Programowanie współbieżne
Lista 3

Algebraiczna specyfikacja kolejki nieskończonej

Sygnatura
```
 empty       : -> Queue
 enqueue     : Elem * Queue -> Queue
 first       : Queue -> Elem
 firstOption: Queue -> Option[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
 firstOption(enqueue(e1,enqueue(e2,q))) = firstOption(enqueue(e2,q))
 firstOption(enqueue(e1,empty))         = Some(e1)
 firstOption(empty)                     = None
```

1. Zdefiniuj klasę generyczną dla kowariantnej kolejki niemodyfikowalnej, reprezentowanej przez dwie listy.
   W ten sposób reprezentowane są kolejki niemodyfikowalne w językach czysto funkcyjnych, a także w Scali (patrz dokumentacja).
   Wskazówka. Wzoruj się na klasie dla stosu z wykładu 3 (str. 9 i 28) oraz dokumentacji scala.collection.immutable.Queue (zaimplementuj tylko metody z powyższej specyfikacji).
   Zdefiniuj obiekt towarzyszący z metodami apply i empty.
   Utworzenie nowej kolejki ma być możliwe na cztery sposoby:
   new MyQueue   MyQueue()   MyQueue.empty   MyQueue('a', 'b', 'c')

Para list ([x_1; x_2; ...; x_m], [y_1; y_2; ...; y_n]) reprezentuje kolejkę x_1 x_2 ... x_m y_n ... 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; ...; y_n]) → (*xl*, [*y*;y_1; y_2; ...; 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; ...; x_m], *yl*) → ([x_2; ...; x_m], *yl*). Kiedy pierwsza lista zostaje opróżniona, druga lista jest odwracana i wstawiana w miejsce pierwszej: ([], [y_1; y_2; ...; y_n]) → ([y_n; ... y_2; y_1], []). Reprezentacja kolejki jest w postaci normalnej, jeśli nie wygląda tak: ([], [y_1; y_2; ...; 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 zamortyzowany koszt operacji wstawiania i usuwania z kolejki jest stały.

2. Dla drzew binarnych, zdefiniowanych na wykładzie 1, str. 94, napisz funkcję `breadthBT[A] : BT[A] => List[A]` obchodzącą drzewo binarne wszerz i zwracającą listę wartości, przechowywanych w węzłach drzewa. Wykorzystaj kolejkę z zadania 1.

Wszystkie definicje oraz proste testy w obiekcie singletonowym z metodą `main` umieść w pliku Lista3.scala.