"الأعمال العظيمة ما هي الاّ أعمال صغيرة كتب لها الاستمرار"

Algorithmique et Structures de Données 1 Mini-Projet

Mini-projet 1 : Tables de hachage

Dans ce mini-projet, nous allons implémenter une table de hachage qui utilise des listes chaînées pour gérer les collisions ; notre implémentation ne prendra pas en compte le redimensionnement de la table lorsque le taux de collisions devient trop élevé.

Structure du mini projet

Nous travaillerons dans un premier temps sur les fonctions de hachage. Ce sera pour vous l'occasion d'implémenter la fonction standard de Java, qui utilise la constante `31`, mais aussi d'implémenter une autre fonction de hachage. Nous parlerons brièvement de collisions.

Les tables de hachage, pour gérer les collisions, utilisent des listes chaînées. Bien que la bibliothèque standard de Java fournisse déjà des listes chaînées, nous vous demanderons pour ce sujet de les réimplémenter. Cette partie est relativement facile, et vous permettra, si vous ne l'avez jamais fait, de voir comment on implémente une telle structure de données.

Enfin, nous nous intéresserons à l'implémentation d'une table de hachage à proprement parler : pour ce faire, nous combinerons les deux premières parties. La présence de multiples fonctions de hachage vous permettra de comparer leurs performances relatives : en effet, la fonction de hachage ultime n'existe pas, et nous verrons que certaines donnent lieu à moins de collisions que d'autres.

Obtenir le sujet

L'archive est disponible dans le groupe yahoo MI. Téléchargez-la, puis extrayez son contenu dans le dossier de votre choix.

Des objets et des haches

L'idée de ce mini-projet est d'illustrer comment fonctionne le principe des tables de hachage. Une table de hachage stocke des *objets* ; ces objets ont eux-mêmes un *hash*, c'est-à-dire une valeur unique qui leur est associée.

Puisque nous voulons comprendre comment fonctionne le mécanisme de hachage, nous n'allons pas réutiliser le mécanisme de hachage standard de Java (la fonction `hashCode()`). Nous allons au contraire définir une classe `Objet`, qui sera la classe des objets allant dans la table de hachage.

Voici la définition de la classe `Objet` :

```
public abstract class Objet {
  abstract int hash();
  abstract String nom();}
```

Cette classe est *abstraite*, c'est-à-dire qu'elle ne peut pas être instanciée à l'aide de l'opérateur `new` : si vous essayez d'écrire `Objet o = new Object();` ce code sera rejeté par le compilateur Java. En effet, les méthodes `hash()` et `nom()` sont abstraites : elles n'ont pas d'implémentation.

Votre rôle ici sera de créer des classes concrètes qui *héritent* de la classe `Objet` et qui fournissent des implémentations pour les deux méthodes abstraites. Ainsi, vous pourrez créer `Objet1` qui hérite de `Objet`, puis écrire :

```
Objet o = new Objet1();
```

Du point de vue fonctionnel, un `Objet` est simplement une classe qui stocke un nom, et qui offre une méthode pour calculer le hash associé à l'`Objet`.

Ici, pour les besoins de l'exercice, notre fonction s'appelle `hash`.

Commencez par créer le fichier `src/Objet1.java`. La première ligne du fichier sera certainement `public class Objet1 extends Objet {`.

Parlons un peu plus de la vie réelle. Dans un programme Java classique, vous utiliseriez `java.util.Hashtable<K,V>`, la classe de la bibliothèque standard Java qui fournit une implémentation des tables de hachage. Celle-ci est paramétrée par le type `K` des clés et le type `V` des valeurs.

La question cruciale est : comment comparer des objets ? En effet, dans ce mini-projet, la table de hachage *sait* que les éléments sont des `Objet`s, et qu'ils ont un champ `nom`, qui permet de comparer un objet à un autre. On casse ainsi les frontières de l'abstraction au bénéfice d'un mini-projet plus simple. Une vraie implémentation comme celle de la bibliothèque standard se doit d'être *générique*, c'est-à-dire de fonctionner pour n'importe quel type `V` des éléments. Comment fait-on pour comparer des éléments de type `V` alors qu'on n'a, par définition, aucune information sur la nature de `V` ? Il suffit d'exiger que `V` fournisse une méthode `equals` qui implémente la comparaison. C'est vrai de toute classe Java : en effet, chaque classe Java hérite de la classe de base `Object`, qui fournit une fonction `equals`.

Constructeur

Les objets peuvent être construits à l'aide du constructeur `Objet1(String nom)`. Commencez par implémenter ce constructeur, puis écrivez la fonction `public String nom()` qui retourne le nom qui a été fourni auparavant via le constructeur.

Fonction de hash standard

Nous allons d'abord implémenter la fonction de hachage standard de Java. Si s est la chaîne de caractères, alors son hachage est défini par :

$$hash(s) = \sum_{i=0}^{n-1} s[i] \times 31^{n-1-i}$$

En d'autres termes,

$$hash(s) = s[0] \times 31^{n-1} + \ldots + s[n-2] \times 31 + s[n-1]$$

Cette fonction peut s'implémenter à l'aide d'une simple boucle `for` **sans utiliser autre chose que des additions ou des multiplications**. Si vous utilisez `BigInteger` ou `Math.pow`, vous allez avoir des problèmes de conversion entre flottants et nombres entiers, et vous serez de surcroît inefficace.

Implémentez la méthode `hash()` de la classe `Objet1`. Nous rappelons que que le i-caractère de la chaîne `s` s'obtient avec `s.charAt(i)`.

Il est important de comprendre les règles de promotion en Java. La référence complète est un peu aride, voici donc un résumé de ce qui vous concerne pour cette question. Une conversion a lieu quand un opérateur binaire est appliqué à deux opérandes qui n'ont pas la même taille. Par exemple, si `c` est un caractère (type `char`) et que `h` est un entier (type `int`), l'opération `c + h` est une opération binaire (le `+`) appliquée à deux opérandes (`c` et `h`) qui n'ont pas la même taille. En effet, en Java, `c` occupe 16 bits, c'est un *codepoint* UCS-2, tandis que `h` est un entier 32 bits, par définition. Dans cette situation, un *élargissement* a lieu : `c` est **automatiquement** converti en entier 32 bits de manière à garantir que l'opération ne donne pas lieu à une perte de précision. En clair : **vous n'avez pas besoin d'écrire de casts pour implémenter `hash`**.

Cette fonction de hash dépassera rapidement l'entier maximum représentable dans le type java `int` (entiers signés 31 bits). Il se produira alors un dépassement de capacité, et le code de hash

deviendra négatif. C'est le comportement attendu. Le type `int` a l'avantage de faire la même taille sur les machines 32 et 64 bits ; les résultats des tests ne dépendront donc pas de votre architecture.

Testez votre fonction de hachage : le hash de la chaîne `"coucou"` est `-1354586272`. Le hash de la chaîne "Bonjour Coursera" est `-1895827609`. La définition ci-dessus est bien définie pour la chaîne vide ! Quel doit être son hash ?

Activez la fonction `test1a`.

Fonction de hash alternative

Il existe tout un folklore de fonctions de hachage : la plupart ont été élaborées de manière empirique, utilisent des constantes qui, en pratique, donnent une bonne distribution, et sont le fruit de beaucoup d'essais / erreurs. Une fonction de hachage doit bien distribuer les bits de son entrée ; une manière classique de le faire est à l'aide d'un processus itératif qui combine une valeur initiale avec les caractères de la chaîne.

Voici une deuxième fonction de hachage, que nous vous proposons d'implémenter. Elle combine le i-ième caractère avec le hash `h` de manière différente.

$$\text{hash}'(i) = \text{hash}'(i-1) * 33 \oplus s[i]$$
$$\text{hash}'(-1) = 5381;$$

L'opérateur mathématique $\oplus$ est le « ou exclusif » ; il est disponible en Java via l'opérateur `^`. Cette définition se prête naturellement à un calcul itératif ; implémentez la classe `Objet2` qui est en tous points similaire à la classe `Objet1`, si ce n'est que sa fonction `hash` utilise la fonction de hash alternative.

Testez votre fonction de hachage : le hash de la chaîne `"coucou"` est `1544958309`.

Activez la fonction `test1b`.

Fonction de hash fournie

Nous fournissons une classe `Objet3` qui implémente une troisième fonction de hachage, et qui sera utile lors de la troisième partie.

De nombreux ouvrages d'algorithmique traitent de la question des fonctions de hachage. On pourra se référer, au besoin, aux ouvrages de D. Knuth, ou de T. Cormen.

Listes simplement chaînées

Nous avons maintenant besoin de listes pour stocker les éléments qui possèdent le même hash. Des listes simplement chaînées suffisent. L'objet de cette partie est d'implémenter des listes impératives ; pour ne pas compliquer inutilement le sujet, nos listes ne seront **pas** génériques : vous pouvez faire l'hypothèse que les éléments de la liste sont des `Objet`s.

Rappel sur les fonctionnements des listes impératives

Une liste simplement chaînée est constituée d'une successions de cellules ; chaque cellule pointe vers la suivante, et la dernière cellule ne pointe vers rien, c'est-à-dire que son champ `suivant` vaut `null`.

Nous voulons ici une structure de données impérative. Si ma liste s'appelle `l` et que je souhaite y ajouter l'objet `o`, écrire `l.ajouterTete(o);` a pour *effet* de modifier `l` *en place*. Après cette ligne, `l` a changé et contient désormais `o`.

Les structures de données impératives s'opposent aux structures fonctionnelles, ou persistentes : une autre approche aurait consisté à *renvoyer une nouvelle liste*. Ainsi, on aurait pu écrire `Liste l2 = l.ajouterTete(o);`. Dans ce cas-là, `l2` aurait contenu la nouvelle liste, tandis que `l` aurait toujours contenu l'ancienne liste.

Pour implémenter facilement une liste impérative, on peut créer, en plus de la classe `Cellule`, une classe `Liste`, qui contient une référence vers la tête de la liste. Ainsi, `Liste` aura un champ

`private Cellule tete;`. Ce champ sera modifié au fur et à mesure des appels à `ajouterTete` et `supprimerTete` : c'est donc bien une structure de données impératives.

Nous imposons dans ce sujet une contrainte supplémentaire : les fonctions d'ajout et de suppression *renvoient l'objet liste lui-même*. Ceci permet de chaîner les appels, comme dans par exemple `l.ajouterTete(o1).ajouterTete(o2);`.

Créez le fichier `src/Liste.java`.

Implémentation attendue

Vous êtes libre de choisir la représentation que vous voulez pour les cellules. En revanche, vos listes chaînées doivent implémenter la classe appelée `Liste` et offrir les fonctions suivantes :

- `Liste()`, le constructeur par défaut;
- `public Liste ajouteTete(Objet val)`, qui ajoute `val` en tête de la liste, et renvoie la liste elle-même ;
- `public Liste supprimeTete()`, qui lance une exception `java.util.NoSuchElementException` si la liste est vide, ou supprime l'élément en tête de la liste, et renvoie la liste elle-même sinon ;
- `public boolean contient(Objet o)`, qui renvoie `true` si la liste contient un élément portant le même nom que `o` (attention, les chaînes de caractères `s1` et `s2` se comparent avec `if (s1.equals(s2)) ...`) ;
- `public int longueur()` qui renvoie la longueur de la liste.

Vous devrez probablement importer `java.util.NoSuchElementException` en tête de votre fichier.

Tests

Vérifiez bien que vos fonctions ont la sémantique attendue. N'oubliez pas que le code suivant doit réussir, car on compare les objets de manière *structurelle* : c'est le nom des objets qui est comparé, et pas l'adresse de l'objet. Votre code de `Liste` doit donc tenir compte de ce fait.

```
Liste l = new Liste();
l.ajouteTete(new Objet1("toto"));
assert l.contient(new Objet1("toto"));
```

Un programme java idiomatique aurait ici écrasé la méthode par défaut `equals` de la classe `Objet`.

Activez la fonction `test2`.

Tables de hachage

Il s'agit maintenant de combiner le travail effectué dans les deux premières parties pour implémenter des tables de hachage.

La table de hachage contiendra un tableau de `Liste`s à usage interne, et fera appel aux méthodes `hash()` des `Objet`s stockés. Encore une fois, la table de hachage ne sera pas générique : vous pouvez donc faire l'hypothèse que tous les objets que vous manipulez sont des `Objet`s. Ceci sera particulièrement utile lors de la gestion des collisions.

Voici la description de la classe `TableDeHachage` que vous devez écrire.

- `TableDeHachage(int n)`, constructeur qui prend la taille initiale de la table (nombre de listes utilisées en interne).
- `public void ajoute(Objet o)`, pour ajouter un élément dans la table.
- `public boolean contient(Objet o)`, pour tester si la table contient un élément.

Nous vous demandons d'écrire une fonction supplémentaire par rapport à l'interface standard des tables de hachage. Cette fonction nous permettra de tester que votre code se comporte correctement.

- `public int[] remplissageMax()`: cette fonction trouve la `Liste` qui est la plus remplie, et renvoie un tableau de deux entiers. Le premier entier est l'index de cette liste dans le

tableau interne, et le second est le nombre d'éléments dans la liste. Dans le cas où plusieurs listes ont le même remplissage, cette fonction doit renvoyer la "première" liste, c'est-à-dire celle dont l'index est le plus petit dans la table. Dit encore différemment, celle qui a le plus petit hash *après* modulo.

Copiez le code de `test4` dans une fonction à vous, et comparez les remplissages maximaux pour les `Objet1`, `Objet2` et `Objet3`. Quelle est la meilleure fonction de hachage ?

Testez votre code avec une fonction de hachage idiote, qui renvoie tout le temps la même valeur. Vous vous retrouverez avec la pire complexité, mais vous pourrez tester facilement votre gestion des collisions.

Attention : nous avons mentionné qu'un hash peut être négatif. Vous aurez besoin de faire appel à l'opérateur modulo pour ramener un hash dans les dimensions de la table interne. Quel est comportement de l'opérateur modulo sur les nombres négatifs en Java ?

Une fois le problème ci-dessus compris, nous vous demandons d'y remédier en *additionant la taille de la table au modulo négatif*. En clair, si `m` est votre modulo, écrivez `if (m < 0) m += table.length`.

Activez la fonction `test3`. Vérifiez que votre table de hachage a bien le comportement attendu sur ce test. Les deux chaînes utilisées n'ont pas été choisies au hasard. Quelle est leur particularité ?

Une fois cette étape validée, activez la fonction `test4`.

Documents à rendre et Calendrier

- Le listing du programme source complet soigneusement commenté.
- Un rapport de programmation.

Le 09/01/2014 est fixé comme dernier délai de remise des documents (tout retard ne sera pas toléré). La soutenance du travail aura lieu une semaine plus tard.

Le projet doit être réalisé en binôme. Ces derniers doivent s'identifier avant le 11/12/2013 auprès des enseignants responsables du module Algorithmique et Structures des Données 1.

Evaluation du mini-projet :

L'évaluation repose sur les éléments suivants :

- Le programme source :
  - Respect de l'énoncé et originalité du travail.
  - Qualité de programmation : efficacité algorithmique, choix de structures de contrôle, ...
  - Présentation du programme : indentation, commentaires et nommage des objets manipulés.
- Le rapport de programmation :
  - Présentation.
  - Argumentation.
  - Rédaction personnel.
- Soutenance du travail :
  - Démonstration du programme.
  - Interrogation individuelle sur le travail réalisé.