Université de Montpellier / Faculté Des Sciences

Implantation des tableaux associatifs avec hachage (ou table d'association ou dictionnaire ou *HashMap*)

Structures de données — HMIN 215

11 février 2020

1 Principe des tableaux associatifs

Un tableau associatif (aussi appelé dictionnaire ou table d'association), map en anglais, est un type de données permettant à ses utilisateurs d'associer un ensemble de clefs à leurs valeurs respectives. La question 9 ci-dessous donne un exemple d'un dictionnaire en Java qui associé chaque fruit à sa valeur calorique. Dans leur version Java, les dictionnaires sont spécifiés par l'interface **Map**, de l'anglais (to map (faire correspondre) et implantés par diverses classes dont **HashMap** ou **HashTable**. Le nom de ces 2 classes vient du fait que leur implantation utilise une fonction dite de hachage (voir plus loin), qui permet de récupérer une valeur en un temps constant, quelle que soit la taille du dictionnaire. Le TP permet à reproduire partiellement et donc de de comprendre l'implantation des classes **HashMap** et **HashTable**.

2 Spécification fonctionnelle

Chaque clé est d'un type donné K et la valeur d'un type donné V . Un dictionnaire, paramétré par les types K et V possède entre autres les méthodes suivantes (inspirée par l'interface Map de Java) :

- **V put(K key, V value) :** place une clé et une valeur associée dans le dictionnaire
- **V get(Object key) :** prend une clé en argument et rend la valeur qui lui est associée dans le dictionnaire ou null s'il n'y en a pas
- **boolean isEmpty() :** dit si le dictionnaire est vide
- **int size() :** retourne le nombre d'associations clé-valeur stockées dans le dictionnaire
- **boolean containsKey(Object key) :** dit si une clé est présente dans le dictionnaire
- **String toString() :** fabrique une chaîne représentant le receveur

3 Spécification de l'implémentation avec des tableaux dynamiques et avec hachage

Un tableau dynamique est un tableau que l'on peut agrandir ; en fait c'est une vue de l'esprit qui représente le fait de remplacer un premier tableau par un plus grand quand le

premier devient trop petit (un peu comme changer de voiture lorsque la famille s'agrandit). L'implantation va utiliser 2 tableaux dynamiques, un premier contenant les clés, un second contenant les valeurs. Une valeur est dans le tableau des valeur au même index que sa clé dans le tableau des clés.

Fonction de hachage. La fonction de hachage qui donne la position (l'index) d'une clé c dans le tableau des clés de taille t est en Java : c.hashCode() % t. % est la fonction "modulo" et hashCode() est une méthode de la classe Object (allez consulter sa documentation).

Gestion des conflits de hachage. Lorsque la fonction de hachage produit une même valeur d'index de rangement pour deux clés différentes, on est dans une situation de conflit de hachage, ou collision. Une méthode de résolution d'une collision est la "recherche séquentielle locale" : on recherche, à partir de l'index donné par la fonction de hachage, l'index de la première case libre (si on arrive en fin de tableau, on recommence au début).

Agrandissement des tableaux. Une bonne gestion des tableaux consiste à les a agrandir quand ils sont pleins à 75%, de façon à limiter les collisions et à garantir qu'il y a toujours une place libre pour la recherche séquentielle locale réalisée en cas de collision.

4 Questions

1. Ecrivez une interface **Idico<K,V>** représentant la spécification fonctionnelle.
2. Définissez une classe **Dico<K,V>** implantant l'interface. Définissez ses attributs et le constructeur sans paramètre.
3. Définissez la méthode publique **isEmpty().**
4. Définissez une méthode protégée **indexOf(K c)**, qui rend l'index auquel est rangé la clé c dans le tableau des clés ou -1 si la clé n'y est pas. Cette méthode doit utiliser la fonction de hachage pour trouver la clé.
5. Définissez la méthode publique **containsKey(K c)**.
6. Définissez la méthode publique **get(K c)**.
7. Définissez une méthode protégée **newIndexOf(K c)**, qui rend l'index auquel pourra être insérée la clé c dans le tableau des clés, mais n'effectue pas l'insertion. Pour cette première version on suppose que les tableaux sont moins que 3/4 pleins.
8. Définissez la méthode publique **put(K c, V v)**. Pour cette première version on suppose que les tableaux sont moins que 3/4 pleins.
9. Ecrivez un main pour tester le dictionnaire.

```
1 Idico calories = new Dico(10);
2 calories.put("abricot",235);
3 calories.put("amande",1023);
4 calories.put("ananas",242);
5 calories.put ("pomme",83);
6 calories.get("amande");
```

10. Trouvez des données qui provoquent un conflit de hachage. Testez que **get** fonctionne toujours.
11. Pour la version finale : Définissez une méthode protégée **mustGrow()** qui dit si le tableau des clés est plus que 3/4 plein. Définissez une méthode protégée **grow()**

qui fait grossir les tableaux de 10% (en fait remplace les anciens par des nouveaux plus grands de 10%). Réécrivez la méthode **newIndexOf()** pour qu'elle traite le cas complet, sans limite sur le nombre de couples dans le dictionnaire. Quand il faut agrandir le tableau, comme l'algorithme dépend de la taille du tableau, il faut bien sûr redistribuer les associations existantes dans le nouveau tableau.

12. Faites un autre programme main utilisant un dictionnaire pour compter le nombre d'occurrences des mots d'une chaîne de caractères.