Examen intermédiaire

CS-108

2026-04-17

1 Ensembles d'entiers empaquetés [25 points]

La classe `PkIntSet64` plus bas contient des méthodes permettant de manipuler un ensemble d'entiers compris entre 0 (inclus) et 64 (exclu), empaqueté dans une valeur de type `long`.

L'idée de cette représentation est que si le bit d'index i de la valeur de type `long` vaut 1, alors l'entier i appartient à l'ensemble, sinon il n'y appartient pas. Par exemple, l'ensemble $\{2, 3, 5, 7, 11\}$ est représenté par la valeur de type `long` valant `0b100010101100`.

```
public final class PkIntSet64 {
  public static final long EMPTY = 0L;

  public static long of(Iterable<Integer> elements) { ... }
  public static int size(long pkIntSet) { ... }
  public static long add(long pkIntSet, int i) {
    return pkIntSet | (1L << Objects.checkIndex(i, Long.SIZE));
  }
  public static void forEachIndexed(long pkIntSet,
                                    IntBiConsumer consumer) { ... }
  public static int[] toArray(long pkIntSet) { ... }
}
```

Partie 1 [6 points] Écrivez le corps de la méthode `of` qui retourne l'ensemble d'entiers empaqueté contenant les mêmes éléments que ceux fournis par la valeur itérable donnée, ou qui lève une `IndexOutOfBoundsException` si l'un de ces éléments n'est pas compris entre 0 (inclus) et 64 (exclu).

Pour écrire votre méthode `of`, vous pouvez utiliser la méthode `add` qui retourne un ensemble empaqueté identique à celui qu'on lui a donné, mais qui inclut l'élément `i`, ou lève une `IndexOutOfBoundsException` si cet élément n'est pas compris entre 0 (inclus) et 64 (exclu).

Souvenez-vous qu'en Java, les valeurs de type `Iterable<...>` sont celles que l'on peut parcourir au moyen d'un itérateur ou de la boucle *for-each*.

Partie 2 [4 points] Écrivez le corps de la méthode `size`, qui retourne la taille de l'ensemble d'entiers empaqueté donné, c.-à-d. le nombre d'entiers qu'il contient.

Partie 3 [8 points] Soit l'interface fonctionnelle `IntBiConsumer` suivante :

```
@FunctionalInterface
public interface IntBiConsumer { void accept(int i1, int i2); }
```

Écrivez le corps de la méthode `forEachIndexed`, qui appelle la méthode `accept` du consommateur donné avec chacun des éléments de l'ensemble, du plus petit au plus grand. En plus de

l'élément, le consommateur reçoit, en *premier* argument, l'index de cet élément, qui va de 0 (inclus) à la taille de l'ensemble (exclue).

Le test JUnit ci-dessous, qui doit s'exécuter avec succès, illustre le fonctionnement de cette méthode.

```
long pkIntSet = PkIntSet64.of(Set.of(20, 26, 4, 17, 10, 15));
StringJoiner j = new StringJoiner(",", "[", "]");
PkIntSet64.forEachIndexed(pkIntSet, (i, e) -> j.add(i + ":" + e));
assertEquals("[0:4,1:10,2:15,3:17,4:20,5:26]", j.toString());
```

Comme on le voit, `forEachIndexed` appelle d'abord la méthode `accept` avec 0 (l'index du plus petit élément) et 4 (le plus petit élément lui-même), puis avec 1 (l'index du second élément) et 10 (cet élément lui-même), et ainsi de suite.

Partie 4 [7 points] Écrivez le corps de la méthode `toArray` qui retourne un tableau contenant les éléments de l'ensemble d'entiers empaqueté donné, triés par ordre croissant. Votre mise en œuvre doit obligatoirement utiliser `forEachIndexed` pour initialiser les éléments du tableau. Le test JUnit ci-dessous, qui doit s'exécuter avec succès, illustre son fonctionnement.

```
long pkIntSet = PkIntSet64.of(Set.of(20, 26, 4, 17, 10, 15));
int[] expected = {4, 10, 15, 17, 20, 26};
assertArrayEquals(expected, PkIntSet64.toArray(pkIntSet));
```

2 Listes immuables [25 points]

L'interface `ImmuList` ci-dessous représente une liste immuable simple, contenant des valeurs de type `E`. La méthode `size` retourne la taille de la liste, c.-à-d. le nombre d'éléments qu'elle contient. La méthode `get` retourne l'élément à l'index donné, ou lève une exception de type `IndexOutOfBoundsException` si cet index n'est pas compris entre 0 (inclus) et la taille de la liste (exclue).

```
public interface ImmuList<E> {
  int size();
  E get(int i);
}
```

Le but de cet exercice est d'écrire plusieurs classes qui implémentent cette interface et représentent différents types de listes. **Attention : vous devez les écrire sans utiliser les collections de la bibliothèque Java !**

Partie 1 [7 points] Écrivez une classe nommée `AbstractImmuList`, abstraite et générique, qui implémente l'interface `ImmuList` mais ne possède aucune autre méthode qu'une redéfinition de la méthode `toString` de `Object`.

Cette méthode retourne la représentation textuelle de la liste, composée des représentations textuelles de chacun des éléments, séparées par des virgules, et entourées de crochets (`[]`). La représentation textuelle d'un élément s'obtient au moyen de sa méthode `toString` à lui.

Le test JUnit ci-dessous, qui doit s'exécuter avec succès, illustre le fonctionnement de cette méthode. Il dépend de la classe `OfArray`, qui hérite de `AbstractImmuList` et est définie plus bas.

```
ImmuList<String> l1 = new OfArray<>(new String[]{});
ImmuList<String> l2 = new OfArray<>(new String[]{"a", "b", "c"});
assertEquals("[]", l1.toString());
assertEquals("[a,b,c]", l2.toString());
```

Partie 2 [6 points] Écrivez une classe nommée `OfArray`, finale et générique, qui hérite de `AbstractImmuList` et permette de construire une liste immuable ayant les mêmes éléments que le tableau passé à son constructeur, comme illustré par le test JUnit de la partie précédente.

Prenez garde au fait que votre classe doit être immuable !

Partie 3 [6 points] Écrivez une classe nommée `Reversed`, finale et générique, qui hérite de `AbstractImmuList` et permette d'obtenir une liste ayant les mêmes éléments qu'une autre, mais dans l'ordre inverse. Le test JUnit ci-dessous, qui doit s'exécuter avec succès, illustre son fonctionnement.

```
ImmuList<String> l1 = new OfArray<>(new String[]{"a", "b", "c"});
ImmuList<String> l1r = new Reversed<>(l1);
assertEquals("[c,b,a]", l1r.toString());
```

Partie 4 [6 points] Écrivez une classe nommée `Concat`, finale et générique, qui hérite de `AbstractImmuList` et permette de construire une liste qui soit la concaténation de deux listes. Le test JUnit ci-dessous, qui doit s'exécuter avec succès, illustre son fonctionnement.

```
ImmuList<String> l1 = new OfArray<>(new String[]{"a", "b"});
ImmuList<String> l2 = new OfArray<>(new String[]{"c", "d"});
ImmuList<String> l12 = new Concat<>(l1, l2);
assertEquals("[a,b,c,d]", l12.toString());
```

3 L-systèmes stochastiques [25 points]

Les L-systèmes de la série d'exercices qui leur était consacrée étaient déterministes, dans le sens où une règle transformait toujours un caractère donné en la même chaîne. Par exemple, le L-système dessinant un arbre comportait la règle suivante :

$$\texttt{F} \rightarrow \texttt{FF+[+F-F-F]-[-F+F+F]}$$

qui récrit tous les caractères `F` dans la chaîne de départ en `FF+[+F-F-F]-[-F+F+F]`.

Pour obtenir des dessins plus intéressants et moins réguliers, il est possible d'utiliser des L-systèmes dits stochastiques (ou probabilistes), dans lesquels une règle peut transformer un caractère en différentes chaînes, selon certaines probabilités. Par exemple, le L-système stochastique ci-dessous peut récrire le caractère `F` en trois chaînes différentes. La probabilité que ce caractère soit récrit en chacune des trois chaînes est donnée au-dessus de la flèche.

$$\texttt{F} \xrightarrow{0.90} \texttt{FF+[+F-F-F]-[-F+F+F]}$$
$$\texttt{F} \xrightarrow{0.09} \texttt{F-[+F-F-F]+[-F+F+F]}$$
$$\texttt{F} \xrightarrow{0.01} \texttt{FFF+[+F-F-F]-[-F+F+F]}$$

Ainsi, avec ce L-système stochastique, la probabilité que le caractère `F` soit récrit en la première chaîne (`FF+...`) est de 0.90 (90%), celle qu'il soit récrit en la seconde chaîne (`F-...`) est de 0.09 (9%) et celle qu'il soit récrit en la troisième chaîne (`FFF+...`) est de 0.01 (1%).

Le but de cet exercice est de compléter l'enregistrement `LSystem` ci-dessous, qui représente un L-système stochastique. Cela se voit au fait que l'attribut `rules` a le type `Map<Character, List<Branch>>`, où `Branch` est un enregistrement imbriqué dans `LSystem` qui représente l'un des résultats possibles d'une règle, accompagné de sa probabilité.

```
public record LSystem(String string,
                      Map<Character, List<Branch>> rules,
                      Set<Character> lineChars,
                      int turningAngle) {
  public record Branch(double probability, String expansion) {}

  public LSystem { ... }
  private static List<Branch> normalize(List<Branch> branches) { ... }
  private String choose(List<Branch> branches,
                       RandomGenerator random) { ... }

  // ... autres méthodes (evolve, etc.)
}
```

Avec cette définition, la règle du L-système présenté plus haut, qui comporte trois branches, peut être définie ainsi :

```
Map<Character, List<LSystem.Branch>> rules = Map.of(
  'F', List.of(new LSystem.Branch(0.90, "FF+[+F-F-F]-[-F+F+F]"),
               new LSystem.Branch(0.09, "F-[+F-F-F]+[-F+F+F]"),
               new LSystem.Branch(0.01, "FFF+[+F-F-F]-[-F+F+F]")));
```

Partie 1 [8 points] Écrivez le corps de la méthode `normalize`, qui prend une liste de branches en argument et retourne une version non modifiable (!) et normalisée de cette liste, dans laquelle la somme des probabilités des branches vaut 1, ou lève une `IllegalArgumentException` si la liste est vide.

Par exemple, appliquée à la liste suivante, cette méthode doit retourner une liste identique à celle associée au caractère `F` ci-dessus.

```
List.of(new LSystem.Branch(90, "FF+[+F-F-F]-[-F+F+F]"),
        new LSystem.Branch( 9, "F-[+F-F-F]+[-F+F+F]"),
        new LSystem.Branch( 1, "FFF+[+F-F-F]-[-F+F+F]"))
```

Partie 2 [9 points] Écrivez le corps de la méthode `choose` qui utilise le générateur donné pour choisir aléatoirement une des branches de la liste donnée, en respectant leurs probabilités, et retourne la chaîne qui lui correspond. Vous pouvez faire l'hypothèse que la liste est normalisée et contient au moins une branche.

Suggestion : obtenez un nombre aléatoire compris entre 0 et 1 au moyen de la méthode `nextDouble` du générateur, et utilisez-le pour choisir la branche.

Partie 3 [8 points] Dans le corrigé de la série sur les L-systèmes, le constructeur compact de l'enregistrement `LSystem` utilise les méthodes `copyOf` de `List` et de `Map` pour garantir l'immuabilité de la classe, ainsi :

```
public LSystem {
  rules = Map.copyOf(rules);
  lineChars = Set.copyOf(lineChars);
}
```

Montrez comment le récrire pour qu'il normalise toutes les règles contenues dans `rules` tout en garantissant l'immuabilité de la classe.

4 Projet [36 points]

Veuillez répondre aux questions suivantes dans le cahier de réponse qui vous a été fourni, et pas sur cette feuille. Attention, chaque réponse fausse annule une réponse correcte.

A : Représentation non empaquetée

Dans le projet, nous représentons le contenu d'une source de tuiles de manière empaquetée. Parmi les types suivants, listez les numéros de ceux qui pourraient être utilisés pour représenter de manière non empaquetée le contenu d'une telle source :

1. `List<TileKind>`,
2. `Set<TileKind.Colored>`,
3. `ArrayList<TileKind.Colored>`,
4. `Map<TileKind, Integer>`,
5. `HashSet<TileKind>`,
6. `Map<TileSource.Factory, List<TileKind>>`,
7. `TileKind[]`.

Les types `List`, `ArrayList`, `Set`, `HashSet`, `Map` et `Integer` sont ceux de la bibliothèque Java, les autres sont ceux du projet.

B : Indexation du mur

Les cases du mur d'un joueur sont indexées de trois manières différentes dans le projet. Parmi les propositions suivantes, listez les numéros de celles qui font partie de ces trois manières d'indexer les cases du mur, et qui désignent une case valide :

10. ligne de motif `PATTERN_0`, colonne 3,
11. ligne de motif `PATTERN_2`, couleur E,
12. case d'index 25,
13. ligne de motif `PATTERN_1`, couleur M,
14. ligne 3, colonne 2, couleur C,
15. ligne de motif `PATTERN_2`,
16. case d'index 12,
17. ligne 2, colonne 1.

C : État des joueurs

Dans la version actuelle du projet, l'état d'un joueur est représenté de manière empaquetée au moyen de 4 entiers de type `int` : le premier contient les lignes de motif, le second la ligne plancher, le troisième le mur et le quatrième le nombre de points.

Pour économiser encore un peu de place, pourrait-on n'utiliser que les trois premiers entiers pour chaque joueur, et utiliser ensuite un unique entier `int` pour stocker les points de *tous* les joueurs ? Pour répondre, listez les numéros des réponses correctes parmi les suivantes :

20. non, car il n'y a pas assez de place dans un entier de type `int`,
21. oui, mais uniquement pour les parties à 2 ou 3 joueurs,
22. oui,
23. non, car la taille nécessaire dépend du nombre de joueurs.

(Suite à la page suivante)

D : Échantillonnage par réservoir

La méthode `sampleColoredInto` du projet prélève un échantillon d'un ensemble de tuiles colorées au moyen de la technique dite d'échantillonnage par réservoir. Admettons que l'on appelle cette méthode avec :

— l'ensemble de tuiles dont la représentation textuelle est `{3*A,4*C,5*E}`,
— un tableau (réservoir) de taille 5,
— un déplacement (`offset`) de 0.

Parmi les propositions suivantes, listez les numéros de celles qui représentent le contenu que le tableau passé à la méthode pourrait avoir après l'appel :

30. `[A,C,C,C,E]`,
31. `[A,E,A,C,A]`,
32. `[E,E,E,E,E]`,
33. `[A,A,A,A,C]`,
34. `[A,A,B,C,C]`,
35. `[C,C,C,E,C]`.

E : Sources uniques

Les quatre lignes ci-dessous montrent les tuiles qui se trouvent sur la zone centrale (0) et les 5 fabriques (1 à 5) d'une partie à 2 joueurs. Les tuiles colorées sont représentées par la lettre majuscule qui correspond à leur couleur, et le marqueur de premier joueur par la lettre `M`.

40. `0: M  1: AAAA  2: BBBB  3: AAAA  4: BCDD  5: BBBB`,
41. `0: ABBB  1: ABCD  2: ABBB  3:  4: BBCC  5: CDDE`,
42. `0:  1: ABEE  2: ABDE  3:  4:  5: ABBC`,
43. `0: M  1:  2:  3: ABCD  4: BCDE  5: EEEE`,
44. `0: CD  1: ABBC  2: ABBC  3: ABBC  4:  5: ABBC`,
45. `0: CCDD  1: CCDD  2:  3: ABCD  4: CCDD  5: DDDE`.

Chacune des valeurs ci-dessous est celle qui est retournée par `pkUniqueTileSources` de `ReadOnlyGameState` dans l'une des quatre configurations ci-dessus :

50. `0b100110`,
51. `0b000011`,
52. `0b101011`,
53. `0b110111`,
54. `0b111000`,
55. `0b010110`.

Associez à chacune des configurations des sources plus haut la valeur de retour qui lui correspond. Par exemple, si vous pensez que la première configuration correspond à la première valeur de retour, écrivez (40,50).

Formulaire

Ce formulaire présente toutes les parties de la bibliothèque Java nécessaires à cet examen. De nombreuses méthodes ont été omises et les types parfois simplifiés.

Interface `List`

L'interface `List` représente les listes. Elle est entre autres implémentée par la classe `ArrayList`, qui possède un constructeur de copie prenant une liste en argument.

```
public interface List<E> {
  // Retourne une copie immuable de l.
  static <E> List<E> copyOf(List<E> l);

  // Retourne la taille de la liste.
  int size();

  // Ajoute l'élément e à la fin de la liste et retourne true.
  boolean add(E e);

  // Retourne l'élément à l'index i, ou lève une
  // IndexOutOfBoundsException si l'index est invalide.
  E get(int index);
}
```

Interface `Map`

L'interface `Map` représente les tables associatives. Elle est implémentée, entre autres, par les classes `HashMap` et `TreeMap`. Ces classes possèdent un constructeur de copie prenant une table associative en argument.

```
public interface Map<K, V> {
  // Retourne une copie immuable de m.
  static <K, V> Map<K, V> copyOf(Map<K, V> m);

  // Associe la valeur v à la clef k.
  V put(K k, V v);

  // Retourne une vue sur l'ensemble des paires clef/valeur.
  Set<Map.Entry<K, V>> entrySet();
}
```

Interface `Map.Entry`

L'interface `Map.Entry` représente une paire clef/valeur.

```
public interface Map.Entry<K, V> {
  // Retourne la clef de la paire.
  K getKey();

  // Retourne la valeur de la paire.
  V getValue();
}
```

Classe `StringJoiner`

La classe `StringJoiner` représente un «joigneur» de chaîne, qui permet de construire des chaînes de caractères constituées d'une séquence de chaînes séparées par un séparateur et entourée d'un préfixe et d'un suffixe.

```
public class StringJoiner {
  // Construit un « joigneur » de chaînes avec le séparateur delimiter,
  // le préfixe prefix, et le suffixe suffix.
  StringJoiner(String delimiter, String prefix, String suffix);

  // Ajoute la chaîne string à la séquence.
  void add(String string);

  // Retourne la chaîne constituée de la concaténation du prefixe, des chaînes
  // ajoutées jusqu'à présent et séparées par le séparateur, et du suffixe.
  String toString();
}
```

Classe `Long`

La classe `java.lang.Long` contient entre autres des méthodes statiques travaillant sur les entiers de type `long`.

```
public class Long {
  // Retourne le nombre de bits valant 1 dans l.
  static int bitCount(long l);
}
```

Interface `RandomGenerator`

L'interface `RandomGenerator` représente un générateur de valeurs aléatoires.

```
public interface RandomGenerator {
  // Retourne une valeur aléatoire comprise entre 0 (inclus) et 1 (exclu).
  double nextDouble();
}
```