Programming Languages and Techniques (CIS120)

Lecture 15

Feb 17, 2012

Iteration
Announcements

• Homework 5 is available on the web
  – due February 23rd at 11:59:59pm
  – make sure your version doesn’t include “counter”

• Midterm 1 grades can be checked via the “check your grades link” from the HW web pages:
  – Class Average: ~82/100, Median: ~86/100
  – Rough grade distribution:
    >= 90  A
    >= 75  B
    >= 60  C
  – Solutions online now
  – Check your exams with Laura Fox starting Monday (Levine 308)
  – Regrades in writing (deadline: March 16th)
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
Datatypes for Mutable Queues

```ocaml
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
Example Queues in the Heap

An empty queue

A queue with one element

A queue with two elements
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 “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.
The code for \texttt{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.

\begin{verbatim}
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
\end{verbatim}
Queue Length

• Suppose we want to extend the interface with a length function:

```ocaml
module type QUEUE =
  sig
      (* type of the data structure *)
      type 'a queue

      ...

      (* Get the length of the queue *)
      val length : 'a queue -> int
  end
```

• How can we implement it?
length (recursively)

(* Calculate the length of the list using recursion *)
let length (q:'a queue) : int =
  let rec loop (no: 'a qnode option) : int =
    begin
      match no with
      | None -> 0
      | Some n -> 1 + (loop n.next)
    end
  in
  loop q.head

• This code for length uses a helper function, loop:
  – the correctness depends crucially on the queue invariant
  – what happens if we pass in a bogus q that is cyclic?

• The height of the ASM stack is proportional to the length of the queue
  – That seems inefficient... why should it take so much space?
Evaluating length

Workspace

length q

Stack

| length | q |

Heap

fun (q:'a queue) ->
let rec loop (no:…) : int = …
in
loop q.head

head

tail

v 1

next

v 2

next
Evaluating length

Workspace

```
length q
```

Stack

```
length
q
```

Heap

```
fun (q:'a queue) ->
let rec loop (no:…) : int =
  ...
in
loop q.head
```

```
head
tail
```

```
v 1
next
```

```
v 2
next
```
Evaluating length

Workspace

Stack

Heap

fun (q:'a queue) ->
  let rec loop (no:…) : int = …
in
  loop q.head

head

tail

v 1

next

v 2

next
Evaluating length

Workspace

Stack

Heap

fun (q:'a queue) ->
let rec loop (no:...) : int =
  ... in
  loop q.head

head
tail

v 1
next

v 2
next
 Evaluating length

Workspace

let rec loop (no: ...) : int =
  begin
    match no with
    | None -> 0
    | Some n -> 1 + (loop n.next)
  end
in
loop q.head

Stack

Heap

fun (q:a queue) ->
let rec loop (no:...) : int =
  ...
in
loop q.head

length
q

(v 1)
next
head
tail

(v 2)
next

Evaluating length

Workspace

```
let loop = fun (no: ...) ->
  begin match no with
    | None -> 0
    | Some n -> 1 + (loop n.next)
  end
in
loop q.head
```

Stack

```
<table>
<thead>
<tr>
<th>length</th>
</tr>
</thead>
<tbody>
<tr>
<td>q</td>
</tr>
<tr>
<td>(</td>
</tr>
</tbody>
</table>

Heap

```
fun (q:'a queue) ->
  let rec loop (no:...) : int =
    ...  
  in
  loop q.head
```

```
<table>
<thead>
<tr>
<th>head</th>
</tr>
</thead>
<tbody>
<tr>
<td>q</td>
</tr>
<tr>
<td>tail</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>v</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>v</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
</tr>
<tr>
<td>next</td>
</tr>
</tbody>
</table>
```
Evaluating length

Workspace

\[
\text{let loop} = \text{fun (no: \ldots) \rightarrow}
\begin{align*}
\text{begin match no with} \\
\text{None \rightarrow 0} \\
\text{Some n \rightarrow 1 + (loop n.next)} \\
\text{end} \\
\text{in loop q.head}
\end{align*}
\]

Stack

- length
- q
- ()

Heap

- fun (q:'a queue) \rightarrow
- let rec loop (no:\ldots) : int = ...
- in loop q.head

- head
- tail
- q
- v 1
- next
- v 2
- next
Evaluating length

Workspace

loop q.head

Stack

length

q

Heap

fun (q:'a queue) ->
let rec loop (no:…) : int =
  ...
in
loop q.head

head

q

tail

loop

v 1

next

v 2

next

fun (no: …) ->
begin match no with
  | None -> 0
  | Some n -> 1 + (loop n.next)
end

CIS120 / Spring 2012
Evaluating length

Workspace

Stack

Heap

fun (q:'a queue) ->
   let rec loop (no:...) : int = ...
      in
      loop q.head

After a few steps...

(From here on, we’ll take some shortcuts in the ASM animations.)

fun (no: ...) ->
   begin
      match no with
      | None -> 0
      | Some n -> 1 + (loop n.next)
   end
begin match no with
    | None  -> 0
    | Some n  -> 1 + (loop n.next)
end

fun (q:'a queue)  ->
  let rec loop (no:...) : int =
    ... in
  loop q.head

fun (no: ...)  ->
  begin match no with
    | None  -> 0
    | Some n  -> 1 + (loop n.next)
  end
begin match _ with
| None  -> 0
| Some n -> 1 + (loop n.next)
end

fun (q: 'a queue) ->
let rec loop (no: ...) : int = ...
in
loop q.head

fun (no: ...) ->
begin match no with
| None  -> 0
| Some n -> 1 + (loop n.next)
end
Evaluating length

Workspace

begin match no with
| None -> 0
| Some n -> 1 + (loop n.next)
end

Stack

fun (q: 'a queue) ->
let rec loop (no: ...) : int = ...
in
loop q.head

Heap

fun (no: ...) ->
begin match no with
| None -> 0
| Some n -> 1 + (loop n.next)
end

begin match length with
| None -> 0
| Some q -> 1 + (loop q.head)
end

(q)

(head)

(tail)

(loop)

(v 1)

(next)

(v 2)

(next)
Evaluating length

Workspace

begin match no with
| None -> 0
| Some n -> 1 + (loop n.next)
end

Stack

<table>
<thead>
<tr>
<th>length</th>
</tr>
</thead>
<tbody>
<tr>
<td>q</td>
</tr>
</tbody>
</table>

Heap

fun (q: 'a queue) ->
let rec loop (no: ...) : int =
  ... in
  loop q.head

begin match no with
| None -> 0
| Some n -> 1 + (loop n.next)
end
Evaluating length

Workspace

1 + (loop n.next)

Stack

<table>
<thead>
<tr>
<th>length</th>
</tr>
</thead>
<tbody>
<tr>
<td>q</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>(</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

Heap

fun (q:'a queue) ->
let rec loop (no:...) : int =
  ...
in
loop q.head

head

tail

loop

<table>
<thead>
<tr>
<th>v 1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>(</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>v 2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

fun (no: ...) ->
begins match no with
  | None -> 0
  | Some n -> 1 + (loop n.next)
end
Evaluating length

Workspace

Stack

Heap

fun (q:'a queue) ->
let rec loop (no:…) : int =
  ...
in
loop q.head

fun (no: …) ->
begin match no with
  | None -> 0
  | Some n -> 1 + (loop n.next)
end

1 + ( )

length

q

( )

head

q

tail

loop

( )

no

v 1

next

v 2

next

loop q.head

CIS120 / Spring 2012
begin match no with
  | None -> 0
  | Some n -> 1 + (loop n.next)
end
...after a few steps...
Evaluating length

Workspace

Stack

Heap

fun (q:'a queue) ->
let rec loop (no:...) : int =
  ...
in
  loop q.head

fun (no: ...) ->
begin match no with
  | None -> 0
  | Some n -> 1 + (loop n.next)
end

1 + (loop n.next)
...after a few more steps...
begin match no with
| None -> 0
| Some n -> 1 + (loop n.next)
end

fun (q:'a queue) ->
let rec loop (no:…) : int =
    ...
in
loop q.head
Workspace

begin match no with
    | None -> 0
    | Some n -> 1 + (loop n.next)
end
fun (q:'a queue) ->
let rec loop (no:…) : int =
  ...
  in
  loop q.head

fun (no:…) ->
begin match no with
| None -> 0
| Some n -> 1 + (loop n.next)
end
func (q:'a queue) ->
let rec loop (no:...) : int =
  ...
  in
  loop q.head

fun (no: ...) ->
begin match no with
  | None -> 0
  | Some n -> 1 + (loop n.next)
end
fun (q:'a queue) ->
  let rec loop (no:...) : int =
    ...
    in
    loop q.head

fun (no: ...) ->
  begin match no with
    | None -> 0
    | Some n -> 1 + (loop n.next)
  end
fun (q:'a queue) ->
let rec loop (no:…) : int =
  …
in loop q.head

fun (no: …) ->
begin match no with
  | None -> 0
  | Some n -> 1 + (loop n.next)
end
EvaluaAng

length

Workspace

Stack

Heap

function (q: 'a queue) ->
let rec loop (no: ...) : int =
  ... in
  loop q.head

fun (no: ...) ->
begin match no with
  None -> 0
  Some n -> 1 + (loop n.next)
end

1 + 1
fun (q: 'a queue) ->
let rec loop (no: ...) : int =
  ...
in
  loop q.head

fun (no: ...) ->
begin match no with
| None -> 0
| Some n -> 1 + (loop n.next)
end
Evaluation of length

Workspace

Stack
- length
- q

Heap
- fun (q:'a queue) ->
  let rec loop (no:...) : int =
    ...
    in
    loop q.head

POP!
Evaluating length

Workspace

2

Stack

<table>
<thead>
<tr>
<th>length</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td>q</td>
</tr>
</tbody>
</table>

Heap

fun (q: 'a queue) ->
let rec loop (no:...) : int =
  begin
    match no with
    | None -> 0
    | Some n -> 1 + (loop n.next)
  end

DONE!
Iteration

loops
length (using iteration)

(* Calculate the length of the list using iteration *)
let length (q:'a queue) : int =
  let rec loop (no:'a qnode option) (len:int) : int =
    begin
      match no with
        | None -> len
        | Some n -> loop n.next (1+len)
    end
  in
  loop q.head 0

• This code for length also uses a helper function, loop:
  – This loop takes an extra argument, len, called the accumulator
  – Unlike the previous solution, the computation happens “on the way down” as opposed to “on the way back up”
  – Note that loop will always be called in an empty workspace—the results of the call to loop never need to be used to compute another expression. In contrast, we had (1 + (loop ...)) in the recursive version.
Tail Call Optimization

• Why does it matter that ‘loop’ is only called in an empty workspace?

• We can optimize the abstract stack machine:
  – The workspace pushed onto the stack tells us “what to do” when the function call returns.
  – If the pushed workspace is empty, we will always ‘pop’ immediately after the function call returns.
  – Therefore, we do not need to save the ‘empty’ workspace on the stack.
  – Moreover, any local variables that were pushed so that the current workspace could evaluate will no longer be needed, so we can eagerly pop them too.

• The end result is that we have turned recursion into a true loop. (Just like a ‘while’ or ‘for’ loop in Java or C.)
Tail Calls and Iterative length

Workspace

length q

Stack

| length | q |

Heap

fun (q:'a queue) ->
let rec loop (no:…) (len:int)= …
in
loop q.head 0

Bindings above this line are top-level declarations.
Tail Calls and Iterative length

Workspace

Stack

Heap

fun (q:'a queue) ->
let rec loop (no:...) (len:int)=
  ...
in
  loop q.head 0

Tail Call!
Tail Calls and Iterative length

Workspace

```
let rec loop (no:'a qnode option) (len:int) : int =
    begin match no with
    | None -> len
    | Some n -> loop n.next (1+len)
    end
    in loop q.head 0
```

Stack

```
<table>
<thead>
<tr>
<th>length</th>
</tr>
</thead>
<tbody>
<tr>
<td>q</td>
</tr>
</tbody>
</table>
```

Heap

```
fun (q:'a queue) ->
    let rec loop (no:...) (len:int)=
        ...
        in loop q.head 0
```

Note:

(1) No workspace is saved – there is no need to do that for tail calls

(2) We pop all the locals (up to the last saved workspace). In this case, there are none.
Tail Calls and Iterative length

Workspace

```
(loops q.head 0)
```

Stack

```
<table>
<thead>
<tr>
<th>length</th>
</tr>
</thead>
<tbody>
<tr>
<td>q</td>
</tr>
<tr>
<td>q</td>
</tr>
<tr>
<td>loop</td>
</tr>
</tbody>
</table>
```

Heap

```
fun (q:'a queue) ->
let rec loop (no:...) (len:int)=
  ...
in
loop q.head 0
```

A detail we’ve been sweeping under the rug until now:

The *closure* of the local recursive function `loop` includes a binding for the loop function itself!

Why? The loop body mentions the loop identifier.

CIS120 / Spring 2012
Tail Calls and Iterative length

Workspace

Stack

Heap

fun (q:'a queue) ->
let rec loop (no:…) (len:int)= …
in
loop q.head 0

fun (no:'a qnode option)
(len:int) -> begin match no with
| None -> len
| Some n -> loop n.next (1+len)
end
Tail Calls and Iterative length

Workspace

( 0 )

Stack

length

q

q

Heap

fun (q:'a queue) ->
let rec loop (no:...) (len:int)= ...
in
loop q.head 0

head
tail

v 1
next

v 2
next

loop

fun (no:'a qnode option)
(len:int) -> begin match no with
  None -> len
  Some n -> loop n.next (1+len)
end
Tail Calls and Iterative length

Workspace

begin match no with
  | None -> len
  | Some n -> loop n.next (1+len)
end

Stack

length

q

Heap

fun (q:'a queue) ->
  let rec loop (no:...) (len:int)=
    ...
in
  loop q.head 0

loop

no

len 0

Notes:
- no workspace is saved on the stack
- pop the old locals (q and loop)
- push the closure and argument bindings
- the new workspace is just the body of the function

This binding comes from the closure.
Tail Calls and Iterative length

Workspace

begin match with
  | None -> len
  | Some n -> loop n.next (1+len)
end

Stack

length

q

loop

Heap

fun (q:'a queue) ->
  let rec loop (no:...) (len:int)=
    ...
    in
    loop q.head 0

head
tail

v 1
next

v 2
next
loop

fun (no:'a qnode option)
  (len:int) -> begin match no with
    | None -> len
    | Some n -> loop n.next (1+len)
  end
Tail Calls and Iterative length

Workspace

```
begin match with
| None -> len
| Some n -> loop n.next (1+len)
end
```

Stack

<table>
<thead>
<tr>
<th>length</th>
</tr>
</thead>
<tbody>
<tr>
<td>q</td>
</tr>
</tbody>
</table>

Heap

```
fun (q:'a queue) ->
let rec loop (no:...) (len:int)=
    ...
in
loop q.head 0
```

```
fun (no:'a qnode option)
(len:int) -> begin match no with
| None -> len
| Some n -> loop n.next (1+len)
end
```
Tail Calls and Iterative length

Workspace

begin match with
  | None -> len
  | Some n -> loop n.next (1+len)
end

Stack

length

loop

Heap

fun (q:'a queue) ->
let rec loop (no:…) (len:int)=
  …
in
  loop q.head 0

head

tail

len 0

v 1

next

v 2

next

loop

fun (no:'a qnode option)
  (len:int) -> begin match no with
  | None -> len
  | Some n -> loop n.next (1+len)
end

CIS120 / Spring 2012
Tail Calls and Iterative length

Workspace

```
(loop n.next (1+len))
```

Stack

```
<table>
<thead>
<tr>
<th>length</th>
</tr>
</thead>
<tbody>
<tr>
<td>q</td>
</tr>
</tbody>
</table>

Heap

```
fun (q:'a queue) ->
  let rec loop (no:…) (len:int)=
      ...
    in
    loop q.head 0
```
Tail Calls and Iterative length

Workspace

Stack

Heap

fun (q:'a queue) -> 
let rec loop (no:...) (len:int)= 
  ... 
in
  loop q.head 0

fun (no:'a qnode option) 
(len:int) -> begin match no with 
  None -> len 
  Some n -> loop n.next (1+len) 
end
Tail Calls and Iterative length

Workspace

( (1+len))

Stack

<table>
<thead>
<tr>
<th>length</th>
</tr>
</thead>
<tbody>
<tr>
<td>q</td>
</tr>
</tbody>
</table>

| loop |
| no    |
| len   |
| 0     |

Heap

fun (q:'a queue) ->
  let rec loop (no:...) (len:int)=
    ...
in
  loop q.head 0

loop

head

tail

len

(v 1)

next

(v 2)

next

loop

fun (no:'a qnode option)
  (len:int) -> begin match no with
    None -> len
    Some n -> loop n.next (1+len)
  end
Tail Calls and Iterative length

Workspace

Stack

Heap

fun (q:'a queue) ->
let rec loop (no:…) (len:int)= …
in
  loop q.head 0

let rec loop (no:…) (len:int)= …
in
  loop q.head 0

fun (no:'a qnode option)
  (len:int) -> begin match no with
    | None -> len
    | Some n -> loop n.next (1+len)
  end
Tail Calls and Iterative length

Workspace

```
begin match no with
  | None -> len
  | Some n -> loop n.next (1+len)
end
```

Stack

```
<table>
<thead>
<tr>
<th>length</th>
</tr>
</thead>
<tbody>
<tr>
<td>q</td>
</tr>
</tbody>
</table>
```

Heap

```
fun (q:'a queue) ->
let rec loop (no:...) (len:int)=
  ...
in
loop q.head 0
```

```
<table>
<thead>
<tr>
<th>head</th>
</tr>
</thead>
<tbody>
<tr>
<td>no</td>
</tr>
<tr>
<td>tail</td>
</tr>
<tr>
<td>len 1</td>
</tr>
</tbody>
</table>
```

```
| v    |
|      |
| 1    |
| next |
```

```
| v    |
|      |
| 2    |
| next |
```

Note: we popped the old values of loop, no, len, and n when we did the tail call. Then we pushed the new values of loop, no, and len.

This leaves the stack in almost the same shape as when we first called loop.

Effectively, we have updated the stack slots for no and len.
Tail Calls and Iterative length

Workspace

```
begin match no with
  None -> len
| Some n -> loop n.next (1+len)
end
```

Stack

```
fun (q:'a queue) ->
  let rec loop (no:...) (len:int)=
    ...
in
  loop q.head 0
```

Heap

```
loop
no
len 1
q
```

```
head
tail
```

```
v 1
next
```

```
v 2
next
```

```
loop
```

```
fun (no:'a qnode option) (len:int) ->
  begin match no with
    None -> len
    | Some n -> loop n.next (1+len)
  end
```
Tail Calls and Iterative length

Workspace

```
begin match with
  | None -> len
  | Some n -> loop n.next (1+len)
end
```

Stack

- length
- q
- loop
- no
- len 1

Heap

```
fun (q:a queue) ->
  let rec loop (no:...) (len:int)=
    ...
  in
  loop q.head 0
```

CIS120 / Spring 2012
Tail Calls and Iterative length

Workspace

Stack

Heap

```ocaml
fun (q:'a queue) ->
let rec loop (no:...) (len:int)=
  ...
in
  loop q.head 0
```

```
fun (no:'a qnode option) (len:int) -> begin match no with
  None -> len
  Some n -> loop n.next (1+len)
end
```
Tail Calls and Iterative length

Workspace

Stack

Heap

fun (q: 'a queue) ->
  let rec loop (no:…) (len:int)=
  ... in
  loop q.head 0

fun (no: 'a qnode option)
  (len:int) -> begin match no with
  | None -> len
  | Some n -> loop n.next (1+len)
  end
Tail Calls and Iterative length

Workspace

( (1+len))

Stack

<table>
<thead>
<tr>
<th>length</th>
</tr>
</thead>
<tbody>
<tr>
<td>q</td>
</tr>
<tr>
<td>loop</td>
</tr>
<tr>
<td>no</td>
</tr>
<tr>
<td>len 1</td>
</tr>
</tbody>
</table>

Heap

fun (q:'a queue) ->
let rec loop (no:) (len:int)=
  ...
  in
  loop q.head 0

fun (no:'a qnode option)
(len:int) -> begin match no with
           | None -> len
           | Some n -> loop n.next (1+len)
           end
Tail Calls and Iterative length

Workspace

(1+len)

Stack

length

q

loop

Heap

fun (q:'a queue) ->
    let rec loop (no:) (len:int)=
        ...
in
    loop q.head 0

head

tail

len 1

next

v 1

next

v 2

next

None

loop

fun (no:'a qnode option)
(len:int) -> begin match no with
    | None -> len
    | Some n -> loop n.next (1+len)
end
Tail Calls and Iterative length

Workspace

( (\(1+1\)) )

Stack

length

q

loop

no

len 1

Heap

fun (q:'a queue) ->
let rec loop (no:…) (len:int)=
  …
in
  loop q.head 0

head
tail
len 1

next

v 1

next

v 2

next

None

fun (no:'a qnode option)
  (len:int) ->
  begin
    match no with
    | None ->
      len
    | Some n ->
      loop n.next
        (1+len)
  end

CIS120 / Spring 2012
Tail Calls and Iterative length

Workspace

(2)

Stack

length

q

loop

Heap

fun (q: a queue) ->
let rec loop (no: …) (len: int) = …
in
loop q.head 0

head

tail

len 1

next

v 2

next

None

v 1

next

loop

fun (no: a qnode option)
(len: int) -> begin match no with
| None -> len
| Some n -> loop n.next (1 + len)
end

CIS120 / Spring 2012
Note: Again, the tail call leaves the stack as before, but effectively updates the values of `no` and `len`.

We may as well call this in-place-update of the stack, even though technically these bindings are not mutable!
begin match no with
| None -> len
| Some n -> loop n.next (1+len)
end

fun (q:'a queue) ->
let rec loop (no:...) (len:int)=
  ...
in
  loop q.head 0

fun (no:'a qnode option)
  (len:int) -> begin match no with
  | None -> len
  | Some n -> loop n.next (1+len)
  end
Tail Calls and Iterative length

Workspace

Stack

Heap

fun (q: 'a queue) ->
  let rec loop (no:…) (len:int) =
    ...
in
  loop q.head 0

begin match with
  | None -> len
  | Some n -> loop n.next (1+len)
end
Tail Calls and Iterative length

Workspace

Stack

Heap

fun (q:'a queue) ->
let rec loop (no:...) (len:int)=
  ...
  in
  loop q.head 0

fun (no:'a qnode option) (len:int) -> begin match no with
  | None -> len
  | Some n -> loop n.next (1+len)
  end
Tail Calls and Iterative length

Workspace

Stack

Heap

fun (q:'a queue) ->
let rec loop (no:…) (len:int)= …
in
loop q.head 0

CIS120 / Spring 2012
Tail Calls and Iterative length

Workspace

Stack

Heap

fun (q: 'a queue) ->
let rec loop (no: ...) (len: int) =
    ...
in
    loop q.head 0

DONE!
Some Observations

• Tail call optimization lets the stack take only a fixed amount of space.

• The “recursive” call to loop (effectively) updates some of the stack bindings in place.
  – We can think of these bindings as the state being modified by each iteration of the loop.

• These two properties are the essence of iteration.
  – They are the difference between recursion and iteration
  – Most imperative programs provide iteration with “while” “for”, and the related “break” and “continue” operations.
  – Tail recursion generalizes all of these.
**to_list (using iteration)**

```ocaml
(* Retrieve the list of values stored in the queue, ordered from head to tail. *)
let to_list (q: 'a queue) : 'a list =
  let rec loop (no: 'a qnode option) (l:'a list) : 'a list =
  begin match no with
    | None -> List.rev l
    | Some n -> loop n.next (n.v::l)
  end
  in loop q.head []
```

- Here, the state maintained across each iteration of the loop is the queue “index pointer” no and the (reversed) list of elements traversed.
- The “exit case” post processes the list be reversing it.
Infinite Loops

• This program will go into an infinite loop.

• Unlike a recursive program, which uses some space on each recursive call, there is no resource being exhausted, so the program will “silently diverge” and never produce an answer...

```ocaml
(* Accidentally go into an infinite loop... *)
let accidental_infinite_loop (q:'a queue) : int =
  let rec loop (qn:'a qnode option) (len:int) : int =
    begin match qn with
    | None -> len
    | Some n -> loop qn (len + 1)
    end
  in loop q.head 0
```
print (using iteration)

```ocaml
let print (q:'a queue) (string_of_element:'a -> string) : unit =
  let rec loop (no: 'a qnode option) : unit =
    begin match no with
      | None -> ()
      | Some n -> print_endline (string_of_element n.v);
      loop n.next
    end
  in
  print_endline "--- queue contents ---";
  loop q.head;
  print_endline "--- end of queue ------"
```

• Here, the only state needed is the queue “index pointer”.
get_tail (using iteration)

```ocaml
(* get the tail (if any) from a queue *)
let rec get_tail (q: 'a queue) : 'a qnode option =
  let rec loop (qn: 'a qnode) (seen: 'a qnode list)
    : 'a qnode option =
    begin match qn.next with
    | None -> Some qn
    | Some n ->
      if contains_alias n seen then None
      else loop n (qn::seen)
    end
  in loop q.head []
```

- This function does not assume that q has no cycles.
- It returns Some n if n is a “tail” reachable from q.head
- It returns None if there is a cycle in the queue found by following next pointers.
- The state is an index pointer and a list of all the nodes seen.
  - contains_alias is a helper function that checks to see whether n has an alias in the list