Programmation et modélisation orientées objet
(USAL5A & USAL2H)

Interface `Map` et classe `HashMap`

On souhaite implanter des listes d'association, encore appelées dictionnaires (ou *map* en anglais), c'est-à-dire une collection de couples (clé, valeur). Une telle collection est en fait un ensemble, car chaque clé peut apparaître au plus une fois (une autre clé « égale » ne peut pas être présente). Voici un exemple où les clés sont des entiers et les valeurs sont des chaînes de caractères :

```
[(72, "craie"), (31, "cheval"), (6, "arbre"), (22, "train"), (21, "pain")]
```

L'interface de la JDK pour ce genre de dictionnaire est `Map<K,V>` :

https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/Map.html

Il en existe deux implantations efficaces, une basée sur les arbres de recherche équilibrés (`TreeMap<K,V>`) et une basée sur le hachage (`HashMap<K,V>`). Ce TP est consacrée à l'implantation de `HashMap<K,V>`, mais nous commencerons par une implantation naïve où le dictionnaire est implanté par une simple liste de couples.

L'interface pour ce type de couples dans la JDK est `Map.Entry<K,V>` :

https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/Map.Entry.html

L'interface des listes dans la JDK est `List<E>` et elle est décrite ici :

https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/List.html

1 Utilisation de `Map<K,V>`

Utiliser la classe `HashMap<K,V>` de la JDK pour tester l'exemple ci-dessus. On pourra créer un package séparé pour ces tests (pour éviter les conflits de noms). Puis on testera les méthodes suivantes :

```
public int size()
public boolean isEmpty()
public boolean containsKey(Object)
public boolean containsValue(Object)
public V get(Object)
public V put(K, V)
public V remove(Object)
```

2 Implantation `Map.Entry<K,V>`

Créer une classe `SimpleEntry<K,V>` qui implante l'interface `Map.Entry<K,V>` (on pourra s'inspirer de la classe `Pair` déjà étudiée).

3 Implantation naïve : `ListMap<K,V>`

Implanter la classe `ListMap<K,V>` en étendant la classe `AbstractMap<K,V>` fournie, et où les entrées sont stockées dans une liste.

Cette implantation est naïve dans la mesure où l'accès à une entrée nécessite de parcours la liste (la moitié de la liste en moyenne).

Le constructeur ne prend pas d'argument et crée une `Map` vide. Les seules méthodes qui restent à implanter sont celles déjà énumérées ci-dessus.

```
public class ListMap<K, V> extends AbstractMap<K, V> {

    List<Entry<K,V>> entries;
    ...
}
```

Tester votre implantation en reprenant vos tests de la première question.

4 Implantation efficace : `HashMap<K,V>`

Pour rendre l'implantation plus efficace, on éclate la liste en n listes qui sont stockées dans une table (on utilisera ici `ArrayList` pour implanter la table). Pour savoir dans quelle liste est stockée un élément on utilise son code de hachage[1] *modulo* la taille du tableau (cette implantation est appelée le hachage « ouvert »).

Le code de hachage d'un objet est donné en Java par la méthode suivante, qui est disponible pour tout objet :

```
public int hashCode()
```

Considérons notre exemple simple, où les clés sont des entiers de type `Integer`. Dans ce cas, le code de hachage est l'entier lui-même. Supposons de plus que n vaut 10 (la fonction *modulo* 10 renvoie alors le dernier chiffre du code de hachage), la table correspondant à notre exemple peut alors être représentée ainsi :

```
0 ⟶ []
1 ⟶ [(31, "cheval"), (21, "pain")]
2 ⟶ [(72, "craie"), (22, "train")]
3 ⟶ []
4 ⟶ []
5 ⟶ []
6 ⟶ [(6, "arbre")]
7 ⟶ []
8 ⟶ []
9 ⟶ []
```

Implanter la classe `HashMap<K,V>` en étendant la classe `AbstractMap<K,V>` fournie, et où les entrées sont stockées dans un tableau de `ListMap<K,V>` de capacité donnée n (appelée `initialCapacity`).

Le constructeur prend en argument la capacité initiales et crée une `Map` vide (donc une table dont chaque case contient une liste vide). Les seules méthodes qui restent à implanter sont toujours celles énumérées ci-dessus.

```
public class HashMap<K, V> extends AbstractMap<K, V> {

    ArrayList<ListMap<K,V>> entries;
    ...
}
```

Tester votre implantation en reprenant vos tests de la première question.

1. La fonction de hachage (`hashCode`) peut être quelconque, mais elle doit toujours être compatible avec l'égalité (`equals`) : si deux objets sont égaux, ils doivent avoir même code de hachage. Pour être efficace, les codes de hachage doivent être, dans la mesure du possible, différents si les objets sont différents.