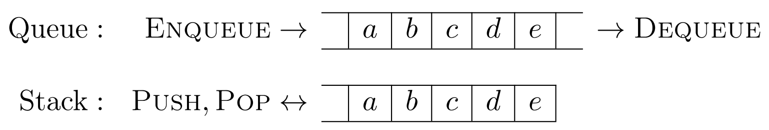
Answers to the mid-term on Erlang

Christian Rinderknecht

25 April 2008

A *queue* is a like stack where items are pushed on one end and popped on the other end. Adding an item in a queue is called *enqueuing*, whereas removing one is called *dequeuing*. Compare the following figures:

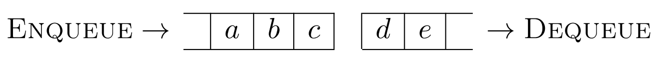


Let `enqueue(E,Q)` be the queue where `E` is the last item in and `Q` is the remaining queue. Let `dequeue(Q)` be the pair `{E,R}` where `E` is the first item out of queue `Q` (if there is none, `dequeue(Q)` is undefined) and `R` is the remaining queue.

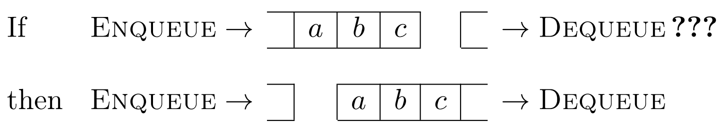
Question. Define `enqueue/2` and `dequeue/1` in the following two cases.

1. **A one-stack implementation.** A simple idea to implement one queue is to use one stack. In this case, ENQUEUE is simply PUSH and DEQUEUE removes the item at the bottom of the stack.

2. **A two-stack implementation.** We can implement one queue with two stacks instead of one: one for enqueuing, one for dequeuing.

   So ENQUEUE is PUSH on the first stack and DEQUEUE is POP on the second. If the second stack is empty, we swap the stacks and reverse the (new) second:

   Let the pair `{S,T}` denote the queue where `S` is the stack for enqueuing and `T` the stack for dequeuing.

Answer.

1. **A one-stack implementation.**

```
-module(queue1).
-export([enqueue/2,dequeue/1,dequeue1/1,dequeue2/1]).

enqueue(E,Q) -> [E|Q].

% Recursive but not tail-recursive:
%
dequeue([E])   -> {E,[]};
dequeue([E|Q]) -> {F,R} = dequeue(Q), {F,[E|R]}.

% Not recursive:
%
dequeue1([H|T]) -> [E|R] = rev([H|T]), {E,rev(R)}.

rev(L)       -> rev(L,[]).
rev([],A)    -> A;
rev([H|T],A) -> rev(T,[H|A]).

% Tail recursive:
%
dequeue2([H|T]) -> deq_aux1([H],T).

deq_aux1([I|A],   []) -> deq_aux2(I,[],A);
deq_aux1(    A,[I|T]) -> deq_aux1([I|A],T).

deq_aux2(I,A,   []) -> {I,A};
deq_aux2(I,A,[J|L]) -> deq_aux2(I,[J|A],L).
```

2. **A two-stack implementation.**

```
-module(queue2).
-export([enqueue/2,dequeue/1,dequeue1/1]).

enqueue(E,{S,T}) -> {[E|S],T}.

% Not tail-recursive:
%
dequeue({S,[E|T]})  -> {E,{S,T}};
dequeue({[H|S],[]}) -> [E|R] = rev([H|S]), {E,{[],R}}.

rev(L)       -> rev(L,[]).
rev([],A)    -> A;
rev([H|T],A) -> rev(T,[H|A]).

% Tail-recursive:
%
```

```
dequeue1({S,[E|T]})  -> {E,{S,T}};
dequeue1({[H|S],[]}) -> deq_aux1([H],S).

deq_aux1([I|A],   []) -> {I,{[],A}};
deq_aux1(    A,[I|S]) -> deq_aux1([I|A],S).
```

Question. Which of the one-stack or two-stack implementation is the most efficient for dequeuing? What are the best and worst cases for each?

Answer. In the one-stack implementation, every time we dequeue, we must traverse all the items in the stack. In the two-stack implementation, we only traverse the second stack, which is always shorter or equal than the stack in the first case.

1. The best case for the one-stack implementation is when the queue contains only one item, and the best case for the two-stack implementation is when the second stack is not empty: any dequeuing then has a constant cost (a pop).
2. All configurations of the one-stack implementation is the worst case: the cost of dequeuing is always the size of the stack, whilst the worst case for the two-stack implementation is when the second stack is empty, in which case the first stack has to be reversed, so the total cost is the number of items in the queue (which are then all in the first stack).

As a conclusion, the two-stack implementation is more efficient.

For a more quantified answer, we can compute the number of steps for each dequeuing function in the worst case (if any):

- `queue1:dequeue(L)` always takes `len(L)` steps.
- `queue1:dequeue1(L)` always takes `2*len(L)+3` steps.
- `queue1:dequeue2(L)` always takes `2*len(L)+2` steps.
- `queue2:dequeue({S,T})` takes `len(S)+2` steps when `T=[]`.
- `queue2:dequeue1({S,T})` takes `len(S)+1` steps when `T=[]`.

Note that `queue1:dequeue(L)` is the fastest, but that is because the pushes `[E|R]` are done on the stack, so they are not counted...