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

GREAT JOB!
Mutable Queues: Recap

singly linked data structures
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

```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:

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

- 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)
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
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}
(* 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
\end{verbatim}
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
Calling Enq on a non-empty queue

Workspace

enq 2 q

Stack

enq
q

Heap

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

head

v 1

tail

next
Calling Enq on a non-empty queue

Workspace

```
enq 2 q
```

Stack

```
| enq |
| q |
```

Heap

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

```
| head |
| tail |
| v 1 |
| next |
```
Calling Enq on a non-empty queue

Workspace

2 q

Stack

enq
q

Heap

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

head
tail

v 1
next
Calling Enq on a non-empty queue

Workspace

Stack

Heap

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

head
tail

v 1

next
Calling Enq on a non-empty queue

Workspace

2

Stack

enq

q

Heap

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

head
tail

v 1

next
Calling Enq on a non-empty queue

Workspace (2)

Stack

Heap

fun (x: 'a) (q: 'a queue) ->
  let newnode = {v=x; next=Nothing}
  in begin
    match q.tail with
    | None -> ...
    | Some n -> ...
  end

head
tail

v 1
next
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;
  | Some n -> n.next <- Some newnode
end
```

Stack

```
| enq |
- q |
________
| () |
| x 2 |
- q |
```

Heap

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

```
| head |
| tail |
| v 1 |
| next |
```
Calling Enq on a non-empty queue

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

Workspace

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

Stack

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

Heap

```
head
```

```
tail
```

```
v
next
```

Workspace Stack Heap
Calling Enq on a non-empty queue

Workspace

```haskell
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

- `enq`
- `q`

Heap

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

```
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
```

- `head`
- `tail`
- `v` (1)
- `next`
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

```
| enq |
| q   |
```

Heap

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

```
| head | tail |
| v    | 1    |
| x 2  | q    |
```
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

```
enq
q
```

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

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

Workspace

Stack
- enq
- q

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

- head
- tail
- x 2
- v 1
- next
- v 2
- next
Calling Enq on a non-empty queue

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

Workspace

Stack

Heap

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

head
tail
q
newnode

v 1
next

v 2
next
Calling Enq on a non-empty queue

Workspace

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

Stack

enq
q

Heap

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

head
tail

v 1
next

v 2
next
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

enq
q

Heap

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

head

x 2
q
newnode

tail

v 1
next

v 2
next
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

enq

q

Heap

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

head

tail

v 1

next

v 2

next

Workspace Stack Heap

_____
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

enq

q

(heap)

newnode

v = 1
next

Heap

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

head

v = 2
next

tail

x = 2

 Stack

Ahead

Workspace

newnode

v = 1
next

tail

x = 2

 Stack
Calling Enq on a non-empty queue

Workspace

begin match with
  | None ->
  | Some n ->
  | q.tail <- Some newnode
end

Stack

enq
q

Heap

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

Workspace

Stack

(())

Heap

head
tail
q

newnode

v 1
next

v 2
next
Calling Enq on a non-empty queue

Workspace

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

Stack

- `enq`
- `q`

Heap

```
fun (x: 'a) (q: 'a queue) ->
let newnode = {v=x; next=Nothing}
in begin match q.tail with
  | None -> ...
  | Some n -> ...
end
```

- `head` 1
- `tail` 2
- `x` 1
- `q`
- `newnode` 2
- `v` 2
- `next` 2
- `v` 1
- `next` None
- `next` None
Calling Enq on a non-empty queue

Workspace

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

Stack

enq

q

Heap

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

head

tail

v 1

newnode

v 2

next

next
Calling Enq on a non-empty queue

Workspace

\[
n.\text{next} \leftarrow \text{Some newnode} \\
q.\text{tail} \leftarrow \text{Some newnode}
\]

Stack

\[
\text{enq} \\
q
\]

Heap

\[
\text{fun } (x: \text{'a}) (q: \text{'a queue}) \rightarrow \\
\text{let newnode = } \{\text{v=x; next=None}\} \\
in \text{begin} \text{match } q.\text{tail} \text{ with} \\
| \text{None} \rightarrow \ldots \\
| \text{Some } n \rightarrow \ldots \\
\text{end}
\]

Note: \(n\) points to a qnode, not a qnode option.
Calling Enq on a non-empty queue

Workspace

n.next <- Some newnode;
qu.tail <- Some newnode

Stack

enq
q

(_,)

x 2
q
newnode

Heap

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

head
tail

v 1
next

v 2
next
Calling Enq on a non-empty queue

Workspace

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

Stack

```
| enq |
| q   |
```

Heap

```
fun (x: 'a) (q: 'a queue) ->
  let newnode = {v=x; next=Nothing}
  in begin match q.tail with
    | None -> ...
    | Some n -> ...
  end
```

```
head
```
```
tail
```
```
v 1
```
```
newnode
```
```
v 2
```
```
next
```
```
v 2
```
```
next
```
Calling Enq on a non-empty queue

Workspace

Workspace

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

\[ \text{.next <- Some } \; ; \]
\[ \text{q.tail <- Some newnode} \]

Stack

\[
\begin{array}{c}
\text{enq} \\
\text{q} \\
() \\
\text{x} \; 2 \\
\text{q} \\
\text{newnode} \\
\text{v} \; 1 \\
\text{next} \\
\text{next} \\
\text{v} \; 2 \\
\text{next}
\end{array}
\]

Heap

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

Diagram:

- Workspace: `.next <- Some ; q.tail <- Some newnode`
- Stack:
  - `enq`:
  - `q`:
  - `()`:
  - `x 2`
  - `q`:
  - `newnode`:
  - `v 1`
  - `next`:
  - `next`:
  - `v 2`
  - `next`:
- Heap:
  - `fun (x: 'a)(q: 'a queue) ->`
  - `let newnode = \{v=x; next=\text{None}\}`
  - `in begin match q.tail with`
  - `| None -> ...`
  - `| Some n -> ...`
  - `end`
Calling Enq on a non-empty queue.

Workspace

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

Stack

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

Heap

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

head
tail
```

```geometry
<table>
<thead>
<tr>
<th>v 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>newnode</td>
</tr>
<tr>
<td>next</td>
</tr>
</tbody>
</table>
```

```geometry
<table>
<thead>
<tr>
<th>v 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
</tr>
</tbody>
</table>
```

```
(head, tail) x 2
```

Workspace

```
( )
```

```geometry
x 2
q
```

```geometry
v
next
n
```
Calling Enq on a non-empty queue

Workspace

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

Stack

```
  enq
  q
```

Heap

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

head

tail

v 1
newnode
next

v 2
next
```

Workspace

```
.next <- 
q.tail <- Some newnode
```
Calling Enq on a non-empty queue

Workspace

\[
\begin{array}{l}
\text{();} \\
\text{q.tail <- Some newnode}
\end{array}
\]

Stack

\[
\begin{array}{l}
\text{enq} \\
\text{q}
\end{array}
\]

Heap

\[
\begin{array}{l}
\text{fun (x: 'a) (q: 'a queue) ->} \\
\text{let newnode = \{v=x; next=\text{None}\}} \\
\text{in begin match q.tail with} \\
\text{ | None -> ...} \\
\text{ | Some n -> ...} \\
\text{end}
\end{array}
\]

\[
\begin{array}{l}
\text{head} \\
\text{tail} \\
\text{x 2} \\
\text{q} \\
\text{newnode} \\
\text{n}
\end{array}
\]

\[
\begin{array}{l}
\text{v 1} \\
\text{next}
\end{array}
\]

\[
\begin{array}{l}
\text{v 2} \\
\text{next}
\end{array}
\]
Calling Enq on a non-empty queue

Workspace

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

Stack

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

Heap

```
Let us assume that v1 = x and n.next = None.
```

```
head
```

```
tail
```

```
v 1
```

```
newnode
```

```
x 2
```

```
v 2
```

```
next
```

```
n
```

```
v
```

```
next
```
Calling Enq on a non-empty queue

Workspace

q.tail <- Some newnode

Stack

enq
q

Heap

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

(head)

x 2
q
cnewnode

v 1
next

v 2
next
Calling Enq on a non-empty queue

Workspace

\[ q \text{.tail} \leftarrow \text{Some newnode} \]

Stack

\[
\begin{align*}
&\text{enq} \\
&q
\end{align*}
\]

Heap

\[
\text{fun} \ (x: \ 'a) \ (q: \ 'a \ \text{queue}) \rightarrow \\
\text{let} \ \text{newnode} = \{v=x; \ \text{next}=\text{None}\} \\
\text{in} \ \text{begin} \ \text{match} \ q.\text{tail} \text{ with} \\
&| \ \text{None} \rightarrow \ldots \\
&| \ \text{Some n} \rightarrow \ldots \\
\text{end}
\]

Workspace:

- `q.tail <- Some newnode`

Stack:

- `enq`
- `q`

Heap:

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

Diagram:

- `q.tail`
- `Some newnode`