Topic 12: Stacks and FIFO Queues[1]
(Version of 10th November 2010)

Pierre Flener

Computing Science Division
Department of Information Technology
Uppsala University
Sweden

Course 1DL201:
Program Construction and Data Structures

[1] Based on original slides by Yves Deville

UPPSALA UNIVERSITET

Outline

1 Stacks

2 FIFO Queues

Outline

Stacks

Stacks ("last-in first-out lists") of elements of any type: `'a stack`

- Insertion (called push) of an element only to the top.
- Extraction (called pop) only of the element at the top.

Value and Some Operations

```
empty
TYPE:  'a stack
VALUE: the empty stack

isEmpty S
TYPE: ''a stack -> bool
PRE:  (none)
POST: true if S is empty, and false otherwise

push (S, t)
TYPE: 'a stack * 'a -> 'a stack
PRE:  (none)
POST: the stack S with t added as new top element
```

More Operations

```
top S
TYPE: 'a stack -> 'a
PRE:  S is nonempty
POST: the top element of S

pop S
TYPE: 'a stack -> 'a * 'a stack
PRE:  S is nonempty
POST: (t, S'), where t is the top element of S,
      and S' is S without t

walk S
TYPE: 'a stack -> 'a list
PRE:  (none)
POST: the representation of S in list form,
      with the top of S as head, and so on
```

Representation 1

Representation of a stack by a list:

```
datatype 'a stack = Stack of 'a list
REPRESENTATION CONVENTION: the head of the list is the top of
  the stack, the 2nd element of the list is the element below
  the top, and so on
REPRESENTATION INVARIANT: (none)
```

where `stack` is a type constructor
and `Stack` is a value constructor.

```
val empty = Stack []
fun isEmpty (Stack S) = (S = [])
fun push (Stack S, t) = Stack(t::S)
fun top (Stack(t::S)) = t
fun pop (Stack(t::S)) = (t, Stack S)
fun walk (Stack S) = S
```

Exercise: All these operations always take $\Theta(1)$ time.

Representation 2

Definition of a recursive new constructed type:

```
datatype 'a stack = EmptyStack | >> of 'a stack * 'a
infix >>
EXAMPLE: EmptyStack >> 5 >> 2 represents the stack with top 2
REPRESENTATION CONVENTION: the right-most value is the top of
the stack, its left neighbour is the element below the top, etc.
REPRESENTATION INVARIANT: (none)

val empty = EmptyStack
fun isEmpty S = (S = EmptyStack)
fun push (S, t) = S>>t
fun top (S>>t) = t
fun pop (S>>t) = (t, S)
VARIANT: |S|;  TIME COMPLEXITY: Θ(|S|)          (* Exercise! *)
fun walk EmptyStack = []
  | walk (S>>t) = t :: (walk S)
```

We have thus defined a new list constructor, with $\Theta(1)$ time (direct) access to the elements from the right!

Outline

First-In First-Out (FIFO) Queues

First-in first-out queues of elements of any type: `'a queue`

- Insertion (enqueue) of an element only to the rear (tail).
- Extraction (dequeue) only of element in front (head).

Value and Some Operations

```
empty
TYPE:  'a queue
VALUE: the empty queue

isEmpty Q
TYPE: ''a queue -> bool
PRE:  (none)
POST: true if Q is empty, and false otherwise

head Q
TYPE: 'a queue -> 'a
PRE:  Q is nonempty
POST: the head element of Q
```

More Operations

```
enqueue (Q, t)
TYPE: 'a queue * 'a -> 'a queue
PRE:  (none)
POST: the queue Q with t added as new tail element

dequeue Q
TYPE: 'a queue -> 'a * 'a queue
PRE:  Q is nonempty
POST: (h, Q'), where h is the head element of Q,
      and Q' is Q without h

walk Q
TYPE: 'a queue -> 'a list
PRE:  (none)
POST: the representation of Q in list form,
      with the head of Q as head, and so on
```

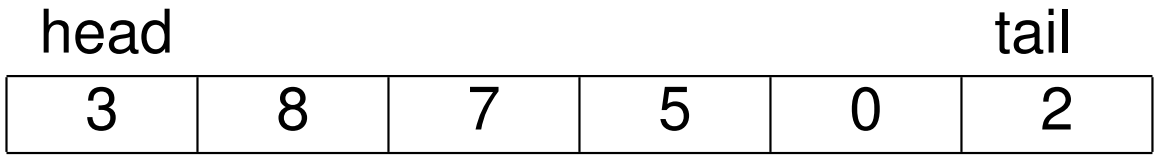
Representation 1

Representation of a FIFO queue by a list:

```
datatype 'a queue = Queue of 'a list
REPRESENTATION CONVENTION: the head of the list is the head of
  the queue, the 2nd element of the list is behind the head of
  the queue, and so on, and the last element of the list is the
  tail of the queue
REPRESENTATION INVARIANT: (none)
```

Example

The queue



is represented by the list `[3,8,7,5,0,2]`.

Exercises

- Implement this representation of the `'a queue` type.
- What is the time complexity of enqueuing an element?
- What is the time complexity of dequeuing an element?

Representation 2

Representation of a FIFO queue by a pair of lists:

```
datatype 'a queue = Queue of 'a list * 'a list
REPRESENTATION CONVENTION: the term
  Queue([x1,x2,...,xn],[y1,y2,...,ym]) represents the
  queue x1,x2,...,xn,ym,...,y2,y1 with head x1 and tail y1
REPRESENTATION INVARIANT: (see next slide)
```

Analysis

- It is now possible to enqueue in $\Theta(1)$ time.
- It is still possible to dequeue in $\Theta(1)$ time, but only if $n \geq 1$.
- What if $n = 0$ and $m > 0$?!
- The same queue can be represented in different ways.

Normalisation

Objective: Avoid the case where $n = 0$ and $m > 0$.
When this case appears, transform (or: normalise)
the term $\texttt{Queue}([\,], [y_1, \ldots, y_m])$ with $m > 0$ into
the term $\texttt{Queue}([y_m, \ldots, y_1], [\,])$,
which represents the same queue.

```
REPRESENTATION INVARIANT: a non-empty queue is
  never represented by Queue([],[y1,y2,...,ym])

normalise Q
TYPE: 'a queue -> 'a queue
PRE:  (none)
POST: if Q is of the form Queue([],[y1,y2,...,ym]),
      then Queue([ym,...,y2,y1],[]),
      else Q
```

Operations

```
val empty = Queue([],[])
TIME COMPLEXITY: Θ(1) always
fun isEmpty Q = (Q = (Queue([],[])))
TIME COMPLEXITY: Θ(1) always
fun head (Queue(h::xs,ys)) = h

local
  TIME COMPLEXITY: Θ(|Q|) at worst              (* Exercise! *)
  fun normalise (Queue([],ys)) = Queue(rev ys,[])
    | normalise Q = Q
in
  TIME COMPLEXITY: Θ(1) always                  (* Exercise! *)
  fun enqueue (Queue(xs,ys), t) = normalise (Queue(xs,t::ys))
  TIME COMPLEXITY: Θ(1) on average
  fun dequeue (Queue(h::xs,ys)) = (h, normalise (Queue(xs,ys)))
end

TIME COMPLEXITY: Θ(|Q|) always                  (* Exercise! *)
fun walk (Queue(xs,ys)) = xs @ (rev ys)
```