Fiche résumé nº 1 – Les listes

1. Introduction

Définition : une liste est une suite finie d'éléments du même type.

N.B. :

- suite => il y a donc une notion d'ordre (1er élément, 2e élément, etc.)
- même type => pas d'éléments hétérogènes

En pratique on voudrait pouvoir utiliser ce type de données comme un type élémentaire du langage : avoir l'outillage pour rédiger des algorithmes manipulant cette SDD « Liste » au moyen des opérations dont elle est dotée (ajout, suppression, etc.).

Fonctions du TAD Liste :

```
sorte Liste
utilise Booléen, Objet
fonctions
    new :                ⟶ Liste
    estVide :    Liste   ⟶ Booléen
    entête :     Liste   ⟶ Liste
    enqueue :    Liste   ⟶ Liste
    succ :       Liste  ⟶/ Liste
    préd :       Liste  ⟶/ Liste
    estSorti :   Liste   ⟶ Booléen
    valec :      Liste  ⟶/ Objet
    modifec :    Liste × Objet  ⟶/ Liste
    ajouterG :   Liste × Objet  ⟶/ Liste
    ajouterD :   Liste × Objet  ⟶/ Liste
    ôterec :     Liste  ⟶/ Liste
    vider :      Liste   ⟶ Liste

préconditions
    Soit une Liste ℓ , soit un Objet e
    pré(valec(ℓ))       = non estSorti(ℓ)
    pré(modifec(ℓ))     = non estSorti(ℓ)
    pré(oterec(ℓ))      = non estSorti(ℓ)
    pré(pred(ℓ)  = non estSorti(ℓ)
    pré(succ(ℓ)  = non estSorti(ℓ)
    pré(ajouterG(ℓ,e)) = estVide(ℓ) ou non estSorti(ℓ)
    pré(ajouterD(ℓ,e)) = estVide(ℓ) ou non estSorti(ℓ)
```

Crée une liste vide

4 opération pour déplacer la position de l'élément courant (ec) = la position actuellement observée

Indique si la position de l'ec est arrivée hors limite

Consulter ou modifier la valeur de l'ec

Ajouts à gauche ou à droite de l'ec

N.B. En rester à cette ébauche ne fournissant pas la partie axiomes n'est pas bien :-) Il est vrai que la sémantique de certaines opérations est évidente, mais d'autres mériteraient d'être clarifiées par leurs axiomes correspondants.

N.B. Les préconditions sur ajouterG et ajouterD plus élaborées que les autres permettent ainsi de ne pas bloquer lors du tout premier ajout, sinon le TAD serait inutilisable.

Exemple d'utilisation de ce TAD :

```
algorithme afficherListe(ℓ: Liste)
    ℓ ← entête(ℓ)
    tant-que non(estSorti(ℓ)) faire
          afficher(valec(ℓ))
          ℓ ← succ(ℓ)
    fin
fin
```

2. Mise en œuvre (Java)

Dans la plupart des langages (y compris Java) ce type n'est absolument pas un type élémentaire de base, il faut le programmer afin d'obtenir les opérations souhaitées.

=> On met en place une interface Java qui traduit les fonctions du TAD, que l'on va ensuite pouvoir mettre en œuvre dans différentes implémentations.

```
/** Une liste d'objets. */
public interface Liste {
    /** Indique si la liste est vide. */
    boolean estVide();
    /** Positionne l'élément courant en début de liste. */
    void entête();
    /** Fournit la valeur de l'élément courant. */
    Object valec() throws SortiException;
    ...
}
```

3. Représentation contiguë (tableau) : ListeTabulée

1re solution => ranger simplement les éléments dans un tableau, les uns derrière les autres.

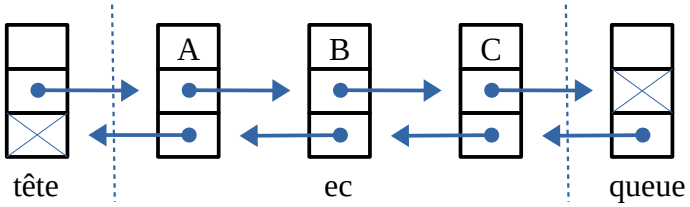
```
public class ListeTabulée implements Liste {
    private final static int N = 10;
    private Object[] éléments; // stockage des éléments
    private int taille; // nombre d'éléments actuels
    private int ec; // position de l'élément courant

    public ListeTabulée() {
          éléments = new Object[N];
          taille = 0;
          entête();
    }

    public boolean estVide() {
          return taille==0;
    }

    public void entête() {
          ec = 0;
    }

    ...
}
```

N.B. Le problème évident est la nécessité de tout décaler à chaque insertion/suppression, ce qui est inefficace pour des tailles de listes importantes.
N.B. La taille *N* initiale devra croître car le TAD n'envisage pas de restrictions « si plein ».

4. Représentation chaînée

Autre solution => relier des éléments stockés en mémoire, mais pas forcément contigus (pour éviter les décalages en cas d'ajout/suppression).

Cas classique : double chaînage (par « pointeurs », ou références Java) avec deux éléments fictifs marqueurs d'extrémités (= faux élément de tête et faux élément de queue).



- Suppression/insertion efficaces (pas de décalages).
- Suppression/insertion facilitées (un seul algorithme car les éléments fictifs permettent d'éviter les cas particuliers).

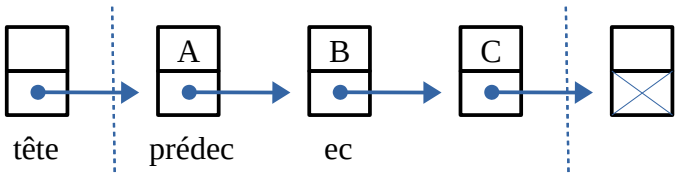
```
public class ListeDoublementChaînée implements Liste {
    private Maillon tête, ec, queue;

    // Classe auxiliaire à usage interne
    private class Maillon {
           Object val;  // valeur du maillon
           Maillon s;   // suivant
           Maillon p;   // précédent
    }

    public ListeDoublementChaînée() {
           tête  = new Maillon();
           queue = new Maillon();
           tête.s  = queue;
           queue.p = tête;
           entête();
    }

    // ...
}
```

N.B. La classe Maillon étant pour notre usage exclusif, on s'épargne les *getters* & *setters*. Cependant, ici il serait plus adéquat d'ajouter static à la définition de la classe Maillon.

5. Cas particuliers

a) Simple chaînage si traitement unidirectionnel

Dans certaines applications on n'énumère la liste que dans un sens : le chaînage inverse ne sert a rien. Prenons le cas du seul sens utilisé vers les suivants :



Alors, pour parvenir à gérer le chaînage (lors d'ajout/suppression) il est commode d'ajouter un attribut qui indique, à chaque instant, le prédécesseur de l'ec : c'est le rôle de prédec.

De plus, on se rend compte qu'on peut alors se passer de mémoriser la position de queue.

```
class ListeSimplementChaînée implements Liste {
    private Maillon tete, predec, ec;

    // Classe auxiliaire à usage interne.
    private static class Maillon {
          Object val;  // valeur du maillon
          Maillon s;   // suivant
    }

    // ...
}
```

Concernant les méthodes devenues hors sujet par rapport à ce traitement unidirectionnel :

- Soit on arrive à les implémenter mais de manière coûteuse
- Soit (en Java) on lance l'exception `UnsupportedOperationException` (qui dérive de `RuntimeException` : des exceptions que l'on n'est pas obligé de mentionner dans la signature, donc compatible avec l'interface imposée). C'est la technique utilisée par Java pour ses propres « méthodes optionnelles » documentées ainsi dans certaines interfaces.

Bilan :

- Implémentation simplifiée
- Économie de mémoire
- On peut bien sur conserver seulement le second chaînage si c'est celui-là le sens utile

b) Traitement récursif

Souvent il est agréable de pouvoir traiter des listes récursivement :

On peut ajouter classiquement 2 méthodes à l'interface Liste :

```
Object tete(); // l'élément de tête
Liste reste(); // la sous-liste sans l'élément de tête
```

Exemple d'utilisation :

```
public void afficherListe(Liste l) {
      if( !l.estVide() ) {
            System.out.println(l.tete());
            afficherListe(l.reste());
      }
}
```

On ajoute alors aussi un constructeur pour fabriquer la liste tete + reste :

```
Liste l = new MaListe(tete,reste);
```