Paradygmaty programowania – ćwiczenia

Lista 7

Algebraiczna specyfikacja kolejki nieskończonej

Sygnatura

```
empty : -> Queue
enqueue : Elem * Queue -> Queue
first : Queue -> Elem
dequeue : Queue -> Queue
isEmpty : Queue -> bool
```

Aksjomaty

```
For all q:Queue, e1,e2: Elem
isEmpty (enqueue (e1,q))            = false
isEmpty (empty)                     = true
dequeue (enqueue(e1,enqueue(e2,q))) =
      enqueue(e1,dequeue(enqueue(e2,q)))
dequeue (enqueue(e1,empty))         = empty
dequeue (empty)                     = empty
first (enqueue(e1,enqueue(e2,q)))   = first(enqueue(e2,q))
first (enqueue(e1,empty))           = e1
first (empty)                       = ERROR
```

1. (OCaml) Dana jest następująca sygnatura dla kolejek czysto funkcyjnych.

```
module type QUEUE_FUN =
sig
  type 'a t
  exception Empty of string
  val empty: unit -> 'a t
  val enqueue: 'a * 'a t -> 'a t
  val dequeue: 'a t -> 'a t
  val first: 'a t -> 'a
  val isEmpty: 'a t -> bool
end;;
```

   Napisz moduł, zgodny z powyższą sygnaturą, w którym kolejka jest reprezentowana:

   a) przez zwykłą listę,

   b) przez parę list.

   Reprezentacja z punku a) jest mało efektywna, ponieważ operacja wstawiania do kolejki (lub usuwania z kolejki) ma złożoność liniową. W lepszej reprezentacji kolejka jest reprezentowana przez parę list.

Para list $([x_1; x_2; \ldots x_m], [y_1; y_2; \ldots y_n])$ reprezentuje kolejkę $x_1 x_2 \ldots x_m y_n \ldots y_2 y_1$. Pierwsza lista reprezentuje początek kolejki, a druga – koniec kolejki. Elementy w drugiej liście są zapamiętane w odwrotnej kolejności, żeby wstawianie było wykonywane w czasie stałym (na początek listy).

enqueue(y, q) modyfikuje kolejkę następująco: $(xl, [y_1; y_2; \ldots y_n]) \to (xl, [y; y_1; y_2; \ldots y_n])$. Elementy w pierwszej liście są pamiętane we właściwej kolejności, co umożliwia szybkie usuwanie pierwszego elementu.

dequeue(q) modyfikuje kolejkę następująco: $([x_1; x_2; \ldots x_m], yl) \to ([x_2; \ldots x_m], yl)$.

Kiedy pierwsza lista zostaje opróżniona, druga lista jest odwracana i wstawiana w miejsce pierwszej: $([], [y_1; y_2; \ldots y_n]) \to ([y_n; \ldots y_2; y_1], [])$.

Reprezentacja kolejki jest w postaci normalnej, jeśli nie wygląda tak: $([], [y_1; y_2; \ldots y_n])$ dla $n \geq 1$. **Wszystkie operacje kolejki mają zwracać reprezentację w postaci normalnej**, dzięki czemu pobieranie wartości pierwszego elementu nie spowoduje odwracania listy. Odwracanie drugiej listy po opróżnieniu pierwszej też może się wydawać kosztowne. Jeśli jednak oszacujemy nie koszt pesymistyczny (oddzielnie dla każdej operacji kolejki), ale koszt zamortyzowany (uśredniony dla całego czasu istnienia kolejki), to okaże się, że koszt operacji wstawiania i usuwania z kolejki jest stały.

W ten sposób reprezentowane są kolejki w językach czysto funkcyjnych.

2. (OCaml) Dana jest następująca sygnatura dla kolejek modyfikowalnych.

```
module type QUEUE_MUT =
sig
  type 'a t
    (* The type of queues containing elements of type ['a]. *)
  exception Empty of string
    (* Raised when [first q] is applied to an empty queue [q]. *)
  exception Full of string
    (* Raised when [enqueue(x,q)] is applied to a full queue [q]. *)
  val empty: int -> 'a t
    (* [empty n] returns a new queue of length [n], initially empty. *)
  val enqueue: 'a * 'a t -> unit
    (* [enqueue (x,q)] adds the element [x] at the end of a queue [q]. *)
  val dequeue: 'a t -> unit
    (* [dequeue q] removes the first element in queue [q] *)
  val first: 'a t -> 'a
    (* [first q] returns the first element in queue [q] without removing
       it from the queue, or raises [Empty] if the queue is empty. *)
  val isEmpty: 'a t -> bool
    (* [isEmpty q] returns [true] if queue [q] is empty,
       otherwise returns [false]. *)
  val isFull: 'a t -> bool
    (* [isFull q] returns [true] if queue [q] is full,
       otherwise returns [false]. *)
end;;
```

Napisz moduł, zgodny z powyższą sygnaturą, w którym kolejka jest reprezentowana przez tablicę cykliczną.

kolejka pusta *kolejka pełna*