SDD

9

```
public void add (E el, int rang) {
    LeftRightNode <E> nouveau = new LeftRightNode ();
    nouveau.setValeur (el);
    this.positionnerCurseur (rang);
    nouveau.setLeft (curseur.getLeft());
    nouveau.setRight (curseur);
    curseur.setLeft (nouveau);
    curseur.getLeft().setRight (nouveau);
    size++;
}
```

SDD
10

Chapitre Tables.

I] Spécification algébrique.

Une table est une fonction qui associe une valeur à une clé (entrée).

Les opérations principales :
- consulter la valeur attachée à une clé.
- ajouter une clé et sa valeur associée (ajouter un couple $< k, v >$)
- vérifier l'existence d'une clé.
- supprimer une clé et la valeur associée.

Opérations supplémentaires :
- nombre de clés
- la table est-elle vide ?

opérations →
Type Dictionary $< k, v >$
empty : → Dictionary$<k,v>$ -- crée une table vide.
add : Dictionary $< k, v >$ × k × v → Dictionary $< k, v >$.
delete : Dictionary $< k, v >$ × k → Dictionary $< k, v >$.
-- supprime l'élément de clé donnée.
has : Dictionary $< k, v >$ × k → booléen.
value : Dictionary $< k, v >$ × k → v.
-- valeur attachée à la clé
size : Dictionary $< k, v >$ → entier
isEmpty : Dictionary $< k, v >$ → booléen.

Préconditions :

add (t, k, v) est définie ssi non has (t, k).
value (t, k) est définie ssi has (t, k).

Axiomes :

has (empty(), k) = false.
has (add (t, k, v), k) = true
has (add (t, k1, v), k2) = has (t, k2) si k1 ≠ k2.
has (delete (t, k, v), k) = false
has (delete (t, k1, v), k2) = has (t, k2) si k1 ≠ k2
value (add (t, k, v), k) = v
value (add (t, k1, v), k2) = value (t, k2) si k1 ≠ k2.
value (delete (t, k1, v), k2) = value (t, k2) si k1 ≠ k2
size (empty()) = 0
size (add (t, k, v)) = 1 + size (t).
size (delete (empty(), k)) = 0
size (delete (add (t, k, v), k)) = size (t)
size (delete (add (t, k1, v), k2)) = size (delete (t, k2)) + 1 si t1 ≠ t2.
isEmpty ... [même chose que size].

II) Tests.

```
public abstract class TestDico {
    protected Dictionary<Integer, String> dico;

    testHas();
    testValue();
    ⋮
}

public testHas() {
    setUpVide();
}
public abstract setUpVide();
```

Pour la vérification des préconditions,
=> class DecorateurDico
qui implante l'interface Dictionary en la décorant avec des assert prenant en compte les ensembles de définitions des fonctions.

III) Implantation de Dictionary.

$< K, V >$: implantation par liste chaînée : $< k_1, v_1 > \rightsquigarrow \dots \rightsquigarrow < k_n, v_n >$.

① Recherche d'un élément : $O(n)$ maxi soit n comparaisons.

Si la liste est ordonnée (ordre sur les clés) :
n comparaison dans le pire des cas.

② Opération add.

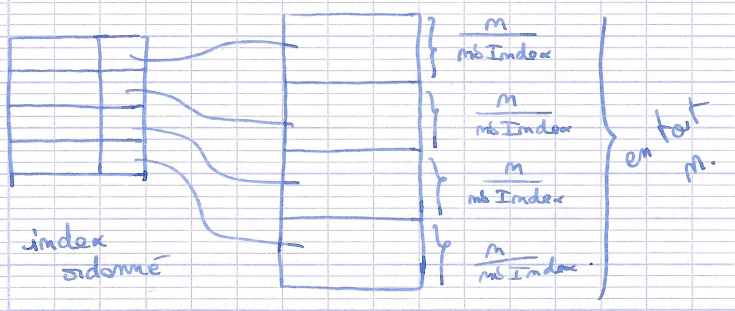
Si la liste n'est pas ordonnée, on fait l'adjonction en tête en temps constant. Les autres opérations : has, value ne sont pas "efficaces".

Diviser pour régner:

Découper la table en sous-tables.

n éléments:

nbIndex : nombre d'index.



Recherche:

$$\log(nbIndex) + \log\left(\frac{n}{nbIndex}\right)$$

Hash-code: idée de sous-table et à partir de la clé, détermine la sous-table

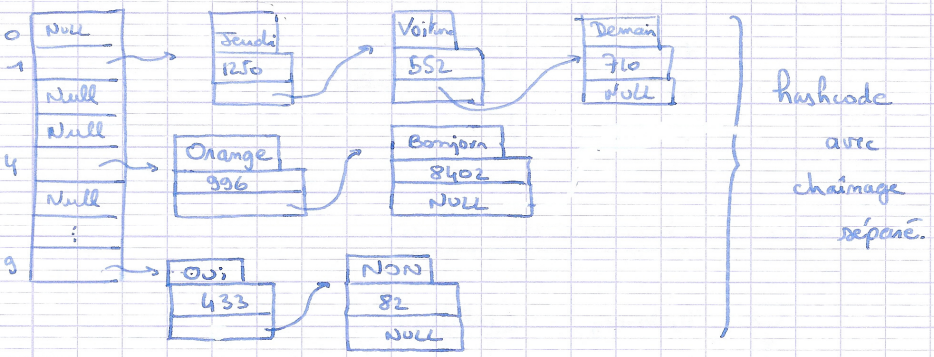
SDD

12

clé valeur

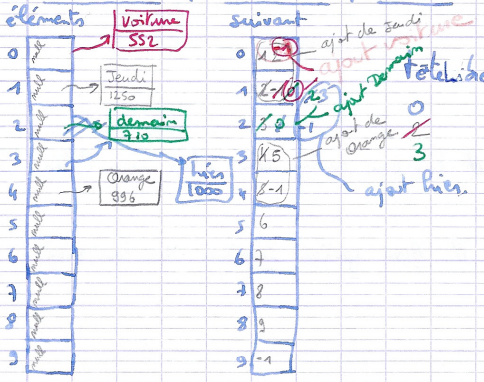
< jeudi, 1250 >
< voiture, 552 >
< Demain, 710 >
< Orange, 996 >
< Bonjour, 8402 >
< Oui, 433 >
< Non, 82 >

Hashcode (jeudi) = 1
(Voiture) = 1
(Demain) = 1
(Orange) = 4
(Bonjour) = 4
(Oui) = 9
(Non) = 9



Hashcode avec chaînage séparé.

Autre représentation possible.



Le tableau éléments est utilisé pour stocker les éléments de la table.

Le tableau suivant modélise les chaînages
- des différentes sous-tables
- des places vides.

tetelibre et l'indice de la première case libre

tetelibre : 0 ⇝ 1 ⇝ 2 ⇝ 3 ... ⇝ 9.

ajout de Jeudi ← tetelibre : 0 ⇝ 2 ⇝ 3 ... ⇝ 9. et 1 dans tableau.

ajout de Orange ← tetelibre : 0 ⇝ 2 ⇝ 3 ⇝ 5 ⇝ 6 ⇝ 7 ⇝ 8 ⇝ 9 et 1 et 4.

ajout voiture ← tetelibre : 2 ⇝ 3 ⇝ 5 ⇝ 6 ⇝ 7 ⇝ 8 ⇝ 9 et 1 ⇝ 0 et 4.

tetelibre : 3 ⇝ 5 ⇝ 6 ⇝ 7 ⇝ 8 ⇝ 9 et 1 ⇝ 2 ⇝ 0 et 4 ⇝

On remarque que la case d'indice [2] est occupée par un élément < Demain, 710 > dont le hashcode est [1] cela signifie qu'il n'y a pas encore d'éléments de hashcode [2] dans la table, on va donc "créer" une sous-table dont les éléments ont comme hashcode 2. L'élément < Demain, 710 > est appelé un intrus. On va déplacer l'intrus rendre sa place à l'élément < Hier, 1000 >.

d'où : 5 ⇝ 6 ⇝ 7 ⇝ 8 ⇝ 9
1 ⇝ 3 ⇝ 0
4
2
} 3 sous-tables.

Java : hiérarchie des classes :

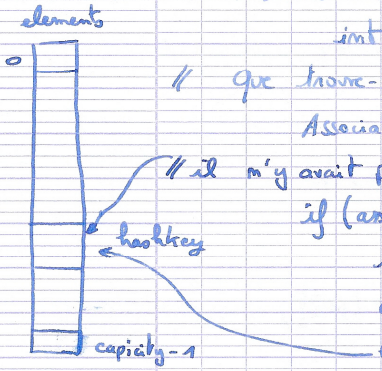
```
public class HashLinkedDictionaryAgcl
                extends HashLinkedDictionary {

    protected int teteLibre;
// on retire l'element placé de la zone libre.
    protected void allouer(int place) {
        if (place == teteLibre)
            teteLibre = suivant[teteLibre];
        else {
            int pl = teteLibre;
            while (suivant[pl] != place) {
                pl = suivant[pl];
            }
            suivant[pl] = suivant[suivant[pl]];
        }
    }

// fournit une place dans la zone libre.
    protected int allouer() {
        int placeLibre = teteLibre;
        teteLibre = suivant[teteLibre];
        return placeLibre;
    }
}
```

fonction add (ajout d'un élément dans la table)
↳ dans la class HashLinkedDictionary

class Association
(avec un attribut key et un value).

Dans HashLinkedDictionary

```
public void add (Object key, Object value) {
        int placeLibre;
// On crée la nouvelle association a ranger
        Association nouvAssoc = new Association(key, value);
// HashCode de la clé de l'élément à ajouter.
        int hashKey = hashCode(key)
// que trouve-t-on à cette place ?
        Association assoc = elements[hashKey];
// il n'y avait pas encore d'association à ce hashKey => création sous-table
        if (assoc == null) {
            int place = hashKey;
            allouer(place);
            elements[place] = nouvAssoc;
            suivants[place] = -1;
        }
        else { // il y a déjà un élément
            placeLibre = allouer();
            int hashCode = hashCode(assoc.key()) // on récupère le hashCode.
            if (hashKey == hashAssoc) { // il existe déjà un él. avec le même hashCode
                elements[placeLibre] = nouvAssoc; // => pas un intrus,
// il faut effectuer le chaînage en 2e position.   // élément de la même sous-table
                suivants[placeLibre] = suivants[hashKey];
                suivants[hashAssoc] = placeLibre;
            }
```



```
            else {  // la place est occupée par un intrus
                    // on déplace l'intrus à placeLibre.
                elements[placeLibre] = assoc;
                suivants[placeLibre] = suivant[hashKey];
                //mise à jour du chaînage après l'intrus.
                // on cherche le prédécesseur de l'intrus.
                int predecIntrus = hashAssoc;
                while (suivant(predecIntrus) != hashKey) {
                    predecIntrus = suivants[predecIntrus];
                }
                suivants[predecIntrus] = placeLibre;
                  // la place pour la clé à insérer est maintenant libre.
                elements[hashKey] = nouvAssoc;
                suivants[hashKey] = -1;
            }
        } // else
        size++;
} // add.
```

Recherche de la place d'une clé
(on retourne -1 si la clé n'existe pas).

```
protected int rechercher(Object key) {
        // calcul du hashCode de la clé
        int hashKey = hashCode(key);
        Association assoc = elements[hashKey];
        place = -1;
        if ((assoc != null) && !(hashKey == hashCode(assoc.key()))) {
            place = hashKey;
```

```
            while (place != -1 && !key.equals(elements[place].key())){
                place = suivant[place];
            }
            return place;
        } // if
    } // rechercher.
```

Supprimer un élément donné connaissant sa clé.

```
public void delete (Object key) {
    int l = -1;     // place à libérer.
    int h = hashCode(key); // hashCode de la clé à supprimer
    if (elements[h] != null) {
        h2 = hashCode(elements[h].key());
        if (h != h2); // pas d'élément avec ce hashCode
            // rien
        else if (key.equals(elements[h].key())) {
                // suppression en tête
                l = suivants[h];
```