Pratique de la programmation orientée-objet
Examen intermédiaire

9 avril 2014

Indications :
— l'examen dure de 13h15 à 15h00,
— indiquez votre nom, prénom et numéro SCIPER ci-dessous et sur toutes les éventuelles feuilles additionnelles que vous rendriez,
— placez votre carte d'étudiant sur la table.
Bon travail !

Nom : _____

Prénom : _____

SCIPER : _____

Itérateur de liste [10 points]

La classe `LinkedList` développée dans le cadre du mini-projet *collections* représente une liste doublement chaînée circulaire avec nœud d'en-tête. Sa définition est rappelée ci-dessous, sans la plupart des méthodes.

```
public final class LinkedList<E> implements List<E> {
  private int size;
  private final Node<E> head;

  public LinkedList() {
    Node<E> head = new Node<>(null);
    head.succ = head.pred = head;
    this.size = 0;
    this.head = head;
  }

  méthodes omises (isEmpty, size, add, etc.)

  public Iterator<E> reverseIterator() {
    return new Iterator<E>() { à faire };
  }
  private void removeNode(Node<E> n) { code omis }

  private static class Node<E> {
    public Node<E> succ, pred;
    public E value;
    constructeur omis
  }
}
```

Partie 1 [10 points] Ecrivez le corps de la méthode `reverseIterator` déclarée ci-dessus, qui retourne un itérateur parcourant les éléments de la liste dans l'ordre inverse, du dernier au premier. La classe de l'itérateur doit être une classe anonyme imbriquée dans la méthode `reverseIterator`.

Pour écrire la méthode `remove` de cet itérateur, vous pouvez vous aider de la méthode privée `removeNode`, qui supprime de la liste le nœud qu'on lui passe. Son code a été omis afin d'alléger la présentation.

Réponse :

(suite à la page suivante)

(suite de la page précédente)

Grands nombres [20 points]

Les types entiers en Java (**`int`**, **`long`**, etc.) ont tous une capacité limitée, c'est-à-dire qu'ils ne peuvent pas représenter des valeurs arbitrairement grandes. Ainsi, le plus grand d'entre eux, **`long`**, ne permet pas de représenter de valeur positive supérieure à $2^{63} - 1$.

Pour contourner cette limitation, on peut représenter un entier par une séquence de chiffres dans une base choisie. Par exemple, l'entier 10^{19} est supérieur à 2^{63}, donc pas représentable directement au moyen d'un type entier de Java. On peut par contre le représenter par une séquence de 20 chiffres en base 10, sous la forme $1, 0, \ldots, 0$ (le chiffre 1 suivi de 19 fois le chiffre 0). Chacun de ces chiffres peut lui être représenté par un entier Java, p.ex. de type **`int`**.

La classe `BigInteger` ci-dessous, finale et immuable, représente un entier *positif* de taille arbitraire au moyen d'une liste de chiffres en base 10. Les chiffres sont stockés dans le champ `digits`, ordonnés par poids *croissant* — le premier chiffre de la liste est donc celui des unités — et sans 0 superflu. Par exemple l'entier 2014 est représenté par la liste de quatre chiffres $4, 1, 0, 2$, dans cet ordre. L'entier 0 est pour sa part représenté par une liste vide.

```
public final class BigInteger {
  private final List<Integer> digits;

  public BigInteger(List<Integer> digits) { à faire }
  public String toString() { à faire }
  public BigInteger add(BigInteger that) { à faire }
}
```

Partie 1 [6 points] Ecrivez le corps du constructeur, qui construit un grand nombre avec la séquence de chiffres passée en argument et ordonnée également par poids croissant. Ce constructeur lève l'exception `IllegalArgumentException` si la liste de chiffres qu'il reçoit contient un chiffre invalide (c-à-d hors de l'intervalle $[0; 9]$) ou si elle se termine par un 0.

N'oubliez pas que la classe `BigInteger` doit être immuable !

Réponse :

Partie 2 [5 points] Ecrivez le corps de la méthode `toString`, qui retourne la représentation textuelle du nombre en base 10. Cette représentation ne doit pas comporter de zéro initial. Par exemple, la représentation textuelle du nombre 2014 est la chaîne `2014` et aucune autre. La seule exception à cette règle est bien entendu l'entier 0 lui-même, dont la représentation textuelle est la chaîne `0`.

Réponse :

Partie 3 [9 points] Ecrivez le corps de la méthode `add`, qui retourne la somme du nombre auquel on l'applique et du nombre passé en argument.

Réponse :

Multi-ensemble [15 points]

Les *multi-ensembles* sont une généralisation des ensembles dans lesquels un élément peut apparaître plus d'une fois. L'interface `MultiSet` ci-dessous, quasiment identique à l'interface `Set` du cours, décrit un multi-ensemble.

```
public interface MultiSet<E> extends Iterable<E> {
  public boolean isEmpty();
  public int size();
  public void add(E newElem);
  public void remove(E elem);
  public int count(E elem);
  public Iterator<E> iterator();
}
```

Cette interface diffère de l'interface `Set` du cours en un point : la méthode `contains` est remplacée par la méthode `count`, qui retourne le nombre d'occurrences de l'élément passé en argument dans le multi-ensemble.

Un multi-ensemble peut être représenté par une table associative qui associe un nombre d'occurrences à chaque élément. Par exemple, le multi-ensemble de chaînes de caractères contenant les éléments { `"a"`, `"a"`, `"b"`, `"c"`, `"c"`, `"c"` } peut être représenté par une table associative associant l'entier 2 à la chaîne `"a"`, l'entier 1 à la chaîne `"b"` et l'entier 3 à la chaîne `"c"`.

La classe `MultiHashSet` ci-dessous met en œuvre un multi-ensemble en stockant le nombre d'occurrences de chaque élément dans une table associative.

```
public class MultiHashSet<E> implements MultiSet<E> {
  private int size = 0;
  private final Map<E, Integer> counts = new HashMap<>();

  public boolean isEmpty() { return size == 0; }
  public int size() { return size; }
  à compléter
}
```

Afin de ne pas gaspiller de la mémoire, la table associative contenant le nombre d'occurrences des éléments du multi-ensemble (`counts` ci-dessus) ne doit contenir que les éléments dont le nombre d'occurrences est *supérieur* à 0.

Partie 1 [2 points] Ecrivez la méthode `count`, qui retourne le nombre d'occurrences dans le multi-ensemble de l'élément passé en argument.

Réponse :

Partie 2 [3 points] Ecrivez la méthode `add`, qui ajoute au multi-ensemble une occurrence de l'élément passé en argument.

Réponse :

Partie 3 [4 points] Ecrivez la méthode `remove`, qui supprime du multi-ensemble *une* occurrence de l'élément passé en argument, ou ne fait rien si aucune occurrence ne s'y trouve.

Cette méthode doit garantir que lorsque le nombre d'occurrences d'un élément est nul, il n'apparaît pas dans la table associative.

Réponse :

(Suite à la page suivante)

Partie 4 [6 points] Ecrivez la méthode `iterator`, qui retourne un itérateur sur les éléments du multi-ensemble. La méthode `remove` de cet itérateur doit simplement lever l'exception `UnsupportedOperationException`.

Les éléments peuvent être produits dans un ordre quelconque, mais le nombre d'occurrence doit bien entendu être respecté, c-à-d qu'un élément apparaissant n fois dans le multi-ensemble doit être produit n fois par l'itérateur.

Réponse :

Formulaire

Ce formulaire présente toutes les parties de la bibliothèque standard Java dont vous avez besoin pour cet examen. Notez que de nombreuses méthodes inutiles ont été omises afin de ne pas encombrer la présentation.

Interface `List`

L'interface `java.util.List` représente les listes. Elle est implémentée, entre autres, par les classes `LinkedList` (listes chaînées) et `ArrayList` (tableaux-listes).

```
interface List<E> extends Iterable<E> {
  // Ajoute l'élément e à la fin de la liste et retourne vrai.
  boolean add(E e);

  // Retourne un itérateur sur les éléments de la liste.
  Iterator<E> iterator();
}
```

Les classes qui implémentent cette interface offrent toutes un constructeur de copie (c-à-d un constructeur qui prend une valeur de type `List<E>` en argument et l'utilise pour initialiser la liste construite).

Interface `Map`

L'interface `java.util.Map` représente les tables associatives. Elle est implémentée, entre autres, par la classe `HashMap`.

```
interface Map<K, V> {
  // Associe la valeur value à la clef key. Retourne la valeur qui était associée
  // à la clef, ou null s'il n'y en avait pas.
  V put(K key, V value);

  // Supprime la valeur associée à key. Retourne la valeur qui était associée
  // à la clef, ou null s'il n'y en avait pas.
  V remove(K key);

  // Retourne la valeur associée à key, ou null s'il n'y en a aucune.
  V get(K key);

  // Retourne vrai si et seulement si la table contient la clef key.
  boolean containsKey(K key);

  // Retourne l'ensemble des clefs de la table.
  Set<K> keySet();
}
```

Les classes qui implémentent cette interface offrent toutes un constructeur de copie (c-à-d un constructeur qui prend une valeur de type `Map<K, V>` en argument et l'utilise pour initialiser la table associative construite).

Interface `Iterable`

L'interface `java.lang.Iterable` représente les objets itérables, c-à-d ceux dont le contenu peut être parcouru au moyen d'un itérateur ou de la boucle *for-each*.

```
interface Iterable<E> {
  // Retourne un itérateur sur les éléments de l'objet.
  Iterator<E> iterator();
}
```

Interface `Iterator`

L'interface `java.util.Iterator` représente les itérateurs.

```
interface Iterator<E> {
  // Retourne vrai ssi cet itérateur peut encore livrer des éléments.
  boolean hasNext();

  // Retourne le prochain élément, ou lève l'exception
  // NoSuchElementException s'il n'y en a plus.
  E next();

  // Supprime la valeur retournée par le dernier appel à next, ou lève
  // l'exception IllegalStateException si next n'a pas encore été
  // appelée, ou si remove est appelée deux fois de suite.
  void remove();
}
```

Interface `StringBuilder`

La classe `java.lang.StringBuilder` est un bâtisseur pour les chaînes de caractères. Elle permet de construire petit-à-petit de telles chaînes.

```
class StringBuilder {
  // Ajoute la représentation textuelle de l'entier passé à la chaîne en cours
  // de construction et retourne this.
  StringBuilder append(int i);

  // Ajoute la représentation textuelle de l'objet passé à la chaîne en cours
  // de construction et retourne this.
  StringBuilder append(Object o);

  // Inverse la chaîne en cours de construction et retourne this.
  StringBuilder reverse();

  // Retourne une nouvelle chaîne avec le contenu ajouté jusqu'à présent.
  String toString();
}
```