UNIVERSITÉ DE LORRAINE | IUT nancy Charlemagne Département Informatique

STRUCTURES DE DONNEES

PUBLIC CONCERNÉ : formation initiale, semestre 1

NOM DES AUTEURS : Y. Belaïd

DATE : 2015/2016

IUT NANCY-CHARLEMAGNE
2 TER BD CHARLEMAGNE - CS 55227
54052 NANCY CEDEX
TÉL. 03 54 50 38 20/22
FAX 03 54 50 38 21
iutnc-info@univ-lorraine.fr
http://iut-charlemagne.univ-lorraine.fr

LES TYPES ABSTRAITS 1

1 DEFINITION D'UN TYPE ABSTRAIT 1
2 IMPLANTATION D'UN TYPE ABSTRAIT 2
3 UTILISATION DU TYPE ABSTRAIT 2
4 GENERICITE 2

LES STRUCTURES LINEAIRES 3

1 LES LISTES 3
1.1 Définition abstraite 3
1.2 Description informelle des opérations 4
1.3 Parcours de listes 8
1.4 Exercices : algorithmes logiques sur les listes 10
2 REPRESENTATION CONTIGUË DES LISTES 14
2.1 Représentation contiguë dans tableau 14
2.2 Représentation contiguë dans un fichier séquentiel 16
2.3 Conclusion 16
3 REPRESENTATION CHAINEE DES LISTES 16
3.1 Généralités 16
3.2 Représentation chaînée dans un tableau ou fichier direct 17
3.3 Représentation chaînée à l'aide de pointeurs 22
3.4 Exercice 22
4 LES PILES ET LES FILES 24
4.1 Les piles 24
4.2 Les files 28
4.3 Les files avec priorité 30
5 LES LISTES CIRCULAIRES ET LES LISTES SYMETRIQUES 32
5.1 Les listes circulaires 32
5.2 Les listes symétriques 34
6 EXERCICES RECAPITULATIFS 35

LES TABLES 42

1 LES TABLES ET LEURS OPERATIONS 42
1.1 Définition abstraite 42
1.2 Description informelle des opérations 43
1.3 Exemples 43
1.4 Exercices 47
2 ETUDE DU CAS PARTICULIER DES TABLES « SIMPLES » 49
3 REPRESENTATION DES TABLES " NON SIMPLES " 53
3.1 Adressage (et rangement) calculé 54
3.2 Adressage (et rangement) associatif 57
3.3 Partage (découpage) de la table 59
3.4 Résumé 71

EXERCICES SUR LES LISTES ET LES TABLES 73

1 EXERCICE 1 73
2 EXERCICE 2 78

ANNEXE 1 82

CONVENTIONS POUR L'ECRITURE DES ALGORITHMES 82

ANNEXE 2 87

EXPRESSION DES FONCTIONS LOGIQUES ATTACHEES A LA STRUCTURE DE LISTE, LORSQU'ON UTILISE UNE REPRESENTATION CONTIGUË DANS UN TABLEAU. 87

ANNEXE 3 88

GESTION DE L'ESPACE LIBRE AVEC MARQUAGE ET RECUPERATION DES PLACES LIBRES 88

LES TYPES ABSTRAITS

La conception d'un algorithme un peu compliqué se fait toujours en plusieurs étapes qui correspondent à des raffinements successifs. La première version de l'algorithme est autant que possible indépendante d'une implémentation particulière. La représentation des données n'est pas fixée.

A ce premier niveau, les données sont considérées de manière abstraite : on se donne une notation pour les décrire ainsi que l'ensemble des opérations qu'on peut leur appliquer et les propriétés de ces opérations. On parle alors de *type abstrait de données* (TAD). La conception de l'algorithme (que nous appellerons *algorithme logique*) se fait en utilisant les opérations du TAD. Les différentes représentations du TAD permettent d'obtenir différentes versions de l'algorithme (que nous appellerons *algorithmes de programmation*) si le type abstrait n'est pas un type du langage que l'on veut utiliser.

Une structure de données est une donnée abstraite dont le comportement est modélisé par des opérations abstraites. Elle peut être décrite par un TAD.

Dans ce chapitre, nous verrons comment spécifier une structure de données à l'aide d'un TAD. Dans les chapitres suivants, nous présenterons plusieurs structures de données fondamentales que tout informaticien doit connaître. Il s'agit des structures linéaires et des tables.

Nous rappelons en annexe 1 les conventions retenues pour l'écriture des algorithmes.

1 Définition d'un type abstrait

Un TAD est décrit par sa signature qui comprend :

- une déclaration des ensembles définis et utilisés ;
- une description fonctionnelle des opérations : nom des opérations et leurs profils ; le profil précise à quels ensembles de valeurs appartiennent les arguments et le résultat d'une opération ;
- une description axiomatique de la sémantique des opérations : nous ne détaillerons pas cette partie ; les opérations seront décrites de manière informelle.

Exemple :

Pour le type abstrait *Ensemble* :

- ensembles définis et utilisés : *Ensemble*, Elément, booléen ;
- description fonctionnelle des opérations :

êtreVide :	Ensemble (Elément)	→	booléen
appartenir :	Ensemble (Elément) X Elément	→	booléen
ajouter :	Ensemble (Elément) X Elément	→	
enlever :	Ensemble (Elément) X Elément	→	
union :	Ensemble (Elément) X Ensemble (Elément)	→	Ensemble (Elément)

Les opérations *ajouter* et *enlever* modifient l'ensemble donné en paramètre.

2 Implantation d'un type abstrait

L'implantation est la façon dont le TAD est programmé dans un langage particulier. Il est évident que l'implantation doit respecter la définition formelle du TAD pour être valide.

L'implantation consiste donc :
- à choisir les structures de données concrètes, c'est-à-dire des types du langage d'écriture pour représenter les ensembles définis par le TAD,
- et à rédiger le corps des différentes fonctions qui manipuleront ces types. D'une façon générale, les opérations des TAD correspondent à des sous-programmes de petite taille qui seront donc facile à mettre au point et à maintenir.

Pour un TAD donné, plusieurs implantations possibles peuvent être développées. Le choix d'implantation variera selon l'utilisation qui en est faite et aura une influence sur la complexité des opérations.

Le concept de classe des langages à objets facilite la programmation des TAD dans la mesure où chaque objet porte ses propres données et les opérations qui les manipulent. Notons toutefois que les opérations d'un TAD sont associées à l'ensemble, alors qu'elles le sont à l'objet dans le modèle de programmation objet. La majorité des langages à objets permet de conserver la distinction entre la définition abstraite du type et son implantation grâce aux notions de *classe abstraite* ou d'*interface*.

3 Utilisation du type abstrait

Puisque la définition d'un TAD est indépendante de toute implantation particulière, l'utilisation du TAD devra se faire exclusivement par l'intermédiaire des opérations qui lui sont associées et en aucun cas en tenant compte de son implantation.

Les en-têtes des fonctions du TAD et les affirmations qui définissent leur rôle représentent l'interface entre l'utilisateur et le TAD. Ceci permet évidemment de manipuler le TAD sans même que son implantation soit définie, mais aussi de rendre son utilisation indépendante vis à vis de tout changement d'implantation.

4 Généricité

Reprenons l'exemple du TAD *Ensemble*. Sa définition n'impose aucune restriction sur la nature des éléments des ensembles. Les opérations d'appartenance ou d'union doivent s'appliquer aussi bien à des ensembles d'entiers qu'à des ensembles d'ordinateurs, de voitures ou de fruits.

L'implantation du TAD doit alors être générique, c'est-à-dire qu'elle doit permettre de manipuler des éléments de n'importe quel type. Certains langages de programmation (ADA, C++,…) incluent dans leur définition la notion de généricité et proposent des mécanismes de construction de types génériques. D'autres comme le langage C n'offrent pas cette possibilité. Il faut alors définir un type différent en fonction des éléments manipulés, par exemple un type *EnsembleEntiers* et un type *EnsembleOrdinateurs*.

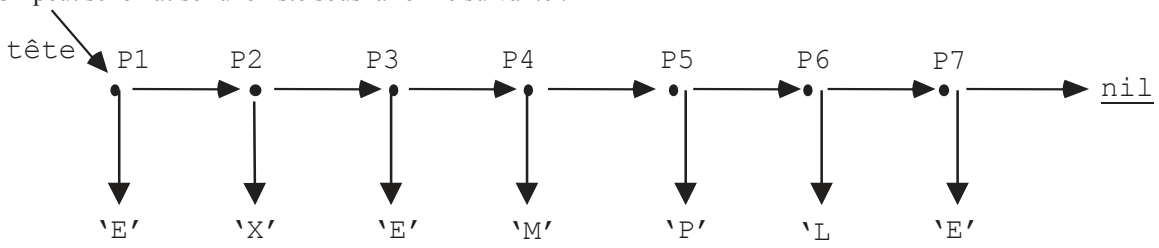
LES STRUCTURES LINEAIRES

Les structures linéaires sont un des modèles les plus élémentaires utilisés dans les programmes informatiques. Elles organisent les données sous forme de séquence d'éléments accessibles de façon séquentielle. Tout élément d'une séquence, sauf le dernier, possède un successeur. Les opérations d'ajout et de suppression d'éléments sont les opérations de base des structures linéaires. Selon la façon dont procèdent ces opérations, nous distinguerons différentes sortes de structures linéaires. Les *listes* autorisent des ajouts et des suppressions d'éléments n'importe où dans la séquence, alors que les *piles* et les *files* ne les permettent qu'aux extrémités. On considère que les files et les piles sont des formes particulières de liste linéaire. Dans ce chapitre, nous commencerons par présenter la forme générale puis nous étudierons quelques formes particulières.

1 Les listes

Une liste est une séquence finie d'éléments de même type repérés selon leur rang. On accède séquentiellement à un élément à partir du premier. L'ordre des éléments dans une liste est fondamental. Il faut remarquer que ce n'est pas un ordre sur les éléments, mais un ordre sur les places des éléments. Ces places sont totalement ordonnées c'est-à-dire qu'il existe une fonction de succession, *suc*, telle que toute place est accessible en appliquant *suc* de manière répétée à partir de la première place de la liste.

On peut schématiser une liste sous la forme suivante :



Remarque :
Qu'est-ce qui distingue les trois 'E' ? Leur place.
'E', 'X', 'M', 'P', 'L' sont les valeurs des éléments de la liste.

1.1 Définition abstraite

Soit *Valeur* l'ensemble des valeurs des éléments d'une liste (par exemple des entiers). On appelle type **Liste de Valeur** et on note **Liste(Valeur)** l'ensemble des listes dont les valeurs sont des éléments de *Valeur*.

Ensembles définis et utilisés : *Liste*, Valeur, Place (ensemble des places y compris *nil* qui est une place fictive)

Description fonctionnelle des opérations :

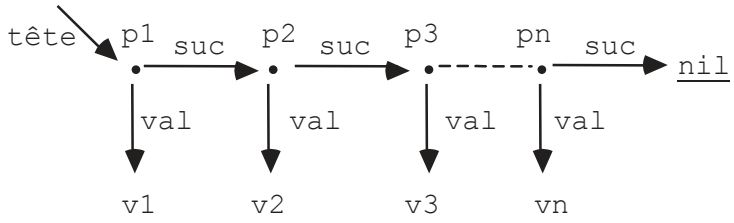
-	tête :	Liste (Valeur)	→	Place
-	val :	Liste (Valeur) x Place-{nil}	→	Valeur
-	suc :	Liste (Valeur) x Place-{nil}	→	Place
-	finliste :	Liste (Valeur) x Place	→	booléen
-	lisvide :		→	Liste (Valeur)
-	adjtlis :	Liste (Valeur) x Valeur	→	

- suptlis : Liste (Valeur)-{lisvide ( )} →
- adjqlis : Liste (Valeur) x Valeur →
- supqlis : Liste (Valeur)-{lisvide ( )} →
- adjlis : (Liste (Valeur)-{lisvide ( )}) x (Place-{nil}) x Valeur →
- suplis : (Liste (Valeur)-{lisvide ( )}) x (Place-{nil}) →
- chglis : (Liste (Valeur)-{lisvide ( )}) x (Place-{nil}) x Valeur →

Les opérations adjtlis, suptlis, adjqlis, supqlis, adjlis, suplis, chglis modifient la liste donnée en paramètre.

1.2 Description informelle des opérations

Nous expliquons ici le rôle de chaque opération et nous l'illustrons par des schémas. Soit une liste l contenant n éléments dont les valeurs sont v1, v2, v3,… vn . On peut représenter la liste l par le schéma suivant :



L'ensemble des places est {p1, p2, p3, …pn, nil}.

1.2.1 Les opérations de parcours

tête :

La fonction *tête* désigne la place du premier élément de la liste. Par exemple, tête (l) rend p1.

val :

La fonction *val* désigne la valeur associée à une place. Par exemple, val (l, p3) rend v3.

suc :

La place qui suit une place p est celle occupée par l'élément suivant dans la liste; elle est désignée par la fonction *suc* (pour successeur). Par exemple, suc (l, p2) rend p3.

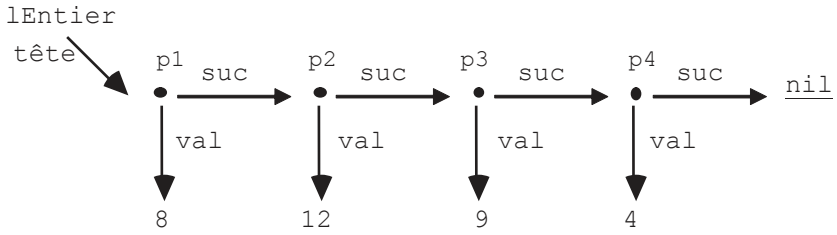
finliste :

La fonction *finliste* permet de tester si une place donnée est la place *nil* c'est-à-dire si on est positionné à la fin de la liste. Notez bien que finliste (l, pn) rend faux et finliste (l, suc (l, pn)) rend vrai.

1.2.2 Exemples d'utilisations des opérations de parcours

Exemple 1

Soit la liste *lEntier* schématisée ci-après :



Evaluer les expressions suivantes :

- val (lEntier, tête (lEntier))
- val (lEntier, suc (lEntier, suc (lEntier, tête (lEntier))))
- finliste (lEntier, suc (lEntier, suc (lEntier, suc (lEntier, tête(lEntier)))))
- finliste (lEntier, suc (lEntier, suc (lEntier, suc (lEntier, suc (lEntier, tête (lEntier))))))

Réponses : 8 9 faux vrai

Quelle est la valeur retournée par l'algorithme suivant ?

```
Algorithme logique
début (*1*)
        lEntier ← lire ( )
        place ← tête (lEntier)
        pour (*2*) i de 1 à 3 faire      (* on est sûr ici que la liste contient au moins 3 éléments *)
                place ← suc (lEntier, place)
        fpour(*2*)
        écrire (val (lEntier, place))
fin (*1*)
Lexique
place : Place, ième place de la liste
lEntier: Liste (entier), liste donnée
```

Réponse : 4

Exemple 2

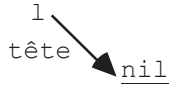
Soit la liste *lEliminés* contenant les nom, prénom et note des candidats ayant échoué à l'épreuve du permis de conduire. Ecrire l'algorithme logique de la fonction qui permet d'afficher le nom et le prénom de la première personne éliminée ou un message si aucune personne n'a échoué.

```
Algorithme logique
Fonction imprimerPremierEliminé (lEliminés : liste (Candidat))
début (*1*)
        placeTête ← tête (lEliminés )
        si finliste (lEliminés, placeTête)
                alors (*2a*)   écrire (« pas d'échec »)
                sinon (*2s*)   candidatEliminé ← val (lEliminés, placeTête)
                               écrire (candidatEliminé.nom,  candidatEliminé.prénom)
        fsi(*2*)
fin(*1*)
Lexique
Candidat = <nom : chaîne, prénom : chaîne, note : entier>
lEliminés : liste (Candidat), liste des candidats éliminés
placeTête : Place, place du premier élément de la liste s'il existe
candidatEliminé : Candidat, premier candidat éliminé s'il existe
```

1.2.3 Les opérations de construction et de mises à jour

lisvide :

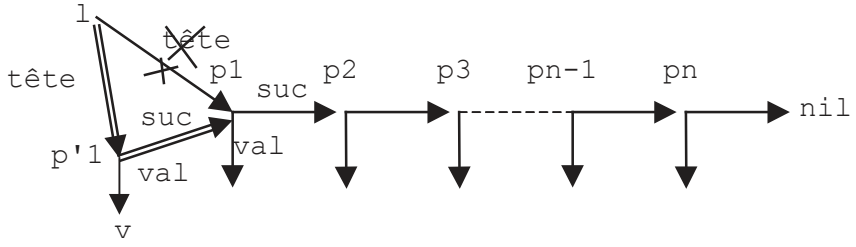
construction d'une liste vide c'est-à-dire d'une liste ne contenant aucun élément.
l ← lisvide() : création de la liste vide *l* :



finliste (l, tête (l)) rend vrai ; c'est le signe que la liste l est vide.

adjtlis :

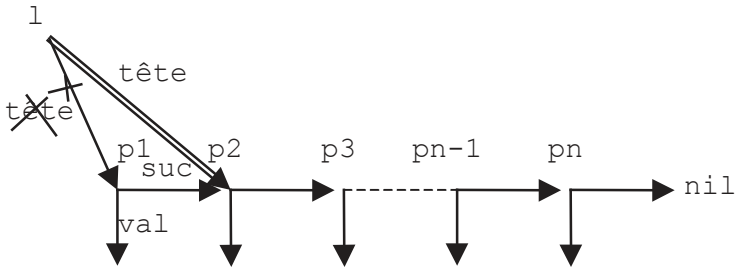
adjonction en tête de la liste ; c'est le cas où l'on insère un élément avant tous les autres ;
adjtlis (l, v) ajoute en tête de *l* un élément de valeur *v* :



Sur le schéma:
- nous dessinons la liste initiale,
- nous barrons les liens qui disparaissent lors de la modification,
- nous notons en double ceux qui apparaissent.

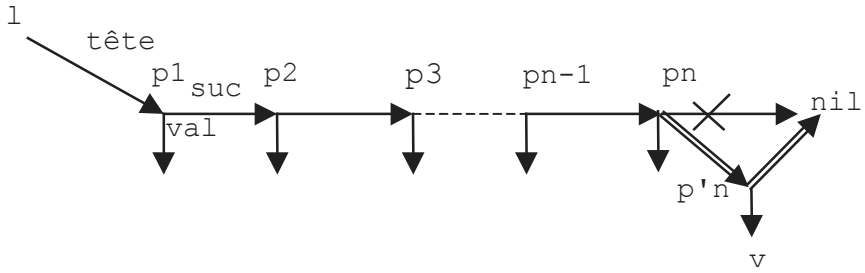
suptlis :

suppression en tête ;
suptlis (l) : c'est le cas où le premier élément est supprimé ;



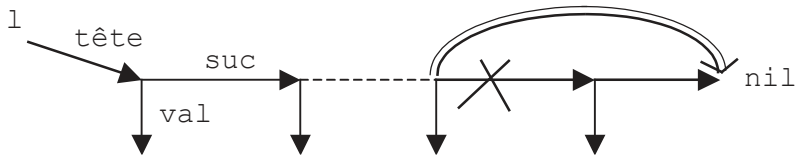
adjqlis :

adjonction en queue de liste ;
adjqlis (l, v) : adjonction d'un élément de valeur *v* en queue de la liste *l* ;



supqlis :

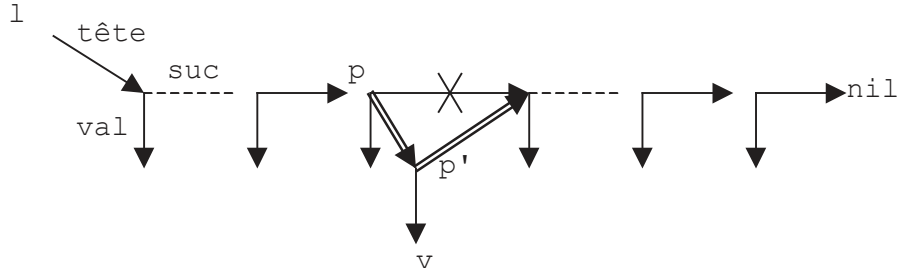
suppression en queue ;
supqlis(l) : c'est le cas où le dernier élément de la liste est enlevé.



Nous venons d'évoquer les adjonctions et suppressions en tête et en queue. Mais il arrive qu'une liste doive subir des adjonctions, suppressions ou modifications ailleurs qu'en tête ou en queue. Nous spécifions par la place p, l'emplacement de la modification.

adjlis :

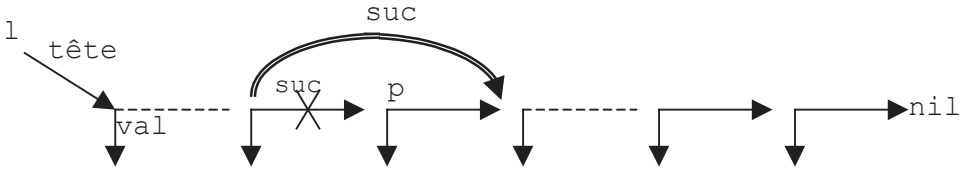
adjonction après une place p ;
adjlis (l, p, v) : adjonction à la liste *l* supposée non vide, juste après l'élément de place *p*, d'un nouvel élément de valeur *v*.



Remarque : ne permet pas de faire une adjonction en tête.

suplis :

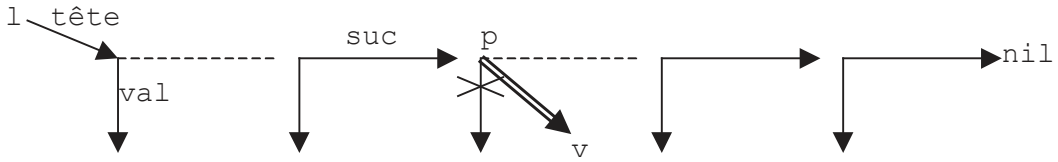
suppression d'un élément à une place donnée ;
suplis(l,p) : suppression dans la liste *l*, supposée non vide, de l'élément situé à la place *p*.



Remarque :
Pour supprimer l'élément à la place *p*, il faut faire le lien entre l'élément précédent *p* et l'élément suivant *p*. Donc, lors du parcours de la liste à la recherche de *p*, il faudra conserver à chaque pas la place précédente.

chglis :

changement de la valeur d'un élément ;
chglis (l, p, v) : modification de la valeur de l'élément situé à la place p.



Remarque importante :

La place p ne peut jamais être connue d'emblée mais est toujours le résultat d'un parcours de la liste (parcours jusqu'à ce qu'une condition C soit réalisée). Nous étudierons ces parcours au paragraphe suivant.

1.2.4 Exemple de création de liste

Prenons l'exemple de la liste d'admission des étudiants dans une école et étudions sa création. On la crée à partir des dossiers déjà triés par ordre de valeur. Au départ, la liste est vide. On ajoute successivement dans la liste les noms et notes figurant sur chacun des dossiers. On s'arrête quand tous les dossiers sont entrés (le nombre de dossiers est donné) .

Données :
nombreDossier
nombreDossier fois : nom
note

Résultats :
liste d'admission

Algorithme logique

```
fonction lEtudSaisir( ) : Liste(Etudiant)
    début (*1*)
        lAdmis ← lisvide( )
        nombreDossier ← lire( )
        pour (*2*) i de 1 à nombreDossier faire
                étudiant ← lire( )
                adjqlis (lAdmis, étudiant)
        fpour (*2*)
        retourne lAdmis
    fin(*1*)
```

Lexique

Etudiant = <nom : chaîne, note : réel >
lAdmis: Liste(Etudiant), liste d'admission
nombreDossier : entier, nombre de dossiers
étudiant : Etudiant, nom et note contenus dans le ième dossier
i : entier, indice d'itération sur les dossiers

1.3 Parcours de listes

Les opérations de parcours sont : **tête**, **val**, **suc** et **finliste**. On peut vouloir parcourir une liste en entier, jusqu'à un certain élément ou à partir d'un certain élément.

1.3.1 Exemples de parcours complet

a) Soit une liste *lEntier* d'entiers, calculer la moyenne des éléments de cette liste.

```
Algorithme logique
    Fonction entierCalculerMoyenne (lEntier : Liste(entier)) : réel
    début (*1*)
        somme ← 0
        nombre ← 0
        place ← tête (lEntier)
        tantque (*2*) non finliste (lEntier, place) faire
                valeur ← val (lEntier, place)
                somme ← somme + valeur
                nombre ← nombre+1
                place ← suc (lEntier, place)
        ftantque (*2*)
        si nombre ≠ 0
                alors moyenne ← somme / nombre
                sinon moyenne ← 0
        fsi
        retourne moyenne
    fin (*1*)
Lexique
    lEntier : Liste(entier), liste des entiers
    moyenne : réel, moyenne des éléments
    place : Place, place courante
    somme : entier, somme des nombre premiers entiers
    nombre : entier, nombre courant de valeurs lues
    valeur : entier,  valeur courante de la liste lEntier
```

b) Soit *lPersonne* une liste des habitants d'une commune. Une personne est décrite par son nom, son adresse et son année de naissance. On désire envoyer un prospectus électoral à toute personne majeure ou le devenant en cours d'année. Ecrire l'algorithme logique de la fonction qui permet d'imprimer les noms et les adresses de ces personnes.

```
Algorithme logique
    fonction lpersonneImprimerMajeure (lPersonne : Liste(Personne), annéeCourante : entier)
    début (*1*)
        place ← tête (lPersonne )
        tantque(*2*) non finliste (lPersonne, place) faire
                personne ← val (lPersonne, place)
                si annéeCourante  -  personne.naissance  ≥ 18
                        alors (*3a*) écrire (personne.nom , personne.adresse)
                fsi(*3*)
                place ← suc (lPersonne, place)
        fintantque(*2*)
    fin(*1*)
Lexique
    Personne = <nom : chaîne, adresse : chaîne, naissance : entier>
    lPersonne : Liste(Personne), liste des personnes de la commune
    place : Place,  place courante
    annéeCourante : entier, année en cours
    personne : Personne, personne courante de la liste
```

1.3.2 Exemple de parcours de liste depuis le début jusqu'à une condition d'arrêt

Reprenons l'exemple précédent en supposant que les personnes sont classées par année de naissance croissante. Ainsi on arrête le parcours dès qu'on rencontre une personne mineure.

```
Algorithme logique
    fonction lpersonneImprimerMajeure (lPersonne : Liste(Personne), annéeCourante : entier)
    début(*1*)
        place ← tête (lPersonne )
        mineur ← faux
        tantque(*2*) (non finliste (lPersonne, place)) et (non mineur) faire
                personne ← val (lPersonne, place)
                si annéeCourante - personne . naissance < 18
                        alors (*3a*)    mineur ← vrai
                        sinon (*3s*)    écrire (personne. nom, personne. adresse)
                                        place ← suc (lPersonne, place)
                fsi(*3*)
        fintantque(*2*)
    fin(*1*)
Lexique
    lPersonne
    place                   |       cf. algorithme précédent
    annéeCourante
    personne
    mineur : booléen, vrai quand la personne courante est mineure
```

1.3.3 Exemple de parcours à partir d'un certain élément de la liste

On définit la place de départ par un parcours de la liste jusqu'à une condition d'arrêt.

Exemples de conditions d'arrêt :

C1 : la valeur associée à la place courante est égale à une valeur donnée.
(Par exemple, parcourir la liste d'admission à partir de la place occupée par Durand).

C2 : la valeur de l'élément précédant l'élément courant est égale à une valeur donnée.
(Par exemple, parcourir la liste d'admission à partir de la place suivant celle occupée par Henri).

C3 : le rang de l'élément à partir duquel on veut faire le parcours est connu.
(Par exemple, parcourir la liste d'admission à partir du 7ème étudiant).

1.3.4 Schéma général de l'algorithme de parcours d'une liste

```
début
    ......
    [* place ← tête (liste) *]
    [* trouve ← faux *]
    tantque non finliste (liste, place) [* et non trouve *] faire
            valeur ← val (liste, place)
            ...
            [* si Condition (valeur)
            alors ...
                    trouve ← vrai
            sinon ...*]
                    place ← suc (liste, place)
            [* fsi *]
    ftantque
    ......
fin
```

où *liste* est la liste à parcourir, *place* la suite des places, *valeur* la suite des valeurs, *trouve* la suite des conditions d'arrêt et *Condition* une fonction à valeur booléenne. Tout ce qui se trouve entre [* et *] n'est à écrire que si le contexte l'exige.

1.4 Exercices : algorithmes logiques sur les listes

Exercice 1 (création et parcours complet)

On souhaite créer une liste de températures et calculer la moyenne des températures d'une liste. Ecrire les algorithmes logiques des fonctions suivantes:

fonction lTempSaisir ( ) : Liste (réel)
    saisit un nombre de températures puis les températures et les range dans une liste
fonction lTempMoyenne (lTemp : Liste (réel)) : réel
    calcule la moyenne des températures de la liste *lTemp*

Exercice 2 (parcours avec C1)

Soit la liste des ouvrages (auteur-titre) d'une bibliothèque classée par ordre alphabétique des noms d'auteurs. Ecrire l'algorithme logique de la fonction qui crée la liste de tous les titres d'un auteur donné.

Exercice 3 (parcours avec C2 et C3)

Soit lPilote, la liste des noms des pilotes d'une course de F1 dans l'ordre d'arrivée.

a) Ecrire l'algorithme logique de la fonction qui permet d'imprimer les noms des pilotes se trouvant après la dixième place.
b) Ecrire l'algorithme logique de la fonction qui permet d'imprimer les noms des 3 pilotes arrivés après A. Prost.

Exercice 4 (adjonction et suppression)

Soit une suite de valeurs entières (toutes comprises entre 0 et 1000). On souhaite construire une liste "lEntier", triée par ordre croissant, contenant les entiers figurant un nombre impair de fois dans la suite des données. Ecrire l'algorithme logique de la fonction réalisant cette construction. On lit le nombre de valeurs puis chaque valeur.

Principe : chaque donnée est cherchée dans la liste partiellement construite ; si elle appartient déjà à la liste, on la supprime, sinon on la rajoute.

Difficulté : initialisation de la place précédente.

Exercice 5 (interclassement de 2 listes triées)

Ecrire l'algorithme logique de la fonction qui interclasse 2 listes d'entiers triées par ordre croissant.

Principe :
- on compare les 1ers éléments des 2 listes, on place le plus petit dans la liste résultat
- on recommence avec l'élément qui reste et l'élément suivant de l'autre liste
...

Correction Exercice 1

Algorithme de la fonction lTempSaisir

```
fonction lTempSaisir ( ) : Liste (réel)
début(*1*)
lTemp ← lisvide()
nbTemp ← lire( )
pour i de 1 à nbTemp faire (*2*)
    température ← lire( )
    adjqlis (lTemp, température)
fpour (*2*)
retourne lTemp
fin(*1*)
```

Lexique

lTemp : Liste (réel), liste des températures
nbTemp : entier, nombre de températures
température : réel, la ième température lue
i : entier, indice d'itération