TP 6: Structures de Données

1 Trouver les Doublons

Pour cet exercice vous devez écrire une fonction avec la signature suivante:

```
public static <T> void findDoubles(List<T> data)
```

Cette fonction prend comme paramètre une liste. Elle parcourt la liste est affiche un message chaque fois qu'elle trouve un doublon, c'est-à-dire un élément qui existe déjà dans la liste. Par exemple, si on donne comme paramètre la liste qui contient les entiers $[0, 1, 2, 3, 1, 2, 1]$ cette fonction doit afficher (avec un `System.out.println`) un message disant que l'élément à position 4 est égale avec l'élément à position 1, et idem pour les positions 5 et 2, et 6 et 1.

Spécifications:

- Votre fonction doit être générique est fonctionner correctement pour n'importe quel type `T`.
- Donnez une implémentation efficace. Il est très facile de donner une implémentation avec deux boucles `for` qui fait toutes les comparaisons entre deux éléments de la liste `data`. **Il faut éviter une telle solution naïve !** Plus généralement, il faut éviter d'utiliser un algorithme qui prend de temps n^2, pour une liste de n éléments.

Comment faire: Le but de cet exercice est de se familiariser avec quelques structures de données disponibles dans la bibliothèque de Java. Les interfaces les plus pertinentes sont `Set` et `Map`, puisque la fonctionnalité des ces interfaces nous permet d'éviter les doublons. Regarder la documentation de Java pour les utiliser. Vous allez constater qu'il vous faudra utiliser une classe qui implémente une de ces interfaces, par exemple la classe `TreeMap` ou la classe `HashMap`. Pour utiliser la classe `TreeMap` il faut que les éléments qui seront inserés dans l'objet `TreeMap` soient comparables. Pour utiliser la classe `HashMap` il faut que la classe `T` donne une redéfinition de la méthode `hashCode`.

Applications: Pour montrer que votre implémentation est correcte donnez deux programmes **qui utilisent la même fonction `findDoubles` sans modification**.

Le premier programme prend comme paramètre sur la ligne de commande le nom d'un fichier qui contient des entiers, un entier par ligne, et affiche tous le doublons du fichier. Le but de cette application est de montrer la performance de votre programme. Vous pouvez utiliser comme entrée le fichier `test.txt` qui se trouve sur la page web.

Le deuxième programme prend comme paramètre sur la ligne de commande les noms de quelques fichiers ppm et trouve, parmi eux, les fichiers qui décrivent la même image. **NB:** votre programme doit être capable de détecter des doublons même si les fichiers ne sont pas identiques (par exemple si on insère des espaces dans un fichier). Exemple d'exécution :

```
$ cp cat.ppm cat2.ppm
$ cp cat.ppm cat3.ppm
$ cp catX.ppm catX2.ppm
$ java Test cat.ppm catX.ppm cat2.ppm catX2.ppm cat3.ppm
Element 2 is the same as Element 0
Element 3 is the same as Element 1
Element 4 is the same as Element 0
```

2 Median Queue

Le but de cet exercice est de programmer une file de priorité où l'élément retourné chaque fois qu'on appele la méthode `pop` est l'élément **médiane**. On commence avec l'interface suivante:

```
1 interface myQueue<T extends Comparable <? super T>> {
2         void push(T t);
3         T top();
4         T pop();
5 }
```

Les spécifications de ces méthodes sont :

- La méthode `push` ajoute un élément dans la file (les doublons sont permis).
- La méthode `top` retourne l'élément médiane de la file, c'est-à-dire, l'élément qui est supérieur ou égale à `size/2` autres éléments de la file. Cette méthode ne modifie pas l'état de la file.
- La méthode `pop` retourne le même élément que `top`, et le supprime de la file.

2.1 Implémentation Facile

Donnez une classe qui implémente l'interface `myQueue<T>` en utilisant une `ArrayList`. Votre classe va maintenir une liste triée de tous les éléments qui ont été inserés dans la file. Il vous sera utile d'utiliser les méthodes suivantes de l'interface `List`

- `boolean add(int index)`: qui permet d'insèrer un élément au milieu du tableau.
- `T remove(int index)`: qui permet de supprimer un élément de la liste.
- `T get(int index)`: qui retourne l'élément à position `index`.

Donner un programme qui utilise votre classe. Votre programme doit prendre comme paramètre (sur la ligne de commande) le nom d'un fichier qui contient des entiers, un entier par ligne. Le programme appelle `push` pour chaque entier du fichier et, une fois le fichier terminé, appelle `pop` sur la file et affiche le résultat jusqu'à ce que la file soit vide.

Exemple: si le fichier contient les nombres $0, 1, \ldots, 9$, ce programme doit afficher $5, 4, 6, 3, 7, 2, 8, 1, 9, 0$.

Pour tester votre programme vous pouvez aussi utiliser le fichier `testRand.txt` qui se trouve sur la page web. Ce fichier contient les entiers $0, \ldots, 199999$ en ordre aléatoire.

2.2 La bonne implémentation

Si vous avez tenté d'exécuter votre programme avec le fichier `testRand.txt`, vous avez sans doute constaté que sa performance n'est pas idéale. La raison est que chaque opération d'insértion/suppression prend $\Theta(n)$ pour une `ArrayList`. (Peut-on améliorer cette performance si on utilise une `LinkedList` à la place de `ArrayList`?). Le but de cette partie est de donner une implémentation plus performante qui utilise des arbres binaires de recherche. Considérez la classe suivante :

```
1 class BinSearchTree<T extends Comparable<? super T>> {
2         T data;
3         BinSearchTree<T> left, right, parent;
```

```
4         int size=1;
5         /* ... methods... ? */
6 }
```

Notez bien qu'on a demandé que la classe `T` soit un sous-type de `Comparable <? super T>`, c'est-à-dire, on a le droit de comparer un objet de type `T` avec un autre en utilisant la méthode `compareTo`. (Regardez la documentation de l'interface `Comparable`). Cette restriction est indispensable pour construire un arbre de recherche.

Un arbre binaire de recherche est une structure qui satisfait les conditions suivantes :

- Chaque noeud de l'arbre contient un élément de type `T` (`data`).
- Pour chaque noeud `v` de l'arbre, tous les éléments contenus dans l'arbre `v.left` sont inférieurs ou égaux à `v.data`. Tous les éléments contenus dans `v.right` sont supérieurs à `v.data`.
- On a `v.size = v.left.size + v.right.size + 1`, ou autrement dit, `v.size` est le nombre d'éléments de l'arbre.

Vous devez programmer (au moins) les méthodes suivantes :

1. `add(T t)` qui ajoute un élément dans l'arbre en respectant les conditions ci-dessus.
2. `find(int index)` qui retourne un élément de l'arbre qui est supérieur à `index` autres éléments. Autrement dit, cette méthode retourne l'élément qui serait dans la position `index` si on mettait tous les éléments de l'arbre dans un tableau trié. (`find(0)` retourne alors l'élément minimum, et `find(size/2)` l'élément mediane).

En utilisant cette base, programmez une classe qui implémente l'interface `myQueue<T>` et donne la fonctionnalité d'une file médiane.

Pour montrer que votre classe est beaucoup plus efficace que l'implémentation naïve de l'exercice 2.1, exécuter le même programme sur le fichier `testRand.txt`, mais avec la nouvelle classe à la place de la classe qui utilise les `ArrayList`. Pouvez-vous garantir que cette implémentation est toujours la meilleure ? Quelle est l'entrée qui représente le pire des cas pour cette classe ?