TD3 : Tables de Hachage

L'efficacité des tables de hachage dépend de nombreux paramètres : la taille de la table, la fonction de hachage et la gestion des collisions. L'implémentation d'une telle structure est complexe, mais heureusement, le langage **Java** fournit plusieurs implémentations efficaces dans le framework `Collections`, notamment l'interface `Map` et sa classe principale `HashMap`.

Une `HashMap` permet d'associer des **clés** à des **valeurs**. Chaque clé est unique et permet d'accéder rapidement (en temps constant $O(1)$) à la valeur correspondante. Une association entre une clé et une valeur forme ce qu'on appelle une **entrée** (*entry* en anglais). Autrement dit, chaque entrée représente un couple (*clé*, *valeur*) stocké dans la table de hachage. La `HashMap` est un **type générique**, c'est-à-dire qu'il faut préciser le type des clés et celui des valeurs lors de sa création.

Par exemple, le code suivant crée une map dont les clés sont des entiers et les valeurs des chaînes de caractères :

```
1 import java.util.HashMap;
2
3 HashMap<Integer, String> m = new HashMap<>();
```

Principales méthodes de `HashMap`

Méthode	Description
`m.put(key, value)`	Ajoute une entrée dans la map (remplace la valeur si la clé existe déjà).
`m.get(key)`	Retourne la valeur associée à la clé, ou `null` si la clé n'existe pas.
`m.containsKey(key)`	Retourne `true` si la clé est présente, `false` sinon.
`m.remove(key)`	Supprime l'entrée associée à la clé. Retourne la valeur supprimée, ou `null` si la clé n'existait pas.
`m.size()`	Retourne le nombre d'entrées de la map.
`m.clear()`	Supprime toutes les entrées de la map.
`m.keySet()`	Retourne l'ensemble des clés contenues dans la map.
`m.values()`	Retourne la collection des valeurs contenues dans la map.
`m.entrySet()`	Retourne l'ensemble des couples (clé, valeur).

Exemple d'utilisation :

```java
import java.util.HashMap;

public class ExempleHashMap {
    public static void main(String[] args) {
        // Creation de la HashMap
        HashMap<String, Integer> m = new HashMap<>();

        // Ajout des entrees (cle, valeur)
        m.put("George", 10);
        m.put("Stevie", 18);
        m.put("Eric", 15);
        m.put("Ed", 5);

        // Affichage de valeurs associees a certaines cles
        System.out.println("m.get(\"George\") =" + m.get("George"));
        System.out.println("m.get(\"Eric\") = " + m.get("Eric"));
        System.out.println("m.get(\"Jimi\") = " + m.get("Jimi"));
        // Cle absente -> null

        // Verification de la presence de certaines cles
        System.out.println("La cle \"Eric\" est-elle presente ? " +
           m.containsKey("Eric"));
        System.out.println("La cle \"Jimi\" est-elle presente ? " +
           m.containsKey("Jimi"));

        // Taille de la map
        System.out.println("La taille est = " + m.size());

        // Suppression d'elements
        System.out.println("On retire la cle \"Ed\" : " +
                                        (m.remove("Ed") != null));
        System.out.println("Maintenant la taille est =" + m.size());
        System.out.println("On retire la cle \"Jimi\" : " +
                                        (m.remove("Jimi") != null));
    }
}
```

Sortie :

```
m.get("George") = 10
m.get("Eric") = 15
m.get("Jimi") = null
La cle "Eric" est-elle presente ? true
La cle "Jimi" est-elle presente ? false
La taille est = 4
On retire la cle "Ed" : true
Maintenant la taille est = 3
On retire la cle "Jimi" : false
```

Itération sur les données :

On peut itérer sur les clés, les valeurs ou les entrées d'une `HashMap` en utilisant une boucle `for`. Les méthodes `keySet()`, `values()` et `entrySet()` fournissent respectivement des vues sur les clés, les valeurs et les couples (clé, valeur). L'exemple ci-dessous illustre les trois formes d'itération sur la map `m`, telle qu'elle est définie à la fin de l'exemple précédent.

```java
// Creation de la HashMap
HashMap<String, Integer> m = new HashMap<>();

// Ajout des entrees (cle, valeur)
m.put("George", 10);
m.put("Stevie", 18);
m.put("Eric", 15);
m.put("Ed", 5);

// Iteration sur les cles
System.out.println("Iteration sur les cles :");
for (String key : m.keySet()) {
    System.out.println("Cle = " + key);
}

// Iteration sur les valeurs
System.out.println("\nIteration sur les valeurs :");
for (Integer value : m.values()) {
    System.out.println("Valeur = " + value);
}

// Iteration sur les entrees (cle, valeur)
System.out.println("\nIteration sur les entrees :");
for (var entry : m.entrySet()) {
    System.out.println(entry.getKey() + " = " + entry.getValue());
}
```

Sortie :

```
Iteration sur les cles :
Cle = Eric
Cle = George
Cle = Stevie
Cle = Ed

Iteration sur les valeurs :
Valeur = 15
Valeur = 10
Valeur = 18
Valeur = 5

Iteration sur les entrees :
Eric = 15
George = 10
Stevie = 18
Ed = 5
```

Exemple d'utilisation d'une `HashMap` comme compteur

Étant donnée une chaîne de caractères, on souhaite compter combien de fois chaque caractère apparaît dans la chaîne.

Solution : Créer une `HashMap` vide utilisant les caractères comme clés, puis parcourir la chaîne. Si le caractère courant n'est pas encore présent comme clé, on crée une nouvelle paire (clé, valeur) avec 1 comme valeur initiale. Sinon, on incrémente la valeur associée de 1.

```java
// Fonction qui compte les caracteres dans une chaine
public static HashMap<Character, Integer> compter(String chaine) {
    HashMap<Character, Integer> compteur = new HashMap<>();

    for (char c : chaine.toCharArray()) {
        // Si le caractere existe deja, on incremente sa valeur
        if (compteur.containsKey(c)) {
            compteur.put(c, compteur.get(c) + 1);
        } else {
            // Sinon, on l'ajoute avec la valeur 1
            compteur.put(c, 1);
        }
    }
    return compteur;
}

public static void main(String[] args) {
    String chaine = "Hello world !";

    // Appel de la fonction
    HashMap<Character, Integer> resultat = compter(chaine);

    // Affichage du resultat
    System.out.println("Comptage des caracteres :");
    for (var entry : resultat.entrySet()) {
        System.out.println(entry.getKey() + "=" + entry.getValue());
    }
}
```

Sortie :

```
Comptage des caracteres :
  = 1
! = 1
H = 1
d = 1
e = 1
l = 3
o = 2
r = 1
w = 1
```

Exercice 1 : Compter les mots d'une chaine

On suppose qu'une chaîne de caractères est constituée de mots séparés par des espaces. Par exemple, la chaîne suivante :

`"Ceci est une chaine et cette chaine est constituee de dix mots"`

Écrivez une fonction qui prend une chaîne de caractères et retourne, sous forme de `HashMap`, les mots qu'elle contient ainsi que le nombre de fois où chaque mot apparaît.

Exercice 2 : Détection de doublons dans un tableau

Écrivez une fonction qui prend en paramètre un tableau de nombres et qui retourne `true` si, et seulement si, le tableau contient au moins un élément apparaissant plusieurs fois.

Exercice 3 : Suite de Fibonacci avec mémoïsation

Pour rappel, la suite de Fibonacci est définie par :

$$u_n = \begin{cases} 0 & \text{si } n = 0, \\ 1 & \text{si } n = 1, \\ u_{n-1} + u_{n-2} & \text{sinon.} \end{cases}$$

Son implémentation récursive simple est la suivante :

```java
// Version naive (inefficace)
public static int fibonacci(int n) {
    if (n == 0) return 0;
    else if (n == 1) return 1;
    else return fibonacci(n - 1) + fibonacci(n - 2);
}
```

Cette version devient rapidement inefficace lorsque `n` augmente, car elle recalcule plusieurs fois les mêmes valeurs de la suite.

Une manière d'éviter cette redondance consiste à stocker les valeurs déjà calculées dans une `HashMap`, dont les clés représentent les indices n et les valeurs les termes correspondants u_n. Ainsi, avant de calculer une valeur, on vérifie si elle est déjà présente dans la map. Ce principe s'appelle la **mémoïsation**.

Complétez la fonction ci-dessous. La map `memoire` sert à stocker les valeurs connues de la suite. Initialement, on connaît $u_0 = 0$ et $u_1 = 1$.

```java
// Initialisation de la memoire
HashMap<Integer, Integer> memoire = new HashMap<>();
memoire.put(0, 0);
memoire.put(1, 1);

// Fonction a completer
public static int fibonacci(int n, HashMap<Integer, Integer> memoire
    ) {
    // ...
}
```

Exercice 5 : Comptage de séquences de caractères

Écrire une fonction :

```
extraireSequence(String chaine, int n, int indice): String
```

qui extrait et retourne la séquence de n caractères commençant à l'indice `indice` dans la chaîne `chaine`. On suppose que $0 \leq$ `indice` $<$ `chaine.length()` et que $0 < n \leq$ `chaine.length()` $-$ `indice`.

Exemple : `extraireSequence("abcdefgh", 4, 3)` retourne `"defg"`.

Question : Écrire une fonction :

```
compterSequences(String chaine, int n): HashMap<String, Integer>
```

qui compte le nombre d'occurrences de chaque séquence de taille n présente dans la chaîne `chaine`. On suppose que $0 <$ `n` $\leq$ `chaine.length()`.

Exemple : `compterSequences("abcab", 2)` retourne :

$$\{\text{"ab"} = 2, \text{"bc"} = 1, \text{"ca"} = 1\}$$