

POBJ – Examen de rattrapages - Janvier 2007

Tables de Hachage

Durée : 2 heures

Documents autorisés : Polycopiés du cours et notes personnelles uniquement

IMPORTANT: Sur une feuille séparée, un diagramme UML vision fournisseur de toutes les classes (interfaces et classes données en énoncé et réponses aux questions) sera élaboré au fur et à mesure du sujet.

L'objectif de cet examen est de concevoir une implémentation orientée objet et générique d'une structure de donnée de type table associative par hachage. Nous adoptons une démarche de conception dite « par contrats » (cf. Cours « Programmation robuste »).

1. EXERCICE 1 : HACHAGE

La notion de « hachage », en informatique, consiste en la génération d'un indice numérique (le plus souvent un entier) à partir d'une donnée quelconque. Une fonction de hachage implémente un algorithme de conversion entre la donnée et son indice de hachage. En Java, une méthode standard de hachage se trouve dans la classe `Object` : `+hashCode(Object) : int`. Toute sous classe peut redéfinir cette méthode pour spécialiser l'algorithme de hachage. La limite principale de cette approche est que toutes les instances d'une même classe seront « hachées » de la même façon. Dans cet exercice, nous adoptons une approche permettant à différentes instances de la même classe d'utiliser des méthodes de hachage différentes. L'interface devant être implémentée par une fonction de hachage est la suivante:

```
package hash.framework;
/**
 * Interface de fonction de hachage
 * @param <T> le type des objets pouvant être hachés par cette fonction */
public interface HashFunction<T> {
    /**
     * fonction de hachage
     * @param obj l'objet devant être haché par cette fonction
     * @return un entier positif représentant l'indice de hachage
     */
    public int hash(T obj);
}
```

On donne ci-dessous l'implémentation d'une fonction de hachage simple et assez efficace, notamment sur les chaînes de caractères (algorithme provenant de GNU Classpath 0.93):

```
package hash.string;
import hash.framework.HashFunction;

public class GnuCPStrHashFun implements HashFunction<String> {

    public GnuCPStrHashFun() {
    }

    public int hash(String obj) {
        int hashCode = 0;
        for (int i = 0; i < obj.length(); i++)
            hashCode = hashCode * 31 + obj.codePointAt(i);
        return hashCode;
    }
}
```

Question 1.1 : Objets « hacheurs »

Définir dans le paquetage `hash.framework` une classe abstraite `HashObject<T>` (avec `T` quelconque) contenant:

- un attribut `obj` de type `T` représentant l'objet haché par le « hacheur » avec son accesseur
- un attribut `hfun` qui est une fonction de hachage pour le type `T` avec un accesseur et un modificateur publics
- un attribut entier `hashCode` qui sauvegarde la dernière valeur de hachage calculé **sans** accesseur **ni** modificateur
- un constructeur à deux paramètres (objet à hacher de type `T` et fonction de hachage pour le type `T`), on initialisera `hashCode` à la valeur -1 (valeur de hachage impossible)
- un constructeur qui ne prend en paramètre que l'objet à hacher, la fonction de hachage sera initialisée par une méthode abstraite protégée `defaultHashFunction` sans paramètre (**remarque**: il faut ajouter la signature de cette méthode qui retourne bien sûr une fonction de hachage pour le type `T`)
- une méthode `hash` sans paramètre qui calcule et retourne l'indice de hachage pour `obj`. Cette méthode utilise la fonction de hachage `hfun` pour calculer l'indice si aucun indice n'a été calculé précédemment, sinon on utilise l'indice déjà calculé (`hashCode`)
- une méthode `rehash` qui renouvelle le hachage en utilisant systématiquement la fonction de hachage `hfun`.
- pour les méthodes standards, on implémentera `+ hashCode() : int` et `+ equals(Object o) : boolean`

Question 1.2 : Hachage de chaînes

Définir dans le paquetage `hash.string` une classe « hacheuse » de chaîne de caractères, de nom `HashString` sans paramètre de type et sous-classe de la classe définie dans la question précédente. La fonction de hachage utilisée par défaut sera celle définie par la classe `GNUCPStrHashFun` (cf. ci-dessus).

Donner un exemple qui illustre le fait que l'on peut changer dynamiquement de fonction de hachage pour les chaînes de caractères (on pourra supposer une deuxième classe de hachage de chaînes `UneAutreStrHashFun`). Expliquer quel design pattern est mis en oeuvre et associer à chaque consistant du pattern (tel que décrit en cours) une classe ou une interface de cette exercice.

2. EXERCICE 2 : TABLE DE HACHAGE PAR CHAÎNAGE SIMPLE

Voici l'interface principale des tables de hachage:

```
package hash.framework;
import java.util.List;
/**
 *
 * L'interface principale de table de hachage.
 * Toute table de hachage doit implémenter cette interface.
 *
 * @param <K> Le type des clés : type compatible avec HashObject<?>
 * @param <V> Le type des valeurs : type quelconque
 */
public interface HashTable<K extends HashObject<?>,V> {
    /**
     * Ajoute une association dans cette table
     * PREREQUIS:
     * - la clé n'est pas null
     * GARANTIES:
     * - la taille de la table augmente de 1
     * - la nouvelle valeur est bien associée à la clé
     * @param key la clé de l'association
     * @param value la valeur de l'association
     * @throws IllegalArgumentException si la clé est null
     * @throws DuplicateEntryException si l'association existe déjà
     */
    public void put(K key,V value) throws DuplicateEntryException;

    /**
     * Tente de récupérer dans cette table la valeur associée à la clé spécifiée
     * @param key la clé à chercher
     * @return la valeur associée s'il y en a une, null sinon
     */
    public V get(K key);

    /**
     * Enlève les associations de cette table pour la clé spécifiée
     * PREREQUIS:
     * - il existe bien une association pour la clé dans la table
     * GARANTIES:
     * - la taille de la table diminue de 1
     * - la clé n'est plus associée
     * @param key la clé spécifiée
     * @throws KeyNotFoundException si la clé n'a pas d'association dans la table
     */
    public void remove(K key) throws KeyNotFoundException;
}
```

```

/**
 * Teste si la clé spécifiée est présente dans cette table de hachage
 * @param k la clé cherchée
 * @return true si trouvée, false sinon
 */
public boolean containsKey(K k);

/**
 * Récupère la taille de cette table de hachage en nombre d'associations
 * enregistrées.
 * @return la taille de la table en nombre d'éléments
 */
public int getSize();
}

```

Question 2.1 : Chaînage – architecture de classe

L'implémentation que nous proposons dans cet exercice repose sur le principe du chaînage. On définit la classe `ChainingHashTable` qui implémente l'interface ci-dessus pour les mêmes paramètres de type. On déclare les attributs suivants:

- attribut table de type `ArrayList<ChainingHashEntry<K,V>>` (cf. ci-dessous pour la classe `ChainingHashEntry`)
- attribut entier `nbElements` (**remarque:** son accesseur est `getSize()`)
- une constante entière `TABLE_SIZE` de valeur 256 et qui donne la taille par défaut de la table

L'unique constructeur de la classe ne prend aucun paramètre. La taille choisie par défaut pour la liste `table` est de `TABLE_SIZE`. On initialisera à `null` tous les éléments de cette `ArrayList`.

Dans cette question, on donnera uniquement le code des méthodes `getSize()` et `containsKey()` de l'interface `HashTable` des tables de hachage, on les implémentera dans les questions suivantes. On donne en revanche le code de la classe `ChainingHashEntry` mais sans les commentaires:

```

package hash.chaining;
import hash.framework.HashObject;

public class ChainingHashEntry<K extends HashObject<?>,V> {
    private final K key;
    private V value;
    private ChainingHashEntry<K,V> next;

    public ChainingHashEntry(K key, V value) {
        this.key = key;
        this.value = value;
        this.next = null;
    }

    public K getKey() {
        return key;
    }

    public V getValue() {
        return value;
    }

    protected void setValue(V value) {
        this.value = value;
    }
}

```

```

    /* package */ ChainingHashEntry<K,V> getNext() {
        return next;
    }

    /* package */ void changeNext(ChainingHashEntry<K,V> next) {
        this.next = next;
    }

    /* package */ boolean append(ChainingHashEntry<K,V> entry) {
        ChainingHashEntry<K,V> current = this;
        if(current.getKey().equals(entry.getKey())) {
            // duplicated
            return false;
        }
        while(current.next!=null) {
            current = current.next;
            if(current.getKey().equals(entry.getKey())) {
                // duplicated
                return false;
            }
        }
        current.next = entry;
        return true;
    }

    /* package */ ChainingHashEntry<K,V> lookup(K key) {
        ChainingHashEntry<K,V> current = this;
        do {
            if(current.getKey().hash()!=key.hash())
                current=current.next;
            else if(!current.getKey().equals(key))
                current = current.next;
            else
                return current;
        } while(current!=null);
        return null;
    }

    /* package */ ChainingHashEntry<K,V> remove(ChainingHashEntry<K,V> prev, K key) {
        ChainingHashEntry<K,V> previous = prev;
        ChainingHashEntry<K,V> current = this;
        ChainingHashEntry<K,V> first = current;
        do {
            if(current.getKey().hash()!=key.hash()) {
                previous = current;
                current = current.next;
            } else if(!current.getKey().equals(key)) {
                previous = current;
                current = current.next;
            } else {
                if(previous==null)
                    return current.next;
                else {
                    previous.next = current.next;
                    return first;
                }
            }
        } while(current!=null);
        return first;
    }
}

```

Question 2.2 : méthode d'ajout

Ajouter à la classe `ChainingHashTable` une implémentation de la méthode `put` (cf. interface `HashTable`). Cette méthode permet d'ajouter une association dans la table de hachage. L'algorithme est le suivant:

- on vérifie le prérequis de la clé non nulle et on lève une exception du type `IllegalArgumentException` (définie dans `java.lang` et sous-classe de `RuntimeException`, il est donc inutile d'ajouter ce type d'exception à la déclaration `throws` de la méthode).
- on commence par calculer la valeur de hachage `hash` de la clé cherchée
- on calcule ensuite un `indice` qui est la valeur de hachage `hash` modulo la taille de la table interne (attribut `table`). Ainsi on est sûr que l'indice correspond bien à un élément du tableau `table` (pour la taille par défaut de 256, l'indice est donc entre 0 et 255).
- Si aucune association n'est déjà enregistrées pour l'indice dans la table (i.e. `table.get(indice)` est `null`) alors on crée une nouvelle `ChainingHashEntry` pour l'association à ajouter et on place cet entrée à l'indice spécifié, le travail est terminé.
- Sinon, si des associations sont déjà présentes pour `indice`, il y a collision. On invoque pour cela la méthode `append` (cf. classe `ChainingHashEntry`) sur l'élément à la position `indice` dans le tableau `table`. Si le retour de la méthode indique une duplication, on lève l'exception adéquate (cf. interface `HashTable` ci-dessus).

Question 2.3 : méthode d'accès

Donner la définition de la méthode `get` pour accéder à la valeur associée à la clé spécifiée (cf. interface `HashTable`).

Question 2.4 : méthode de retrait

Donner la définition de la méthode `remove` pour enlever une association de la table.

3. EXERCICE 3 : TESTS UNITAIRES ET DIAGRAMMES OBJETS

Nous implémentons dans cet exercice les tests unitaires pour la classe `ChainingHashTable`, sous la forme d'une classe de test junit nommée `ChainingHashTableTest` (remarque: on ne demande pas de définir cette classe).

Question 3.1 : test pour la méthode d'ajout

La méthode suivante décrit le test unitaire de la méthode `put` (cf. question 2 exercice 2):

```
public void testPut() {
    ChainingHashTable<HashString,Integer> htable = new ChainingHashTable<HashString,Integer>();
    assertTrue(htable.getSize()==0);
    HashString key = new HashString("key");
    Integer num = new Integer(12);
    // PREREQUIS: la clé n'est pas null
    assertTrue(key!=null);
    int oldSize = htable.getSize(); // pour tester la garantie 1
    try {
        htable.put(key,num);
    } catch (IllegalArgumentException e1) {
        assertTrue(false); // prérequis validé: on ne passe pas par ici !
    } catch (DuplicateEntryException e2) {
        assertTrue(false); // prérequis validé: on ne passe pas par là non plus !
    }
    // GARANTIE: la taille de la table augmente de 1
    assertTrue(htable.getSize()==oldSize+1);
    // GARANTIE: la nouvelle valeur est bien associée à la clé
    assertTrue(htable.get(key)==num);
}
```

Ce test est-il selon-vous validé ou non ? Donner une représentation de la mémoire à l'issue de l'exécution du test (i.e. diagramme des « patates »).

Question 3.2 : test pour la méthode de retrait

En vous inspirant de la question précédente, définir une méthode de test unitaire complète pour la méthode `remove` définie à l'exercice 2 (question 4). Dans ce test unitaire, on commencera par ajouter un ou deux éléments dans la table, puis on les enlèvera en vérifiant que les prérequis et les garanties de la méthode `remove` sont vérifiés.