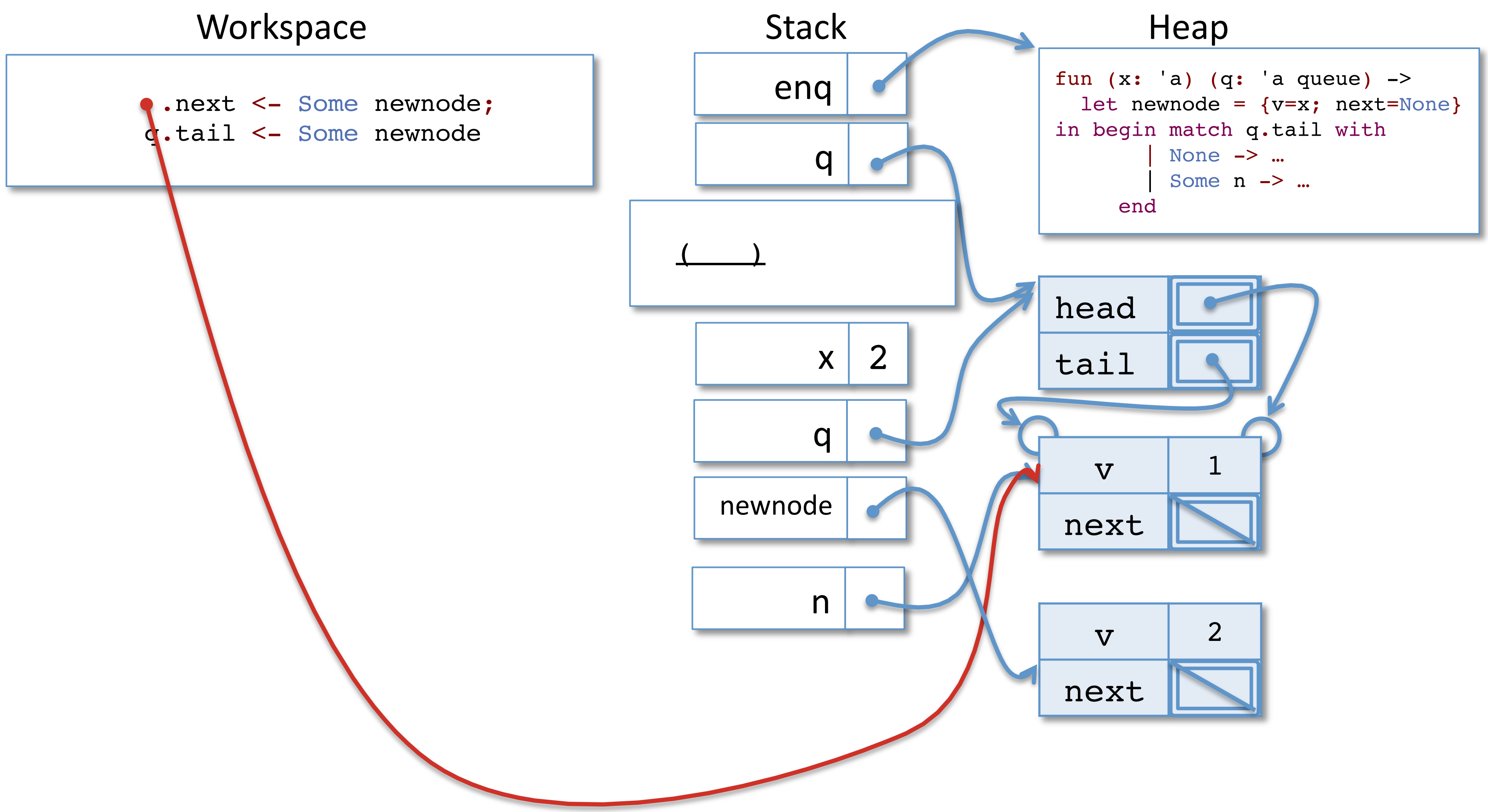
```
n.next <- Some newnode;
q.tail <- Some newnode
```

Stack

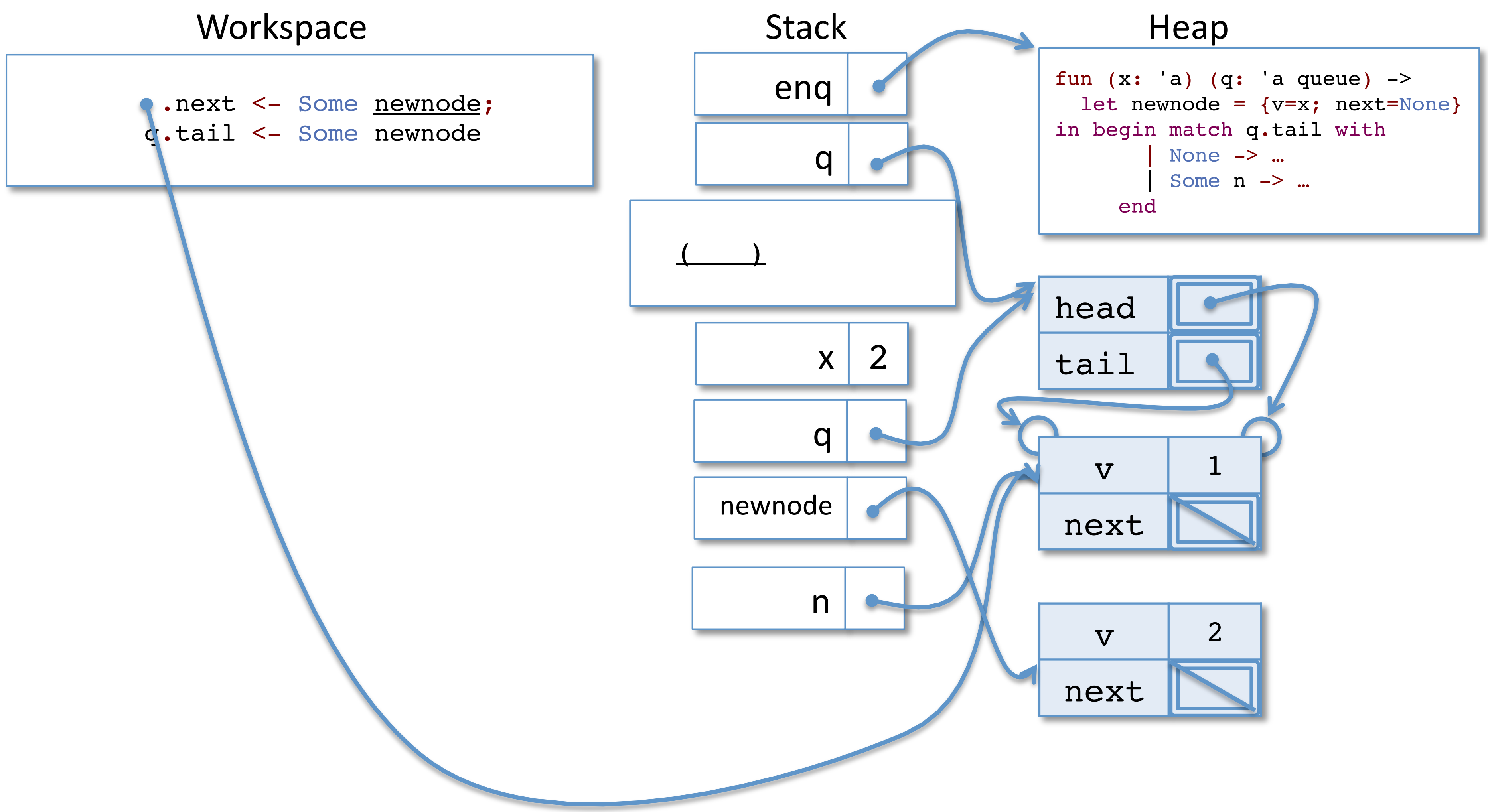
Heap



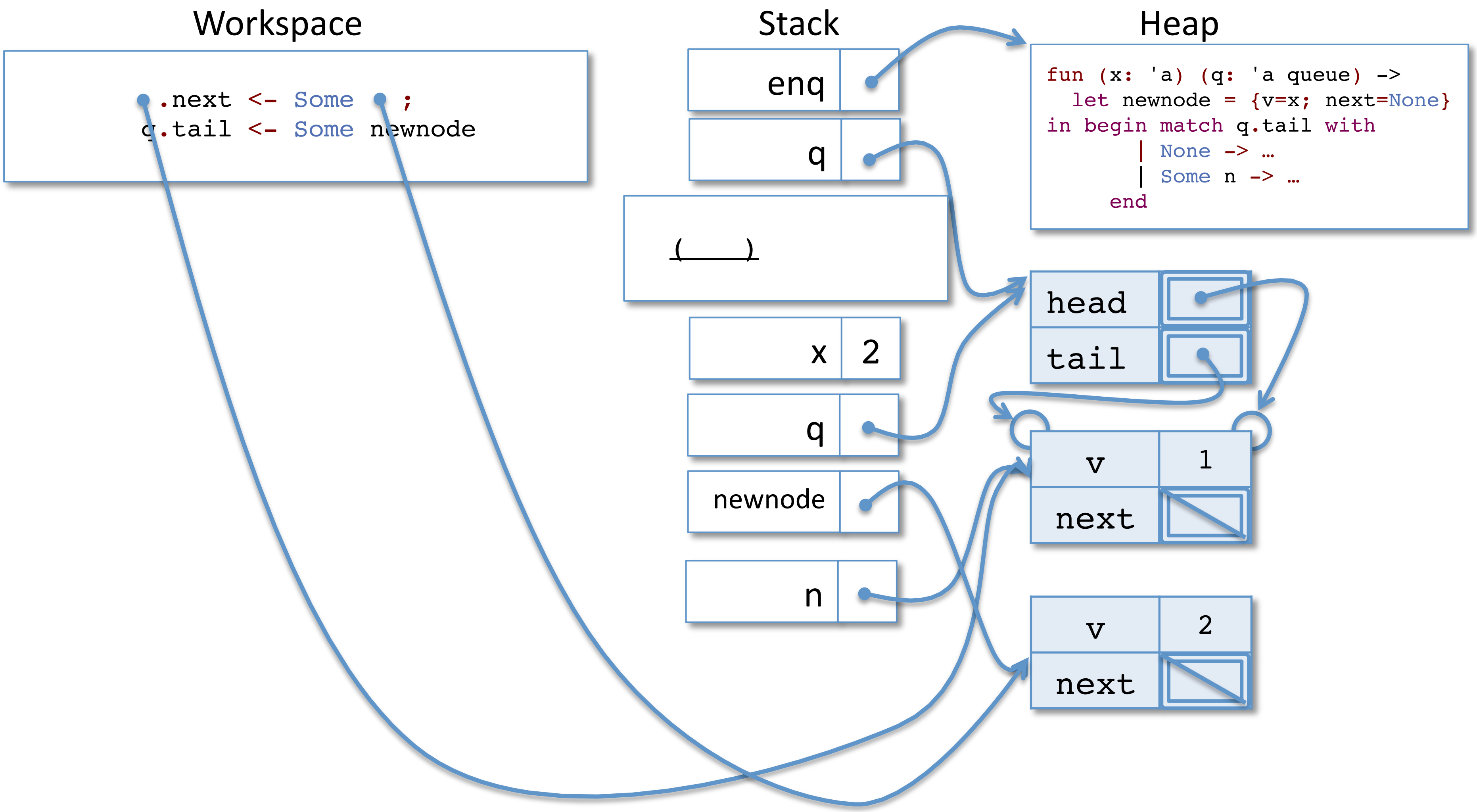
Calling Enq on a non-empty queue



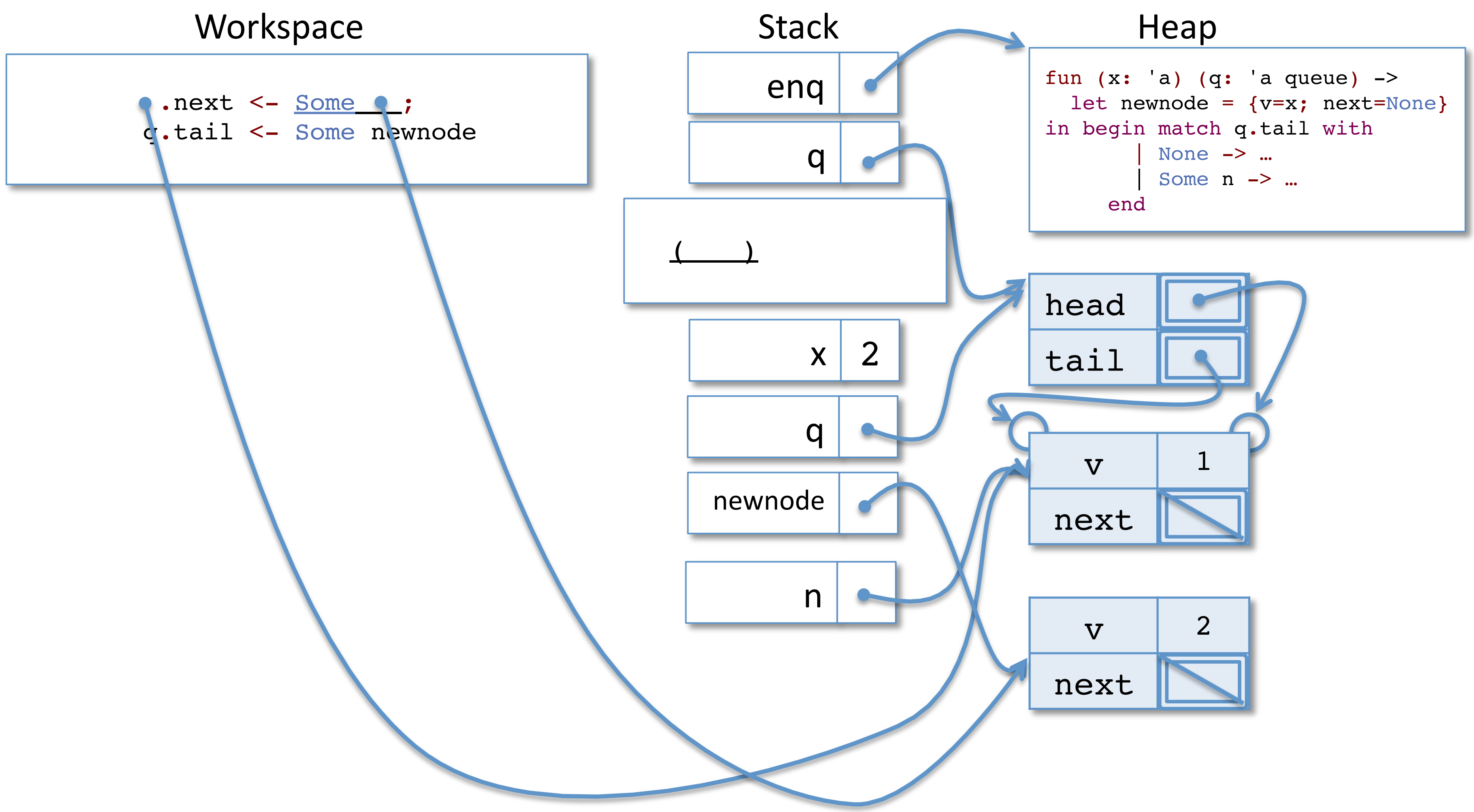
Calling Enq on a non-empty queue



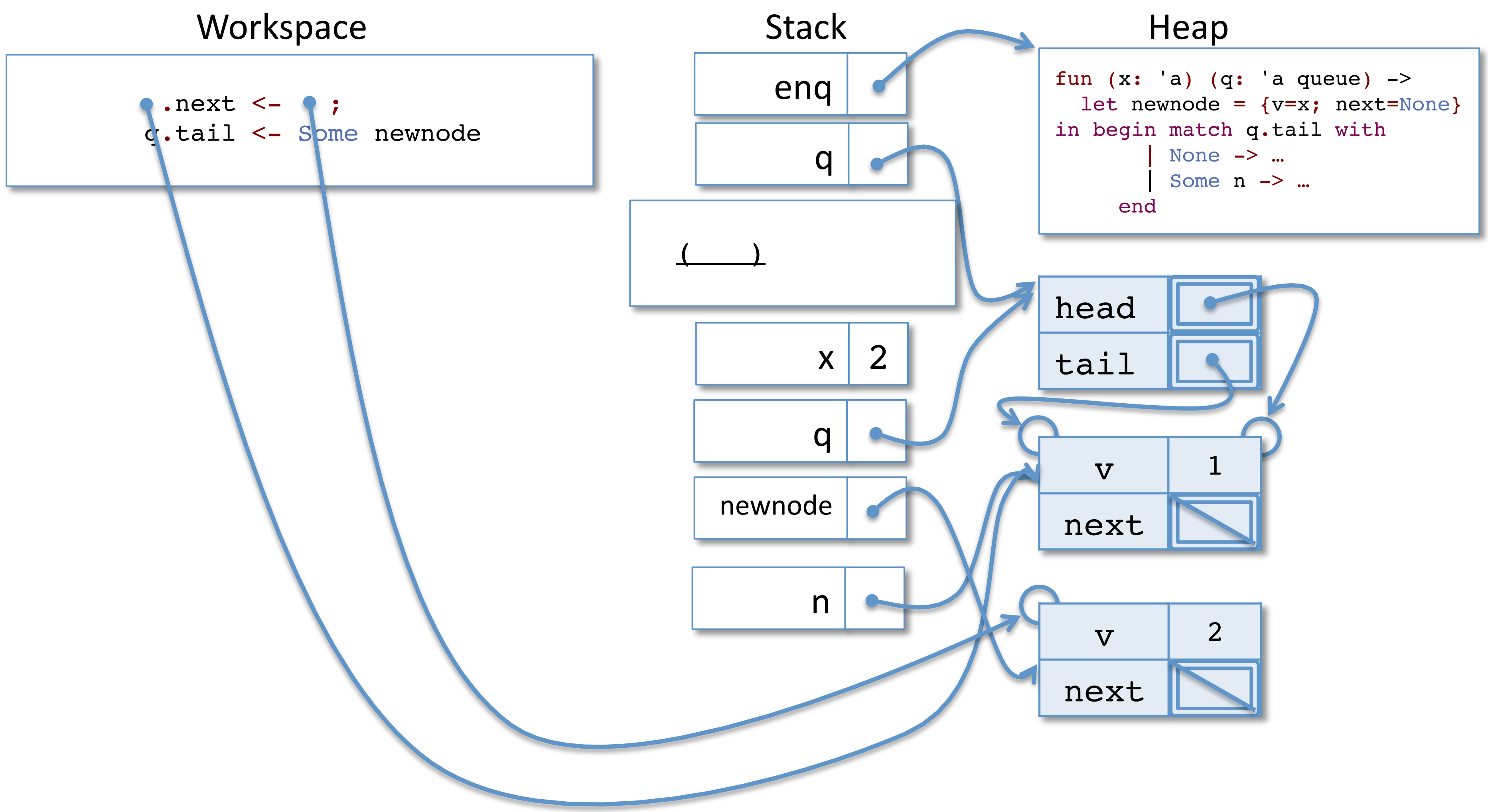
Calling Enq on a non-empty queue



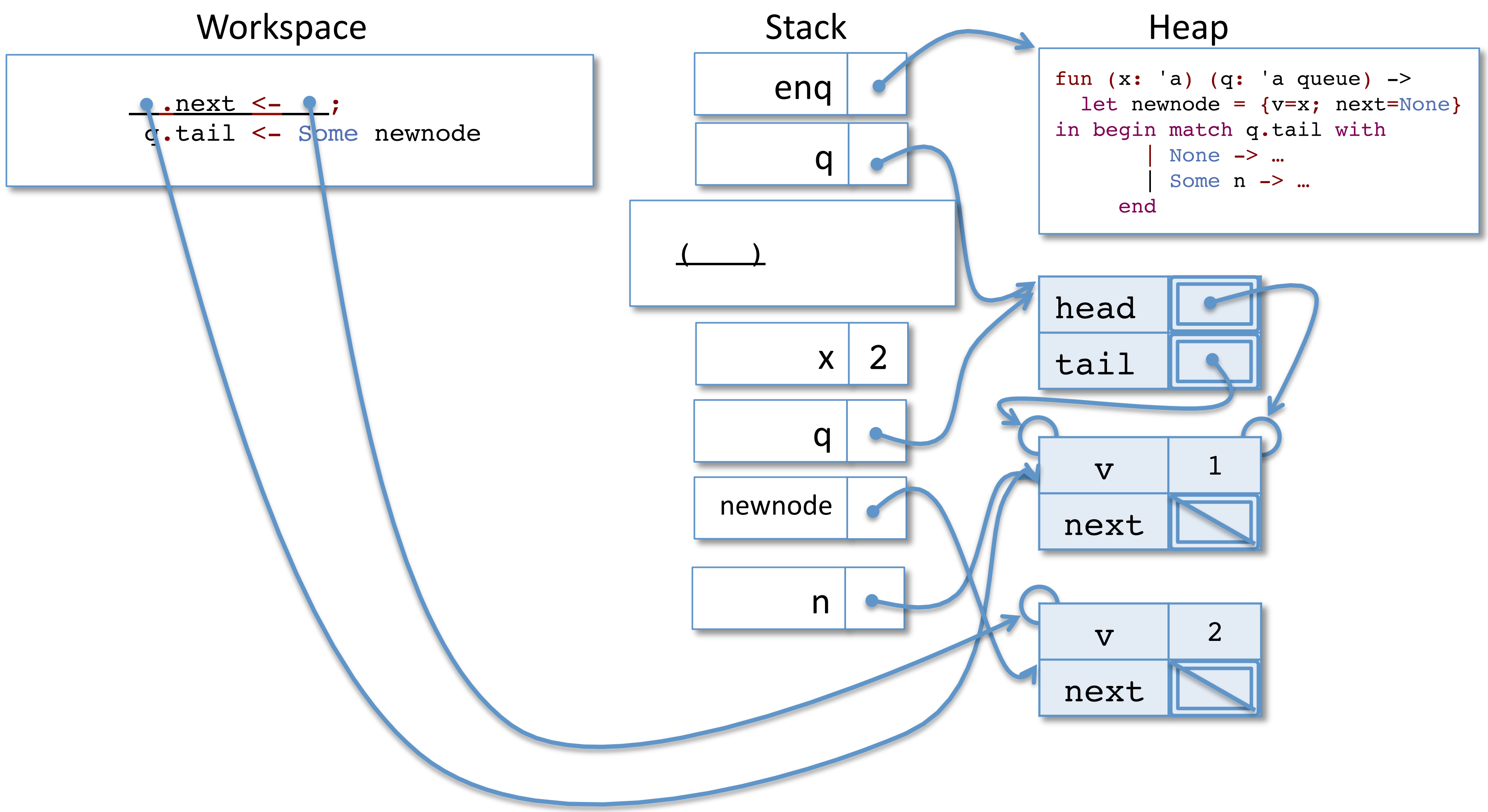
Calling Enq on a non-empty queue



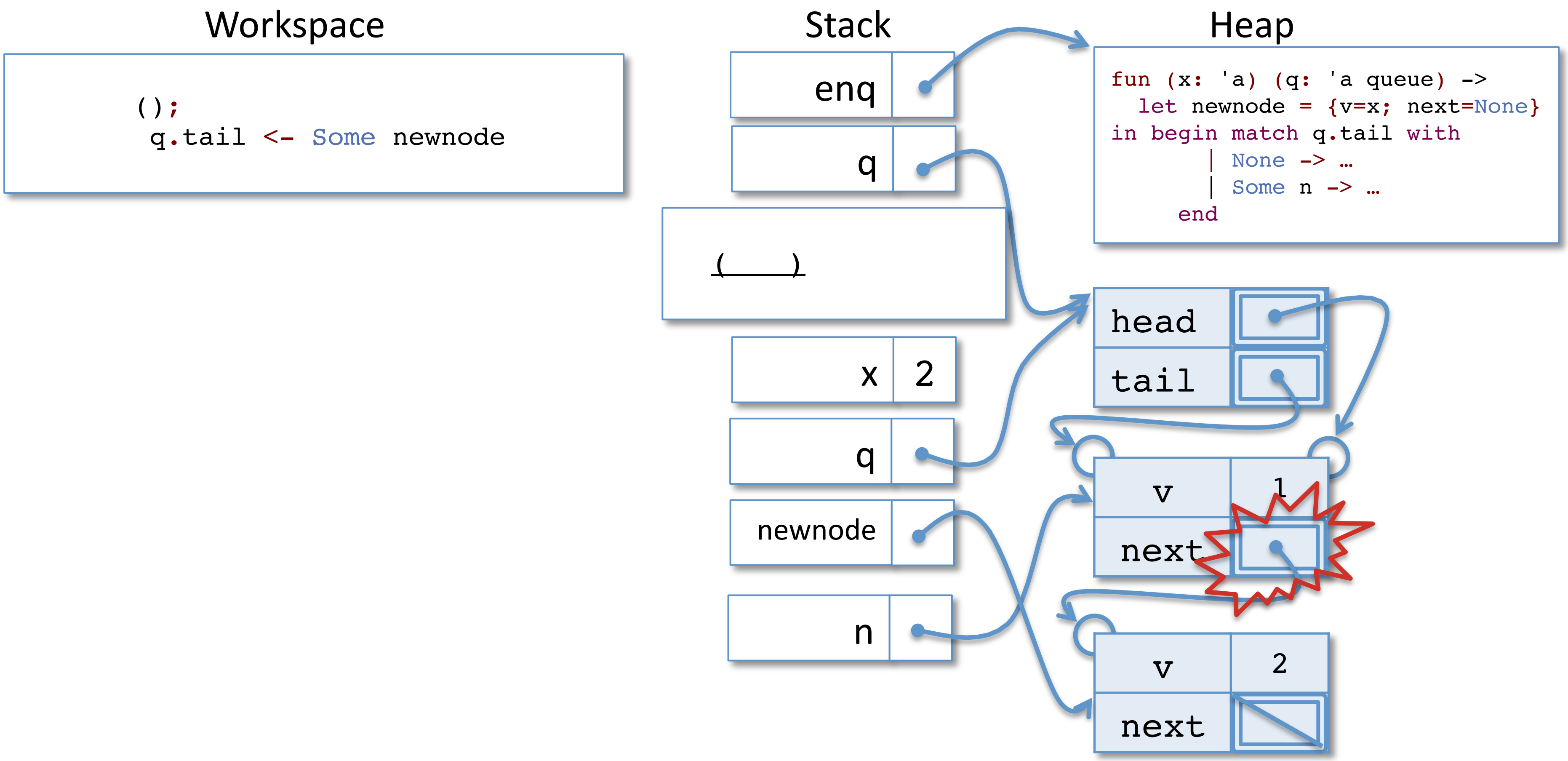
Calling Enq on a non-empty queue



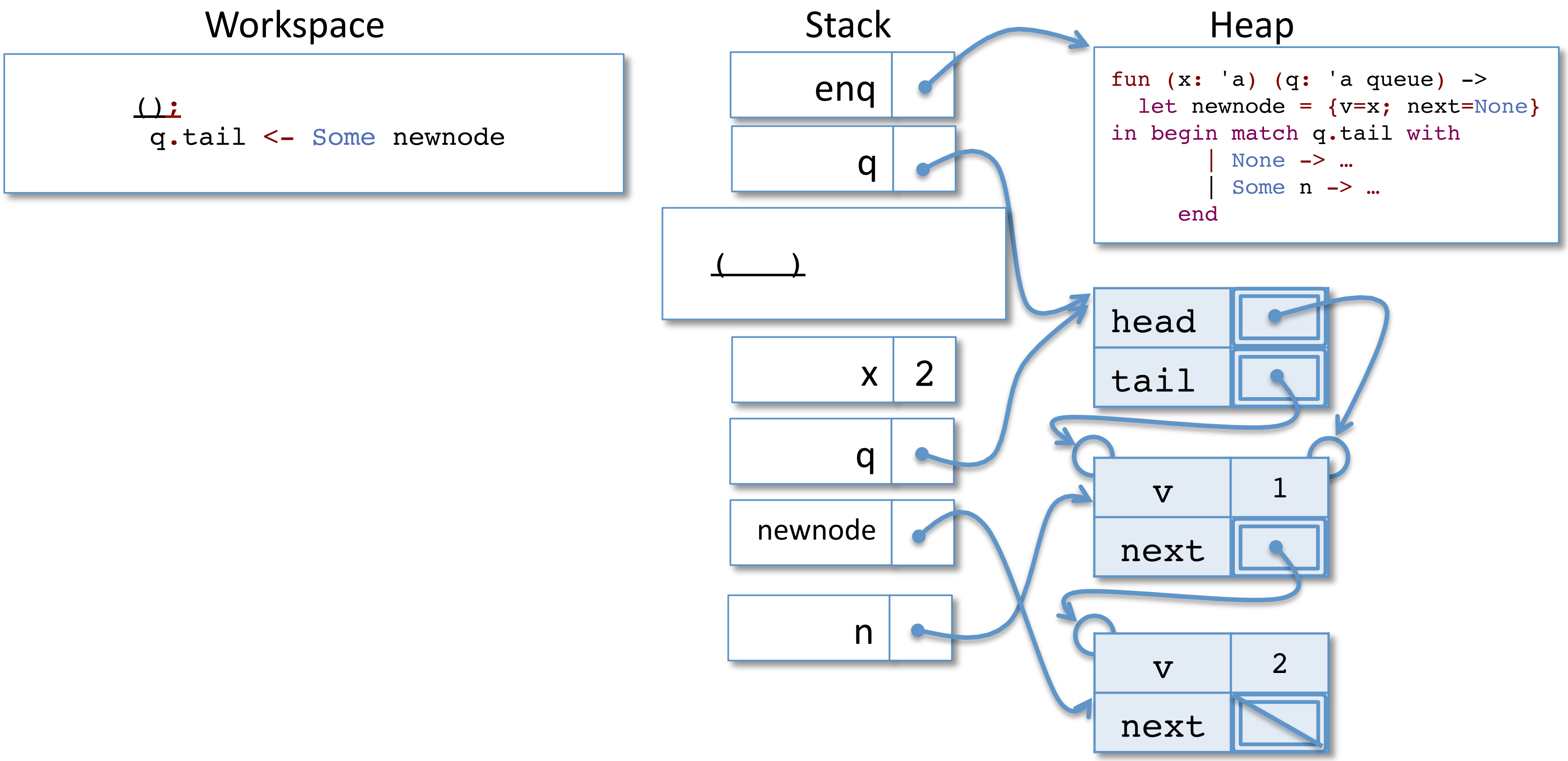
Calling Enq on a non-empty queue



Calling Enq on a non-empty queue



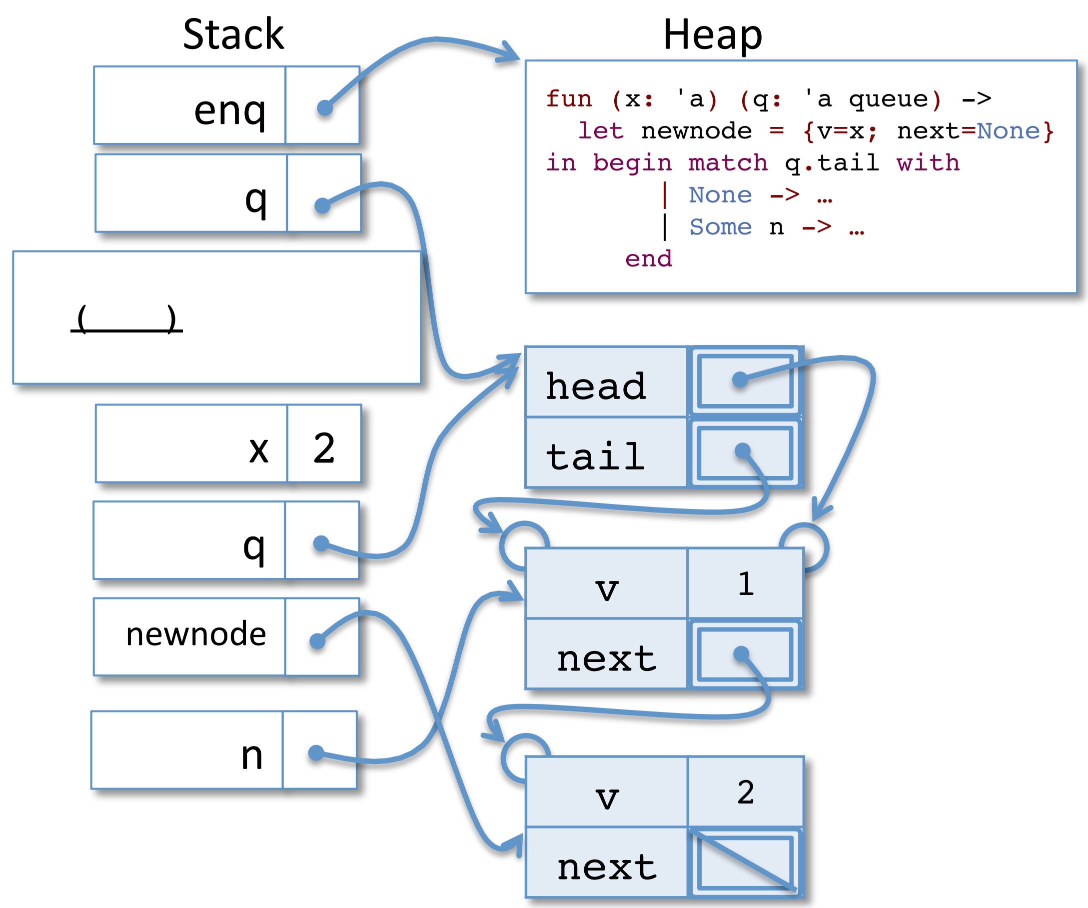
Calling Enq on a non-empty queue



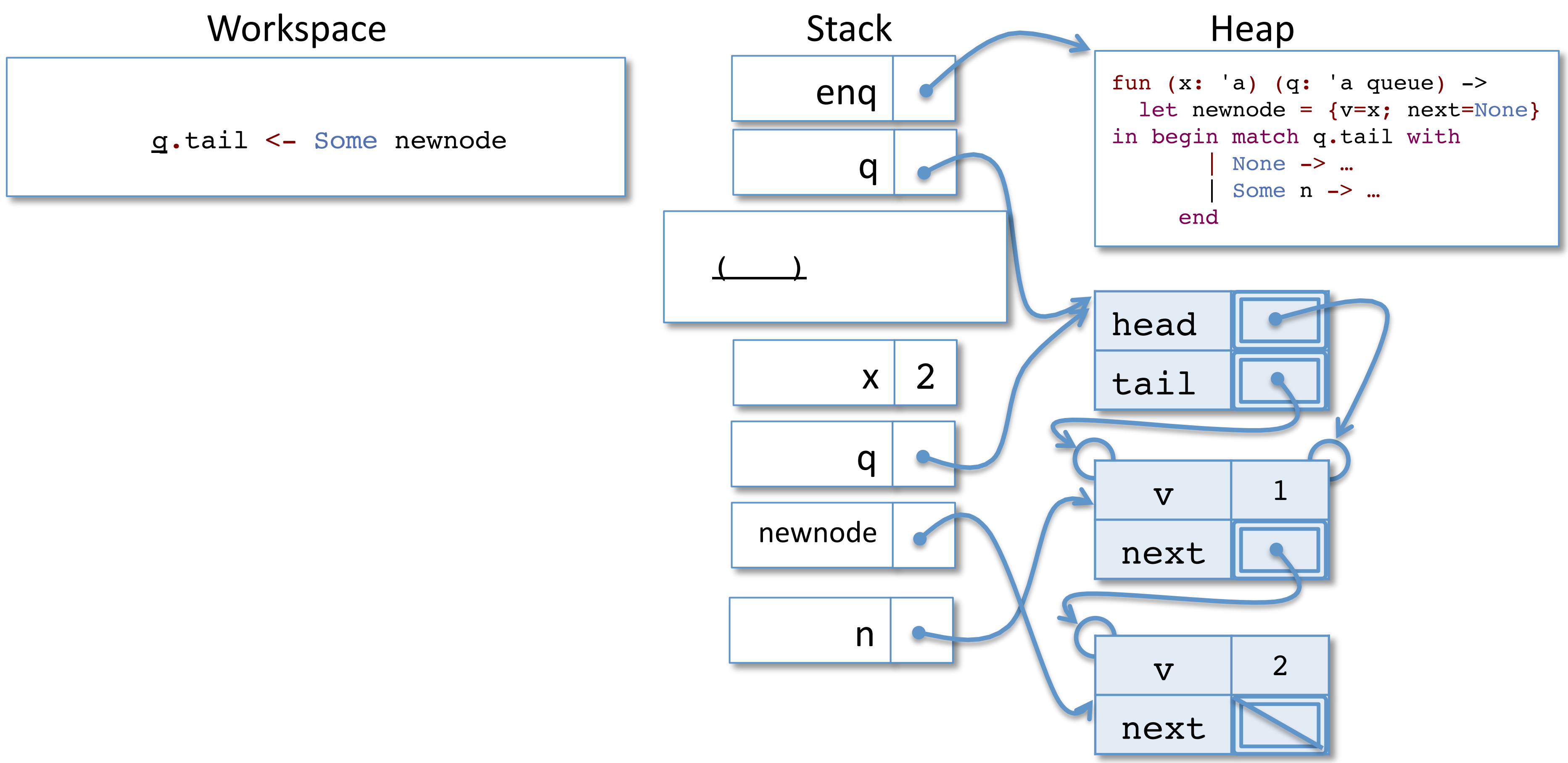
Calling Enq on a non-empty queue

Workspace

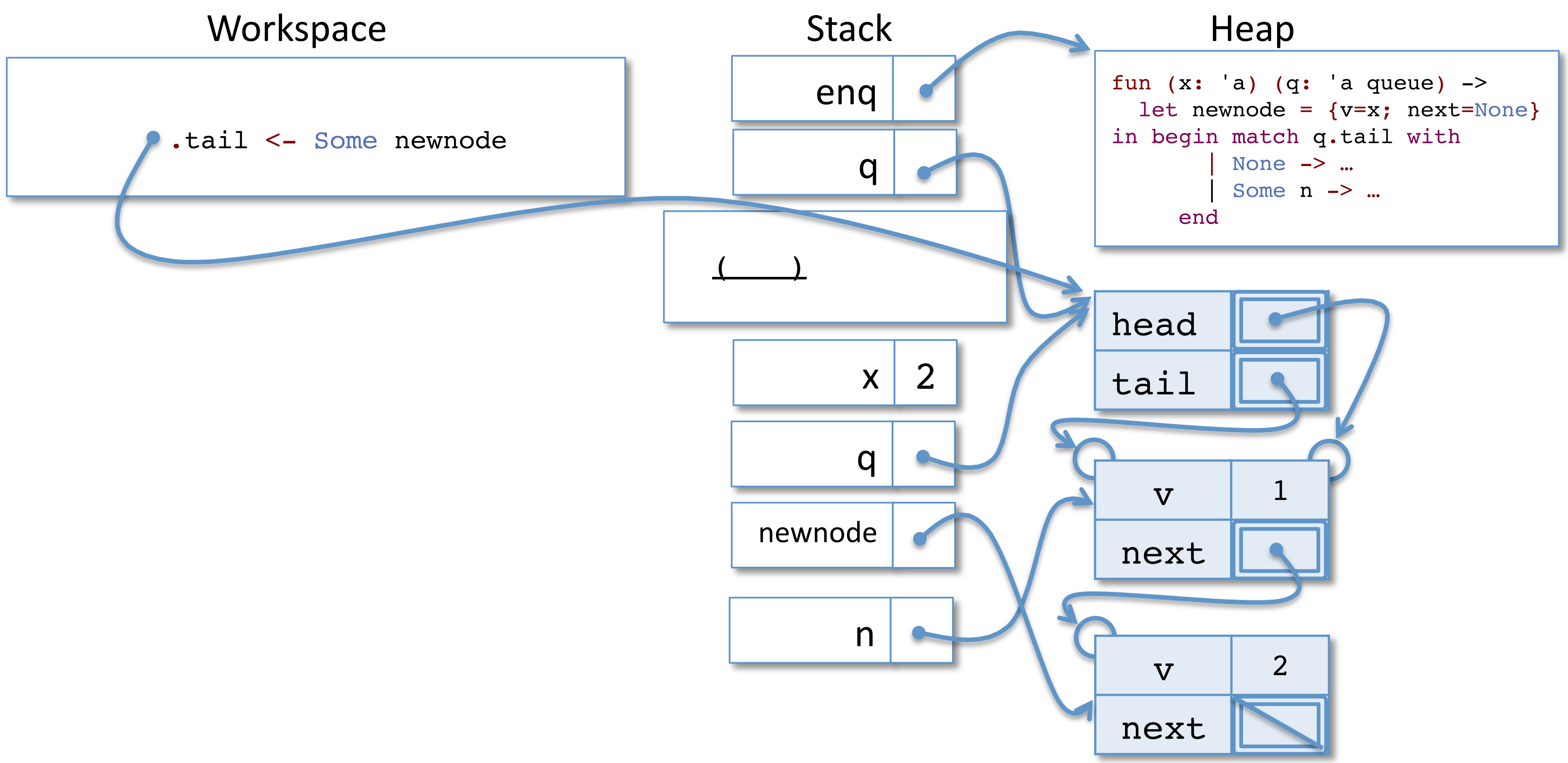
```
q.tail <- Some newnode
```



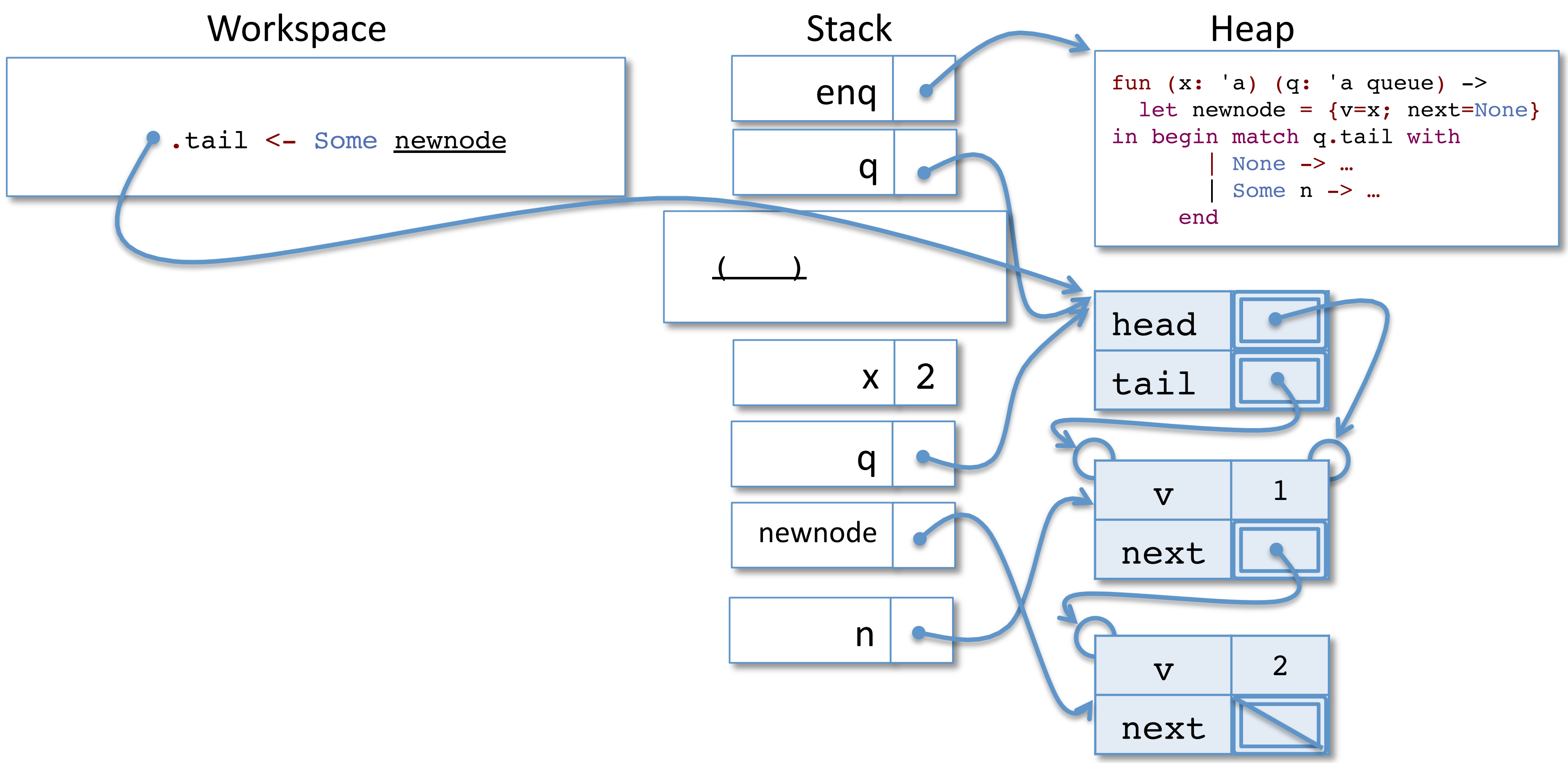
Calling Enq on a non-empty queue



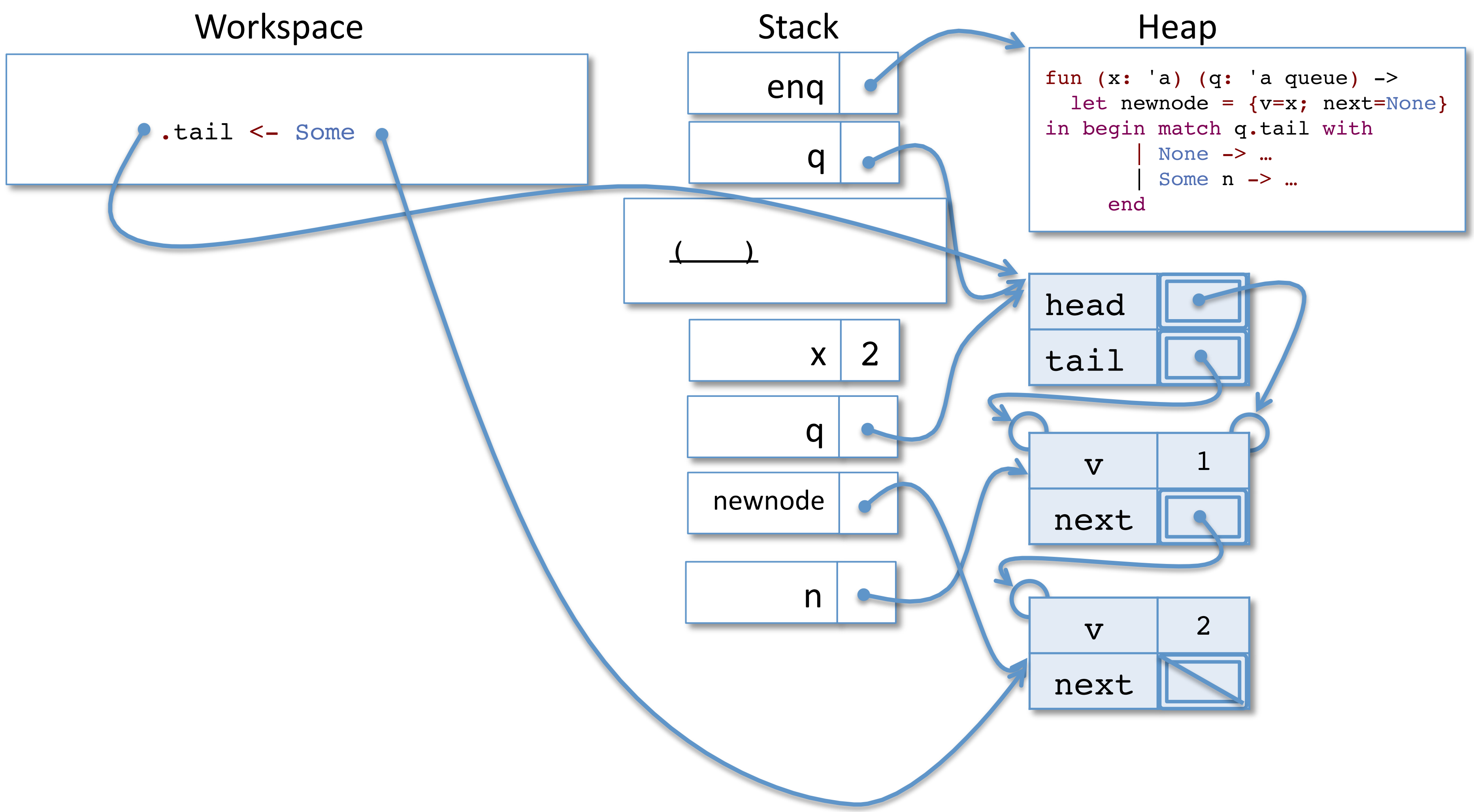
Calling Enq on a non-empty queue



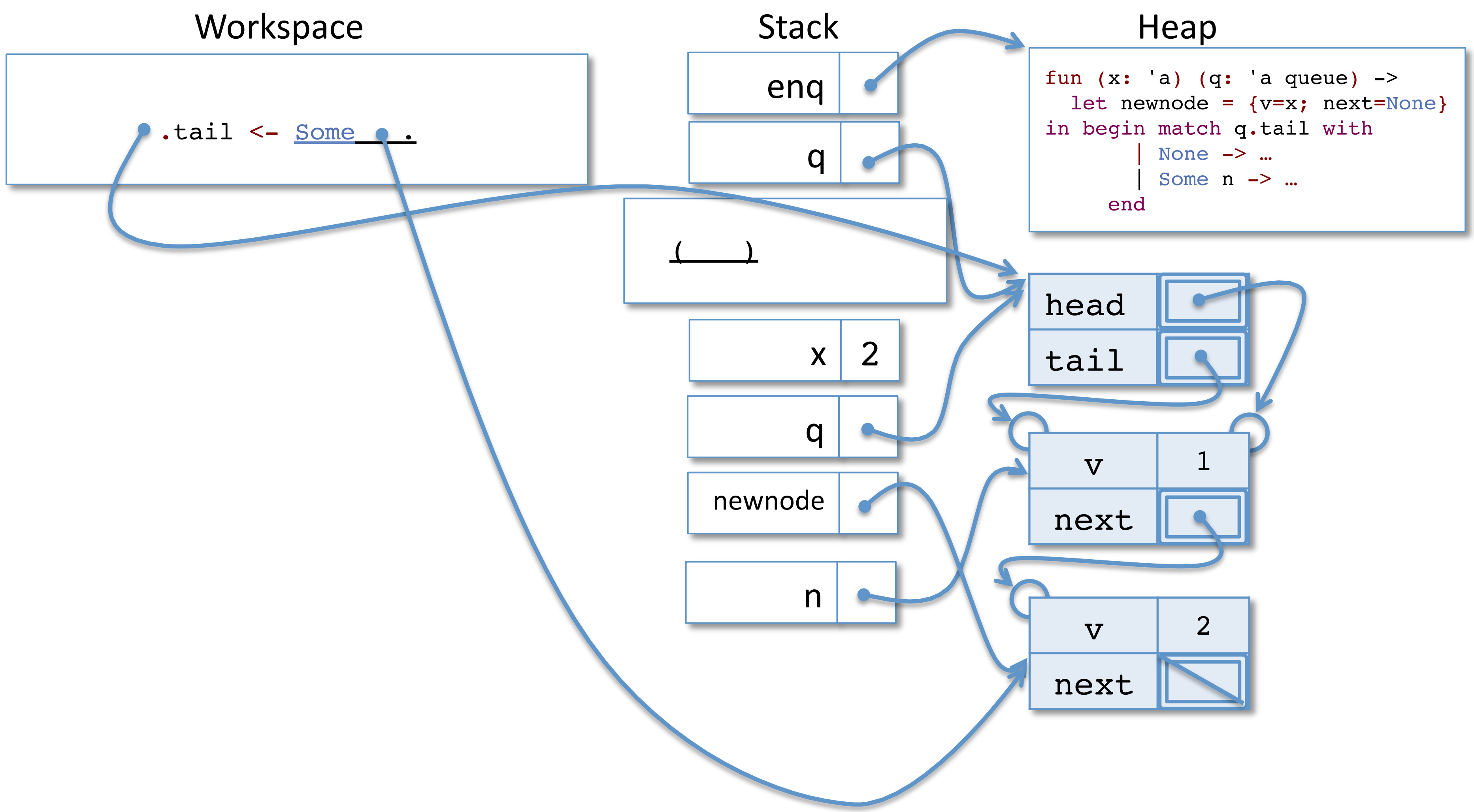
Calling Enq on a non-empty queue



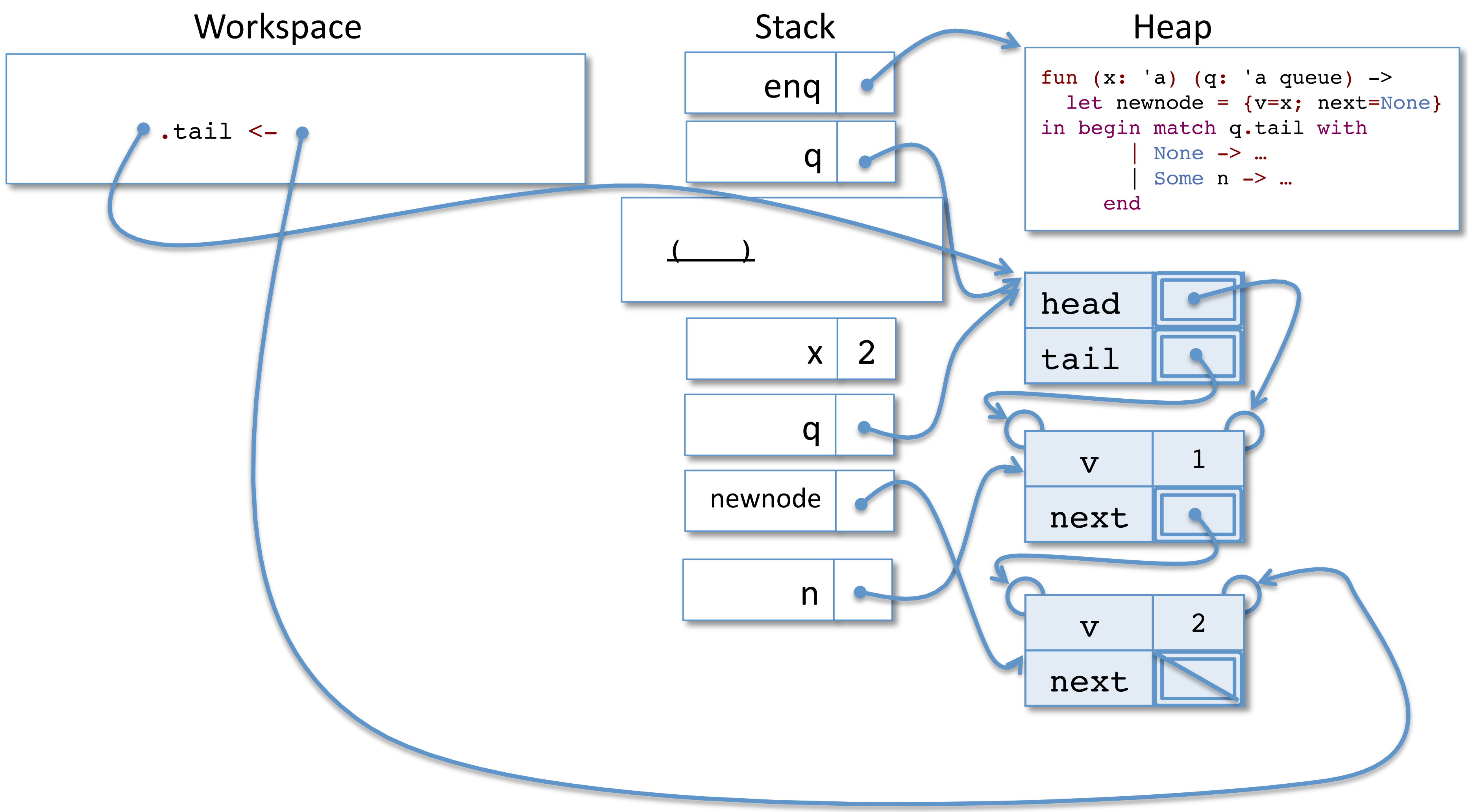
Calling Enq on a non-empty queue



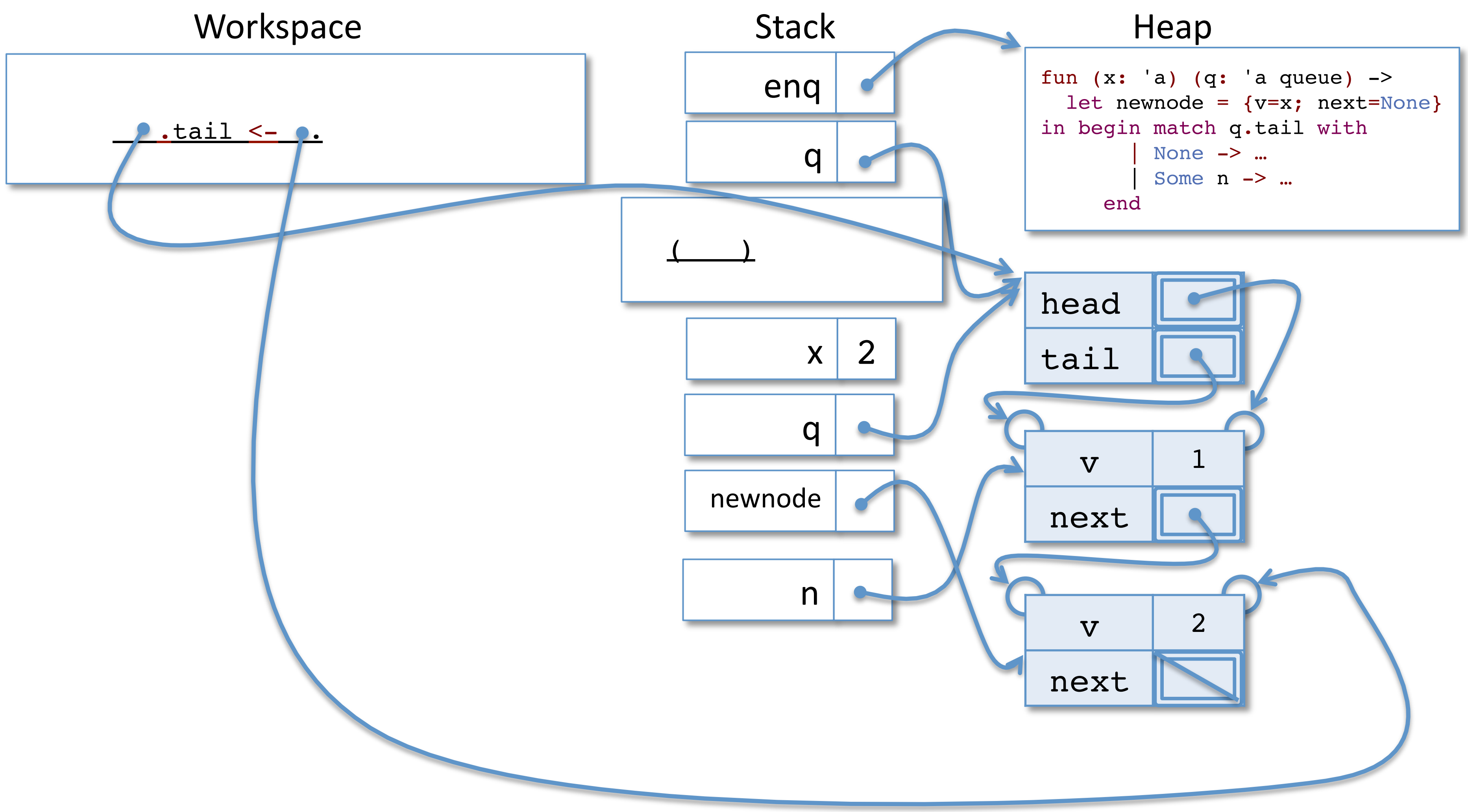
Calling Enq on a non-empty queue



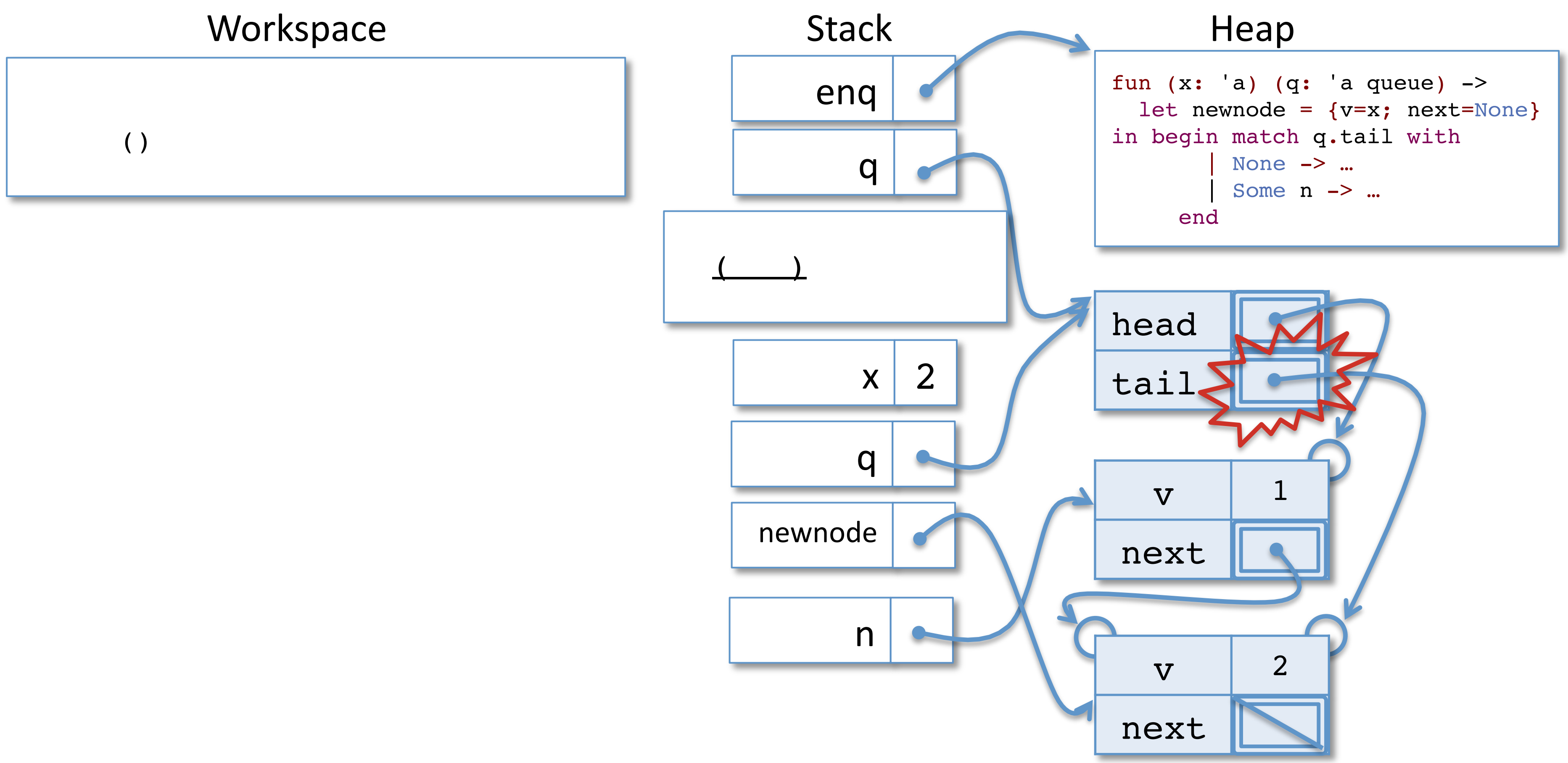
Calling Enq on a non-empty queue



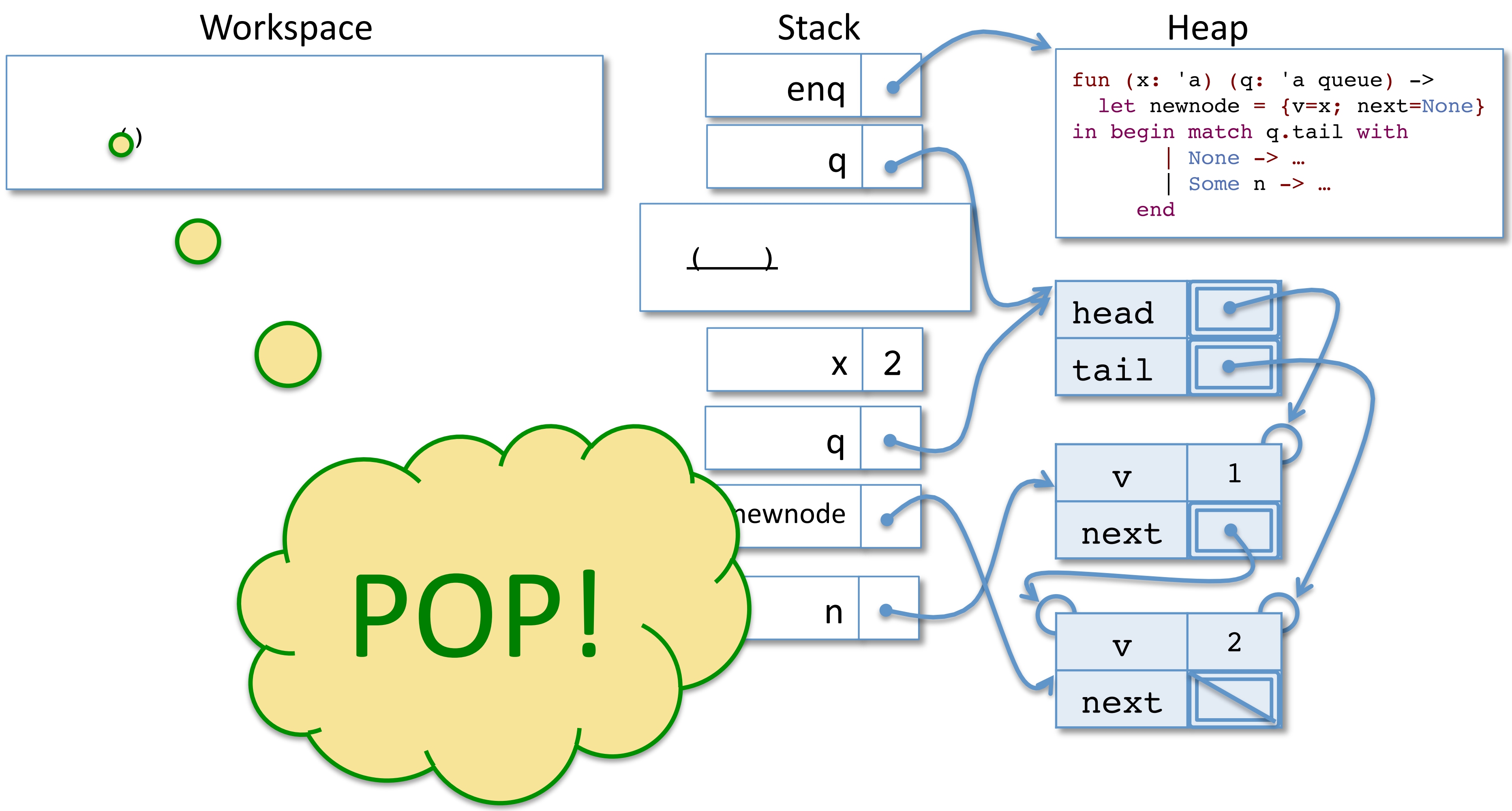
Calling Enq on a non-empty queue



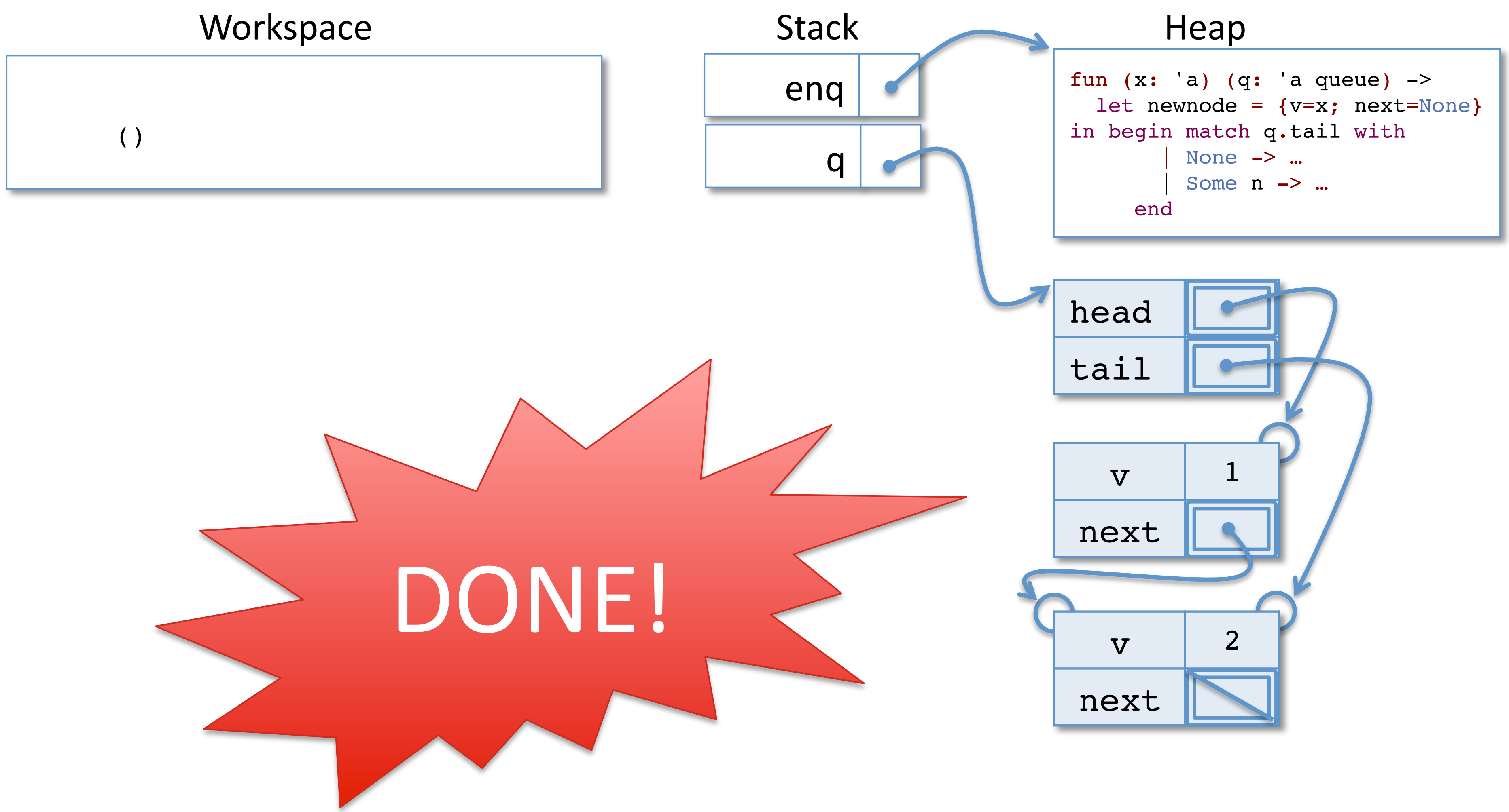
Calling Enq on a non-empty queue



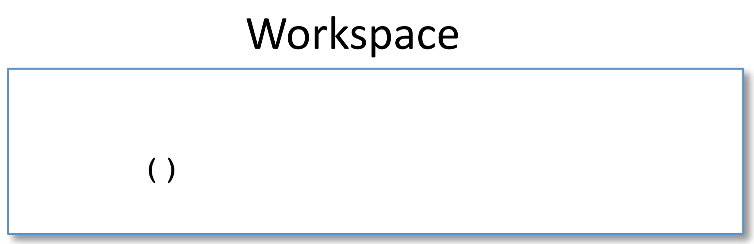
Calling Enq on a non-empty queue

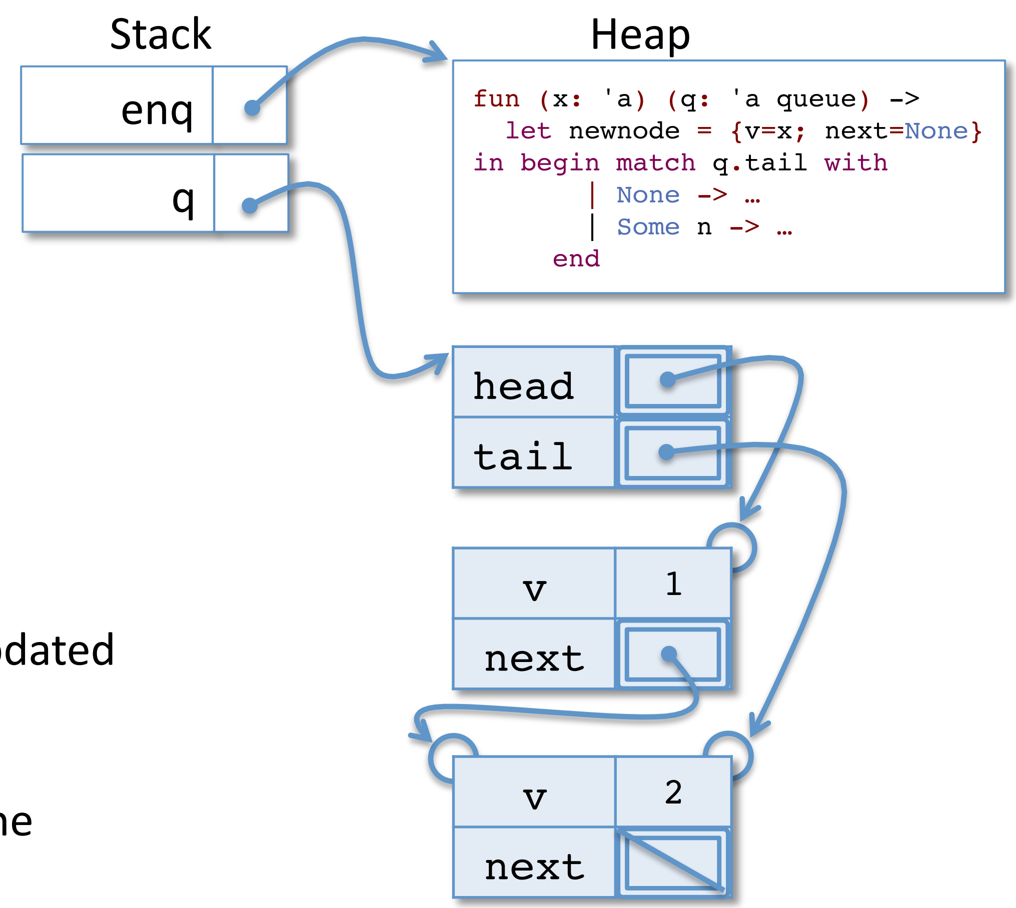


Calling Enq on a non-empty queue



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

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