



UNIVERSITÉ  
DE LORRAINE



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](mailto:iutnc-info@univ-lorraine.fr)  
<http://iut-charlemagne.univ-lorraine.fr>

<b>LES TYPES ABSTRAITS.....</b>	<b>1</b>
1 DEFINITION D'UN TYPE ABSTRAIT .....	1
2 IMPLANTATION D'UN TYPE ABSTRAIT .....	2
3 UTILISATION DU TYPE ABSTRAIT.....	2
4 GENERICITE .....	2
<b>LES STRUCTURES LINEAIRES .....</b>	<b>3</b>
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 CHAÎNÉE 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
<b>LES TABLES .....</b>	<b>42</b>
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 « SIMPLÉS » .....	49
3 REPRESENTATION DES TABLES " NON SIMPLÉS " .....	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
<b>EXERCICES SUR LES LISTES ET LES TABLES .....</b>	<b>73</b>
1 EXERCICE 1.....	73
2 EXERCICE 2.....	78
<b>ANNEXE 1.....</b>	<b>82</b>
CONVENTIONS POUR L'ECRITURE DES ALGORITHMES .....	82
<b>ANNEXE 2.....</b>	<b>87</b>
EXPRESSION DES FONCTIONS LOGIQUES ATTACHEES A LA STRUCTURE DE LISTE, LORSQU'ON UTILISE UNE REPRESENTATION CONTIGUË DANS UN TABLEAU. ....	87
<b>ANNEXE 3.....</b>	<b>88</b>
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 :
 

<i>êtreVide</i> :	<i>Ensemble</i> ( <i>Elément</i> )	→	booléen
<i>appartenir</i> :	<i>Ensemble</i> ( <i>Elément</i> ) X <i>Elément</i>	→	booléen
<i>ajouter</i> :	<i>Ensemble</i> ( <i>Elément</i> ) X <i>Elément</i>	→	
<i>enlever</i> :	<i>Ensemble</i> ( <i>Elément</i> ) X <i>Elément</i>	→	
<i>union</i> :	<i>Ensemble</i> ( <i>Elément</i> ) X <i>Ensemble</i> ( <i>Elément</i> )	→	<i>Ensemble</i> ( <i>Elément</i> )

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*.

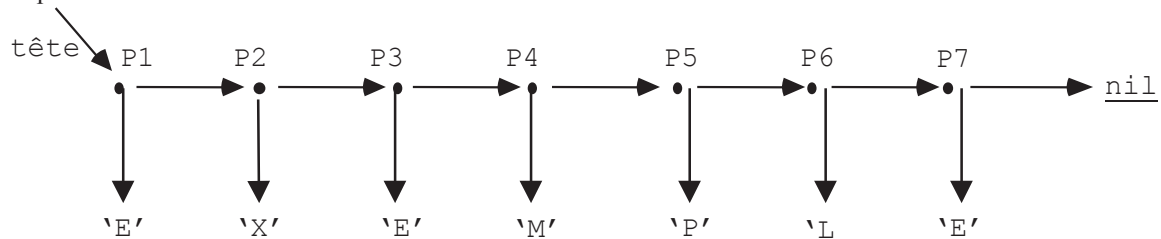
# LES STRUCTURES LINEAIRES

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 *pires* 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 :

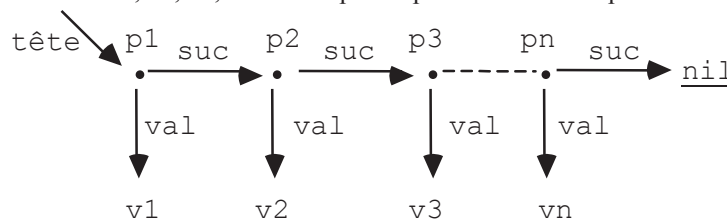
- tête :	Liste (Valeur)	→	Place
- val :	Liste (Valeur) x Place-{nil}	→	Valeur
- suc :	Liste (Valeur) x Place-{nil}	→	Place
- finliste :	Liste (Valeur) x Place	→	booléen
- lisvide :		→	Liste (Valeur)
- adjtlis :	Liste (Valeur) x Valeur	→	

- `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  $\{p1, p2, p3, \dots, pn, nil\}$ .

### 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`.

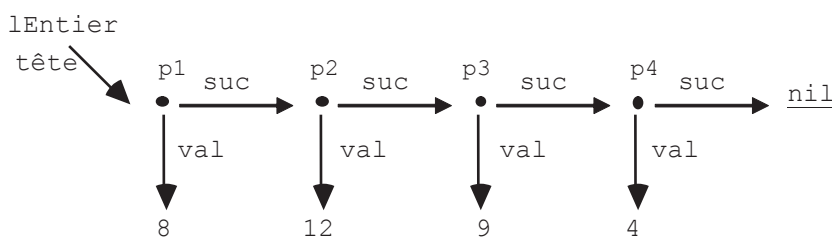
**finliste :**

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 *IEliminés* 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 (IEliminés : liste (Candidat))

```
début (*1*)
  placeTête ← tête (IEliminés)
  si finliste (IEliminés, placeTête)
    alors (*2a*) écrire (« pas d'échec »)
    sinon (*2s*) candidatEliminé ← val (IEliminés, placeTête)
                  écrire (candidatEliminé.nom, candidatEliminé.prénom)
  fsi(*2*)
```

fin(\*1\*)

Lexique

Candidat = <nom : chaîne, prénom : chaîne, note : entier>

IEliminés : 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.

$l \leftarrow \text{lisvide}()$  : création de la liste vide  $l$  :

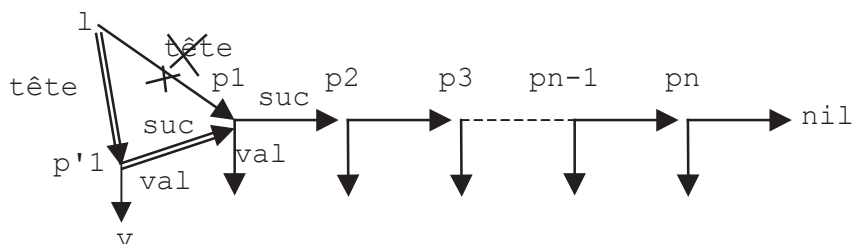


finliste ( $l$ , tête ( $l$ )) rend vrai ; c'est le signe que la liste  $l$  est vide.

### 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  $v$  :



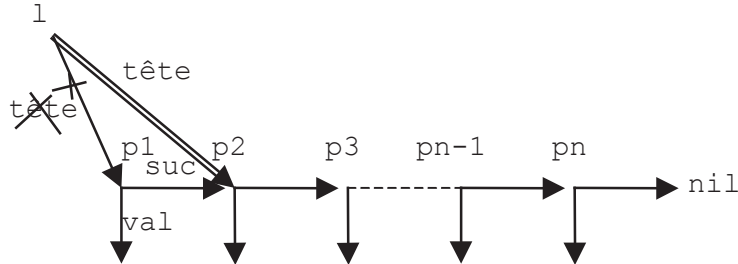
Sur le schéma:

- 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 ;

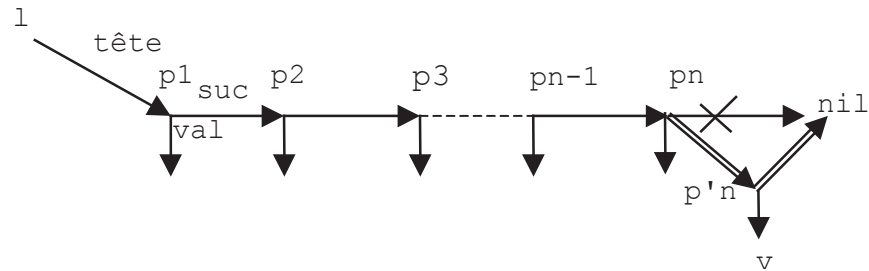
suptlis( $l$ ) : c'est le cas où le premier élément est supprimé ;



### adjqlis :

adjonction en queue de liste ;

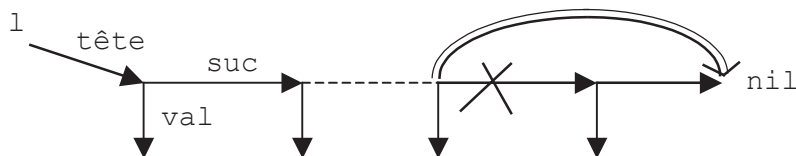
adjqlis( $l, v$ ) : adjonction d'un élément de valeur  $v$  en queue de la liste  $l$  ;



### supqlis :

suppression en queue ;

supqlis( $l$ ) : c'est le cas où le dernier élément de la liste est enlevé.

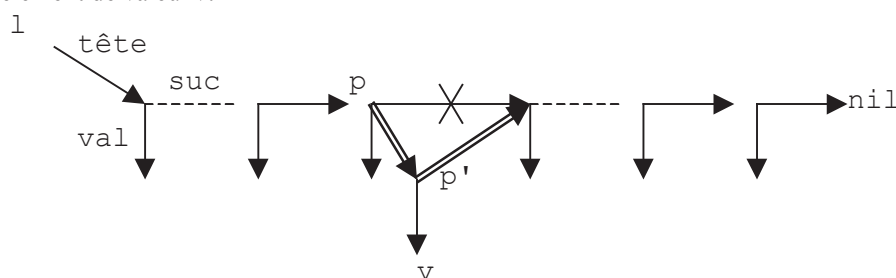


Nous venons d'évoquer les adjonctions et suppressions en tête et en queue. Mais il arrive qu'une liste doive subir des adjonctions, suppressions ou modifications ailleurs qu'en tête ou en queue. Nous spécifions par la place  $p$ , l'emplacement de la modification.

### 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  $v$ .



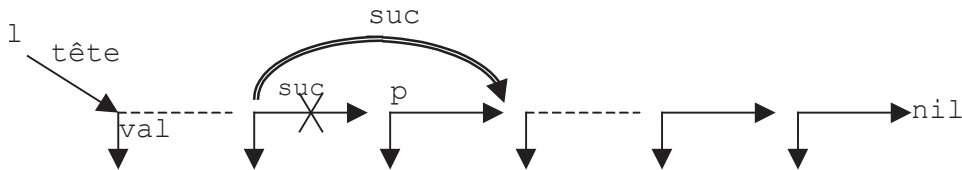
Remarque : ne permet pas de faire une adjonction en tête.



**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  $p$ .



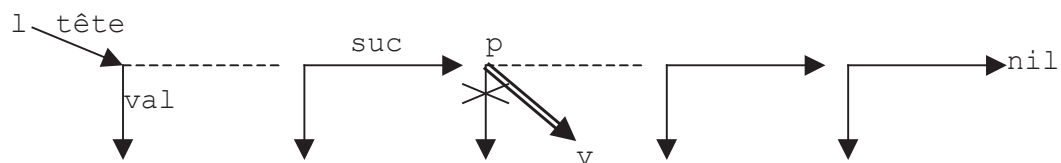
Remarque :

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 *IPersonne* 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 (IPersonne : Liste(Personne), annéeCourante : entier)
début (*1*)
    place ← tête (IPersonne)
    tantque (*2*) non finliste (IPersonne, place) faire
        personne ← val (IPersonne, place)
        si annéeCourante - personne.naissance ≥ 18
            alors (*3a*) écrire (personne.nom , personne.adresse)
        fsi (*3*)
        place ← suc (IPersonne, place)
    ftantque (*2*)
fin (*1*)

```

##### Lexique

Personne = <nom : chaîne, adresse : chaîne, naissance : entier>  
 IPersonne : 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*)
    ftantque(*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.

- Ecrire l'algorithme logique de la fonction qui permet d'imprimer les noms des pilotes se trouvant après la dixième place.
- 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

Algorithme de la fonction lTempMoyenne

```

fonction lTempMoyenne ( lTemp : Liste (réel) ) : réel
  début(*1*)
    somme ← 0
    nombreTempérature ← 0
    place ← tête (lTemp)
    tantque(*2*) non finliste (lTemp, place) faire
      température ← val (lTemp, place)
      somme ← somme + température
      nombreTempérature ← nombreTempérature + 1
      place ← suc (lTemp, place)
    fintantque(*2*)
    si nombreTempérature ≠ 0 alors moyenne ← somme / nombreTempérature
    sinon moyenne ← 0
  fsi
  retourne moyenne
fin(*1*)

```

Lexique

lTemp : Liste (réel), liste des températures  
 température : réel, ième température de la liste  
 moyenne : réel, moyenne des températures ou 0  
 somme : réel, somme des températures  
 nombreTempérature : entier, nombre de températures de la liste  
 place : Place, ième place dans la liste

**Correction Exercice 2**Algorithme de la fonction lLivresCréerListeTitre

```

fonction lLivresCréerListeTitre (lLivres : ListeLivres, auteur : chaîne) : ListeTitre
  début(*1*)
    place ← tête(lLivres)
    arrêt ← faux
    tantque(*2*) non finliste (lLivres, place) et non arrêt faire (* recherche du 1er livre de l'auteur *)
      livre ← val (lLivres, place)
      si (livre.auteur >= auteur)
        alors(*3a*) arrêt ← vrai
        sinon(*3s*) place ← suc (lLivres, place)
      fsi(*3*)
    fintantque(*2*)
    lTitre ← lisvide( )
    si livre.auteur = auteur (* on a trouvé l'auteur *)
    alors(*4a*) (*la place, où on en est, est bien celle du premier livre de l'auteur*)
      adjqlis (lTitre, livre.titre)
      place ← suc (lLivres, place)
      bonAuteur ← vrai
      tantque(*5*) bonAuteur et non finliste (lLivres, place) faire
        livre ← val (lLivres, place)
        si livre.auteur = auteur alors(*6a*) adjqlis (lTitre, livre.titre)
        place ← suc (lLivres, place)
        sinon(*6s*) bonAuteur ← faux
      fsi(*6*)
    fintantque(*5*)
  fsi(*4*)
  retourne lTitre
fin(*1*)

```

Lexique

Livres = < auteur : chaîne, titre : chaîne >  
 ListeLivres = Liste (Livres)  
 ListeTitre = Liste (chaîne)  
 lLivres : ListeLivres  
 auteur : chaîne, auteur donné  
 place : Place, ième place dans la liste  
 arrêt : booléen, à vrai si on trouve l'auteur ou s'il n'y est pas  
 livre : Livre, ième livre dans la liste  
 bonAuteur : booléen, à vrai si le ième livre est de l'auteur *auteur*  
 lTitre : ListeTitre, liste de tous les titres de l'auteur donné

## Correction Exercice 3

### Algorithme question a

```

fonction IPiloteImprimerAprès10 (IPilote : Liste (chaîne))
début(*1*)
  i ← 1
  place ← tête (IPilote)
  tantque(*2*) non finliste (IPilote, place) et i <= 10 faire
    i ← i+1
    place ← suc(IPilote, place)
  ftantque(*2*)
  tantque(*3*) non finliste (IPilote, place) faire
    écrire (val (IPilote, place))
    place ← suc (IPilote, place)
  ftantque(*3*)
fin(*1*)

```

### Lexique

IPilote : Liste (chaîne), liste des pilotes  
 i : entier, numéro de la place courante  
 place : Place, place courante de la liste

### Algorithme question b

```

fonction IPiloteImprimer3AprèsProst (IPilote : Liste (chaîne))
début(*1*)
  trouve ← faux
  place ← tête (IPilote)
  tantque(*2*) non finliste (IPilote, place) et non trouve faire
    si val (IPilote, place) = « A.Prost »
      alors(*3a*) trouve ← vrai
      sinon(*3s*) place ← suc (IPilote, place)
    fsi(*3*)
  ftantque(*2*)
  si trouve alors(*4a*)
    place ← suc (IPilote, place)
    i ← 1
    tantque(*5*) non finliste (IPilote, place) et i <=3 faire
      écrire (val (IPilote, place))
      place ← suc (IPilote, place)
      i ← i+1
    ftantque(*5*)
  sinon(*4s*) écrire (« A. Prost n'est pas dans la liste »)
  fsi(*4*)
fin(*1*)

```

### Lexique

IPilote : Liste (chaîne), liste des pilotes  
 trouve : booléen, à vrai lorsqu'on trouve A. Prost  
 place : Place, place courante  
 i : entier, compteur

## Correction Exercice 4

### Algorithme logique

```

fonction lEntierCréer ( ) : Liste (entier)
début (*1*)
  lEntier ← lisvide ( )
  nbValeur ← lire ( )
  pour i de 1 à nbValeur faire(*2*)
    valeur ← lire ( )
    place ← tête (lEntier)
    si finliste (lEntier, place) (*1a liste est vide*) alors(*8a*) adjqlis (lEntier, valeur)
    sinon(*8s*)
      placePrécédente ← place (* initialisation artificielle *)
      fini ← faux (*recherche de la valeur*)
      tantque(*3*) non finliste (lEntier, place) et non fini faire
        élément ← val (lEntier, place)
        si valeur ≤ élément alors(*4a*) fini ← vrai

```

```

        sinon (*4s*)
            placePrécédente ← place
            place ← suc (lEntier, place)
        fsi (*4*)
    fintantque (*3*)
    si valeur = élément (*valeur appartient déjà à la liste*) alors (*5a*)
        suplis (lEntier, place)
    sinon (*5s*) (*valeur n'appartient pas à la liste*)
        si place = placePrécédente alors (*7a*)
            (*valeur est inférieur à toutes les valeurs*)
            adjtlis (lEntier, valeur)
        sinon (*7s*)
            adjlis (lEntier, placePrécédente, valeur)
        fsi (*7*)
    fsi (*5*)
    fsi (*8*)
    fpour (*2*)
    retourne lEntier
fin (*1*)

```

### Lexique

lEntier: Liste(entier), liste des entiers par ordre croissant  
 nbValeur : entier, nombre de valeurs à traiter  
 i : entier, indice d'itération  
 valeur : entier, ième donnée  
 place : Place, place courante  
 placePrécédente : Place, place précédent la place courante  
 fini : booléen, vrai lorsqu'on a trouvé l'entier ou lorsque la valeur courante est supérieure à la valeur cherchée.  
 élément : entier, valeur courante dans la liste

## Correction Exercice 5

### Algorithme de la fonction lEntierInterclasser

```

fonction lEntierInterclasser (liste1 : Liste(entier), liste2 : Liste(entier)) : Liste(entier)
début (*1*)
    listeRésultat ← lisvide()
    place1 ← tête (liste1)
    place2 ← tête (liste2)
    tant que non finliste (liste1, place1) et non finliste (liste2, place2) faire (*2*)
        entier1 ← val (liste1, place1)
        entier2 ← val (liste2, place2)
        si entier1 < entier2 alors (*3a*)
            adjqlis (listeRésultat, entier1)
            place1 ← suc (liste1, place1)
        sinon (*3s*)
            adjqlis (listeRésultat, entier2)
            place2 ← suc (liste2, place2)
        fsi (*3*)
    ftant (*2*)
    si finliste (liste1, place1)
        alors (*4a*) lentierCopierFinListe (liste2, place2, listeRésultat)
        sinon (*4s*) lentierCopierFinListe (liste1, place1, listeRésultat)
    fsi (*4*)
    retourne listeRésultat
fin (*1*)

```

### Lexique

liste1 : Liste(entier), liste triée par ordre croissant  
 liste2 : Liste(entier), liste triée par ordre croissant  
 listeRésultat : Liste(entier), résultat de l'interclassement de liste1 et liste2  
 place1 : Place, dans liste1  
 place2 : Place, dans liste2  
 entier1 : entier, élément de liste1  
 entier2 : entier, élément de liste2  
 fonction lEntierCopierFinListe (listeA : Liste(entier), placeListeA : Place, listeB InOut: Liste(entier)) copie la fin de listeA à partir de la place placeListeA en fin de listeB

Algorithme de la fonction lEntierCopierFinListe

```

fonction lEntierCopierFinListe (listeA : Liste(entier), placeListeA : Place, listeB InOut: Liste(entier))
  début (*1*)
    placeCourante ← placeListeA
    tant que non finliste (listeA, placeCourante) faire (*2*)
      adjqlis (listeB, val (listeA, placeCourante))
      placeCourante ← suc (listeA, placeCourante)
  fin (*1*)
  fin (*2*)
fin (*1*)

```

Lexique

placeListeA : Place, place à partir de laquelle il faut copier  
 listeB : Liste(entier), liste dans laquelle il faut copier  
 listeA : Liste(entier), liste à partir de laquelle il faut copier  
 placeCourante : Place, place courante dans *listeA*

## 2 Représentation contiguë des listes

Une liste est caractérisée par un ensemble de places et les fonctions : tête, suc, val et finliste. Pour représenter une liste, il faut choisir une représentation pour chacune de ces fonctions. La fonction dont la représentation influe le plus sur les traitements est la fonction *suc*. Dans ce paragraphe, nous étudierons les cas où la fonction *suc* est représentée par une fonction de contiguïté.

### 2.1 Représentation contiguë dans tableau

Les éléments de la liste sont représentés dans un tableau contenant les valeurs rangées successivement à partir du début (borne inférieure, notée *bi*). Une place dans la liste est représentée par un indice d'accès dans le tableau.

place p                      <==>                      indice p

Dans ce cas, val (l, p) signifie "contenu de l'élément d'indice p".

La fin de liste peut être représentée par :

- un entier indiquant le nombre d'éléments de la liste ;
- un enregistrement "bidon" placé après le dernier élément de la liste ;
- un indice d'accès au dernier élément de la liste (indice de queue).

Remarque 1 : Dans certains problèmes particuliers, il peut être intéressant de démarrer la liste à partir d'un rang quelconque dans le tableau. Dans ce cas, il faut gérer un indice de tête.

Remarque 2 : On peut imaginer des cas où val est qualifié d'*indirect* : l'élément d'indice p du tableau contient non pas val (l, p) mais quelque chose qui permet de le trouver, par exemple un indice dans un autre tableau. Cette représentation est utile en particulier si une même valeur peut apparaître de nombreuses fois ou si les diverses valeurs ont des longueurs très disparates. Un exemple est proposé dans l'exercice sur les *tables simples*.

### Exemple

Reprenons, comme exemple, la liste des admissions. Choisissons une représentation contiguë à l'aide d'un couple : tableau et nombre d'éléments.

La liste ladmis est alors représentée par le type composite suivant :

Liste(Etudiant) = < tab : tableau Etudiant [1..MAXNBETUDIANT], nb : entier> .

( rappel: Etudiant = <nom : chaîne, note : réel> )

Le champ *tab* contient le tableau des couples (*nom*, *note*) et le champ *nb* le nombre d'éléments de la liste.



Exemple de représentation de la liste ladmis :

lAdmis.tab		lAdmis.nb
	nom      note	
1	Jean      18	4
2	Paul      15	
3	Marie     14	
4	Germaine 10	
5		
6		
7		

Lorsqu'on choisit une représentation, chaque fonction logique est écrite sous forme d'une fonction dans le langage de programmation choisi. Nous donnons en exemple quelques algorithmes de programmation de ces fonctions.

- fonction tête (lAdmis : Liste(Etudiant)) : entier  
début  
place ← 1  
retourne place  
fin
- fonction val (lAdmis : Liste(Etudiant), place : entier) : Etudiant  
début  
étudiant ← lAdmis.tab[place]  
retourne étudiant  
fin
- fonction suc (lAdmis : Liste(Etudiant), place : entier) : entier  
début  
retourne place + 1  
fin
- fonction finliste (lAdmis : Liste(Etudiant), place : entier) : booléen  
début  
fin ← ( place = 1 + lAdmis.nb )  
retourne fin  
fin
- fonction lisvide ( ) : Liste(Etudiant)  
début  
lAdmis.nb ← 0  
retourne lAdmis  
fin
- fonction adjqlis (lAdmis InOut : Liste(Etudiant), étudiant : Etudiant)  
début  
lAdmis.nb ← lAdmis.nb + 1  
lAdmis.tab [lAdmis.nb] ← étudiant  
fin

## Exercice

Ecrire les algorithmes de programmation des fonctions suptlis et adjlis.

Corrigé :

- fonction suptlis (lAdmis InOut : Liste(Etudiant))  
début  
/\* Pour que la nouvelle liste débute toujours en 1, il faut décaler tous les éléments :\*/  
pour i de 1 à lAdmis.nb - 1 faire  
lAdmis.tab[i] ← lAdmis.tab[i+1]  
fpour  
lAdmis.nb ← lAdmis.nb - 1  
fin

```

- fonction adjlis (lAdmis InOut : Liste(Etudiant), place : entier, étudiant : Etudiant)
  début
  /* Il faut libérer la place suivant place en décalant tous les éléments situés après place vers la fin du tableau :*/
  pour j décroissant de 1 + lAdmis.nb à place + 2 faire
    lAdmis.tab[j] ← lAdmis.tab[j-1]
  fpour
  lAdmis.tab[place+1] ← étudiant
  lAdmis.nb ← lAdmis.nb+1
  fin

```

Les autres fonctions (supqlis, chglis, supplis, adjtlis) sont données en annexe 2.

## 2.2 Représentation contiguë dans un fichier séquentiel

Les éléments de la liste sont stockés dans un fichier séquentiel conservé en mémoire secondaire (sur disquette, disque dur, bande magnétique...).

Reprenons comme exemple la liste des admissions. Elle est représentée par : lAdmis : fichier Etudiant. Les fonctions logiques sont remplacées par des instructions de manipulations de fichiers propres au langage de programmation utilisé. Les adjonctions et suppressions nécessitent des recopies dans un fichier intermédiaire.

Dans le cas d'une représentation contiguë, le choix entre fichier et tableau sera fonction des critères suivants :

### Avantages pour le choix d'un fichier séquentiel

- conservation des informations,
- plus grande capacité,
- pas de surdimensionnement.

### Inconvénients pour ce choix :

- temps d'accès.

## 2.3 Conclusion

Dans une représentation contiguë, les places des éléments sont modifiées en cas d'adjonctions ou de suppressions dans la liste. Dans certains algorithmes, on mémorise des places dans des variables pour pouvoir y accéder après des adjonctions ou suppressions. Dans ce type de problème, le choix d'une représentation contiguë n'est pas possible. Il faudra choisir une représentation chaînée. On verra un tel exemple en exercice.

La représentation d'une liste de manière contiguë (par un tableau ou un fichier séquentiel), nécessite des décalages ou des recopies lors de modifications, ce qui est extrêmement coûteux en temps. Il est préférable de ne l'utiliser que lorsque les adjonctions et suppressions à l'intérieur de la liste sont rares.

# 3 Représentation chaînée des listes

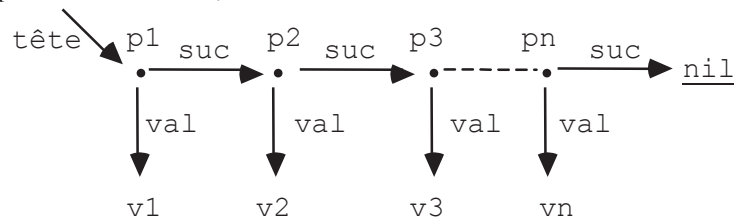
## 3.1 Généralités

Nous allons introduire un nouveau moyen de représenter les listes, qui est plus coûteux en place que le précédent mais économise du temps lors des modifications. Pour choisir l'un ou l'autre mode de représentation (contigu ou, comme nous l'introduisons maintenant, chaîné), il faudra trancher l'habituel dilemme : économiser temps ou place ? Le choix dépendra surtout de la fréquence des modifications mais aussi éventuellement de contraintes sur la place (limite de la saturation) ou le temps (réaction rapide nécessaire).

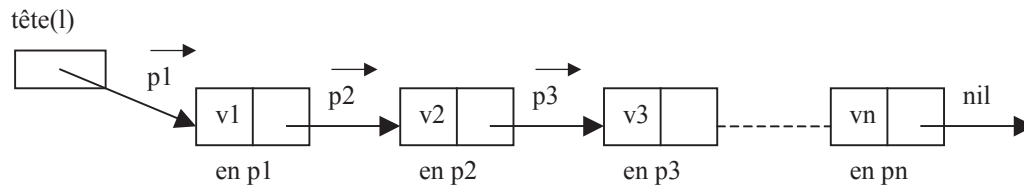
Dans la représentation contiguë, la fonction *suc* était représentée implicitement. Nous allons maintenant la représenter explicitement: chaque élément va comporter l'indication de son successeur. Nous verrons deux façons différentes de le faire. Chaque élément est un couple formé de sa valeur et de l'indication de la place du suivant. Remarquons que c'est la représentation qui ressemble le plus à la structure logique.

**Notation :**

Dans le cas d'une représentation chaînée, la liste l suivante



peut s'illustrer par



Remarque :

la flèche vers la place p est appelée "indicateur de place" et notée  $\vec{p}$ ; nil doit être considéré comme une place fictive et son indicateur. Dans l'exemple, p1 est une place et  $\vec{p1}$  est l'indicateur de la place p1.

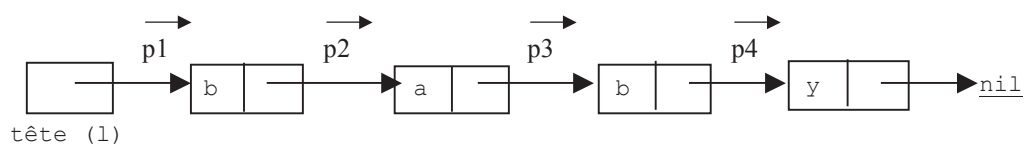
### 3.2 Représentation chaînée dans un tableau ou fichier direct

Nous présentons les représentations par tableaux, celles des fichiers s'en déduisant par analogie. Nous notons l la liste chaînée à représenter.

La liste chaînée est représentée par un tableau de variables composites à 2 champs: *val* et *suc*. Chaque élément contient sa valeur (dans le champ *val*) et l'indice de son successeur dans le tableau (dans le champ *suc*). Il suffit alors de connaître l'indice du premier élément (il est donné par la fonction *tête*) pour avoir accès à tous les éléments de la liste. Souvent, *nil* est représenté par 0 et la tête est représentée dans le champ *suc* de l'élément d'indice 0 du tableau.

#### Exemple :

La liste suivante :



peut être représentée de manière chaînée dans un tableau comme suit:

	0	1	2	3	4	5	6	7		indices
l :		y	b	b		a				val
	3	0	1	5		2				suc
	tête									
	<div style="display: inline-block; width: 20px; height: 15px; background-color: #cccccc; border: 1px solid black;"></div> = places libres									

Les éléments peuvent être placés n'importe où dans le tableau. Lors d'une adjonction, il faut donc trouver une place libre dans le tableau. Lors d'une suppression, il faut signaler que la place libérée peut de nouveau être utilisée. Cette gestion de l'espace libre doit être réalisée lors de la création de la liste et lors de chaque adjonction ou suppression d'éléments. Nous détaillerons dans la suite les trois méthodes les plus couramment utilisées.

Exercice : (jouez à la machine !)

Simulez l'évolution de la liste d'admissions lors de sa création à partir des données suivantes, sachant que cette liste est représentée de manière chaînée dans le tableau ladmis :

Jean 15  
 Marie 13  
 Paul 16  
 Albert 14  
 Germaine 18

Rappel : les éléments sont rangés par note décroissante.

l :	0		0 1 3 5
	1	Jean, 15	0 2 4
	2	Marie, 13	0
	3	Paul, 16	1
	4	Albert, 14	2
	5	Germaine, 18	3
		val	suc

### Expression des fonctions logiques dans le cas d'une représentation chaînée à l'aide d'un tableau :

Nous avons fait le choix d'écrire ces expressions sous forme de fonctions. Il est bien sûr toujours possible de remplacer simplement ces expressions par une suite d'instructions (les instructions qui constituent le corps des fonctions traductions).

- déclaration :

$l : \text{ListeChaînéeTableau} = \text{tableau} < \text{val} : \text{Elément}, \text{suc} : \text{entier} > [0..bs]$   
 où Elément est le type des éléments de la liste

- p est une place  $\langle == \rangle$  p est un indice

- suc (l,p)  $\langle == \rangle$   
fonction suc (l : ListeChaînéeTableau, p : entier) : entier  
début  
 pSuivant  $\leftarrow$  l [p].suc  
retourne pSuivant  
fin

- val (l,p)  $\langle == \rangle$   
fonction val (l : ListeChaînéeTableau, p : entier) : Elément  
début  
 valeur  $\leftarrow$  l [p].val  
retourne valeur  
fin

- tête (l)  $\langle == \rangle$   
fonction tête (l : ListeChaînéeTableau) : entier  
début  
 p  $\leftarrow$  l [0].suc  
retourne p  
fin

```

- finliste (l, p)      <==>
    fonction finliste (l : ListeChainéeTableau, p : entier) : booléen
        début
            fin ← p=0
            retourne fin
        fin

- l ← lisvide ( )      <==>
    fonction lisvide ( ) : ListeChainéeTableau
        début
            l[0].suc ← 0
            initialisation de l'espace libre
            retourne l
        fin

- adjtlis (l, v)       <==>
    fonction adjtlis (l InOut : ListeChainéeTableau, v : Élément)
        début
            recherche d'une place libre pl
            l[pl].val ← v
            l[pl].suc ← l[0].suc
            l[0].suc ← pl
        fin

- suptlis (l)          <==>
    fonction suptlis (l InOut : ListeChainéeTableau)
        début
            l[0].suc ← l[l[0].suc].suc
            restitution de la place libre
        fin

- adjqlis (l, v)       <==>
    fonction adjqlis (l InOut : ListeChainéeTableau, v : Élément)
        début
            recherche d'une place libre pl
            l[pl].val ← v
            l[pl].suc ← 0
            p ← 0
            tantque l[p].suc ≠ 0 faire
                p ← l[p].suc
            ftantque
            l[p].suc ← pl      (* p : place après laquelle se fait l'adjonction, indice de tête si liste vide *)
        fin

```

Lorsque la place intervenant dans la modification est déjà connue, on peut éviter les parcours de la liste donnée. Nous avons mis en italiques les instructions qui peuvent alors être supprimées dans le cas de la fonction *adjqlis*. Dans ce cas, les places concernées doivent bien sûr être passées en paramètre des fonctions traductions qui auraient alors un profil et un nom différents.

### Exercice :

Ecrire les algorithmes de programmation des autres fonctions (supqlis, adjlis, suplis et chglis).

```

- supqlis (l)          <==>
    fonction supqlis (l InOut : ListeChainéeTableau)
        début
            ancP ← 0
            p ← l[0].suc
            tantque l[p].suc ≠ 0 faire
                ancP ← p
                p ← l[p].suc
            ftantque
            l[ancP].suc ← 0      (* ancP est la place précédant celle qu'on supprime ; c'est
                                l'indice de la tête si la liste n'avait qu'un seul élément *)
            restitution de la place libre p
        fin

```