ScalaQueue

Introduction

Ce projet correspond à un TP donné en Master informatique. Le but est de créer une liste chaînée auquel on aurait accès au dernier élément (celui que l'on a inséré en premier dans la liste) avec un temps constant (O(1)). Ce type de liste est donc appelée une queue et permet un accès FIFO (First in, last out).

Le temps d'accès au premier élément doit aussi être en temps constant.

Notre solution consiste à utiliser deux listes (in et out), in contient les éléments ajoutés au fil du temps. Tant que l'on ne retire pas d'élément, out reste vide. Dès que l'on veut retirer un élément, si out est vide, out deviens in en inversé et in devient alors vide, sinon on retire un élément de out.

Travail à réaliser

1. Implantez les méthodes de la classe Queue telles que listées ci-dessous.
2. Implantez une méthode length qui retourne la taille de le queue.
3. Implantez une méthode rearOption qui retourne le dernier élément de la queue (celui inséré en premier) sans la modifier.
4. Implantez une méthode toList qui convertit la Queue en liste simplement chaînée.
5. Implantez une méthode map sur Queue.
6. Implantez une méthode foldLeft sur Queue.
7. La méthode dequeue est partielle. Corrigez-la.
8. Ré-implantez isEmpty avec match.
9. Définissez un ensemble de tests exhaustifs dans un main. Nous n'avons pas encore vu de système de tests unitaires, mais suivez cette façon de faire

Structure de donnée

```
/** Une classe liste FIFO. */
case class Queue[T](in:List[T] = Nil, out:List[T] = Nil) {
    /** Ajoute un élément `x` en tête. */
    def enqueue(x:T):Queue[T] = ???
    /** Retire le dernier élément. */
    def dequeue():(T,Queue[T]) = ???
    /** Accès au premier élément, s'il existe (dernier élément entré). */
    def headOption():Option[T] = ???
    /** Vrai si la liste est vide. */
    def isEmpty:Boolean = in.isEmpty && out.isEmpty
}
```

1. Implantez les méthodes de la classe Queue telles que listées ci-dessus.

Pour cette première étape il fallait donc implantez les méthodes enqueue, dequeue et headOption.

Enqueue

Il fallait simplement ajouté notre élément x à la fin de notre list in.

```
val queue1 = Queue[Int](Nil, Nil)
val queue2 = queue1.enqueue(1) // Queue(List(1),List())
val queue3 = queue2.enqueue(2) // Queue(List(2,1),List())
```

Dequeue

- Si out est vide : On inverse la liste in ou on récupère ensuite la tête et le reste de la liste, puis on retourne la tête ainsi qu'une nouvelle queue ou in vaut Nil et out le reste de la liste.
- Sinon : on renvoie la tête de la liste out et on renvoie une nouvelle queue avec in étant notre liste in courante et out le reste de la liste de out sans la tête qui a été retiré.

```
val queue1 = Queue[Int](List(2,3,5,4), Nil)
val queue2 = queue1.dequeue // (4,Queue(Nil,List(5,3,2))
```

headOption

Retourne * Option.empty si les deux listes sont vides * le dernier élément de out si in est vide * la tête de in sinon

```
val queue1 = Queue[Int](List(2,3,5,4), Nil)
val queue2 = queue1.headOption // 2
```

2. Implantez une méthode length qui retourne la taille de le queue.

Retourne la somme de la longueur de la liste in et la longueur de la liste out

```
val queue1 = Queue[Int](List(2,3,5,4), List(7,6,4))
val queue2 = queue1.length // 7
```

3. Implantez une méthode rearOption qui retourne le dernier élément de la queue (celui inséré en premier) sans la modifier.

Similaire à headOption, juste inversé

```
val queue1 = Queue[Int](List(2,3,5,4), Nil)
val queue2 = queue1.rearOption // 2
```

4. Implantez une méthode toList qui convertit la Queue en liste simplement chaînée.

On retourne la liste in concaténée à la liste out mais inversé

```
val queue1 = Queue[Int](List(2,3,5,4), List(7,6,4))
val queue2 = queue1.toList // List(2,3,5,4,4,6,7)
```

5. Implantez une méthode map sur Queue.

On cré une nouvelle queue en passant in que l'on map grâce à la méthode map de la classe List et on fait pareil pour la liste out.

```
val queue1 = Queue[Int](List(2,3,5,4), List(7,6,4))
val queue2 = queue1.map(a => a*2) // List(4,6,10,8,8,12,14)
```

6. Implantez une méthode foldLeft sur Queue.

Pour cette méthode j'ai utilisé les méthodes foldLeft et foldRight de la classe List de Scala. On fait un foldLeft sur la liste in, que l'on passe en paramètre du foldRight de la liste out. On utilise un foldRight sur la liste out car la liste est inversé.

```
val queue1 = Queue[Int](List(2,3,5,4), List(7,6,4))
val queue2 = queue1.foldLeft(2)((a,b) => a+b) // 33
```

7. La méthode dequeue est partielle. Corrigez-la.

On retourne un Option[T] au lieu d'un [T] ce qui nous permet aussi de traité le cas ou la queue est vide

8. Ré-implantez isEmpty avec match.

On effectue un match sur les deux listes, si les deux valent Nil alors on retourne true, sinon dans tous les autres cas on renvoie false