IFT1025 automne 2009

Miklós Csűrös

17 novembre 2009

Structures de données

type abstraite (interface) : liste (`java.util.List`), ensemble (`java.util.Set`), dictionnaire (`java.util.Map`)
structure de données (implantation) : liste par `ArrayList`, `Vector` et `LinkedList`, ensemble par `HashSet`, dictionnaire par `Hashtable` et `TreeMap`.

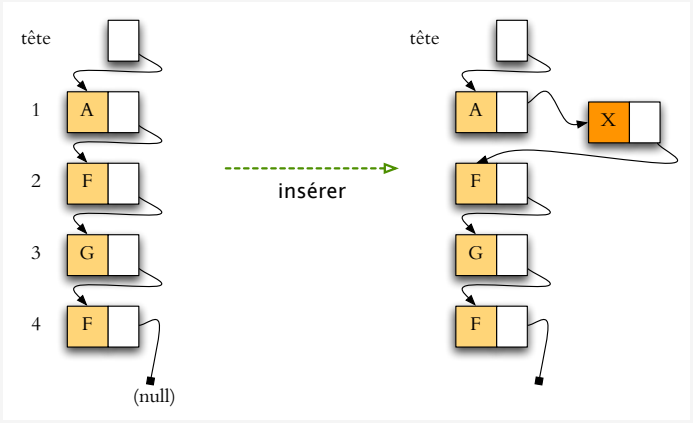
Liste

opérations sur une liste `List`⟨`E`⟩ : insertion **add**(`E` *element*), insertion au milieu **add**(`int` *indice*, `E` *element*) suppression au milieu **remove**(`int` *indice*), accès **get**(`int` *indice*), **contains**(`E` *element*), et parcours **iterator**().

implantation par tableau `E`[] : manipulation lente (décalage des éléments lors d'insertion ou suppression) mais accès rapide

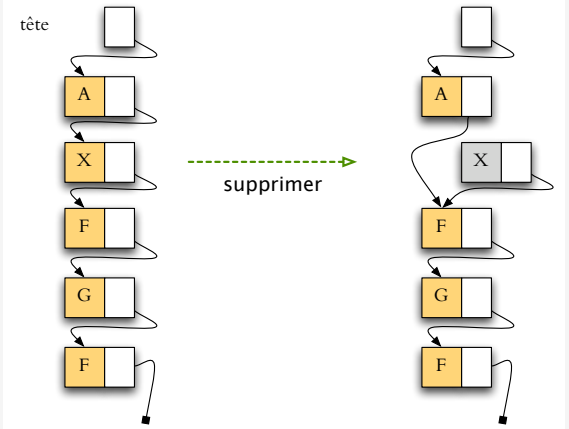
implantation par **liste chaînée** : manipulation rapide mais accès lent (parcours de la chaîne complète)

```
public class LinkedList // liste chaînée
{
    private static class Noeud // liste = séquence de noeuds
    {
        private Object valeur;
        private Noeud prochain;
    }

    private Noeud premier; // tete (premier element) de la liste

    public LinkedList(){Noeud = null;} // liste nulle

    private Noeud newNode(Object element) // "emballage" d'un element
    {
        Noeud N = new Noeud();
        N.valeur = element;
        N.prochain = null;
        return N;
    }
```



Insertion se fait par l'affectation de deux références.

```
public void insertFirst(Object e)
{
  Noeud N = newNode(e);
  N.prochain = premier;
  premier = N;
}
private void insert(Noeud prec, Object e)
{
  Noeud N = newNode(e);
  N.prochain = prec.prochain;
  prec.prochain = N;
}
```



Suppression se fait par l'affectation d'une seule référence.

```
public void deleteFirst()
{
  if (premier==null)
   throw new NoSuchElementException();
  premier = premier.prochain;
}
private void delete(Noeud prec)
{
  if (prec.prochain==null)
   throw new NoSuchElementException();
  prec.prochain=prec.prohain.prochain;
}
```

Classes génériques

Définition d'une classe générique : avec types paramétriques

```
public class LinkedList<E> // E: type d'éléments (paramétrique)
{
    private class Noeud // ne peut pas être statique à cause de E
    {
        private E valeur;
        private Noeud prochain;
    }

    private Noeud premier; // tete (premier element) de la liste

    public LinkedList(){Noeud = null;} // liste nulle

    private Noeud newNode(E element) // "emballage" d'un element
    {
        Noeud N = new Noeud();
        N.valeur = element;
        N.prochain = null;
        return N;
    }
    ...
}
```

Type spécifié lors d'instanciation :

```
    LinkedList<String> liste = new LinkedList<String>();
```

Les types sont vérifiés lors de compilation mais l'information n'est pas retenue dans le code : type E n'est pas disponible lors d'exécution (donc p.e., `x instanceof E` n'est pas possible)

Tableau de hachage

Dictionnaire. (ou tableau de symboles) associations clé-valeur. Les clés sont uniques et permettent l'accès rapide aux valeurs.

Opérations sur `Hashtable`$\langle K, V\rangle$: nouvelle association **put**(`K` *clé*, `V` *valeur*), recherche **containsKey**(`Object` *clé*), accès **get**(`Object` *clé*).

Fonction de hachage : permet la construction d'un indice à partir d'une instance de clé. En Java, on utilise la méthode `int` **hashCode**() définie dans la classe `Object`.

Tableau de hachage : utiliser un tableau de listes chaînées. Deux clés a, b sont sur la même liste dans un tableau de taille T ssi `a.hashCode() % T == b.hashCode() % T`.

```
public class Hashtable<K,V>
{
    private class Noeud
    {
        private K cle;
        private V valeur;
        private Noeud prochain;
    }

    private Noeud[] table;

    public Hashtable(int taille)
    {
        table = new Noeud[taille];
    }

    private Noeud newNode(K cle, V valeur) // "emballage" d'un element
    {
        Noeud N = new Noeud();
        N.cle = cle;
        N.valeur = V;
        N.prochain = null;
        return N;
    }

    public void add(K cle, V valeur)
    {
        Noeud N = newNode(cle, valeur);
        int h = cle.hashCode();
        if (h<0) h=-h;
        int i = h % table.length;
        // insertion sur liste i
        N.prochain = table[i];
        table[i] = N.prochain;
    }
```

Recherche de clé.

```
    // solution par récursion
    private Noeud find(Noeud N, K cle) // recherche sur une liste chaînée
    {
        if (N==null || N.cle.equals(cle)) return N;
        return find(N.prochain, cle); // recursion
    }
    public boolean containsKey(K cle)
    {
        int h = cle.hashCode();
        if (h<0) h=-h;
        int i = h % table.length;
        return find(table[i], cle) != null;
    }
```

hashCode. Accès ou recherche dans le tableau de hachage : pour un argument a (clé recherchée), on doit trouver une telle clé b dans le tableau que soit $a = b =$ `null`, soit `a.equals(b)`. Spécification Java : «il faut toujours assurer que si `a.equals(b)`, alors `a.hashCode()==b.hashCode()`»

```
public class DeuxValeurs
{
    private String x;
    private Integer y;

    public boolean equals(Object a)
    {
        if (a instanceof DeuxValeurs)
        {
            DeuxValeurs dva = (DeuxValeurs) a;
            return x.equals(dva.x) && y.equals(dva.y);
        } else return super.equals(a);
    }

    public int hashCode()
    {
        return x.hashCode()*37 + y.hashCode();
    }
}
```

Efficacité. La performance d'un tableau de hachage dépend de la longueur des listes chaînées : on veut éviter des **collisions**. Pour une fonction de hachage h, une collision entre $x \neq y$ se produit quand $h(x) = h(y)$. Avec une bonne fonction de hachage, les collisions sont aléatoires. «Bonne» fonction de hachage :

- ⋆ p.e. $h(x) = x \bmod M$ est bonne avec M un nombre premier loin de 2^k et 10^m
- ⋆ combinaison $ah_1(x) + bh_2(x)$ est bonne avec a, b des nombres premiers entre eux.