

# TD 3 et 4 – Listes

## Structures de données (IF 2)

Un *modèle de données* est une abstraction utilisée pour décrire des problèmes. Cette abstraction spécifie les valeurs qui peuvent être attribuées aux objets, et les opérations valides sur ces objets. La *structure de données*, quant à elle, est une construction du langage de programmation, permettant de représenter un modèle de données.

Nous nous proposons à travers ce TD d'étudier la *liste*, un modèle de données classique qui permet de décrire une séquence d'objets contenant chacun une valeur d'un type donné. À la base, on peut définir le type  $L =$  liste de  $V$  à partir d'un type  $V$  donné, comme une suite finie d'éléments de  $V$ . Les opérations basiques que l'on peut définir sur une telle liste sont :

- *estVide* :  $L \rightarrow Bool$  indique que la liste est vide
- *premier* :  $L \rightarrow V$  donne la valeur de la tête de liste
- *suivant* :  $L \rightarrow L$  donne la suite de la liste
- *ajouteEnTete* :  $L \times V \rightarrow L$  ajoute une valeur en tête de liste

En plus de ces opérations de base, on peut souhaiter définir des opérations plus complexes qui s'expriment en termes de combinaison de ces opérations de base comme l'adjonction ou la suppression à un endroit quelconque, la recherche d'une valeur dans la liste, voire l'adjonction et la suppression en queue ou le tri ... Le premier type de représentation auquel on pense habituellement est le tableau, que nous avons déjà vu dans les séances précédentes. Dans ce TD, nous manipulerons uniquement des listes d'entiers ( $V = \mathbb{N}$ ).

## 1 Listes chaînées

La liste chaînée est une structure de données que l'on retrouve fréquemment en informatique. Elle nécessite de représenter chaque élément de la liste par un couple (*valeur*, *suivant*), désignant respectivement la *valeur* au point courant et le pointeur sur le chaînon *suivant*. En Java, cela s'écrit très aisément puisque, comme nous l'avons vu, toutes les variables dont le type est défini par une classe sont des *références*.

**Exercice 1** En quoi les listes chaînées sont-elles plus intéressantes, sur le plan de complexité algorithmique, par rapport aux tableaux ?

### 1.1 La classe Cellule

La classe *Cellule* va représenter un *maillon de la chaîne* :

```
private class Cellule {
    private int valeur;
    private Liste suivant; // suite de la liste
}
```

**Exercice 2** Définir le constructeur par défaut qui initialisera la valeur de l'élément à 0.

**Exercice 3** Définir un second constructeur qui prend en argument un entier destiné à l'initialisation de la valeur de l'élément.

**Exercice 4** Écrire les modifieurs et accesseurs nécessaires.

## 1.2 La classe Liste

Nous pouvons à présent définir la classe *Liste* représentant une liste chaînée d'entiers.

```
public class Liste {  
  
    private Cellule tete; // l'élément en tête de liste  
  
    // méthodes publiques  
  
    public boolean est_vider() {  
        return (tete == null);  
    }  
  
    public int tete() {  
        return tete.getValeur();  
    }  
  
    public Liste queue() {  
        return tete.getSuivant();  
    }  
}
```

**Exercice 5** Définir le constructeur par défaut, qui crée une liste vide.

**Exercice 6** Définir un constructeur qui prend en argument un entier  $n$  et une liste  $L$  et crée la liste avec  $n$  en tête et  $L$  en queue.

**Exercice 7** Écrire une méthode *dernier* qui renvoie la valeur du dernier élément d'une liste.

**Exercice 8** Écrire une méthode *longueur* qui calcule le nombre d'éléments d'une liste.

**Exercice 9** Écrire une méthode publique *vider* qui vide la liste.

**Exercice 10** Définir la méthode publique d'affichage *toString* qui renvoie une chaîne de caractères correspondant au contenu de la liste.

**Exercice 11** Définir une méthode publique *ajouteAprès* qui prend en argument deux entiers  $n$  et  $i$ . Cette méthode devra tout d'abord créer un nouvel élément contenant l'entier  $n$  puis l'insérer après l'élément d'indice  $i$  dans la liste (par convention, on pose que l'indice du premier élément est 0).

**Exercice 12** Définir maintenant une méthode publique *ajouteAvant* qui prend en argument deux entiers  $n$  et  $i$  et insère un élément de valeur  $n$  dans la liste avant l'élément d'indice  $i$ .

**Exercice 13** Écrire une méthode publique *enlever* qui prend en argument un entier  $i$  et supprime l'élément d'indice  $i$  de la liste.

**Exercice 14** Écrire une méthode publique *inverse* qui inverse le contenu de la liste. Si par exemple on inverse la liste  $L = [1, 2, 3]$  alors la fonction renverra la liste  $[3, 2, 1]$ .

**Exercice 15** Écrire enfin une méthode publique *recherche* qui prend en argument un entier  $n$  et renvoie l'indice du premier élément contenant  $n$  (ou  $-1$  si  $n$  n'apparaît pas dans la liste).

## 2 Listes doublement chaînées

**Exercice 16** Comment faut-il adapter les classes *Cellule* et *Liste* pour avoir des listes doublement chaînées? Modifier à cet effet dans la classe *Liste* les méthodes publiques *precedent*, *ajouteAprès*, *ajouteAvant* et *enlever*.