Cours de Programmation & Algorithmique II
1ère session 2014 — PARTIE 1

Consignes à lire impérativement !

L'examen est composé de **2 parties** et d'un total de **4 questions**. Chaque partie dure **2 heures**. Il vous est demandé de respecter les consignes suivantes.

- L'usage d'une calculatrice, d'un téléphone portable ou de notes de cours n'est **pas autorisé**, laissez-les dans votre sac. Pensez à éteindre votre téléphone portable !
- A l'issue de l'épreuve, rendez toutes les feuilles d'énoncé, les réponses ainsi que les feuilles de brouillon.
- Commencez par **écrire vos nom, prénom et section (math, info, ...) sur chaque feuille** que vous rendrez, y compris les feuilles de brouillon.
- Répondez à chacune des questions.
- Faites attention à la clarté et à l'organisation de vos réponses. Respectez les règles grammaticales et orthographiques.
- Veuillez utiliser pour vos réponses les cadres prévus à cet effet. Si une partie de votre réponse n'entre pas dans le cadre, utilisez le dos de la feuille ou une feuille supplémentaire et indiquez clairement à quelle question la réponse correspond.
- Vous devez terminer cette partie de l'examen avant de pouvoir sortir de la salle (pour aller à la toilette par exemple).

Question 1: Elément le plus fréquent (/4)

Soit un tableau dont les éléments sont des références vers des objets quelconques. L'objectif de cette question est d'implémenter une méthode en Java appelée `mostFrequent` qui détermine l'élément le plus fréquent du tableau, c.-à-d. celui qui apparaît le plus grand nombre de fois dans le tableau. Si plusieurs éléments correspondent à ce critère, l'algorithme retourne l'un d'entre eux.

La signature de la méthode `mostFrequent` est donnée à la Figure 1. Un exemple d'utilisation est également fourni. Un tableau de `Double` est initialisé avec les valeurs [5, 5, 1, 2, 5, 2]. La méthode `mostFrequent` retourne comme résultat la référence vers le premier élément (de valeur égale à 5).

Il vous est demandé de : Fournir une implémentation de la méthode `mostFrequent`. La méthode `mostFrequent` doit respecter la contrainte suivante : il est interdit d'utiliser une structure de données intermédiaire telle qu'un autre tableau, une liste ou une table de hachage. Il existe plusieurs façons d'implémenter `mostFrequent`. Pour information, il est possible, tout en respectant la contrainte ci-dessus, de trouver une solution de complexité $O(n.log(n) + n)$. Nous ne nous attendons pas à ce que vous trouviez une solution avec une telle complexité.

Attention ! La méthode prend en argument un tableau de références vers des objets et non d'éléments de type primitif. Par conséquent, le test d'égalité ne doit pas être fait n'importe comment !

```
public class MostFrequent
{

  public static <E> E mostFrequent(E [] array)
  {
    /* à implémenter */
  }

  public static void main(String [] args)
  {
    Double [] tab= { 5.0, 5.0, 1.0, 2.0, 5.0, 2.0 };
    System.out.println(mostfrequent(tab)); /* Affiche 5.0 */
  }

}
```

FIGURE 1 – Exemple d'utilisation de la méthode `mostFrequent`.

Q1 **(implémentation de la méthode `mostFrequent`)**

Question 2: Itérateur pour liste doublement chaînée (/6)

Soit une classe nommée `DoubleLinkedList` qui gère une séquence d'éléments à l'aide d'une liste doublement chaînée. Un extrait de l'implémentation de cette classe est fourni à la Figure 2. La classe `DoubleLinkedList` utilise une classe interne `Node` pour représenter les noeuds de la liste. Chaque instance de `Node` peut désigner un successeur et un prédécesseur avec les variables d'instance `next` et `prev` respectivement ainsi qu'une référence vers un élément de type `E` où `E` est le paramètre de type de la classe. Une instance de `DoubleLinkedList` maintient la référence des premier et dernier noeuds de la liste avec respectivement les variables `head` et `tail`. Ces variables valent `null` lorsque la liste est vide.

```
public class DoubleLinkedList<E>
{

  private class Node {
    Node next; /* successeur */
    Node prev; /* predecesseur */
    E    data; /* donnee associee */
  }

  private Node head; /* Debut de la liste */
  private Node tail; /* Fin de la liste */

  public ListIterator<E> iterator() { /* Implémentation non montrée */ }

  /* Implementation du restant de la classe non montrée */

}
```

FIGURE 2 – Extrait de la classe `DoubleLinkeList`.

L'objectif de cette question est de fournir pour la classe `DoubleLinkedList` une implémentation de l'interface `ListIterator`. Cette permet de se déplacer d'avant en arrière dans une collection, sans savoir comment cette collection est implémentée. Un extrait de la définition de `ListIterator` est donné à la Figure 3. Le déplacement en avant est réalisé avec `hasNext` et `next` tandis que le déplacement en arrière est réalisé avec `hasPrevious` et `previous`. Par ailleurs, la méthode `remove` de `ListIterator` permet également de supprimer l'élément qui a été retourné précédemment avec `next` ou `previous`. Un extrait de la définition de l'interface `ListIterator` est donné ci-dessous.

La Figure 4 donne un exemple d'utilisation d'un tel itérateur. Une instance de `DoubleLinkedList` est créée et on suppose que les éléments 17, 33 et 54 y sont ajoutés, dans cet ordre. L'ajout des éléments n'est pas montré dans l'exemple. Un itérateur est obtenu sur la liste à l'aide de la méthode `iterator`. Ensuite, plusieurs déplacements sont

```
public interface ListIterator<E> {
  boolean hasNext();
  boolean hasPrevious();
  E       next();
  E       previous();
  void    remove();
}
```

FIGURE 3 – Extrait de l'interface `ListIterator`.

effectués avec `next` et `previous`. Finalement, une suppression est effectuée avec `remove`. Dans l'exemple, l'élément supprimé est 17.

```
DoubleLinkedList<Integer> l= new DoubleLinkedList<Integer>();
/* Ajout des éléments 17, 33 et 54, dans cet ordre (non montré). */
ListIterator iter= l.iterator();
iter.next();
iter.next();
iter.previous();
iter.remove(); /* supprime l'élément 17 *.
```

FIGURE 4 – Exemple d'utilisation d'un itérateur avec `DoubleLinkedList`.

Il vous est demandé de : Fournir un classe `MyIterator` qui implémente l'interface `ListIterator` pour la classe `DoubleLinkedList`. La classe `MyIterator` est une classe interne de `DoubleLinkedList`. Vous ne devez implémenter que les méthodes `hasPrevious`, `previous`, `next` et `remove`. Si nécessaire, vous devez également implémenter un constructeur pour `MyIterator`.

Note : pour des raisons de lisibilité et de place, l'implémentation de la classe `MyIterator` ci-dessous ne doit pas être placée dans la classe `DoubleLinkedList` bien qu'elle en soit une classe interne. Considérez que tout le code écrit dans les cadres ci-dessous se trouve dans la classe `DoubleLinkedList`.

Q2 **(implémentation de la classe `MyIterator`)**

Q2 **(suite de l'implémentation de la classe `MyIterator`)**