Feuille 8 - Files de Priorité

1 File de priorité

Considérez les interfaces suivantes :

```
1 import java.util.*;
2 interface myCollection<T> {
3         void add(T t);
4         boolean isMember(T t);
5 }
6 interface myPQ<T extends Comparable<T>> extends myCollection<T> {
7         T getFirst();
8 }
```

L'interface `myCollection` n'est qu'une version très (trop ?) simplifiée de l'interface `Collection` de Java. La fonctionnalité décrite dans cette interface dit simplement qu'on a le droit d'ajouter un élément à la Collection, et de tester si un élément y appartient.

L'interface `myPQ` ("my Priority Queue") étend l'interface `myCollection`. Elle est censée représenter une structure de données appelée une file de priorité. Alors que pour une file classique la méthode `get` nous retourne les plus ancien élément de la structure, pour une file de priorité la méthode `getFirst` nous retourne l'élément de plus grande priorité. Pour decider quel est cet élément, on suppose que les éléments qui sont contenus dans la file sont comparables – notez bien qu'on a imposé cette condition en utilisant la généricité en disant que `T extends Comparable<T>`. Puisque les éléments du type `T` qui sont inserés dans la file implémentent l'interface `Comparable<T>` on peut supposer qu'il existe une méthode `public int compareTo(T t)` dans la classe `T`. Quand on applique cette méthode dans deux objets de cette classe, par example comme `t1.compareTo(t2)`, elle retourne un entier negatif ssi `t1` est plus petit que `t2`. (Regardez la documentation de l'interface `Comparable`). Étant donné cette comparabilité, la file de priorité retourne, avec la méthode `getFirst`, l'objet qui paraît le plus grande de la collection.

2 Implémentation

Donnez une classe qui implémente l'interface des files de priorité. Commencez avec la définition ci-dessous.

```
1 class myPQArray<T extends Comparable<T>> implements myPQ<T> {
2         List<T> data;
3         public myPQArray(){ ... }
4         public void add(T t){ ... }
5         public boolean isMember(T t){ ... }
6         public T getFirst(){ ... }
```

```
7           ...
8 }
```

Questions à considérer :

- Peut-on utiliser un tableau de `T` à la place de la liste `data` ? Pourquoi ?
- Comment peut-on initialiser l'attribut `data` dans le constructeur, vu que son type est une interface ?
- Peut-on programmer les méthodes sans connaitre le type réel de `data` ? Quelles sont les méthodes de l'interface `List` dont on a besoin ?
  - Rappel : `List` représente une collection ordonnée. On a les méthodes : `T get(int index)` qui retourne l'élément à position `index`, `T set(int index, T t)` qui remplace l'élément à position `index` avec l'élément `t`, et la méthode `T remove(int index)` qui supprime l'élément à position `index`.

Donnez une implémentation directe et simple de la classe en utilisant les méthodes de l'interface `List`.

3 Efficacité

Supposons qu'on a le programme suivant :

```
1 public static void main(String [] args){
2         myPQ<Integer> pq = new myPQArray<>();
3         for(int i=0; i<100000; i++) pq.add(i);
4         for(int i=0; i<100000; i++)
5                 System.out.println("Removed:"+pq.getFirst());
6 }
```

- Va-t-il compiler ?
- Que va-t-il afficher si on implémente la classe avec une `ArrayList` ? Avec une `LinkedList` ?
- Quel est l'impact de notre choix sur la pérformance de ce programme ?

Pour répondre à la dernière question, il faut penser à la pérformance des deux classes qui implémentent `List`. Selon la documentation de Java, pour `ArrayList` : "The ... get, set, ... operations run in constant time. The add operation runs in amortized constant time, that is, adding n elements requires O(n) time. All of the other operations run in linear time (roughly speaking). " En ce qui concerne `LinkedList` la documentation nous dit que "All of the operations perform as could be expected for a doubly-linked list". (Qui va dire quoi ? ?)

Quel est le bon choix ?

- Pour une `ArrayList` la classe maintient un tableau dynamique d'éléments. Ainsi, accéder aux éléments est une opération de coût constant. Le problème est que c'est difficile d'insérer un élément au milieu du tableau (il faut tout déplacer). En plus, ça peut paraître difficile d'élargir le tableau, parce que normalement il faut allouer un nouveau tableau. La classe résout ce problème en gardant un tableau plus grand que le tableau necessaire. Par conséquent, one n'est pas obligé de re-allouer le tableau pour chaque insertion, mais seulement périodiquement.
- Pour une `LinkedList` la classe maintient une liste (doublement) chaînée. Par conséquent, c'est rapide de trouver les premiers (ou derniers) éléments, et d'inserer un nouveau élément, mais dans le pire de cas, pour trouver un élément quelconque il faut parcourir toute la liste.