Programming Languages and Techniques (CIS120)

Lecture 16

February 19, 2016

Queues via Linked Lists

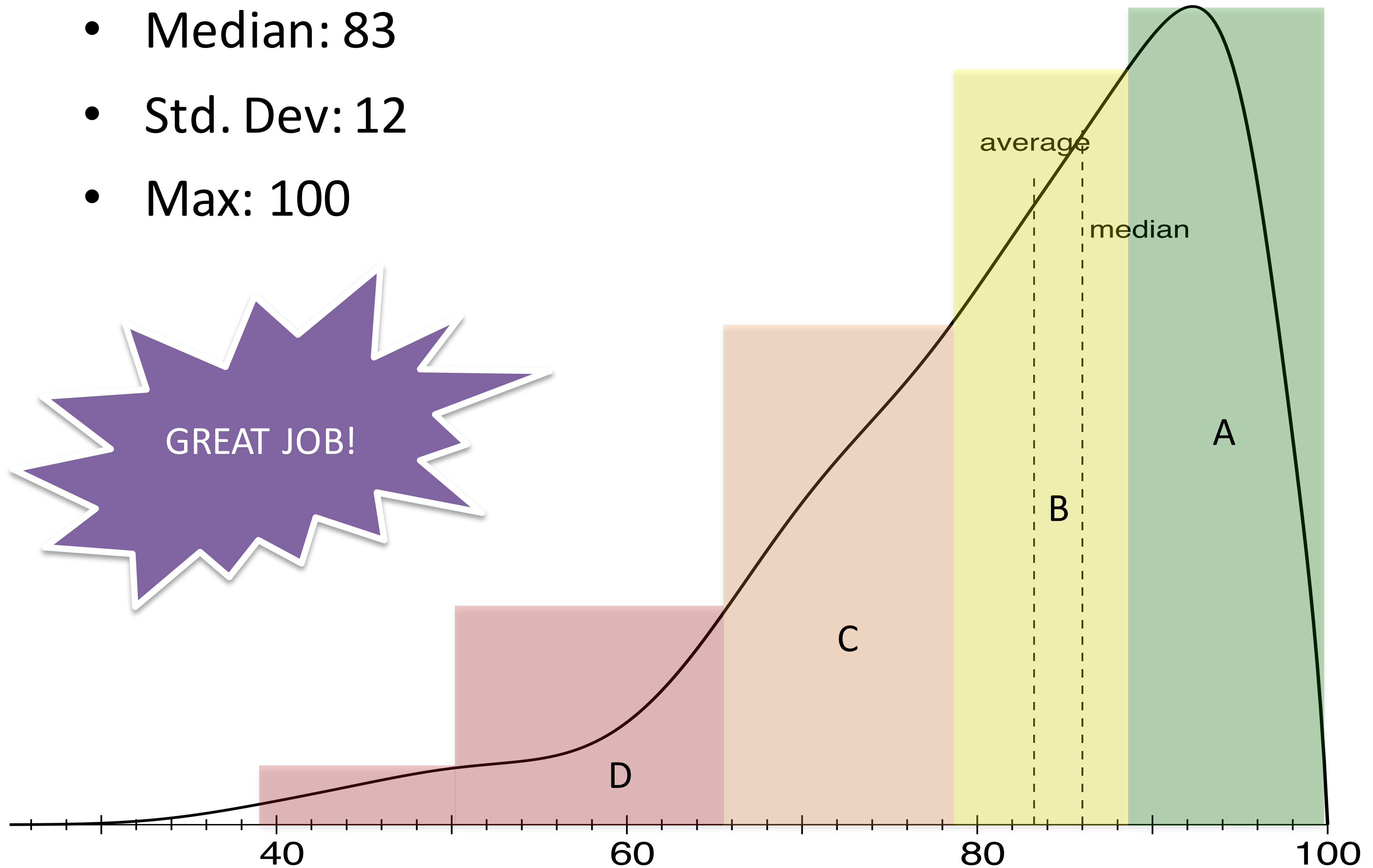
Lecture notes: Chapter 16

Announcements

- HW 4: Queues
  - Due Tuesday, February 23rd at 11:59pm
- Midterm Exam
  - Check scores on "Submit" homework link
  - Graded exam: Will be available for you to examine, copy, etc. in Levine 308 next week
  - Solutions available next week

Midterm 1 Results

- Average: 86
- Median: 83
- Std. Dev: 12
- Max: 100



Mutable Queues: Recap

singly linked data structures

(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
```

Data Structure for Mutable Queues

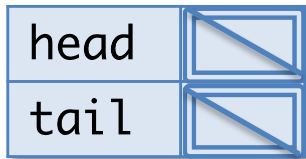
```
type 'a qnode = {
     v: 'a;
     mutable next : 'a qnode option
}

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

There are two parts to a mutable queue:

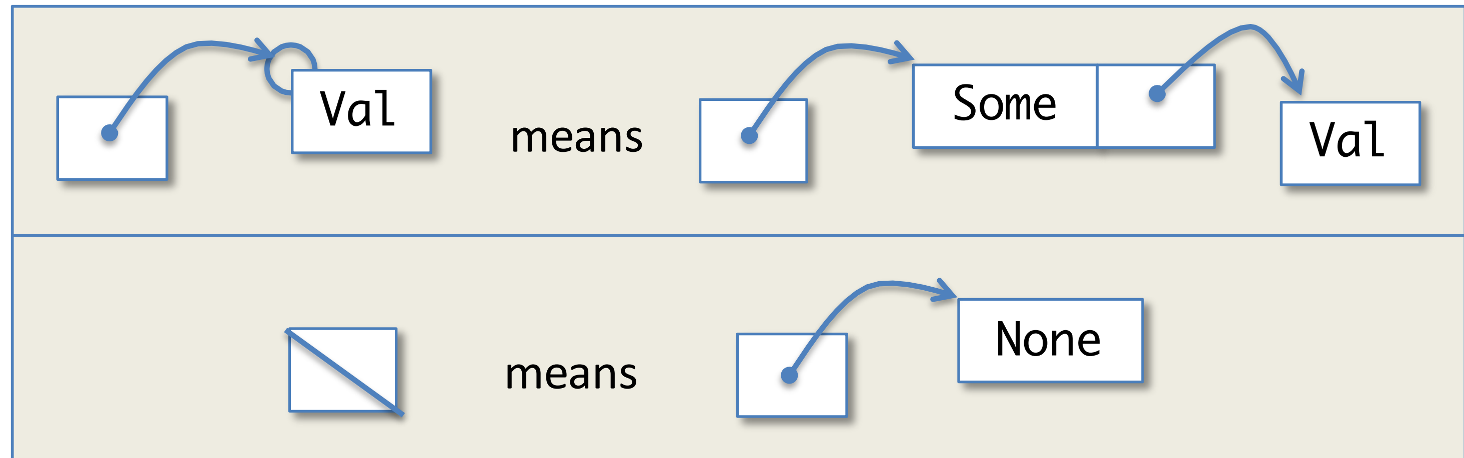
1. the “internal nodes” of the queue, with links from one to the next
2. a record with links to the head and tail nodes

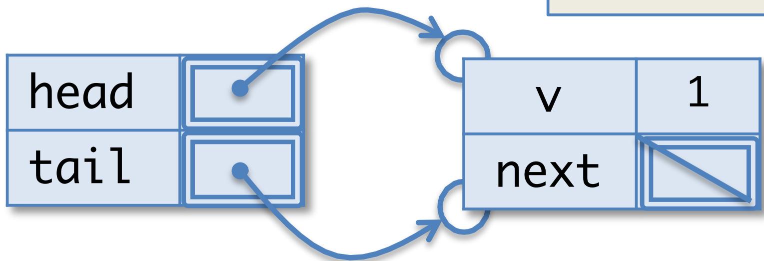
All of the links are *optional* so that the queue can be empty.

Visual Shorthand: Abbreviating Options

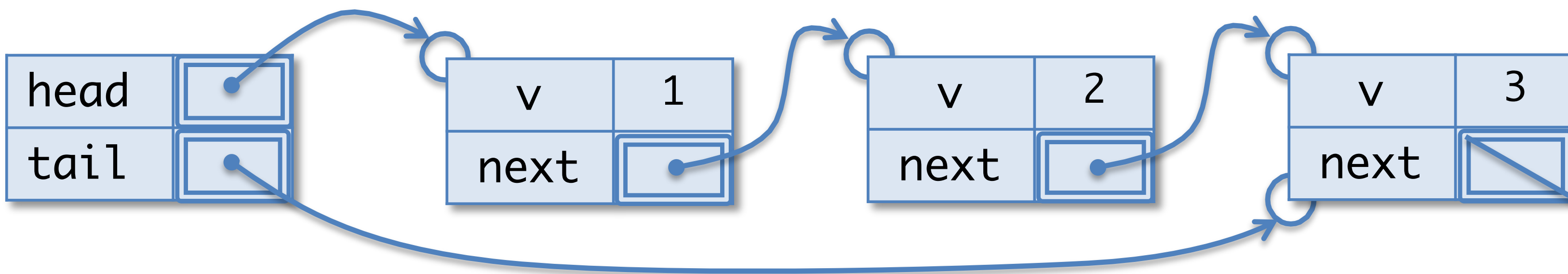


An empty queue



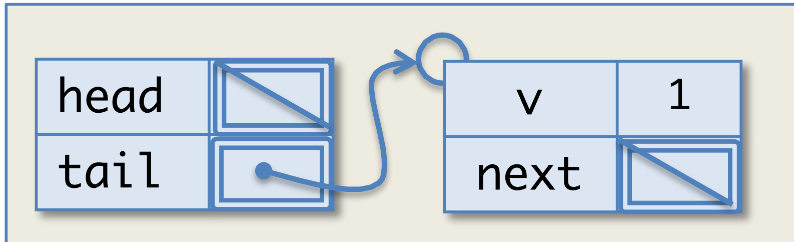


A queue with one element

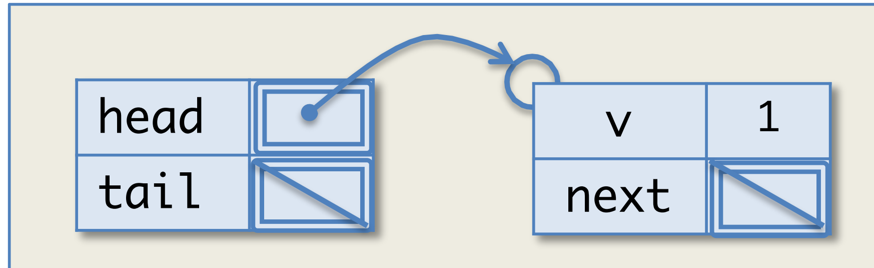


A queue with three elements

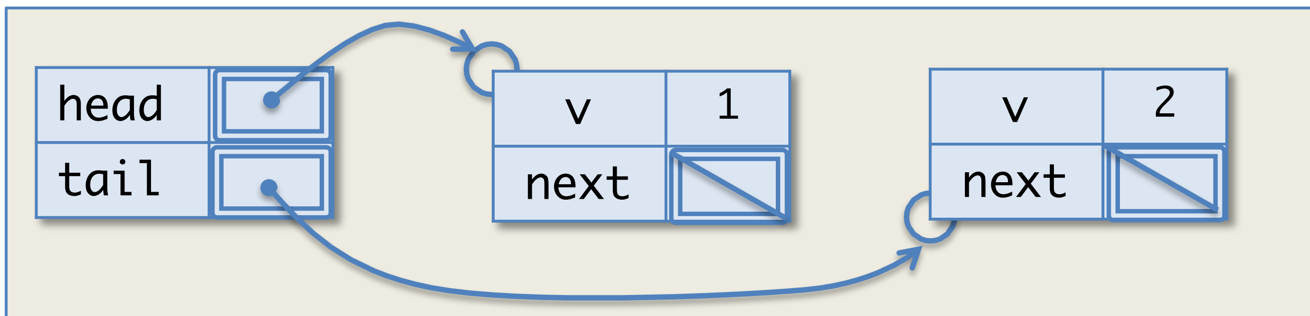
“Bogus” values of type `int queue`



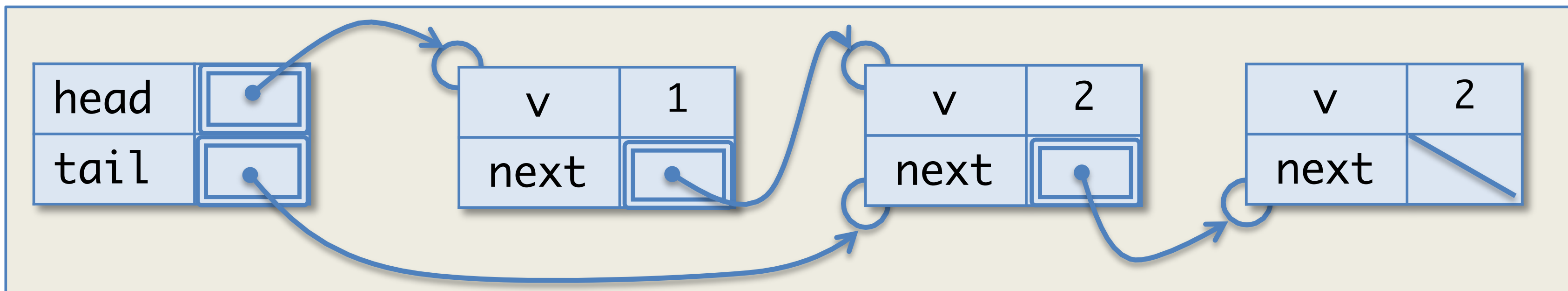
head is None, tail is Some



head is Some, tail is None



tail is not reachable from the head

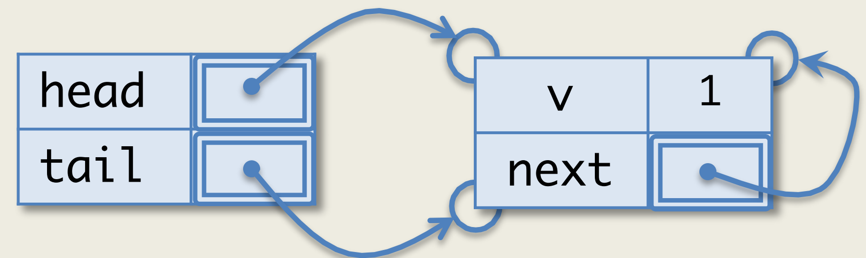


tail doesn’t point to the last element of the queue

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.)

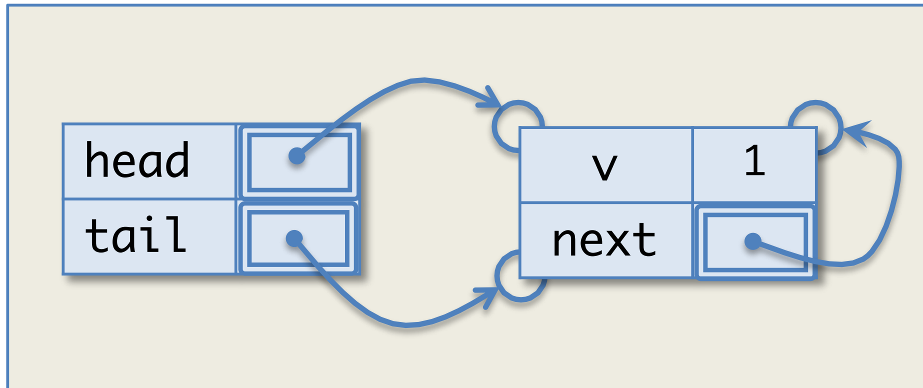
```
type 'a qnode = {
      v: 'a;
      mutable next : 'a qnode option
}

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

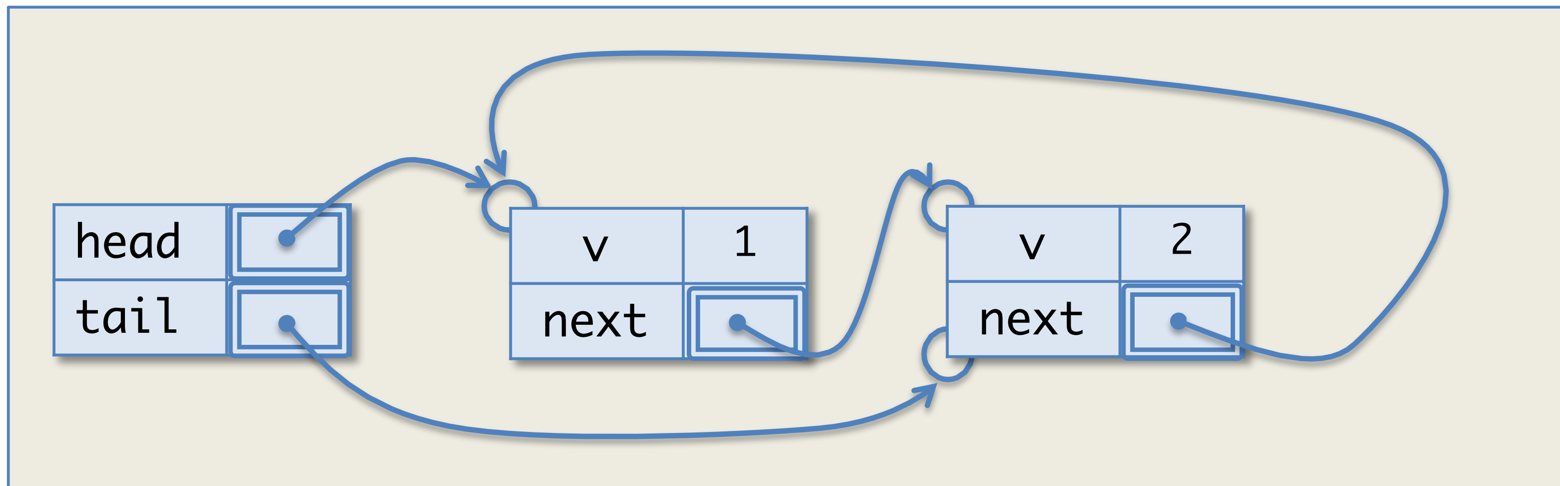
1. yes
2. no
3. not sure



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)

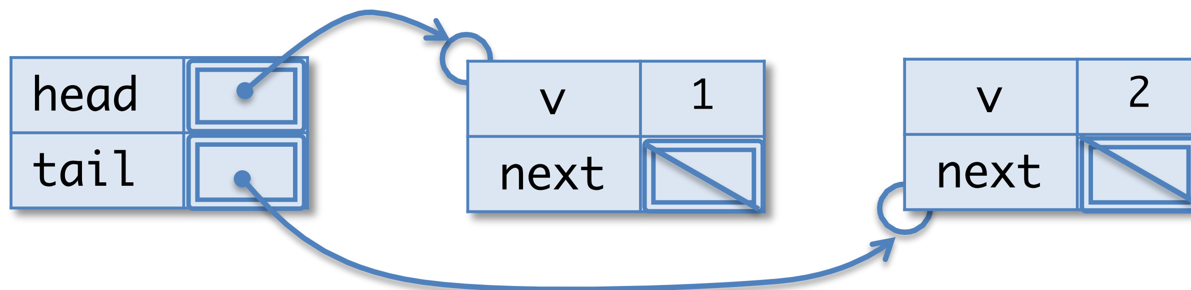
or

(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)

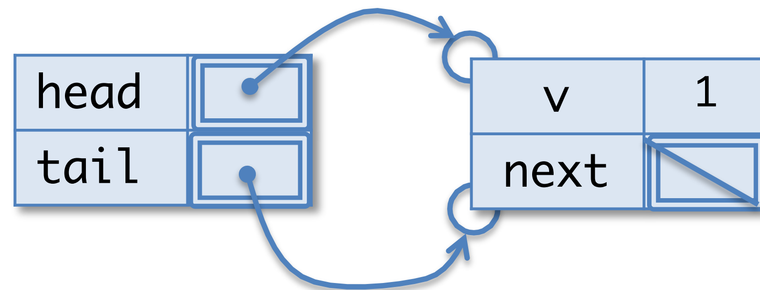
or

(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



Either:

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

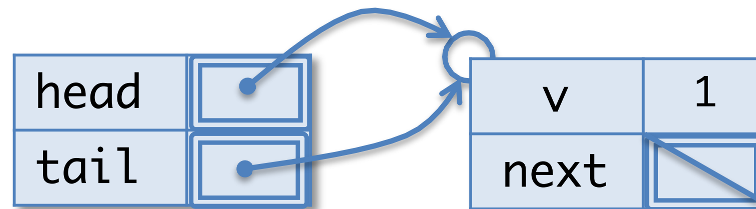
or

(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

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?

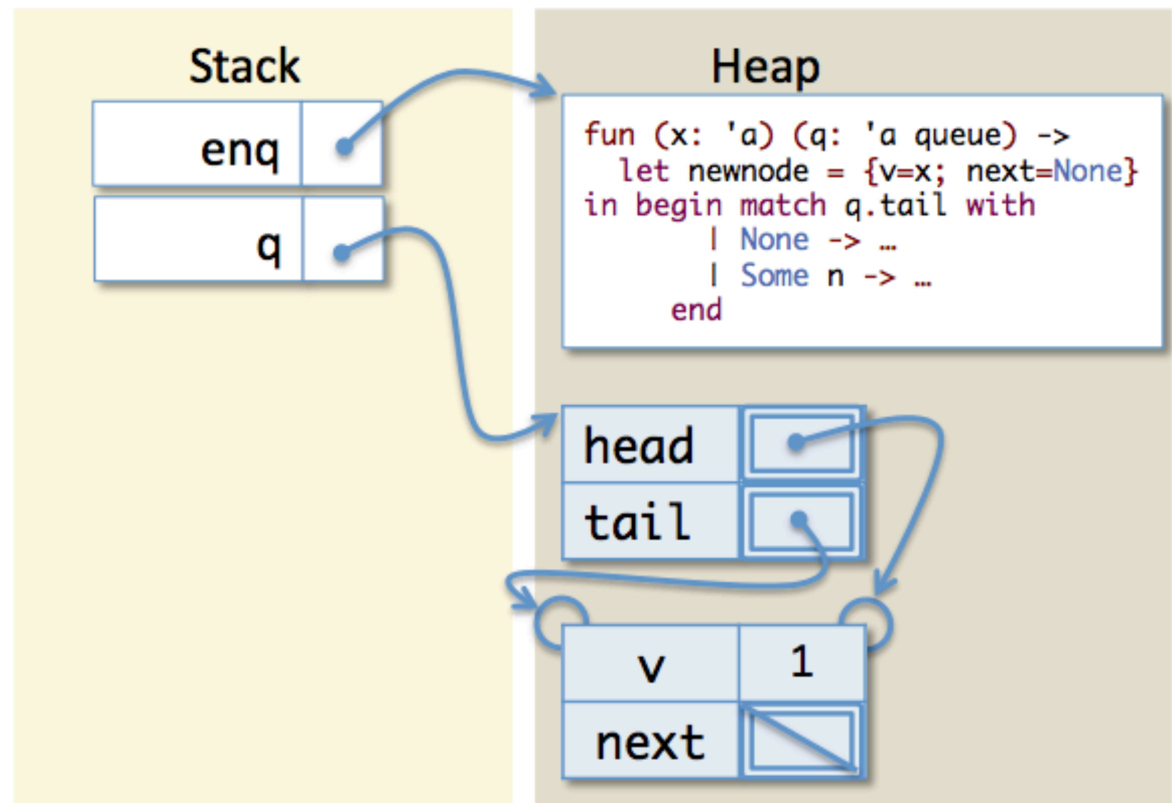
1. got it well under control
2. OK but need to work with it a little more
3. a little puzzled
4. very puzzled
5. very *very* puzzled :-)

Do you want to see an example of enq on the ASM?

1. yes
2. no

Workspace

```
enq 2 q
```

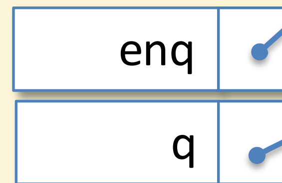


Calling Enq on a non-empty queue

Workspace

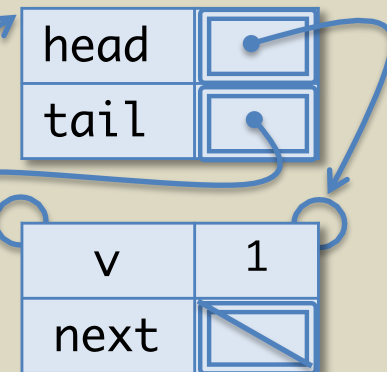
```
enq 2 q
```

Stack



Heap

```
fun (x: 'a) (q: 'a queue) ->
  let newnode = {v=x; next=None}
in begin match q.tail with
      | None -> …
      | Some n -> …
    end
```

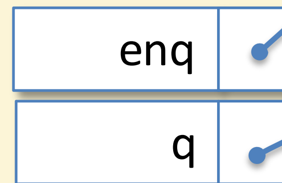


Calling Enq on a non-empty queue

Workspace

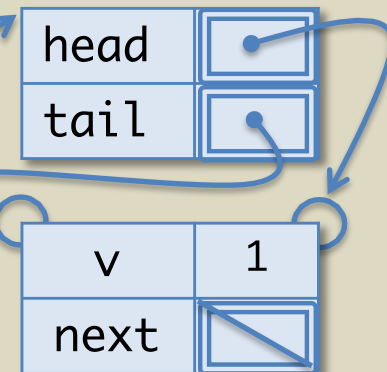
```
enq 2 q
```

Stack

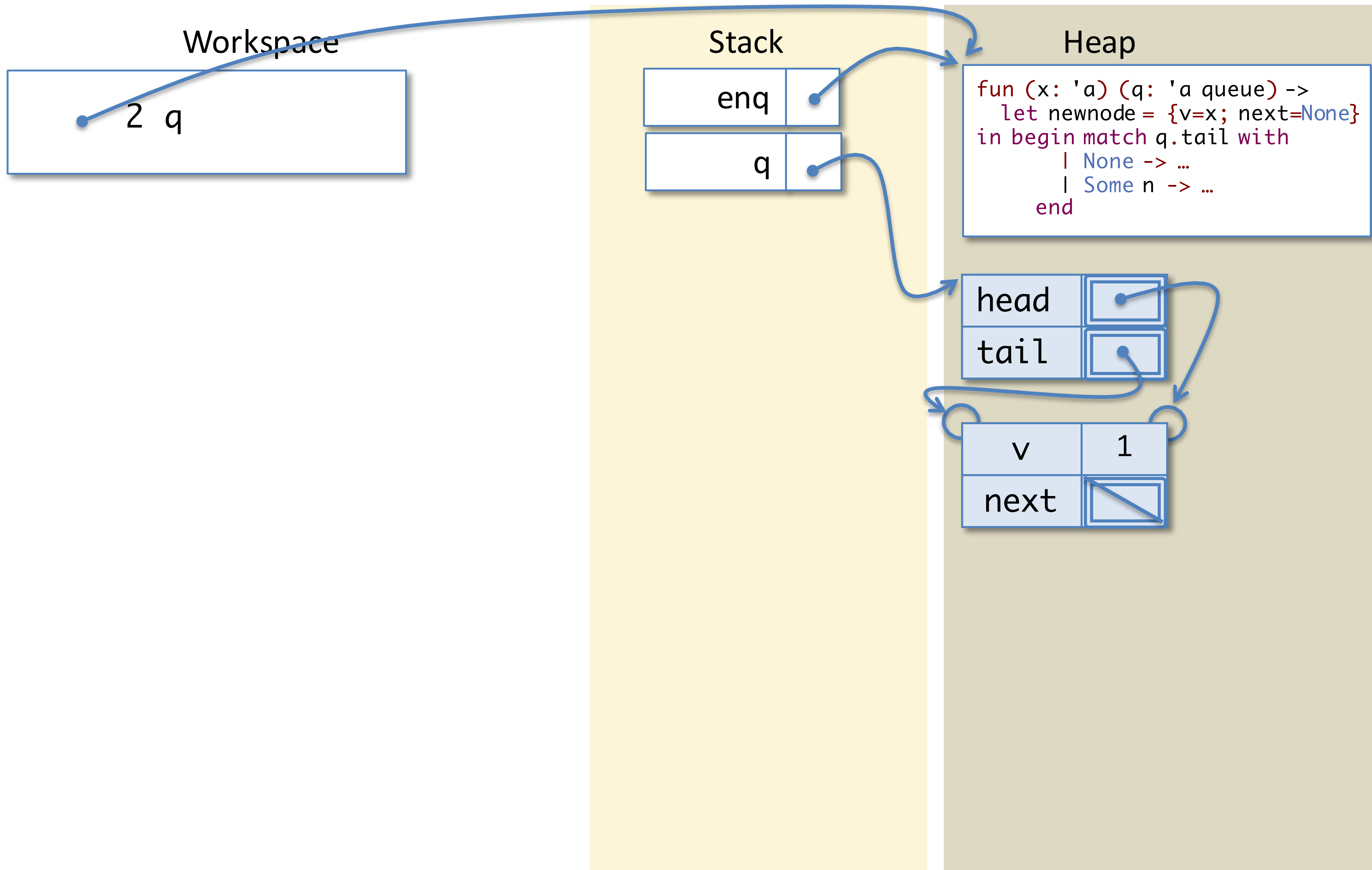


Heap

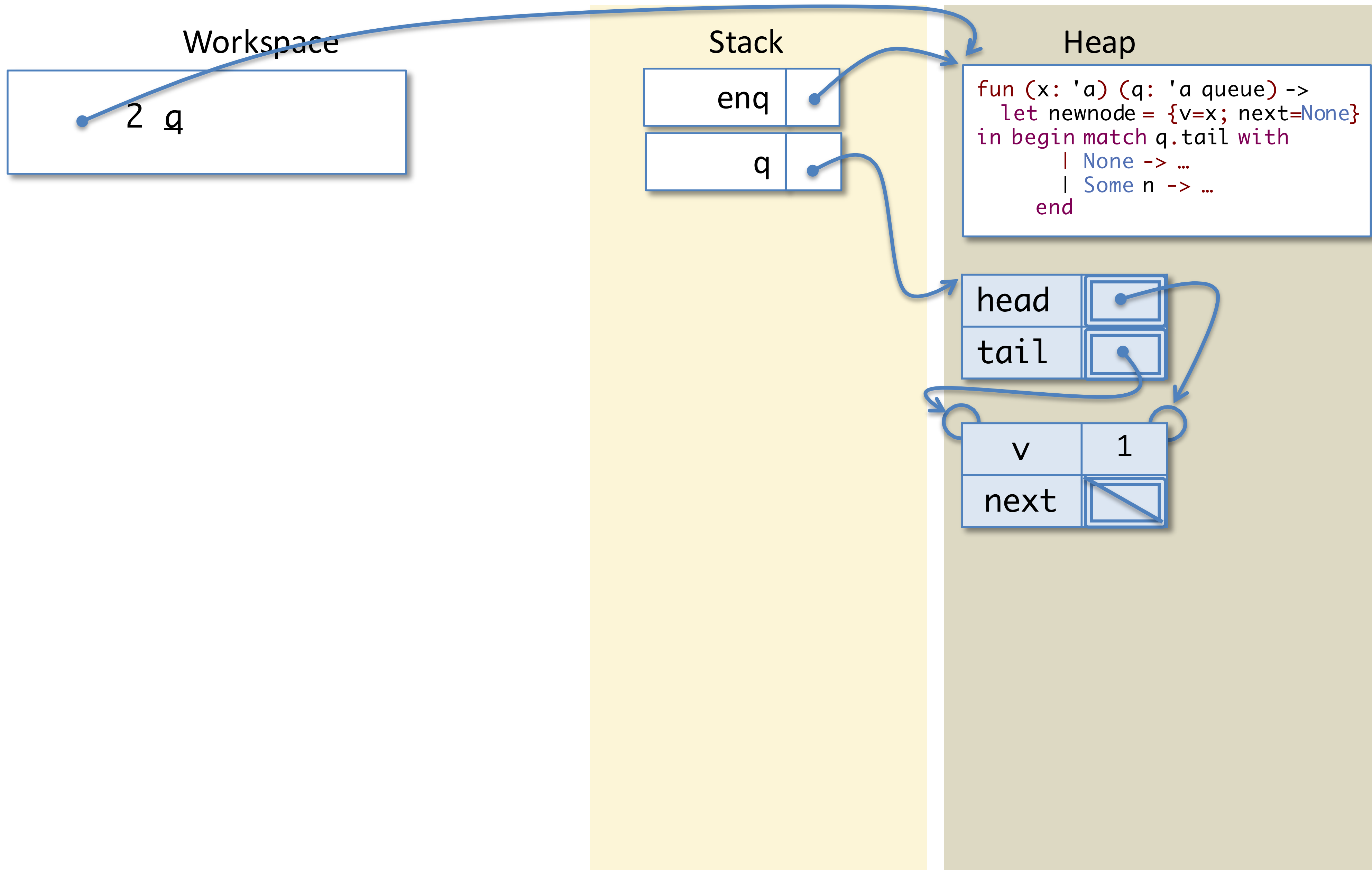
```
fun (x: 'a) (q: 'a queue) ->
  let newnode = {v=x; next=None}
in begin match q.tail with
       | None -> …
       | Some n -> …
     end
```



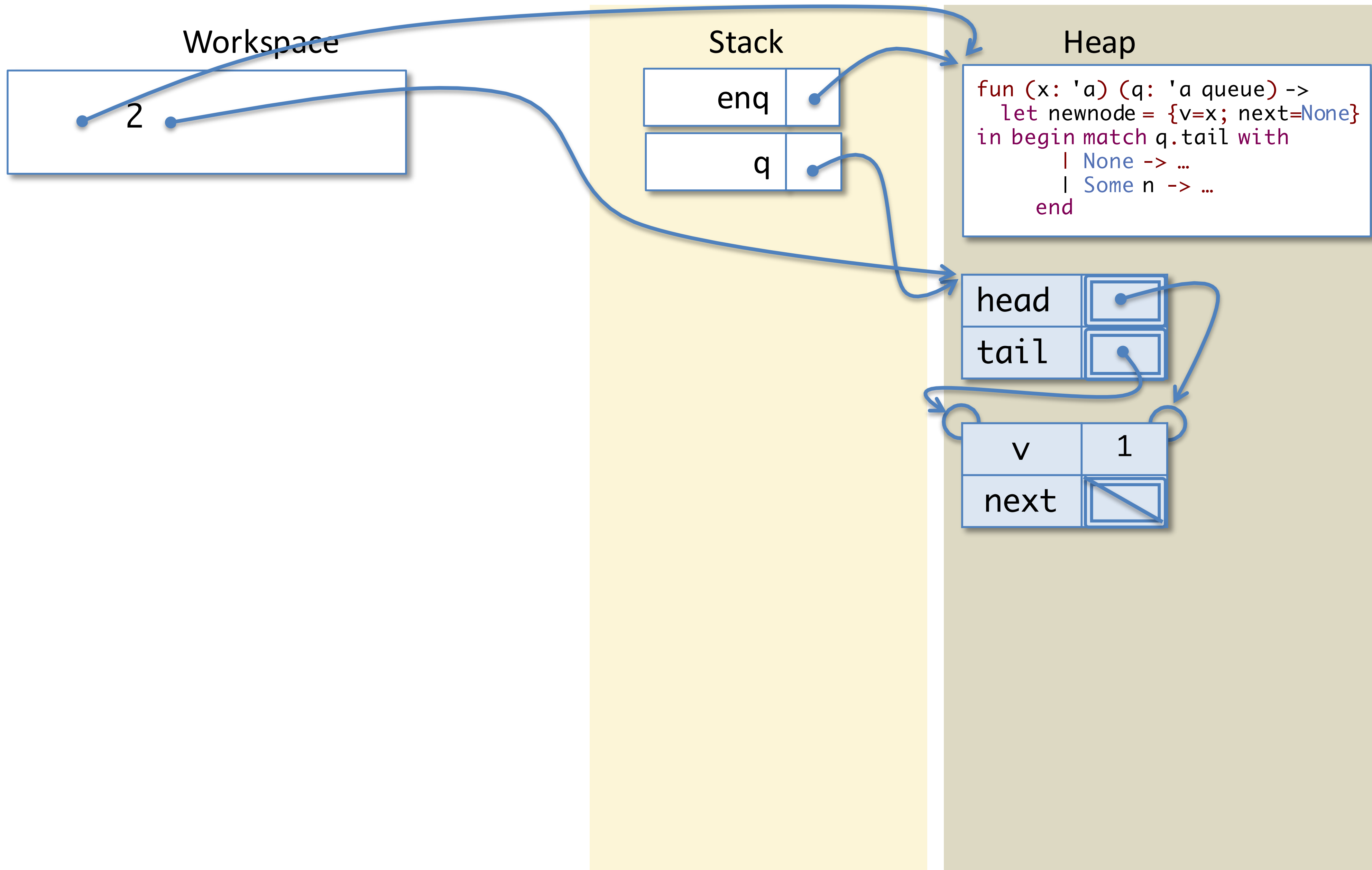
Calling Enq on a non-empty queue



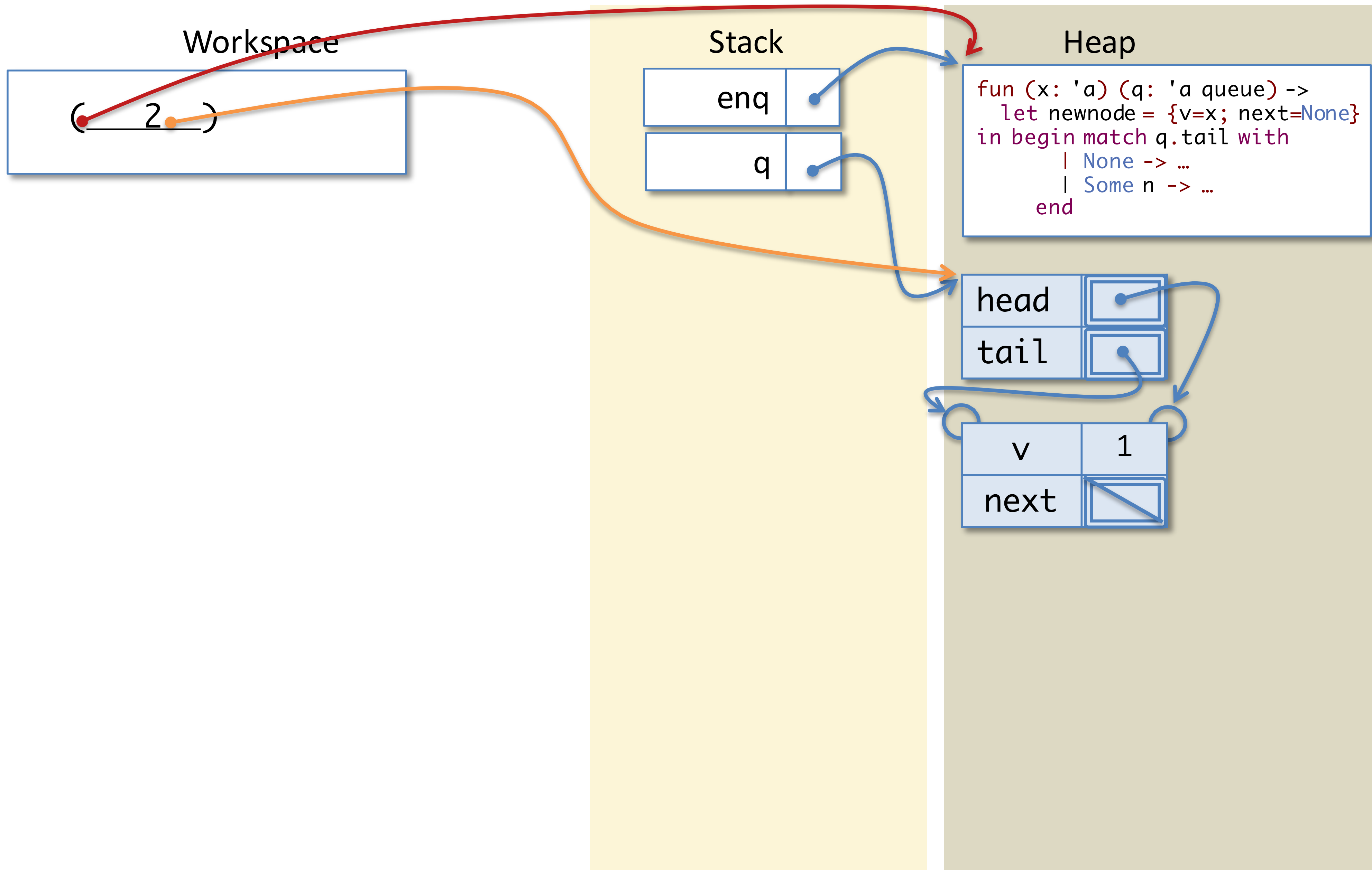
Calling Enq on a non-empty queue



Calling Enq on a non-empty queue



Calling Enq on a non-empty queue

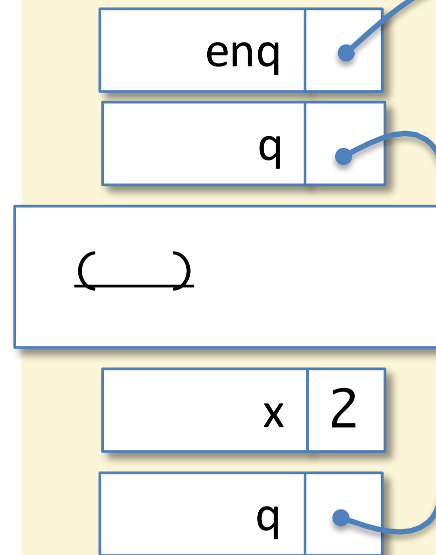


Calling Enq on a non-empty queue

Workspace

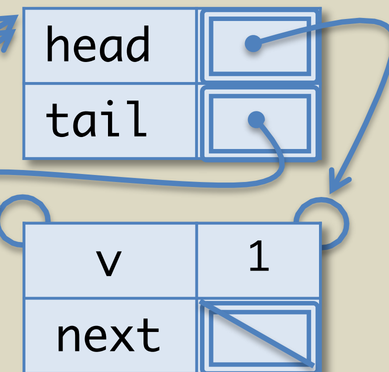
```
let newnode = {v=x; next=None} in
  begin match q.tail with
    | None ->
      q.head <- Some newnode;
      q.tail <- Some newnode
    | Some n ->
      n.next <- Some newnode;
      q.tail <- Some newnode
  end
```

Stack



Heap

```
fun (x: 'a) (q: 'a queue) ->
  let newnode = {v=x; next=None}
in begin match q.tail with
        | None -> …
        | Some n -> …
     end
```

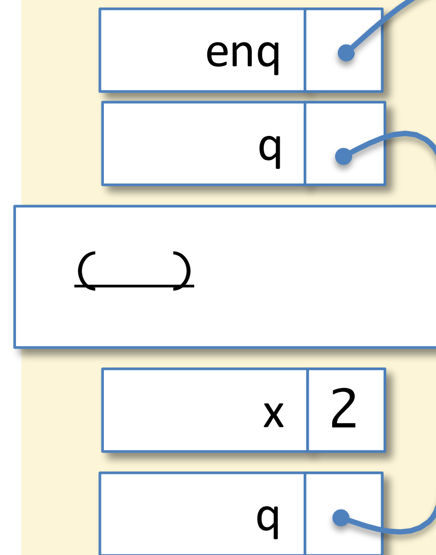


Calling Enq on a non-empty queue

Workspace

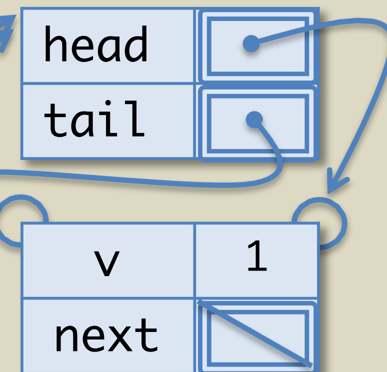
```
let newnode = {v=x; next=None} in
  begin match q.tail with
    | None ->
      q.head <- Some newnode;
      q.tail <- Some newnode
    | Some n ->
      n.next <- Some newnode;
      q.tail <- Some newnode
  end
```

Stack



Heap

```
fun (x: 'a) (q: 'a queue) ->
  let newnode = {v=x; next=None}
in begin match q.tail with
      | None -> …
      | Some n -> …
    end
```

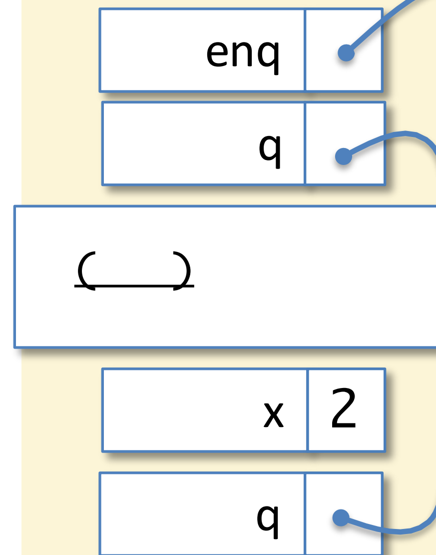


Calling Enq on a non-empty queue

Workspace

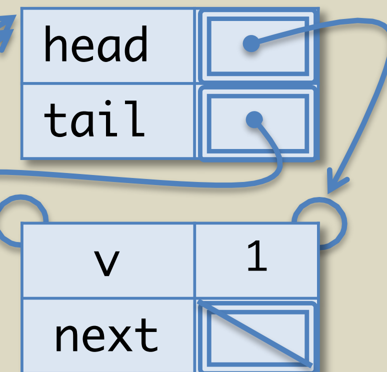
```
let newnode = {v=2; next=None} in
  begin match q.tail with
    | None ->
      q.head <- Some newnode;
      q.tail <- Some newnode
    | Some n ->
      n.next <- Some newnode;
      q.tail <- Some newnode
  end
```

Stack



Heap

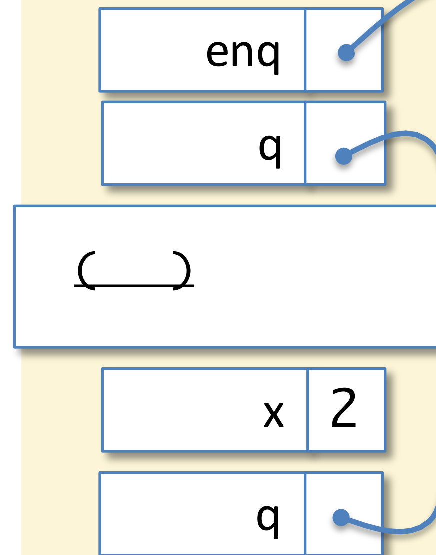
```
fun (x: 'a) (q: 'a queue) ->
  let newnode = {v=x; next=None}
in begin match q.tail with
      | None -> …
      | Some n -> …
    end
```



Calling Enq on a non-empty queue

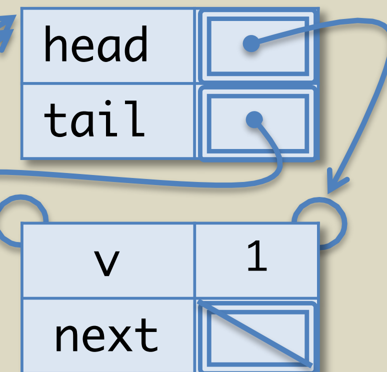
Workspace

```
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
```



Heap

```
fun (x: 'a) (q: 'a queue) ->
  let newnode = {v=x; next=None}
in begin match q.tail with
     | None -> …
     | Some n -> …
   end
```

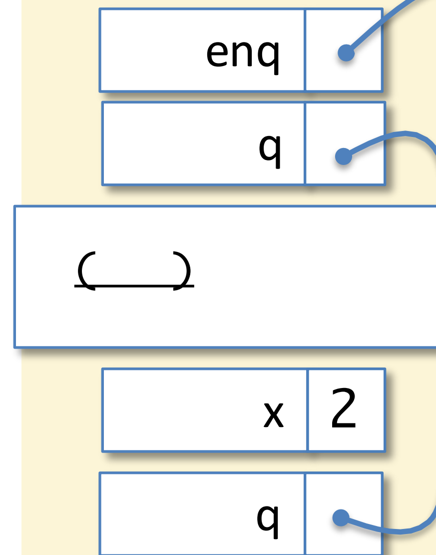


Calling Enq on a non-empty queue

Workspace

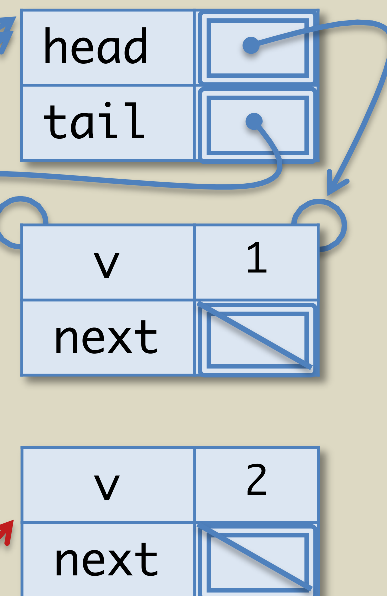
```
let newnode =   in
  begin match q.tail with
    | None ->
      q.head <- Some newnode;
      q.tail <- Some newnode
    | Some n ->
      n.next <- Some newnode;
      q.tail <- Some newnode
  end
```

Stack



Heap

```
fun (x: 'a) (q: 'a queue) ->
  let newnode = {v=x; next=None}
in begin match q.tail with
      | None -> …
      | Some n -> …
    end
```



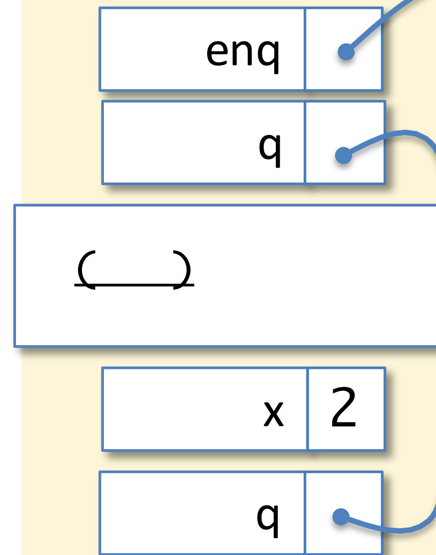
Note: there is no "Some bubble": this is a qnode, not a qnode option.

Calling Enq on a non-empty queue

Workspace

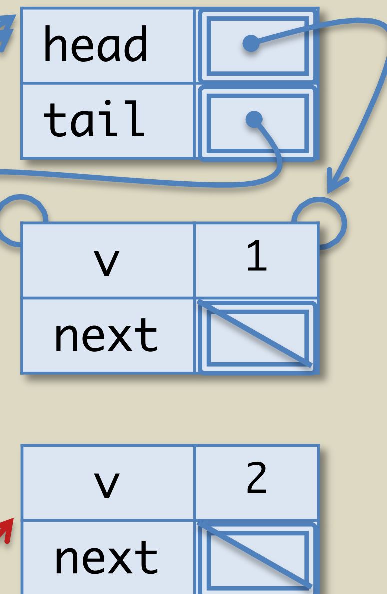
```
let newnode = ● in
  begin match q.tail with
    | None ->
      q.head <- Some newnode;
      q.tail <- Some newnode
    | Some n ->
      n.next <- Some newnode;
      q.tail <- Some newnode
  end
```

Stack



Heap

```
fun (x: 'a) (q: 'a queue) ->
  let newnode = {v=x; next=None}
in begin match q.tail with
      | None -> …
      | Some n -> …
    end
```

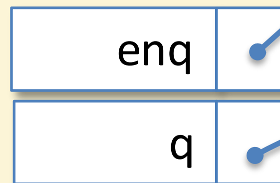


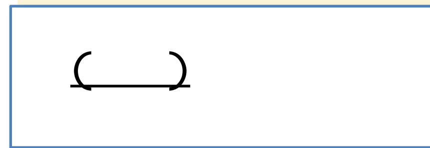
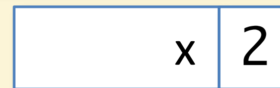
Calling Enq on a non-empty queue

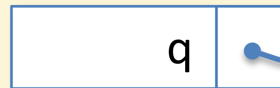
Workspace

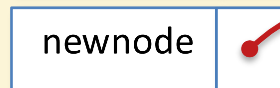
```
begin match q.tail with
  | None ->
     q.head <- Some newnode;
     q.tail <- Some newnode
  | Some n ->
     n.next <- Some newnode;
     q.tail <- Some newnode
end
```

Stack



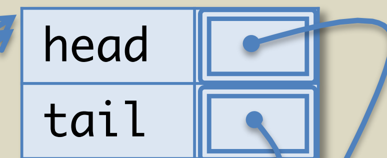



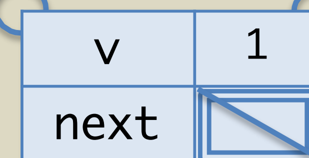


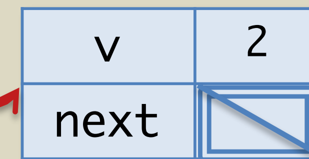


Heap

```
fun (x: 'a) (q: 'a queue) ->
  let newnode = {v=x; next=None}
in begin match q.tail with
       | None -> …
       | Some n -> …
    end
```





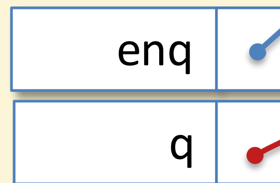


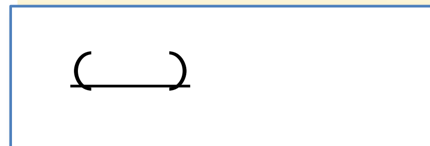
Calling Enq on a non-empty queue

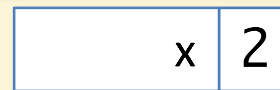
Workspace

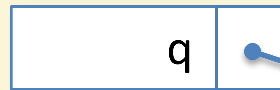
```
begin match q.tail with
  | None ->
    q.head <- Some newnode;
    q.tail <- Some newnode
  | Some n ->
    n.next <- Some newnode;
    q.tail <- Some newnode
end
```

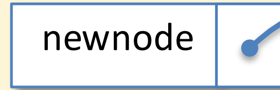
Stack





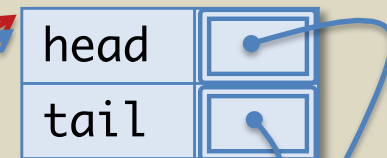


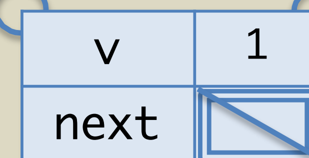


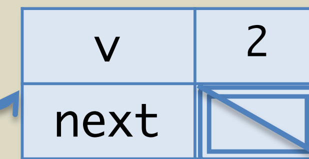


Heap

```
fun (x: 'a) (q: 'a queue) ->
  let newnode = {v=x; next=None}
in begin match q.tail with
      | None -> …
      | Some n -> …
    end
```



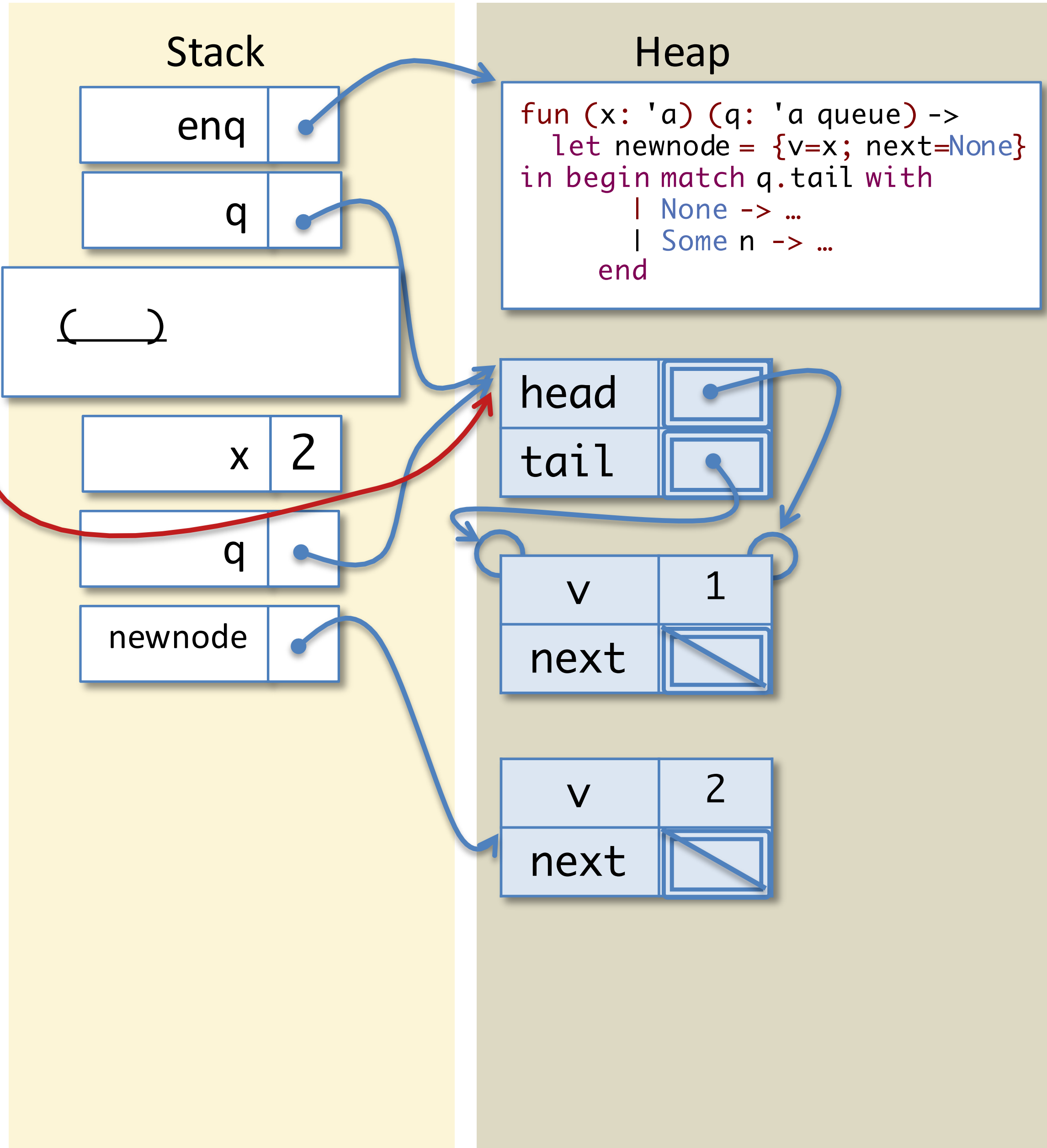




Calling Enq on a non-empty queue

Workspace

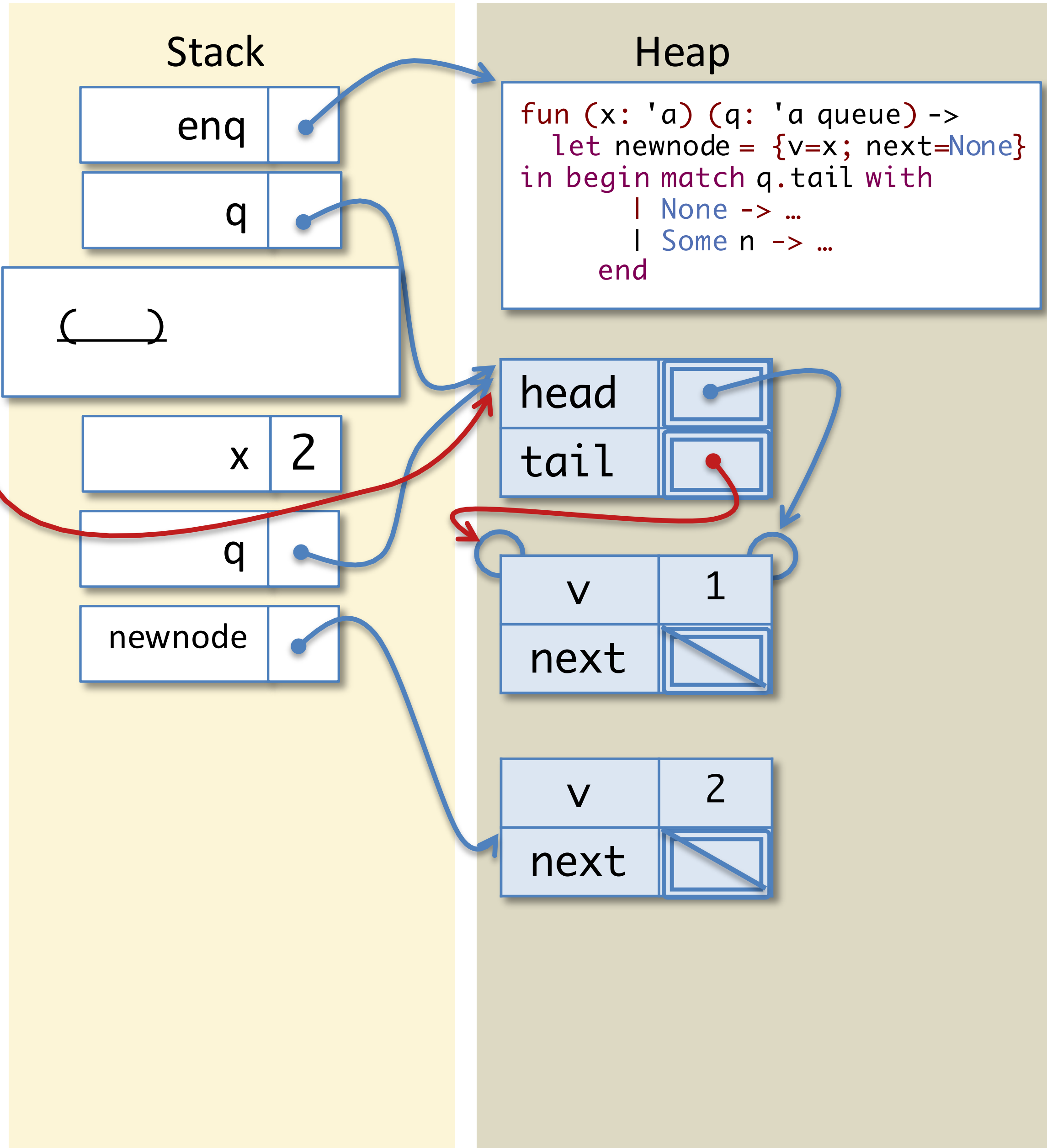
```
begin match •.tail with
  | None ->
    q.head <- Some newnode;
    q.tail <- Some newnode
  | Some n ->
    n.next <- Some newnode;
    q.tail <- Some newnode
end
```



Calling Enq on a non-empty queue

Workspace

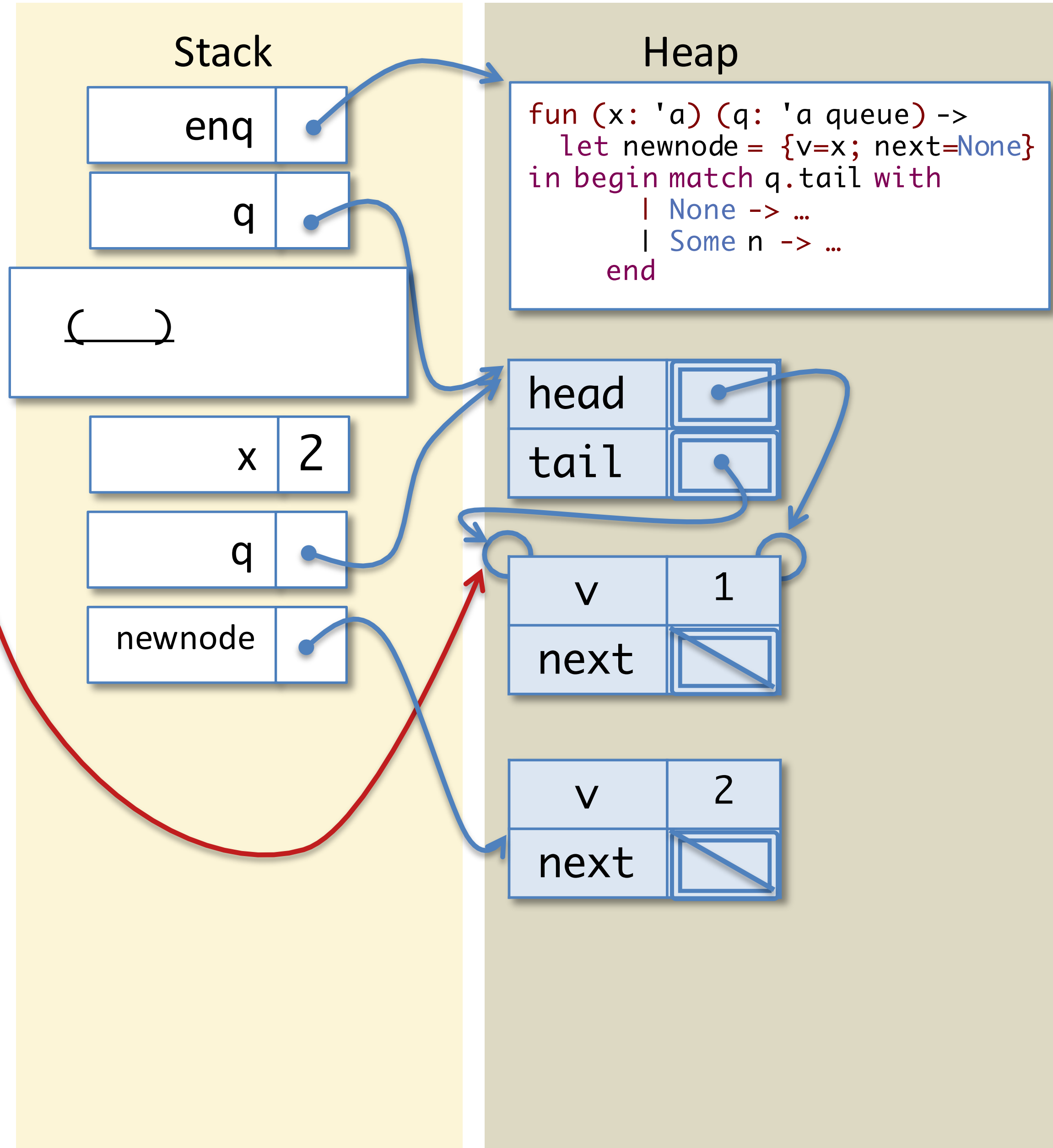
```
begin match ___.tail with
  | None ->
    q.head <- Some newnode;
    q.tail <- Some newnode
  | Some n ->
    n.next <- Some newnode;
    q.tail <- Some newnode
end
```



Calling Enq on a non-empty queue

Workspace

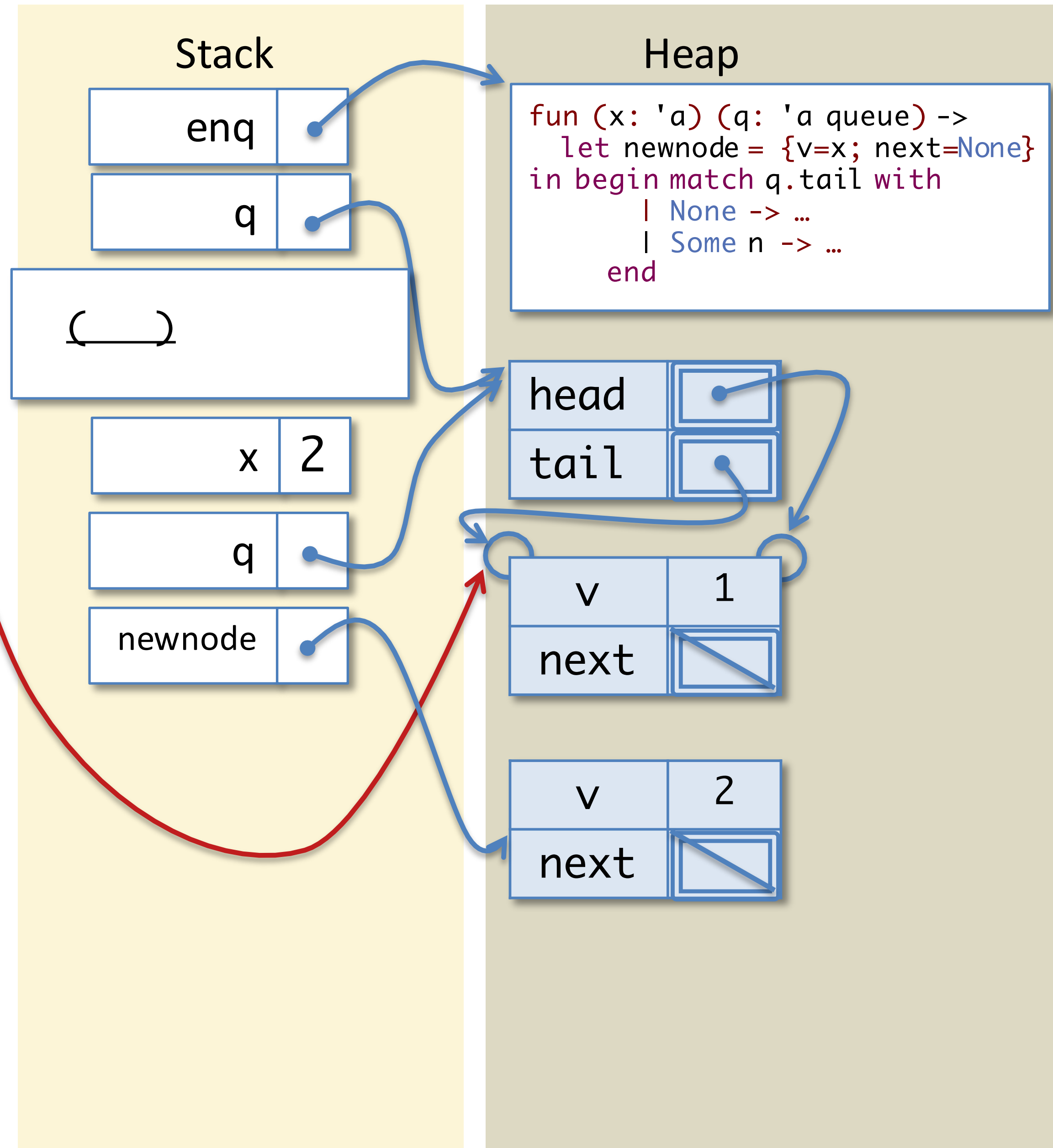
```
begin match • with
  | None ->
    q.head <- Some newnode;
    q.tail <- Some newnode
  | Some n ->
    n.next <- Some newnode;
    q.tail <- Some newnode
end
```



Calling Enq on a non-empty queue

Workspace

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

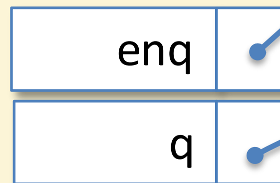


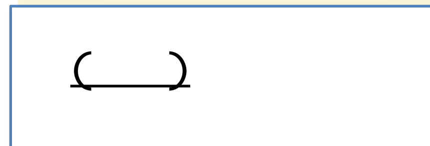
Calling Enq on a non-empty queue

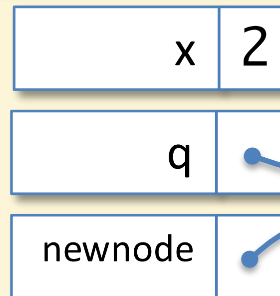
Workspace

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

Stack

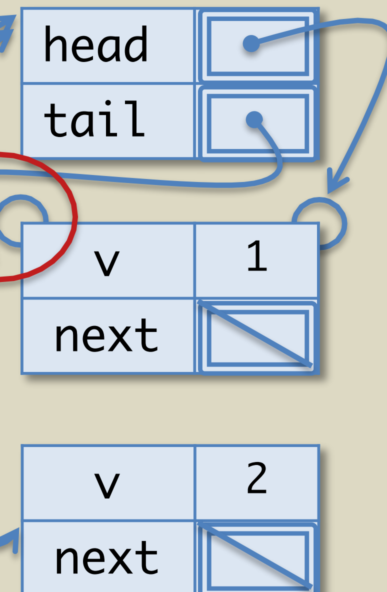




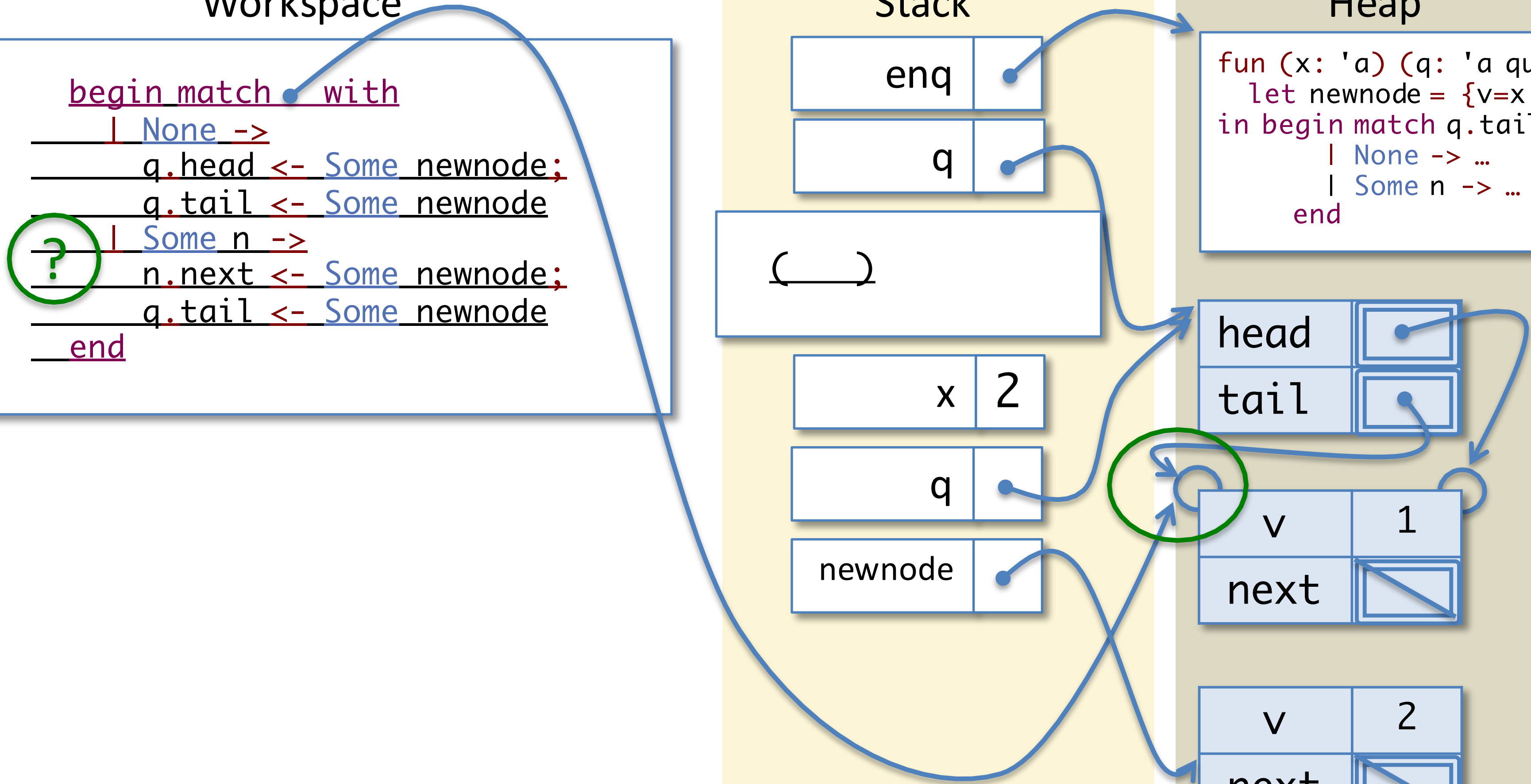


Heap

```
fun (x: 'a) (q: 'a queue) ->
  let newnode = {v=x; next=None}
in begin match q.tail with
      | None -> …
      | Some n -> …
    end
```



Calling Enq on a non-empty queue



Calling Enq on a non-empty queue

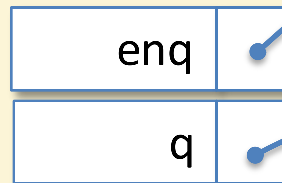
Workspace

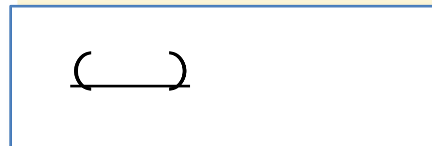
```
n.next <- Some newnode;
q.tail <- Some newnode
```

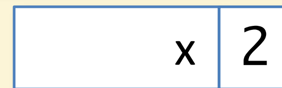
Stack

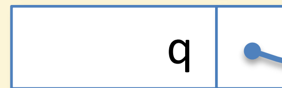
Heap

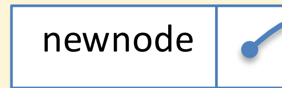
```
fun (x: 'a) (q: 'a queue) ->
  let newnode = {v=x; next=None}
in begin match q.tail with
       | None -> …
       | Some n -> …
     end
```

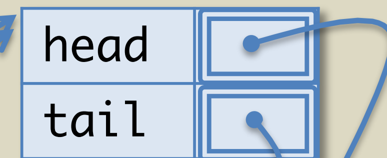


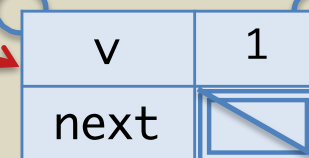


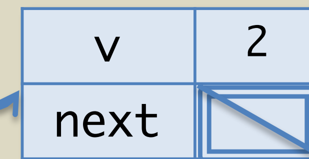












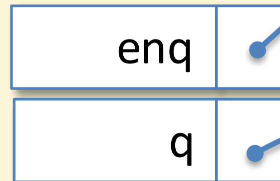
Note: n points to a qnode, not a qnode option.

Calling Enq on a non-empty queue

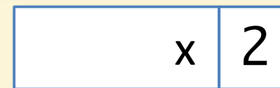
Workspace

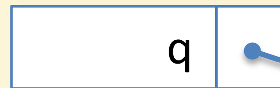
```
n.next <- Some newnode;
q.tail <- Some newnode
```

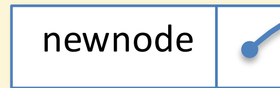
Stack

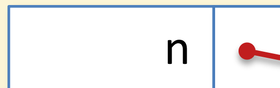


( )



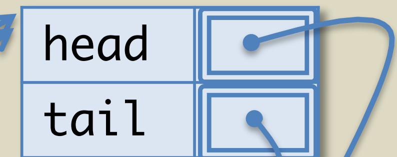


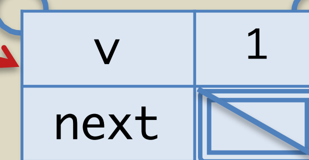


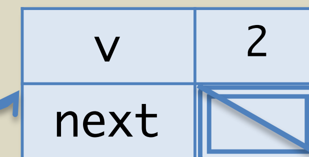


Heap

```
fun (x: 'a) (q: 'a queue) ->
  let newnode = {v=x; next=None}
in begin match q.tail with
       | None -> …
       | Some n -> …
     end
```





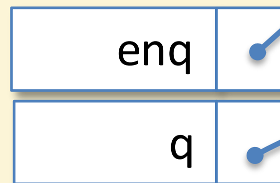


Calling Enq on a non-empty queue

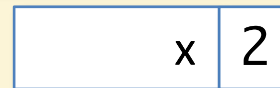
Workspace

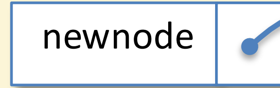
```
 .next <- Some newnode;
q.tail <- Some newnode
```

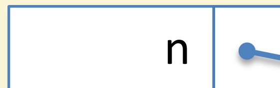
Stack



(___)

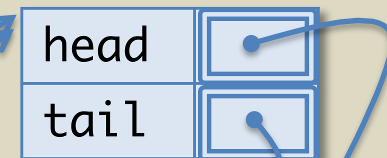


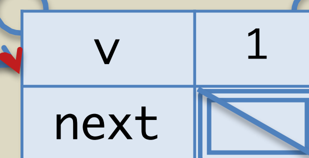


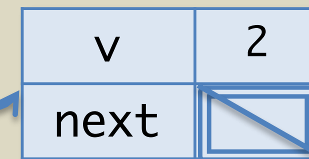


Heap

```
fun (x: 'a) (q: 'a queue) ->
  let newnode = {v=x; next=None}
in begin match q.tail with
       | None -> …
       | Some n -> …
     end
```





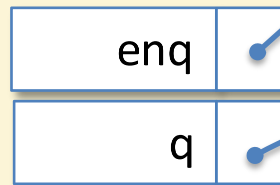


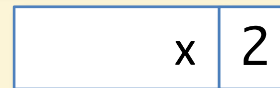
Calling Enq on a non-empty queue

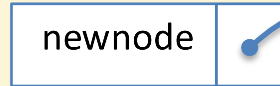
Workspace

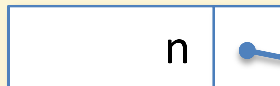
```
 .next <- Some newnode;
q.tail <- Some newnode
```

Stack



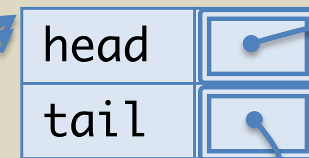


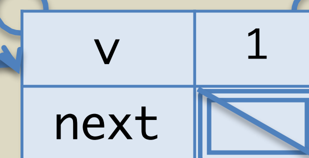


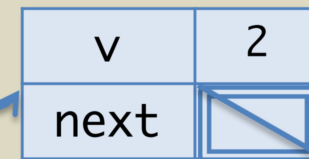


Heap

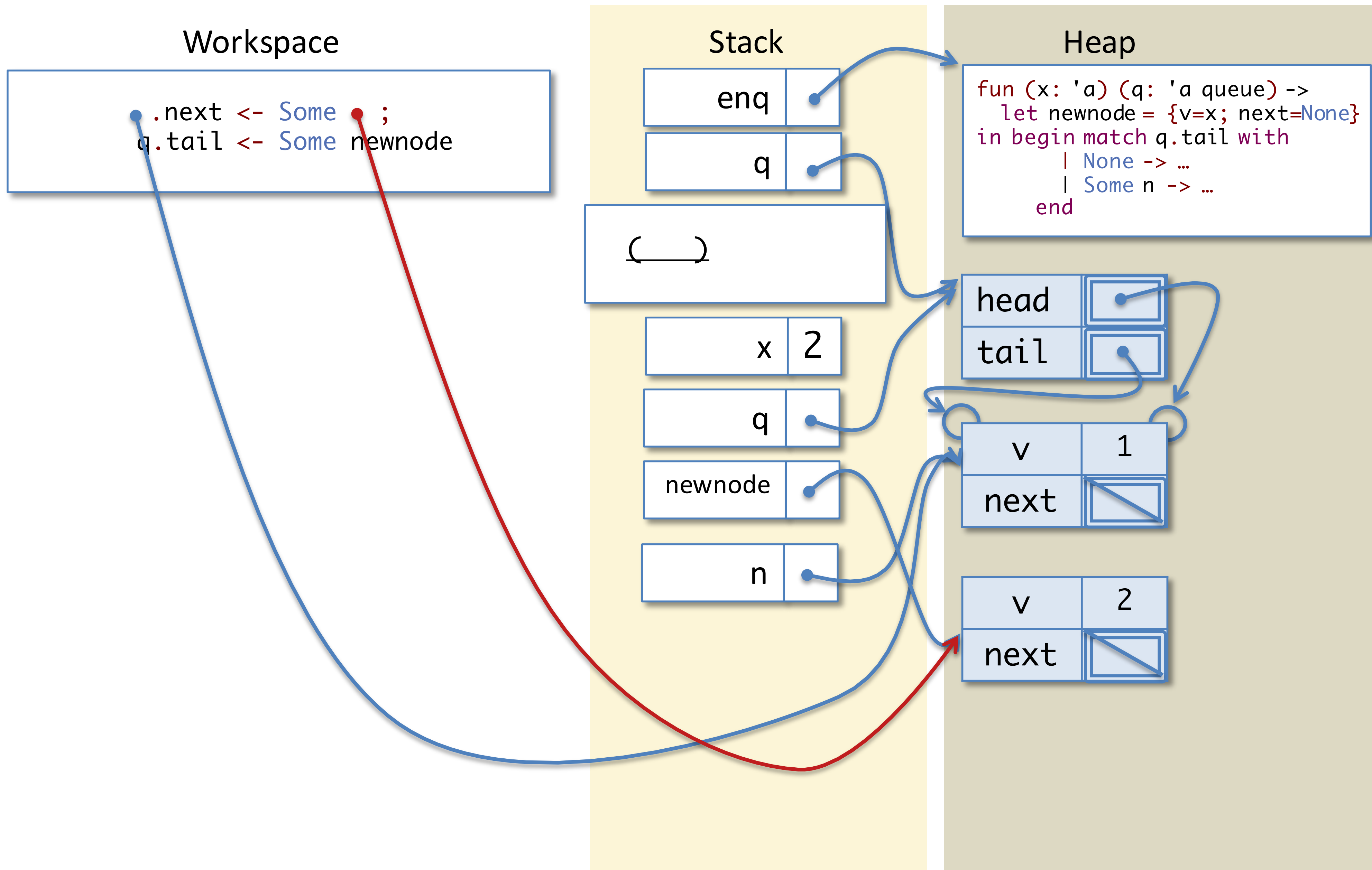
```
fun (x: 'a) (q: 'a queue) ->
  let newnode = {v=x; next=None}
in begin match q.tail with
        | None -> …
        | Some n -> …
     end
```







Calling Enq on a non-empty queue

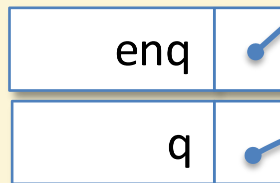


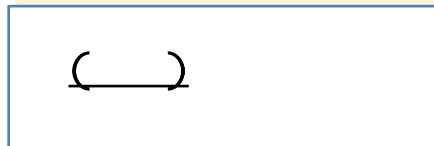
Calling Enq on a non-empty queue

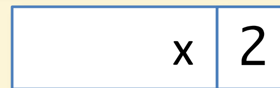
Workspace

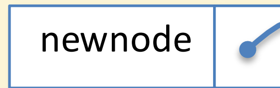
```
 .next <- Some ___;
q.tail <- Some newnode
```

Stack



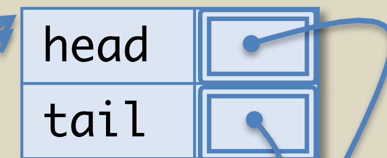


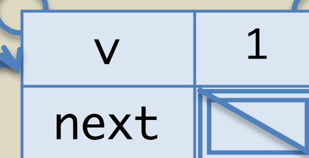


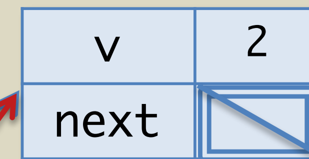


Heap

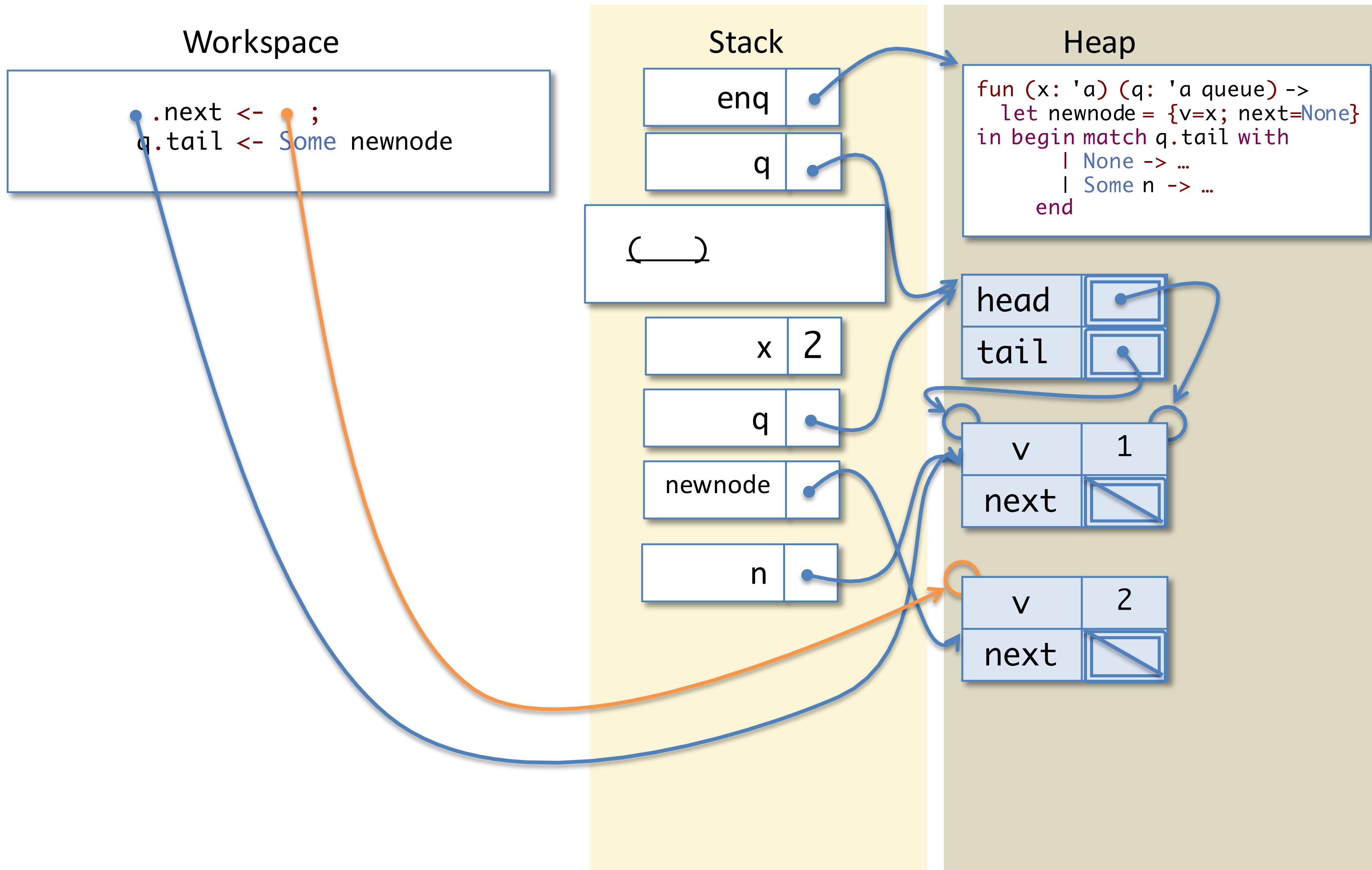
```
fun (x: 'a) (q: 'a queue) ->
  let newnode = {v=x; next=None}
in begin match q.tail with
       | None -> …
       | Some n -> …
     end
```







Calling Enq on a non-empty queue

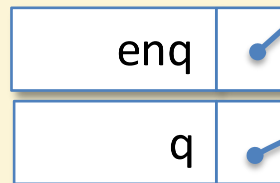


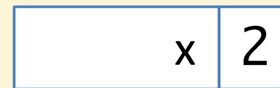
Calling Enq on a non-empty queue

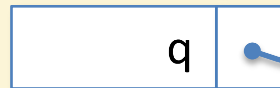
Workspace

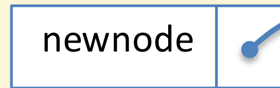
```
___.next <- ___;
q.tail <- Some newnode
```

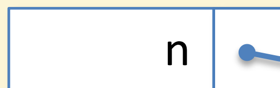
Stack





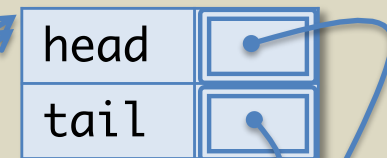


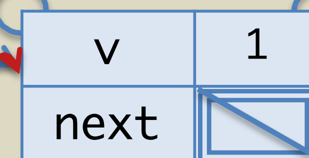


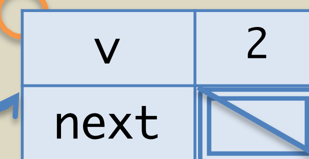


Heap

```
fun (x: 'a) (q: 'a queue) ->
  let newnode = {v=x; next=None}
in begin match q.tail with
       | None -> …
       | Some n -> …
     end
```





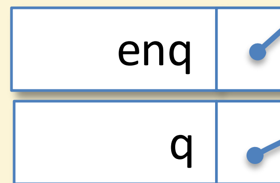


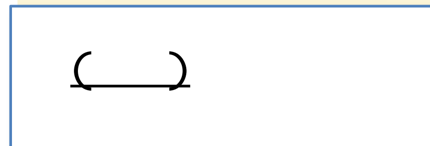
Calling Enq on a non-empty queue

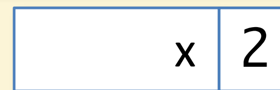
Workspace

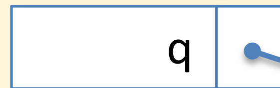
```
();
q.tail <- Some newnode
```

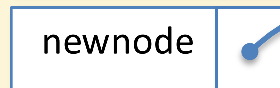
Stack

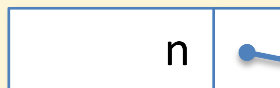






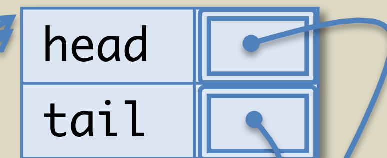


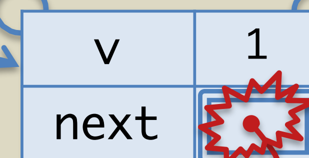


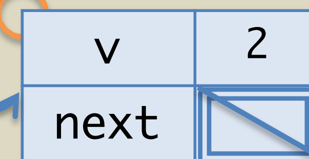


Heap

```
fun (x: 'a) (q: 'a queue) ->
  let newnode = {v=x; next=None}
in begin match q.tail with
       | None -> …
       | Some n -> …
     end
```





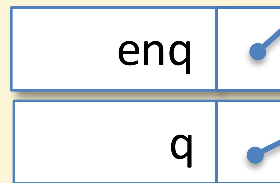


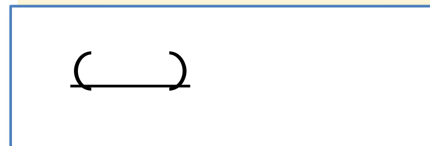
Calling Enq on a non-empty queue

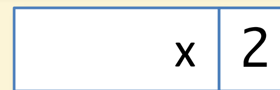
Workspace

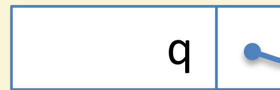
```
();
q.tail <- Some newnode
```

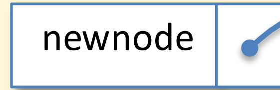
Stack

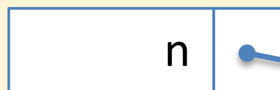






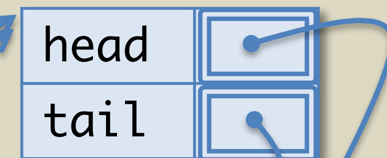


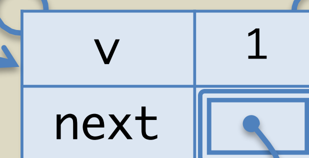


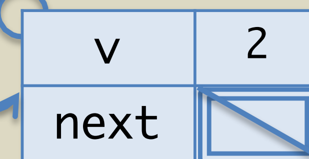


Heap

```
fun (x: 'a) (q: 'a queue) ->
  let newnode = {v=x; next=None}
in begin match q.tail with
        | None -> …
        | Some n -> …
    end
```





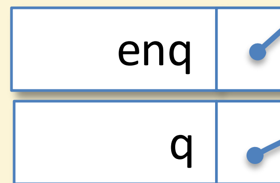


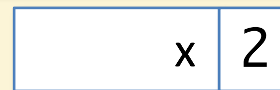
Calling Enq on a non-empty queue

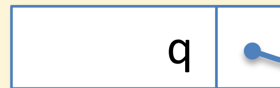
Workspace

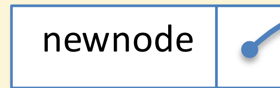
```
q.tail <- Some newnode
```

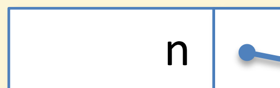
Stack





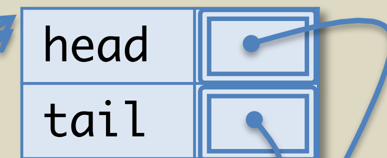


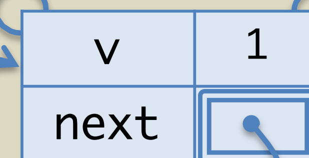


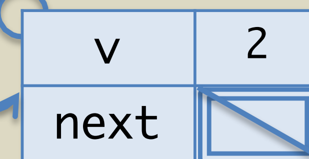


Heap

```
fun (x: 'a) (q: 'a queue) ->
  let newnode = {v=x; next=None}
in begin match q.tail with
        | None -> …
        | Some n -> …
     end
```





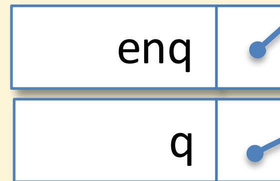


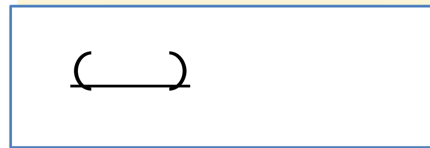
Calling Enq on a non-empty queue

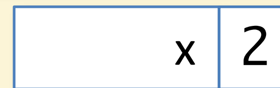
Workspace

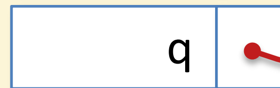
```
q.tail <- Some newnode
```

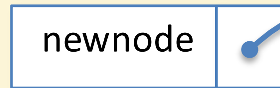
Stack

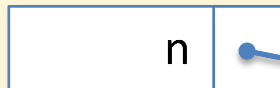












Heap

```
fun (x: 'a) (q: 'a queue) ->
  let newnode = {v=x; next=None}
in begin match q.tail with
       | None -> …
       | Some n -> …
    end
```