Zadanie L: SML - Kolejka O(1)

Zadanie polega na zaimplementowaniu trwałej kolejki przy użyciu stosów.
Operujemy na zestawie 6-ciu stosów:

```
signature STACKS =
sig
      type 'a stacks
      val snil : 'a stacks (* Tworzy zestaw 6-ciu stosów o numerach 0, 1, 2, 3, 4, 5 *)
      val size : int -> 'a stacks -> int (* Zwraca rozmiar stosu o podanym numerze - O(1) *)
      val empty : int -> 'a stacks -> bool (* Sprawdza, czy stos jest niepusty *)

      val pop : int -> 'a stacks -> 'a stacks (* Usuwa element na szczycie stosu *)
      val copy : int -> int -> 'a stacks -> 'a stacks (* Kopiuje element znajdujący się na szczycie p
      val dup : int -> int -> 'a stacks -> 'a stacks (*  Kopiuje cały pierwszy stos na drugi stos. Po
      val clear : int -> 'a stacks -> 'a stacks (* Czyści stos o podanym numerze *)
end
```

Wszystkie operacje wymienione w powyższej sygnaturze działają w czasie O(1).
Kolejka jest zdefiniowana przez sygnaturę:

```
signature RQUEUE =
sig
      type 'a info
      type 'a stacks
      val qnil : 'a stacks * 'a info (* Tworzenie pustej kolejki *)
      val enq : 'a stacks * 'a info -> 'a stacks * 'a info (* Dodawanie elementu na koniec kolejki *)
      val deq : 'a stacks * 'a info -> 'a stacks * 'a info (* Pobieranie i usuwanie elementu na pocza
end
```

Twoim zadaniem jest napisanie funktora:

```
functor Queue (Stacks : STACKS) : RQUEUE =
struct
      type 'a stacks = 'a Stacks.stacks
      datatype 'a info = ...
      val qnil = (Stacks.snil, ...)
      fun enq (stosy, stan) = ...
      fun deq (stosy, stan) = ...
end
```

Struktura info zawiera dodatkowe informacje (stan) wykorzystywane w kolejce. Przed wywołaniem operacji enq nowy element umieszczany jest na szczycie stosu 0. Operacja enq następnie może go przesunąć w inne miejsce (oraz wykonać inne modyfikacje kolejki), Operacja deq ma za zadanie przesunąć początkowy element kolejki na szczyt stosu 0. Zaraz po wykonaniu tej operacji element zostanie zdjęty.

Pojedyncza operacja enq lub deq może wykonywać nie więcej niż 20 operacji pop, copy, dup, clear na stosach. Pozostałe operacje nie są limitowane. Pesymistyczny czas działania funkcji enq i deq musi być $O(1)$.

Przykład

Strukturę implementującą stosy można zaimlementować np. tak:

```
structure SixStacks : STACKS =
struct
      type 'a stacks = (int * 'a list) list
      val snil = [(0, []), (0, []), (0, []), (0, []), (0, []), (0, [])]
      fun pop 0 ((cnt, head::tail)::rest) = (cnt-1, tail) :: rest
        | pop n (head::tail) = head :: (pop (n-1) tail)
      fun empty 0 ((cnt, _)::_) = cnt = 0
        | empty n (_::tail) = empty (n-1) tail
      local
              fun get 0 ((_, h::_)::_) = h
                | get n (_::tail) = get (n-1) tail
              fun put 0 v ((cnt, t)::rest) = (cnt+1, v::t)::rest
                | put n v (head::tail) = head :: (put (n-1) v tail)
      in
              fun copy i j S = put j (get i S) S
      end
      fun dup i j S = List.take (S, j) @ (List.nth (S, i) :: List.drop (S, j+1))
      fun clear i S = List.take (S, i) @ ((0, []) :: List.drop (S, i+1))
      fun size i (S : 'a stacks) = #1 (List.nth (S, i))
end
```

Po zaimplementowaniu funktora można go używać w następujący sposób:

```
- structure X = Queue(SixStacks);
structure X : RQUEUE
- open X;
opening X
  datatype 'a info = ...
  type 'a stacks = 'a SixStacks.stacks
  val qnil : 'a stacks * 'a info
  val enq : 'a stacks * 'a info -> 'a stacks * 'a info
  val deq : 'a stacks * 'a info -> 'a stacks * 'a info
- fun pre_enq ((cnt, head)::stacks_rest, info) v = ((cnt+1, v::head)::stacks_rest, info);
val pre_enq = fn
  : (int * 'a list) list * 'b -> 'a -> (int * 'a list) list * 'b
- fun post_deq ((cnt, head::tail)::stacks_rest, info) = (head, ((cnt-1, tail)::stacks_rest, info));
val post_deq = fn
  : (int * 'a list) list * 'b -> 'a * ((int * 'a list) list * 'b)
- val q = qnil;
val q = ([(0,[]),(0,[]),(0,[]),(0,[]),(0,[]),(0,[])],...)
  : 'a stacks * 'a info
- val q = enq (pre_enq q 10);
val q = ([(0,[]),(0,[]),(1,[10]),(0,[]),(0,[]),(0,[])],...)
  : int stacks * int info
- val q = enq (pre_enq q 20);
val q = ([(0,[]),(1,[20]),(1,[10]),(0,[]),(0,[]),(0,[])],...)
  : int stacks * int info
- val q = enq (pre_enq q 30);
```

```
val q = ([(0,[]),(1,[30]),(1,[10]),(1,[10]),(0,[]),(1,[20])],...)
  : int stacks * int info
- val (res, q) = post_deq (deq q);
val res = 10 : int
val q = ([(0,[]),(1,[30]),(1,[20]),(0,[]),(0,[]),(0,[])],...)
  : (int * int list) list * int info
- val (res, q) = post_deq (deq q);
val res = 20 : int
val q = ([(0,[]),(0,[]),(1,[30]),(0,[]),(0,[]),(0,[])],...)
: (int * int list) list * int info
- val q = enq (pre_enq q 40);
val q = ([(0,[]),(1,[40]),(1,[30]),(0,[]),(0,[]),(0,[])],...)
  : int stacks * int info
- val (req, q) = post_deq (deq q);
val req = 30 : int
val q = ([(0,[]),(0,[]),(1,[40]),(0,[]),(0,[]),(0,[])],...)
  : (int * int list) list * int info
- val (req, q) = post_deq (deq q);
val req = 40 : int
val q = ([(0,[]),(0,[]),(0,[]),(0,[]),(0,[]),(0,[])],...)
  : (int * int list) list * int info
```