TP 8 : Les Collections en Java

christina.boura@uvsq.fr, stephane.lopes@uvsq.fr

8 avril 2016

Le contenu de ce TD fait partie d'un sujet de TD préparé par Luca Castelli Aleardi et traité dans le cadre du cours INF311, enseigné à l'École Polytechnique.

1 Introduction

Le but de ce TD est de fournir une implémentation efficace de certaines des fonctionnalités que l'on retrouve dans les navigateurs GPS actuellement disponibles dans le commerce. Notamment, on veut pouvoir interroger notre navigateur, en lui fournissant les coordonnées (latitude, longitude) de la position où nous nous trouvons, et on désire retrouver en temps réel (très rapidement, même pour des gros jeux de données) la ville la plus proche, parmi celles stockées dans la base de données du navigateur. En d'autres termes (plus informatiques), on veut implémenter un Dictionnaire, qui est une structure de données (abstraite) associant à un ensemble de clés un ensemble de valeurs, et qui est munie d'une opération de requête qui renvoie la valeur stockée correspondante à une clé donnée. Dans ce TD, les clés seront les coordonnées GPS et les valeurs seront les villes.

Structures de données abstraites et interfaces Java Il est utile de séparer la définition abstraite de la structure de données (définie par les opérations agissant sur les données à manipuler) de son implémentation concrète. Pour plus de modularité, on va se donner une interface pour le dictionnaire, et on va dans la suite proposer deux implémentations de cette dernière, dont on comparera les performances. La première implémentation sera basée sur une liste (`LinkedList`) et la deuxième sera basée sur une table de hachage.

2 L'interface `TableVille`

L'interface `TableVille` définit la structure de données abstraites d'un ensemble de villes. Elle est décrite dans le fichier `TablesVilles.java` que vous pouvez télécharger et observer. Cette interface décrit cinq méthodes :

- `public void ajouterVille(Ville v)` qui ajoute une ville dans l'ensemble.
- `public Ville rechercherVille(double latitude,double longitude)` qui cherche si une ville dont les coordonnées sont passées en paramètre est présente dans la liste.
- `public int nombreVilles()` qui renvoie le nombre de villes de l'ensemble.
- `public String infos()` qui donne des informations sur l'ensemble (par exemple la taille).
- `public LinkedList<Ville> traverser()` qui renvoie une liste contenant tous les elements de l'ensemble.

3 La classe `Ville`

Afin de représenter les villes, nous allons écrire une classe `Ville`. Cette classe aura trois attributs :

- `private String nom` : nom de la ville.
- `private double latitude` : latitude de la ville.
- `private double longitude` : longitude de la ville.

Elle doit avoir un constructeur

```
public Ville(String nom, double latitude, double longitude)
```

qui instancie un objet de type `Ville` en fournissant le nom de la ville ainsi que ses deux coordonnées géographiques.

Vous devez maintenant écrire les méthodes suivantes de la classe `Ville` :

- `public String toString()` qui retourne une chaîne de caractères représentant une ville.
- `public double getLatitude()` qui renvoie la latitude de la ville.
- `public double getLongitude()` qui renvoie la longitude de la ville.
- `public String getNom()` qui renvoie le nom de la ville.
- `public int compareTo(Ville v)` qui compare deux villes selon l'ordre lexicographique sur les couples (latitude, longitude). Elle doit renvoyer `-1` si notre élément est plus petit que celui passé en paramètre, `1` s'il est plus grand et `0` s'il y a égalité.
- `public int compareTo(double latitude, double longitude)` qui compare une ville à un couple de coordonnées (latitude, longitude). Cette méthode fonctionne de la même manière que la méthode juste au dessus.

4 Données à télécharger

On vous demande de télécharger le fichier `Fichiers.zip` (à sauvegarder dans le repertoire `src` avec les fichiers `.java` de votre projet). Le fichier `france_all.txt` contient les coordonnées et les noms de toutes les villes de la France. Les fichiers `france_echantillon.txt` et `france_petit_echantillon.txt` sont des sous-ensembles du fichier `france_all.txt`. Il s'agit des fichiers de texte, où chaque ligne correspond à une ville, et dont l'interprétation est très naturelle : le premier nombre décrit la latitude d'une ville (un réel compris entre `-90` et `90`), et un deuxième réel fournit sa longitude (entre `-180` et `180`) ; la ligne se termine par le nom de la ville (éventuellement un nom composé).

Comme toutes les villes appartiennent à la France métropolitaine (plus la Corse), leurs latitudes sont comprises entre `41.3833333` et `51.0833333`, et leurs longitudes entre `-5.1166667` et `9.5333333`.

```
42.9833333   -0.4   Aas
43.2833333   -0.0833333     Aast
50.2333333   3.2166667      Abacourt
48.5333333   5.5    Abainville
49.5833333   1.6333333      Abancourt
49.7         1.7666667      Abancourt
50.2333333   3.2166667      Abancourt
48.9         6.25   Abaucourt
49.2         5.5333333      Abaucourt
47.1333333   5.8833333      Abbans-Dessous
47.1166667   5.8833333      Abbans-Dessus
47.55        -1.5333333     Abbaretz
...
```

5 Une implémentation de l'interface à l'aide de listes chaînées

On vous demande ici de faire une première implémentation de l'interface `TableVille` à l'aide de listes chaînées. Vous devez écrire pour cela la classe `TableVillesListe` :

```
public class TableVillesListe implements TableVilles
```

Cette classe aura un seul attribut de type `LinkedList<Ville>`. Vous devez alors implementer toutes les méthodes dont le squelette est décrit par l'interface `TableVille`. En particulier, la méthode `infos()` doit retourner le nombre de villes de la liste. Le corps de chaque méthode ne devrait pas être très long ; une seule ligne en général, 4-5 lignes pour la méthode `rechercherVille`.

6 Tester votre implementation

Vous pouvez maintenant tester votre implémentation à l'aide de la classe `TestEnsembleVilles` que vous devez télécharger et compléter. Plus précisément, on vous demande de completer les deux méthodes suivantes :

— La méthode `remplir` qui prend en entrée le nom d'un fichier et la table à remplir. Cette méthode doit vous permettre de parcourir le fichier ligne par ligne, créer à chaque fois un nouvel objet `Ville` à partir des informations contenues sur cette ligne et l'insérer dans la table, à l'aide notamment de la méthode `ajouterVille` que vous avez déjà écrit. Afin de parcourir le fichier vous pouvez utiliser la classe `BufferedReader` et vous pouvez lire chaque ligne à l'aide de la méthode `readLine()`. Pour séparer chaque ligne en mots distincts vous pouvez utiliser la classe `StringTokenizer` :

```
StringTokenizer st = new StringTokenizer(uneLigne);
```

Pour récupérer un mot de cette ligne, vous pouvez utiliser la méthode `nextToken()`.

Attention : Certaines villes ont un nom composé de plusieurs mots (par exemple la ville *Les Brousses*). Pour tester ceci vous pouvez utiliser la méthode `hasMoreTokens()` et contatener le nouveau mot au nom de votre ville.

Pensez à gérer les exceptions qui pourraient être levées.

— La méthode `rechercherVille` qui doit chercher si une ville dont les coordonnées géographiques sont données en paramètre fait partie de la table qui est également passée en paramètre. Vous pouvez utiliser alors la méthode `rechercherVille` que vous avez déjà créé.

Si votre code marche bien, l'exécution de cette fonction doit afficher à l'écran un texte de la forme suivante :

```
Temps de calcul construction de l'ensemble:    153 millisecondes
44.2666667 4.1166667 Les Brousses
47.9833333 -4.3166667 Drevez
46.5833333 1.5166667 Argenton
Temps de calcul de la recherche : 7326 microsecondes

Informations sur le jeux de donnees:
Nombre de villes: 53465
Longueur: 53465
```

Pour tester avec un jeu de données différent, il suffit de modifier les commentaires dans la fonction `main`.

7 Tables de hachage

Avant de commencer à coder une table de hachage, nous rappelons quelques notions de base qui pourront nous être utiles : En général, lorsqu'on utilise du hachage il y a 3 paramètres à considérer :

— le nombre n d'objets à stocker ;

— un entier m qui détermine le nombre maximal de valeurs prises par la fonction de hachage $h(x)$ (la valeur $h(x)$ devant être comprise entre 0 et $m-1$)

— la taille N de l'univers U des clés.

Un aspect important concerne le choix de la fonction de hachage $h(x)$

— $h(x)$ doit permettre de traduire facilement un élément en un entier (son "empreinte" ou "haché")

— $h(x)$ doit transformer un ensemble creux de n éléments en un tableau "compact" de taille m (ainsi $m << N$)

Dans le cas d'un ensemble de villes nous avons :

— n est le nombre de villes

— m est la taille du tableau qu'on va allouer pour stocker les listes de villes ;

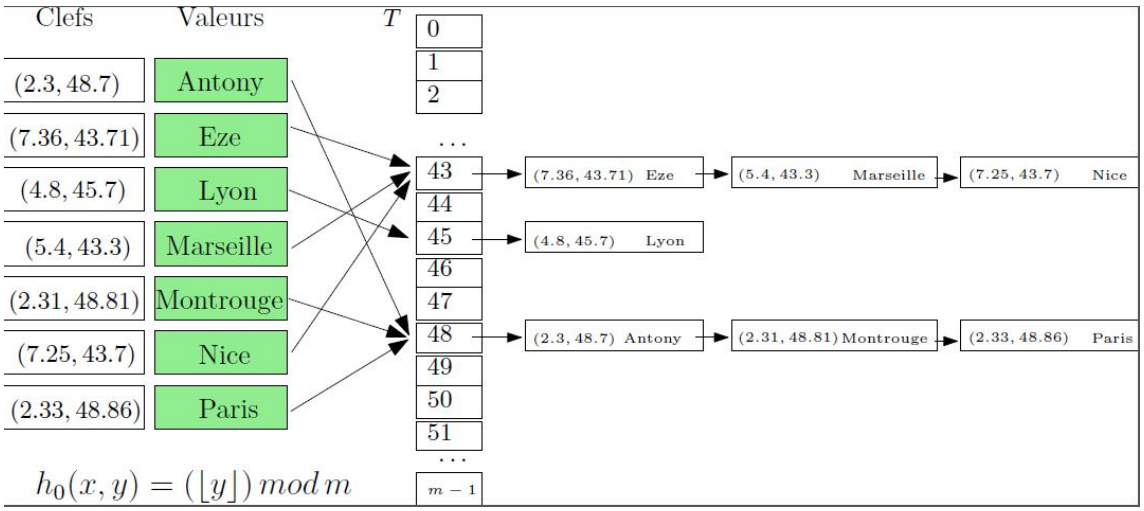
— dans notre cas les clés étant les coordonnées d'une ville (longitude et latitude), la taille de l'univers U dépendra donc des valeurs prises par la latitude et la longitude ; cela dépend donc des valeurs maximales et minimales des coordonnées, ainsi que de la précision avec laquelle on les exprime (le nombre de chiffres après la virgule).

7.1 Hacher les villes de Frances (de manière non efficace)

Rappelons que nous nous proposons ici de stocker des paires (clé, valeur), qui dans notre cas sont : ((longitude, latitude), ville). Pour résoudre le problème des collisions, nous allons ici adopter une solution basée sur des listes : on utilise un tableau pour référencer des listes de villes, chacune contenant des clés ayant le même code de hachage. Une première solution simple à mettre en place (fonction $h_0(x, y)$), consiste à considérer la valeur y de la latitude d'une ville, la discretiser (en prenant la partie entière inférieure), et en à calculer sa valeur modulo m (il faut s'assurer que le code d'hachage ait une valeur entre 0 et $m - 1$).

Quel est le désavantage de cette approche ? Peut-on faire mieux ?

Les exercices qui suivent fourniront une réponse (partielle) à ces questions.



7.2 D'autres fonctions de hachage plus efficaces

Dans la suite, nous voulons pouvoir facilement changer de fonction de hachage (et donc la décomposition de l'ensemble de villes). C'est pourquoi nous allons définir une interface `FonctionHachage` (contenant le code suivant, déjà fourni avec `Fichiers.zip`).

```
public interface FonctionHachage {

   // Fonction de hachage. Le troisieme parametre est la taille de la table de hachage

        public int hashFunction(double latitude, double longitude, int m);

}
```

La fonction d'hachage relative à l'exemple donné ci-dessus, pourrait se coder de la manière suivante :

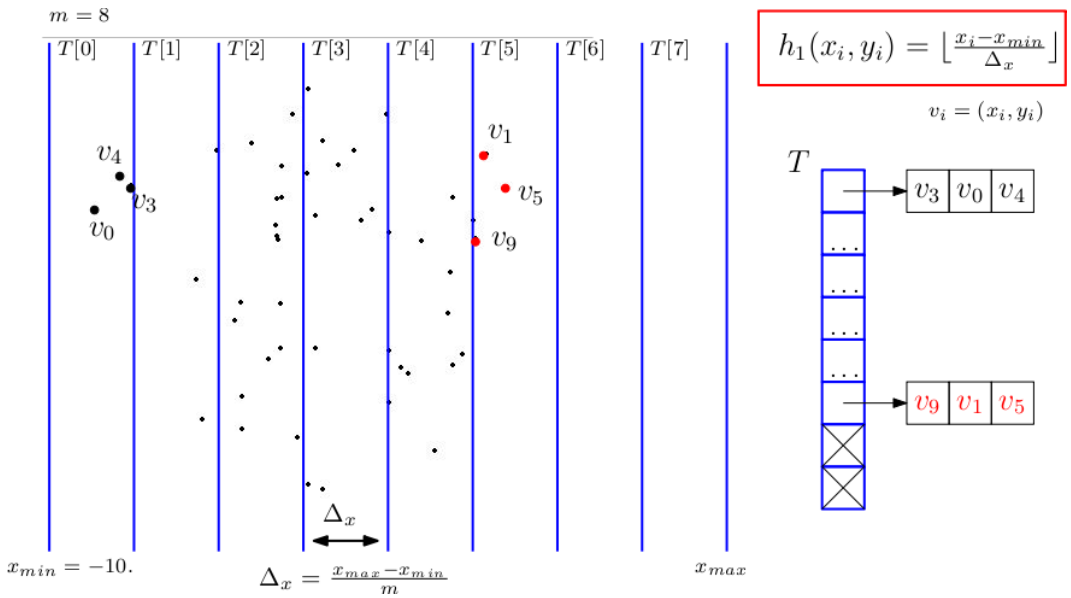
```java
public class H0 implements FonctionHachage {

    public int hashFunction(double latitude, double longitude, int m){
        return modulo((int)Math.floor(latitude), m);
    }

    public static int modulo(int x, int n) {
        if(x>=0) return x%n;
        else return (n-Math.abs(x%n))%n;
    }
}
```

Il n'y a pas une seule et meilleure manière de choisir une fonction de hachage. Nous vous proposons par exemple d'essayer avec deux autres fonctions de hachage correspondant aux décompositions ci-dessous.

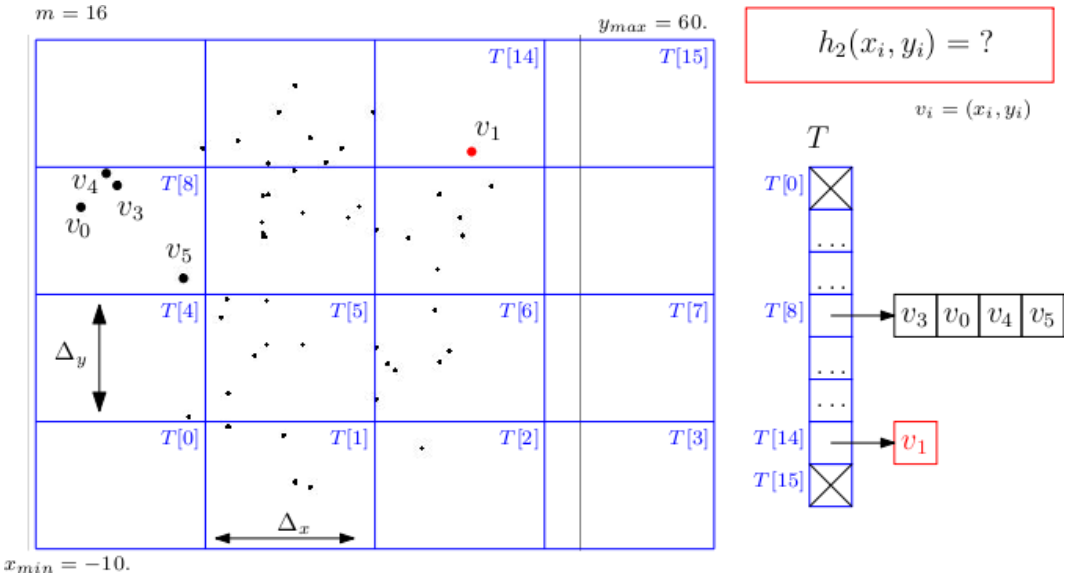
Remarque : Observez que la fonction de hachage (dans le cas de données géométriques), peut dépendre de plusieurs paramètres : le nombre de bandes verticales (paramètre m), ainsi que la taille de la boite englobante (qui est donné par x_{min} et x_{max}, dans le cas ci-dessous).



Complétez la classe `H1` qui fournit une première implémentation de l'interface `FonctionHachage`, correspondant à la fonction de hachage h_1 ci-dessus. Bien entendu, la valeur de $h_1(x, y)$ pour x, y donnés ne doit pas varier au fil des insertions et suppressions dans la table et les valeurs de x_{min} et x_{max} seront des constantes.

Remarque : Vous pouvez utiliser la fonction `static double Math.floor(double a)` pour calculer la partie entière d'un nombre réel.

Ecrivez une classe `H2` qui fournit une autre implémentation de l'interface `FonctionHachage`, codant la fonction de hachage h_2 qui correspond à la decomposition suivante (grille régulière) :



7.3 Ajout et recherche

Le but de cette section est de compléter la classe `TableVillesHachage` dont le squelette vous est déjà fourni.

- Commencez par compléter le constructeur. On rappelle que ce dernier prendra en argument un entier m (la taille du tableau utilisé dans la table de hachage) et une fonction de hachage `FonctionHachage hash`.
- Complétez les fonctions `ajouterVille` et `rechercherVille`.

Remarque : Dans votre implémentation `rechercherVille` pensez à utiliser la fonction `public Ville rechercherVille(double latitude, double longitude)` fournie dans `TableVillesListe`.

7.4 Informations sur la structure

Complétez maintenant les fonctions `remplissage`, `elementMaximal`, `nombreVilles` et `infos`.

Testez. La fonction `info` devra renvoyer une information sur la taille maximale des listes. En testant la fonction h_0 avec paramètre $m = 50$, on devrait avoir :

```
Temps de calcul construction de l'ensemble:    158 millisecondes
44.2666667 4.1166667 Les Brousses
47.9833333 -4.3166667 Drevez
46.5833333 1.5166667 Argenton
Temps de calcul de la recherche : 2850 microsecondes

Informations sur le jeux de donnees:
Nombre de villes: 53465
Nombre de cases utilisés: 11 / element maximal: Longueur: 10715
```

Que se passe-t-il lorsque on change avec d'autres valeurs du paramètre m ? Et si on fait appel aux fonctions h_1 et h_2 ?