Master 1 TIIL - Algorithmique et Programmation Objet

Alain Plantec

La table de hash fermée

Une table de hash est une collection dans laquelle on stocke un ensemble de couples (clé,valeur). La clé et la valeur peuvent être de n'importe quelle classe. La clé d'un couple est unique dans toute la collection. Autrement dit, on ne peut pas trouver deux fois la même clé dans une même table de hash.

Nous avons vu en cours que, contrairement à la mise en oeuvre ouverte, une mise en oeuvre fermée est calibrée à sa création pour un certain nombre de couples. On a donc un tableau de couples à une seule dimension de taille *N*. La taille est définie par construction.

Lors d'une insertion, pour une clé *k*, la position donnée par $p = abs(k.hashCode()) \% N$ constitue le premier indice de stockage à partir duquel une recherche de case vide est effectuée. Le nouveau couple est stocké dans la première cas libre à partir de la position p. Une case libre est une case qui contient *null*.

Pour une recherche, *p* donne la position à partir duquel il faut rechercher le couple.

Le tableau est traité de façon circulaire par les algorithmes d'insertion, de recherche ou de suppression : quand on arrive à la fin du tableau, alors on continue depuis le début jusqu'à atteindre la position *p* de départ.

Si la position de départ est atteinte cela signifie, pour une insertion, que la table est pleine, et pour une recherche ou une suppression, que la clé n'a pas été trouvée.

Exercice 1

Il est demandé de développer et de tester la classe *ClosedHashDict* qui met en oeuvre l'interface *Map* donnée en annexe.

Exercice 2

Comparez la rapidité de la table de hash fermée par rapport la la table de hash ouverte pour les opérations d'ajout et de suppression d'une entrée. Pour cela, vous produisez des courbes à l'aide d'un outil pour afficher des graphiques comme Gnuplot (http://www.gnuplot.info).

Annexes

Interface *Map*

```
public interface Map<K,V> {
    // creer un couple ('key','value') et le stocker dans la table
    V put(K key, V value);
    // retrouver la valeur associee a la cle 'key'
    V get(K key);
    // supprimer le couple dont la cle est 'key'
    V remove(K key);
    // tester l'existance d'un couple dont la cle est 'key'
    boolean containsKey(K key);
    // tester l'existance d'un couple dont la valeur est 'value'
    boolean containsValue(V value);

    int size();
    boolean isEmpty();

    // vider la table
    void clear();

    // retourner un Set contenant toutes les cles
    Set<K> keySet();
    // retourner une Collection contenant toutes les valeurs
    Collection<V> values();
    // retourner un Set contenant tous les couples
    Set<Map.Entry<K,V>> entrySet();

    // la classe interne decrivant un couple
    public interface Entry<K,V> {
        // retourner la cle
        K getKey();
        // retourner la valeur
        V getValue();
        // modifier la valeur
        V setValue(V value);
    }
}
```

Exemple d'itération dans un *Map*

```
public static void printMap(Map mp) {
    Iterator it = mp.entrySet().iterator();
    while (it.hasNext()) {
        Map.Entry pairs = (Map.Entry)it.next();
        System.out.println(pairs.getKey() + " = " + pairs.getValue());
    }
}
```