TP2 - Algorithmique avancée : Tables de Hachage

CERI - Master I Informatique - 2016-2017

Dans ce TP nous allons implanter trois types de hachages utilisant différentes méthodes de gestions de collisions et fonctions de hachage :

1. collisions par chaînage et hachage spécifique détaillé en section 1 ;
2. collisions par hachage linéaire et hachage basé sur `String.hashcode()` ;
3. collisions par hachage double et hachage basé sur `String.hashcode()`.

1 Modélisation

Définir une classe générique `Hachage` permettant de représenter n'importe quel type de hachage. Cette classe doit contenir un entier m représentant le nombre maximal d'entrées de la table. Elle doit également comporter les méthodes :

— `Hachage(m)` : permet de créer une table de hachage de taille m ;
— `add(d)` : permet d'ajouter une donnée à la table ;
— `recherche(d)` : retourne vrai si la donnée `d` est contenue dans la table et faux sinon ;
— `h(d)` : fonction de hachage retournant un entier appartenant à $\{0, 1, \ldots, m-1\}$.

Tout comme pour le précédent TP, les tables de hachage contiendront des données génériques contenant un attribut `cle` de type chaîne de caractères.

2 Hachage avec résolution de collisions par chaînage

On définit tout d'abord la fonction de hachage considérée pour cette table (*i.e.*, comment calculer pour toute donnée, un entier appartenant à $\{0, 1, \ldots, m-1\}$ correspondant à sa clé).

2.1 Fonction de hachage considérée

On appelle **code numérique associé à un caractère** `c` l'entier retourné par l'instruction Java `c.hashCode()`.

On considère un entier a ainsi qu'une donnée d dont la clé contient n caractères c_0, c_1, ..., c_{n-1}. Soit t_i le code numérique associé au caractère i pour tout $i \in \{0, ..., n-1\}$. On définit le **code numérique** `x` **associé à la clé** $c_0c_1...c_{n-1}$ par

$$x = t_0 + at_1 + a^2t_2 + ... + a^{n-1}t_{n-1}$$

La fonction de hachage considérée est la suivante : $h(d) = x \ mod \ m$.

2.2 Implémentation

Définir une classe `HachageCollision` héritant de `Hachage`. Cette classe devra comporter

— un constructeur prenant en argument la taille m ainsi que l'entier a utilisé pour le calcul de la fonction de hachage ;
— une méthode `String getListsSize()` permettant d'avoir une idée globale du remplissage de la table (voir exemple ci-dessous).

Considérons la table de hachage suivante :

Entrée de la table	0	1	2	$3 = m - 1$
Valeur(s) contenue(s)	$\emptyset$	{"algorithmique", "avancée}"	{"ceri"}	$\emptyset$

La méthode `getListsSize()` retournera pour cette table la chaîne de caractères :

```
2 entrée(s) contenant 0 élément(s)
1 entrée(s) contenant 1 élément(s)
1 entrée(s) contenant 2 élément(s)
```

2.3 Fichier de tests

Testez vos différentes méthodes en utilisant **et complétant** le fichier `HachageCollisionTest.java` fourni sur l'ENT.

2.4 Dictionnaire

Testez vos méthodes en stockant les mots du dictionnaire de la langue française dans votre table de hachage.

Utilisez ensuite cette table pour tester qu'il n'y ait pas de fautes de frappe dans un texte. Pour ce faire, écrire une méthode prenant en argument le fichier correspondant au texte et vérifiant que tous les mots utilisés sont bien dans le dictionnaire.

Vous afficherez un tableau qui indiquera pour les valeurs de m suivantes $300 * 10^3$, $400 * 10^3$ et $1000 * 10^3$:

— le nombre moyen de comparaisons pour trouver un mot existant dans le dictionnaire ;

— le nombre moyen de comparaisons pour déterminer qu'un mot n'existe pas dans le dictionnaire ;

— le nombre de comparaisons total ;

— le temps d'exécution.

Vous testerez votre méthode avec les fichiers `ArticleMonde.fr` et `ArticleMondeModifie.fr`.

3 Hachage avec résolution de collisions par hachage linéaire

On considère maintenant une table de hachage permettant une résolution de collisions par hachage linéaire. On utilise la fonction de hachage `|String.hashcode()| mod m`.

Effectuer les mêmes travaux que pour la précédente table de hachage (implémentation, fichier de tests `HachageLineaireTest.java`, dictionnaire). Les seules différences sont les suivantes :

— Le constructeur prend uniquement un entier m en argument ;

— La méthode `getListsSize()` est remplacée par `String toString()` permettant d'obtenir l'indice des éléments non vides de la table ainsi que leur contenu (voir exemples du fichier de tests pour plus de détails).

4 Hachage avec résolution de collisions par hachage double

Même travail qu'en section 3 mais avec une résolution de collisions par hachage double.

Le constructeur de la table doit prendre en paramètre un entier `k` – premier avec `m` – utilisé pour calculer les fonctions d'essai (dans le cadre de ce TP, la valeur de `k` est la même quelque soit la donnée considérée). Le fichier de tests est `HachageDoubleTest.java`.