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

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*.

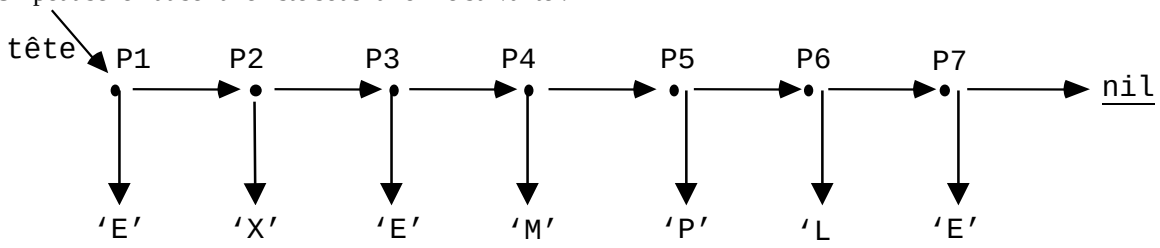
LES STRUCTURES LINEAIRES

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 :

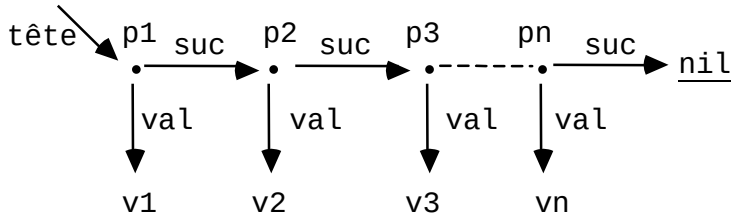
-	tête :	Liste (Valeur)	→	Place
-	val :	Liste (Valeur) x Place-{nil}	→	Valeur
-	suc :	Liste (Valeur) x Place-{nil}	→	Place
-	finliste :	Liste (Valeur) x Place	→	booléen
-	lisvide :		→	Liste (Valeur)
-	adjtlis :	Liste (Valeur) x Valeur	→	

- 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.

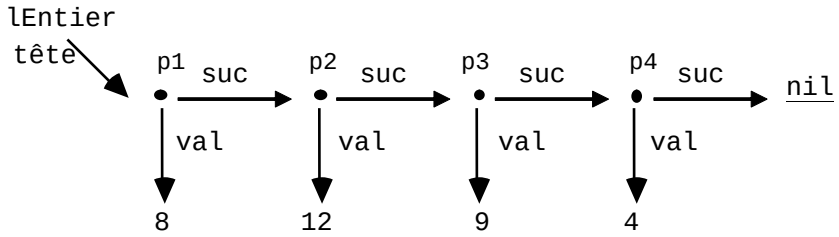
finliste :

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

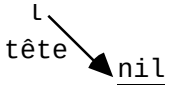
Soit la liste *lEliminés* contenant les nom, prénom et note des candidats ayant échoué à l'épreuve du permis de conduire. Ecrire l'algorithme logique de la fonction qui permet d'afficher le nom et le prénom de la première personne éliminée ou un message si aucune personne n'a échoué.

```
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 :

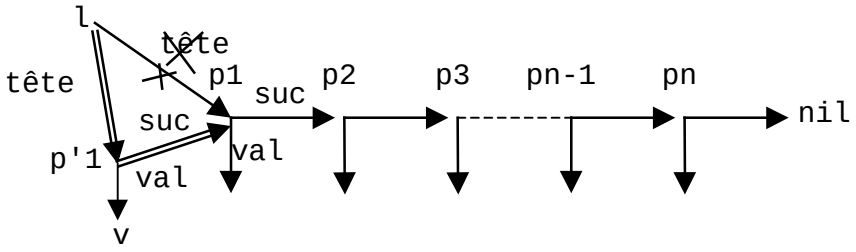
construction d'une liste vide c'est-à-dire d'une liste ne contenant aucun élément.
l ← lisvide() : création de la liste vide *l* :



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

adjtlis :

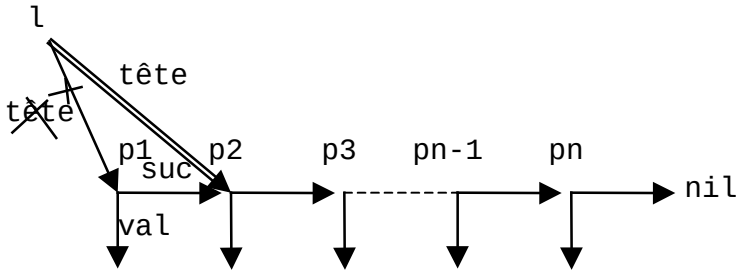
adjonction en tête de la liste ; c'est le cas où l'on insère un élément avant tous les autres ;
adjtlis (l, v) ajoute en tête de *l* un élément de valeur *v* :



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.

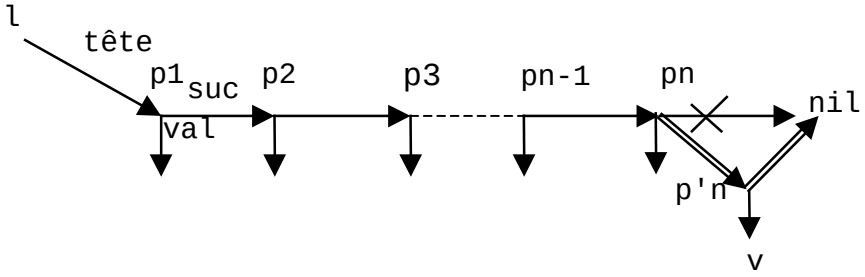
suptlis :

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



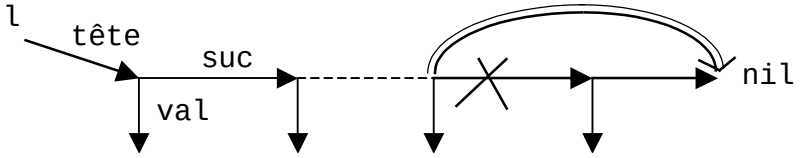
adjqlis :

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



supqlis :

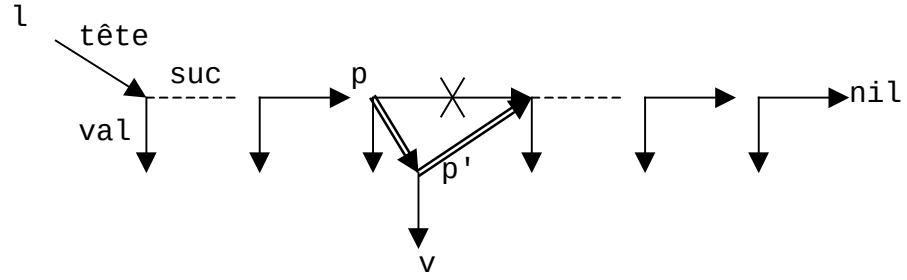
suppression en queue ;
supqlis(l) : c'est le cas où le dernier élément de la liste est enlevé.



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 :

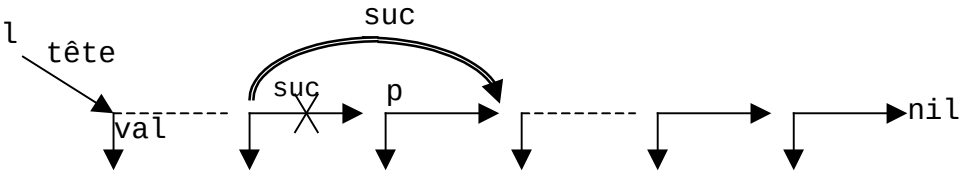
adjonction après une place p ;
adjlis (l, p, v) : adjonction à la liste *l* supposée non vide, juste après l'élément de place *p*, d'un nouvel élément de valeur *v*.



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

suplis :

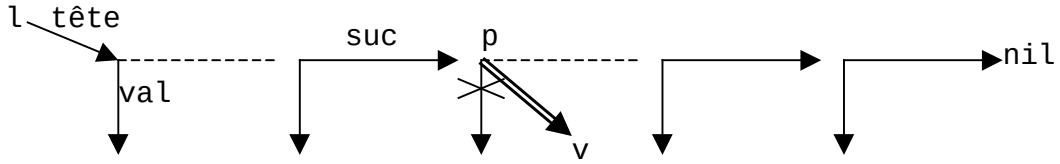
suppression d'un élément à une place donnée ;
suplis(l,p) : suppression dans la liste *l*, supposée non vide, de l'élément situé à la place *p*.



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

```
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 lLivreCréerListeTitre
    fonction lLivreCréerListeTitre (lLivre : ListeLivre, auteur : chaîne) : ListeTitre
    début(*1*)
      place ← tête(lLivre)
      arrêt ← faux
      tantque(*2*) non finliste (lLivre, place) et non arrêt faire  (* recherche du 1er livre de l'auteur  *)
            livre ← val (lLivre, place)
            si (livre.auteur >= auteur)
                  alors (*3a*) arrêt ← vrai
                  sinon (*3s*) place ← suc (lLivre, 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 (lLivre, place)
            bonAuteur ← vrai
            tantque(*5*) bonAuteur et non finliste (lLivre, place) faire
                  livre ← val (lLivre, place)
                  si livre.auteur = auteur alors(*6a*)  adjqlis (lTitre, livre.titre)
                                                        place ← suc (lLivre, place)
                                          sinon(*6s*) bonAuteur ← faux
                  fsi(*6*)
            fintantque(*5*)
      fsi(*4*)
      retourne lTitre
    fin(*1*)
Lexique
    Livre = < auteur : chaîne, titre : chaîne >
    ListeLivre = Liste (Livre)
    ListeTitre = Liste (chaîne)
    lLivre : ListeLivre
    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 lPiloteImprimerAprès10 (lPilote : Liste (chaîne))
début(*1*)
    i ← 1
    place ← tête (lPilote)
    tantque(*2*) non finliste (lPilote, place) et i <= 10 faire
            i ← i+1
            place ← suc(lPilote, place)
    ftantque(*2*)
    tantque(*3*) non finliste (lPilote, place) faire
            écrire (val (lPilote, place))
            place ← suc (lPilote, place)
    ftantque(*3*)
fin(*1*)
```

Lexique

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

Algorithme question b

```
fonction lPiloteImprimer3AprèsProst (lPilote : Liste (chaîne))
début(*1*)
    trouve ← faux
    place ← tête (lPilote)
    tantque (*2*) non finliste (lPilote, place) et non trouve faire
            si val (lPilote, place) = « A.Prost »
                    alors (*3a*) trouve ← vrai
                    sinon (*3s*) place ← suc (lPilote, place)
            fsi(*3*)
    ftantque(*2*)
    si trouve  alors (*4a*)
            place ← suc (lPilote, place)
            i ← 1
            tantque(*5*) non finliste (lPilote, place) et i <=3 faire
                    écrire (val (lPilote, place))
                    place ← suc (lPilote, place)
                    i ← i+1
            ftantque(*5*)
    sinon (*4s*) écrire (« A. Prost n'est pas dans la liste »)
    fsi (*4*)
fin(*1*)
```

Lexique

lPilote : 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) (*la 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)
    ftant (*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

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

lAdmis.nb

4

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 situé 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, suplis,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.

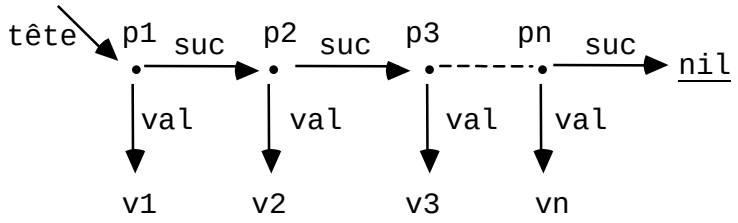
3 Représentation chaînée des listes

3.1 Généralités

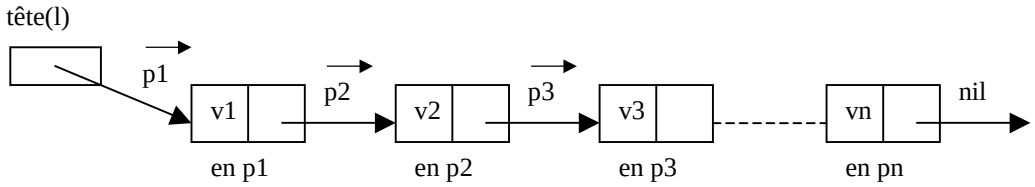
Nous allons introduire un nouveau moyen de représenter les listes, qui est plus coûteux en place que le précédent mais économise du temps lors des modifications. Pour choisir l'un ou l'autre mode de représentation (contigu ou, comme nous l'introduisons maintenant, chaîné), il faudra trancher l'habituel 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.

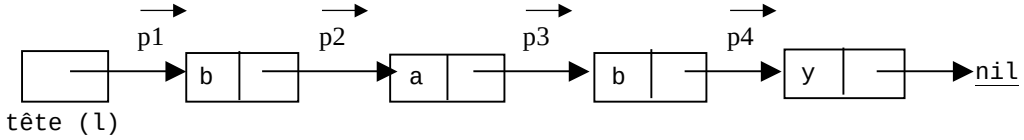
3.2 Représentation chaînée dans un tableau ou fichier direct

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

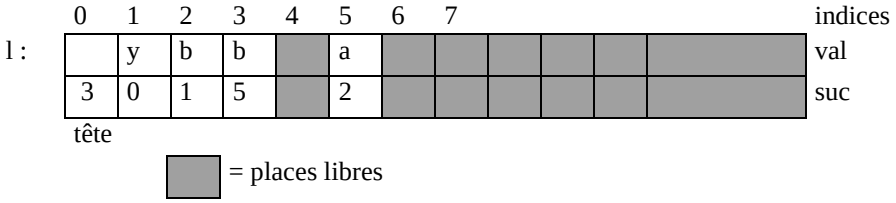
La liste chaînée est représentée par un tableau de variables composites à 2 champs: *val* et *suc*. Chaque élément contient sa valeur (dans le champ *val*) et l'indice de son successeur dans le tableau (dans le champ *suc*). Il suffit alors de connaître l'indice du premier élément (il est donné par la 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:



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 : ListeChainéeTableau = tableau < val : Elément, suc : entier> [0..bs]
  où Elément est le type des éléments de la liste

- p est une place <==> p est un indice

- suc (l,p) <==>

```
fonction suc (l : ListeChainéeTableau, p : entier) : entier
    début
    pSuivant ← l [p].suc
    retourne pSuivant
    fin
```

- val (l,p) <==>

```
fonction val (l : ListeChainéeTableau, p : entier) : Elément
    début
    valeur ← l [p].val
    retourne valeur
    fin
```

- tête (l) <==>

```
fonction tête (l : ListeChainéeTableau) : entier
    début
    p ← 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 : Elément)
        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 :Elément)
        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
```

- adjlis (l, p, v) <==>

```
fonction adjlis (l InOut : ListeChainéeTableau, p : entier, v : Elément)
        début
        recherche d'une place libre pL
        l[pL].val ← v
        l[pL].suc ← l[p].suc
        l[p].suc ← pL
        fin
```

- suplis (l, p) <==>

```
fonction suplis (l InOut : ListeChainéeTableau , p : entier)
        début
        ancP ← 0
        tantque l[ancP].suc ≠ p faire
                ancP ← l[ancP].suc
        ftantque
        l[ancP].suc ← l[p] .suc        (* ancP est la place précédant la place p à supprimer ; c'est l'indice
                                          de la tête si l'élément à supprimer était le 1er de la liste *)
        restitution de la place libre p
        fin
```

- chglis (l,p,v) <==>

```
fonction chglis (l InOut : ListeChainéeTableau , p : entier, v : Elément)
        début
        l[p].val ← v
        fin
```

Gestion de l'espace libre sans récupération des places libérées

On se positionne en début de tableau et on consomme les places les unes après les autres sans s'occuper de restituer les places libérées lors des suppressions. Un marqueur noté pLibre dans la suite, permet de désigner la première place non encore utilisée dans le tableau . Au fur et à mesure des adjonctions, cette place avance dans le tableau. Elle peut être mémorisée dans le champ suc du tableau à l'indice -1. Le tableau sera alors défini sur l'intervalle [-1..bs] .

Exemple :

	-1	0	1	2	3	4	5	**6**					
val			y	b	b		a						
suc	**6**	3	0	1	5		2						

↑
pLibre

Voyons comment se modifient les expressions des fonctions logiques :

initialisation de l'espace libre :

```
l[-1].suc ← 1
```

recherche d'une place pl libre :

```
pl ← l[-1].suc
l[-1].suc ← l[-1].suc + 1
```

Remarque :

Lorsque le marqueur de place libre arrive en fin de tableau, on peut faire une opération de "ramasse-miettes". Cette opération consiste à concentrer toutes les valeurs en début de tableau afin de récupérer les places libres dispersées dans le tableau.

Gestion de l'espace libre avec marquage et récupération des places libres

Cette solution permet de profiter des suppressions d'éléments pour récupérer de la place et la réutiliser. Elle consiste à marquer les places libres en y mettant une valeur particulière, par exemple -1 dans le champ *suc*.

Lors de l'initialisation de la liste à "vide", on marque à -1 le champ suc de toutes les places. Chaque fois qu'on supprime un élément, on indique que la place qu'il occupait est libre en mettant -1 dans le champ suc. Chaque fois qu'on veut ajouter un élément, on cherche dans le tableau une case dont le champ suc est marqué à -1.

Exemple :

	0	1	2	3	4	5	6	7							indices
l		y	b	b		a									val
	3	0	1	5	**-1**	2	**-1**	**-1**	**-1**	**-1**	**-1**	**-1**	**-1**	**-1**	suc

tête

Les algorithmes correspondant à cette gestion sont donnés en annexe 3.

Variantes :

- On peut aussi marquer les places libres en mettant une valeur "bidon" dans le champ val. Mais il n'est pas toujours possible de trouver une valeur "bidon".
- On peut ne pas marquer les places libres au fur et à mesure des libérations mais seulement en cas de problème (on consomme en suivant, tant qu'on peut). Cela se fait en marquant alors les places occupées, par un parcours de la liste. Les places libres sont les autres. Ennui: il faut un tableau supplémentaire, de booléens: lOccup.

Gestion de l'espace libre à l'aide d'une liste ("liste libre")

Le tableau l contient deux listes :

- la liste sur laquelle on travaille (liste de travail) ;
- la liste des places inoccupées (liste libre).

Les créations, suppressions et adjonctions mettent les deux listes à jour. Lorsque l'on veut ajouter un élément dans la liste de travail, on prend l'élément de la tête de la liste libre. Lorsqu'on supprime un élément dans la liste de travail, on ajoute sa place dans la liste libre (en tête).

Souvent, on représente la tête de la liste libre par le champ suc de l'élément d'indice -1 du tableau l. Nous retiendrons cette hypothèse.

Exemple :

	-1	0	1	2	3	**4**	5	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**
val			y	b	b		a								
suc	**4**	3	0	1	5	**6**	2	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**0**

↑

tête de la liste libre

Remarque :

Ceci fait partie du problème plus général de la gestion simultanée de plusieurs listes.

Voici comment compléter les expressions des fonctions logiques :

Déclarations

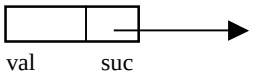
l : tableau <val : Elément, suc : entier > [-1..bs] où Elément est le type des éléments de la liste.

```
Initialisation de l'espace libre
        l [-1].suc ←1
        pour i de 1 à bs-1 faire
                l[i].suc ← i+1
        fpour
        l [bs].suc ← 0

Recherche d'une place libre pl  (en tête)   (on suppose qu'il y en a)
        pl ← l [-1].suc
        l [-1].suc ← l [pl].suc

Restitution d'une place libre pl  (en tête)
        l [pl].suc ← l [-1].suc
        l [-1].suc ← pl
```

3.3 Représentation chaînée à l'aide de pointeurs

Un **pointeur** est une variable dont la valeur est l'adresse d'une autre variable. On dit que le pointeur "pointe vers" l'autre variable. Les pointeurs permettent de gérer la mémoire de manière dynamique, c'est-à-dire lors de l'exécution du programme: on ne crée des variables que lorsque l'on en a besoin. De la même manière, lorsque l'on n'a plus besoin d'une variable, on peut récupérer la place qu'elle occupait.

Comment représenter une liste chaînée par des pointeurs en ne réservant en mémoire que la place nécessaire ? Une liste chaînée est, comme nous l'avons vu, une suite de couples (valeur, suivant) que l'on peut représenter par le schéma :



Le champ suc contient l'adresse de l'élément suivant c'est-à-dire un élément de type pointeur.

Pour matérialiser la tête de la liste, il faut un pointeur vers le premier élément. La connaissance de ce pointeur donne complètement accès à la liste ; pour cette raison, on confond la liste et son pointeur de tête.

Cette représentation est très utilisée et sera détaillée dans le langage C.

3.4 Exercice

Ecrire l'algorithme de programmation correspondant à l'algorithme logique de l'exercice 4 du paragraphe 1.4 en représentant les listes de manière chaînée dans un tableau et avec une gestion de l'espace libre sans récupération des places libérées.

La solution classique et généralement conseillée consiste à écrire simplement les fonctions traductions des fonctions logiques utilisées sur les listes. La solution demandée dans cet exercice ne devra pas utiliser de fonctions pour les traductions afin de permettre de réaliser facilement des optimisations.

Rappel de l'énoncé :

Soit une suite de valeurs entières (toutes comprises entre 0 et 1000). Construire une liste *lEntier*, triée par ordre croissant, contenant les entiers figurant un nombre impair de fois dans la suite des données.
(Principe : chaque donnée est cherchée dans la liste partiellement construite ; si elle appartient déjà à la liste, on la supprime, sinon on la rajoute.)

Algorithme logique (rappel) :

```
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) alors (*8a*)  (*la liste est vide*)
                        adjqlis (lEntier, valeur)
                sinon (*8s*)
                        placePrécédente ← place  (* initialisation artificielle *)
                        (*recherche de la valeur*)
                        fini ← faux
                        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 alors (*5a*)
                                (*valeur appartient déjà à la liste*)
                                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, suite des données
place : Place, place courante
placePrécédente : Place, place précédent la place *place*
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

Algorithme de programmation

```
fonction lentierCréer ( ) : LentierChaînéeTableau
        début(*1*)
        lentier[0].suc ← 0
        lentier[-1].suc ← 1
        nbvaleur ←  lire( )
        pour i de 1 à nbValeur faire (*2*)
            valeur ← lire( )
            place ← lEntier[0].suc
            si place = 0 alors (*8a*)
                    placeLibre ← lEntier[-1].suc
                    lEntier[-1].suc ←  lEntier[-1].suc + 1
                    lEntier [placeLibre ].val ← valeur
                    lEntier [placeLibre ].suc ← 0
                    lEntier[0].suc ← placeLibre
            sinon (*8s*)
                    placePrécédente ← place
                    fini ← faux
                    (*recherche de la valeur*)
                    tantque(*3*) place ≠ 0 et non fini faire
                            élément ←  lEntier[place].val
                            si valeur ≤ élément alors(*4a*)
                                    fini ← vrai
                            sinon(*4s*)
```

```
                                placePrécédente ← place
                                place ← lEntier[place].suc
                        fsi(*4*)
                ftantque(*3*)
                si valeur = élément alors(*5a*)
                        (*valeur appartient déjà à la liste*)
                        si place = placePrécédente alors(*6a*)
                                (*c'est le 1er élément*)
                                lEntier[0].suc ← lEntier[place].suc
                        sinon(*6s*)
                                lEntier[placePrécédente].suc ← lEntier[place].suc
                                (* en appliquant la traduction systématique, on écrirait :
                                placePrécédente ← 0
                                tantque(*20*) lEntier[placePrécédente].suc ≠ place faire
                                        placePrécédente ← lEntier[placePrécédente].suc
                                ftantque(*20*)
                                lEntier[placePrécédente].suc ← lEntier[place].suc
                                *)
                        fsi(*6*)
                sinon(*5s*) (*valeur n'appartient pas à la liste*)
                        placeLibre ← lEntier[-1].suc
                        lEntier[-1].suc ← lEntier[-1].suc+1
                        si place = placePrécédente alors(*7a*)
                                (*valeur < à toutes les valeurs*)
                                lEntier[placeLibre].val ← valeur
                                lEntier[placeLibre].suc ← lEntier[0].suc
                                lEntier[0].suc ← placeLibre
                        sinon(*7s*)
                                lEntier[placeLibre].val ← valeur
                                lEntier[placeLibre].suc ← lEntier[placePrécédente].suc
                                (*ou lEntier[placeLibre].suc ← place*)
                                lEntier[placePrécédente].suc ← placeLibre
                        fsi(*7*)
                fsi(*5*)
        fsi(*8*)
    fpour(*2*)
    retourne lEntier
    fin(*1*)
Lexique
    LEntierChaînéeTableau = tableau < val : entier, suc : entier > [-1...bs]
    lEntier : LEntierChaînéeTableau, tableau des valeurs et des successeurs
    nbValeur : entier, nombre de valeurs à traiter
    i : entier, indice d'itération
    bs : entier, nombre maximum d'éléments de la liste lEntier
    valeur : entier, suite des données
    place : entier, place courante dans lEntier
    placePrécédente : entier, place précédant place dans lEntier
    fini : booléen, vrai lorsqu'on a trouvé l'entier ou lorsque la valeur courante est supérieure à la valeur cherchée
    élément : entier, valeur courante
    placeLibre : entier, place libre sélectionnée pour la valeur courante
```

4 Les piles et les files

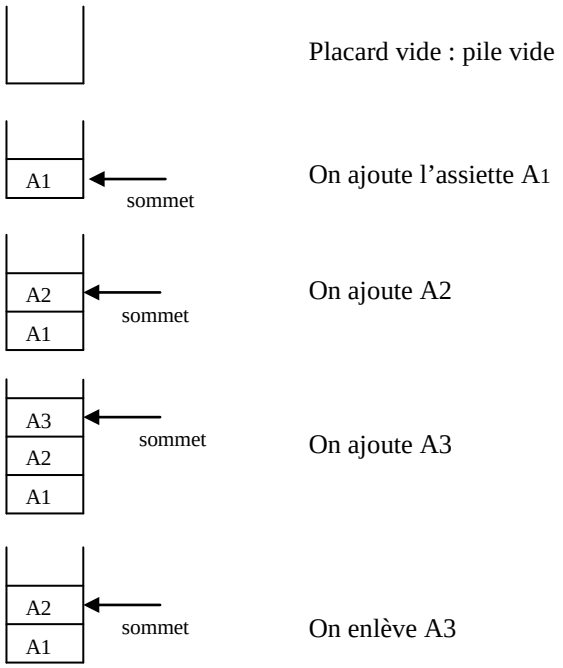
Pour beaucoup d'applications, les seules opérations à effectuer sur les listes sont des insertions et des suppressions aux extrémités.

4.1 Les piles

Une **pile** est une liste dans laquelle toutes les adjonctions et toutes les suppressions se font à une seule extrémité appelée *sommet*. Ainsi, le seul élément qu'on puisse supprimer est le plus récemment entré. Une pile a une structure « LIFO » pour « Last In, First Out » c'est-à-dire « dernier entré premier sorti ». Une bonne image pour se représenter une pile est une pile d'assiettes : c'est en haut de la pile qu'il faut prendre ou mettre une assiette !

Exemple de la pile d'assiettes :

On ne pose d'assiettes qu'au "sommet" de la pile. On n'en enlève également qu'au sommet.



On enlève une assiette : c'est A2.
Etc...

Les opérations sur une pile sont :

- tester si une pile est vide (*estVidePile*) ;
- accéder au sommet (*sommet*);
- empiler un élément (*empiler*);
- retirer l'élément qui se trouve au sommet (*dépiler*),
- créer une pile vide (*pileVide*).

Définition abstraite du type pile :

Soit *Valeur* un ensemble de valeurs (par exemple des entiers). On appelle type **pile de Valeur** et on note **Pile(Valeur)** l'ensemble des piles dont les valeurs sont des éléments de *Valeur*.

Ensembles définis et utilisés : *Pile*, Valeur,booléen

Description fonctionnelle des opérations :

-	pileVide :		→	Pile (Valeur)
-	sommet :	Pile (Valeur)-{pileVide( )}	→	Valeur
-	estVidePile :	Pile (Valeur)	→	booléen
-	empiler :	Pile (Valeur) x Valeur	→	
-	dépiler :	Pile (Valeur)-{pileVide( )}	→	

Les opérations empiler et dépiler modifient la pile donnée en paramètre.

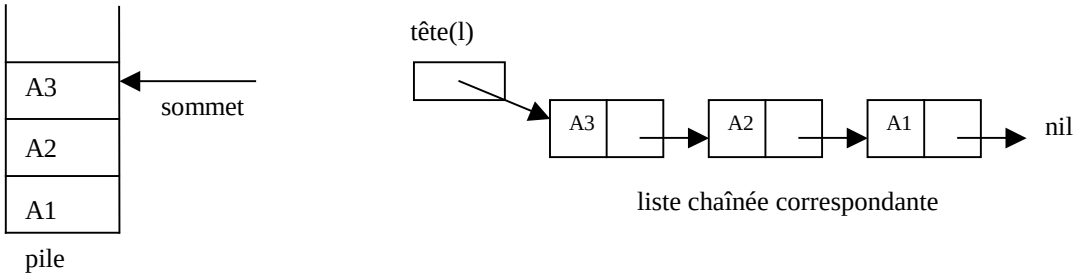
Utilité :

Les piles sont des structures fondamentales, et leur emploi dans les programmes informatiques est très fréquent. Le mécanisme d'appel des sous-programmes suit ce modèle de pile. Les logiciels qui proposent une fonction « undo » se servent également d'une pile pour défaire, en ordre inverse, les dernières actions effectuées par l'utilisateur.

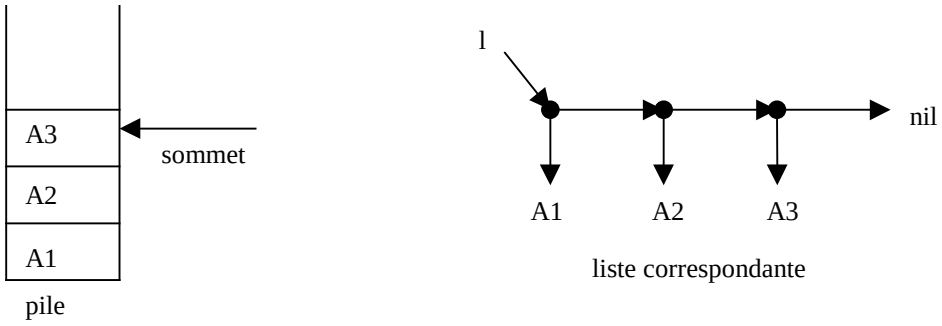
Représentations :

On peut utiliser pour implémenter les piles toutes les représentations étudiées pour les listes. Les opérations empiler et dépiler travailleront soit sur la tête de liste (dans le cas des représentations chaînées), soit sur la queue de liste (dans le cas des représentations contiguës) pour limiter la complexité.

Exemple :
Considérons le cas de la pile d'assiettes contenant 3 éléments. Dans le cas où on choisit une représentation chaînée, cette pile correspond à une liste dans laquelle les adjonctions et suppressions se font uniquement en tête :



Dans le cas où on choisit une représentation contiguë, la pile correspond à une liste dans laquelle les adjonctions et suppressions se font uniquement en queue :



Exercice :

On désire évaluer des expressions postfixées formées d'opérandes entiers positifs et des 2 opérateurs «+» et «-». On rappelle que, dans la notation postfixée, l'opérateur suit ses opérandes. Par exemple, l'expression infixée suivante :

(7 + 12) + (5 – 3) s'écrit 7 12 + 5 3 - +

L'évaluation d'une expression postfixée se fait simplement à l'aide d'une pile. L'expression est lue de gauche à droite. Chaque opérande lue est empilé et chaque opérateur trouve ses 2 opérandes en sommet de pile qu'il remplace par le résultat de son opération. Lorsque l'expression est entièrement lue, sans erreur, la pile ne contient qu'une seule valeur, le résultat de l'évaluation.

Question 1 : Ecrire l'algorithme logique de la fonction d'évaluation d'une expression postfixée supposée syntaxiquement correcte. On suppose disposer d'une fonction *chaineEntierConvertir* qui convertit une chaîne de caractères représentant un entier en cet entier. Les opérateurs et les opérandes sont séparés par des blancs et l'expression est suivie d'un « . ».

Question 2 : On choisit de représenter la pile de manière contiguë par un couple : tableau et indice du sommet, à l'aide du type suivant :

PileEntier = <tab : tableau entier [1..MAXTAB], sommet : entier>

On suppose que la taille du tableau est suffisante et donnée par la constante MAXTAB. Ecrire les algorithmes de programmation des opérations sur les piles.

Corrigé :
Question 1 :

```
fonction ExpPostfEvaluer ( ) : entier
    début (*1*)
    p ← pileVide( )
    chLue ← lire( )
    tant que chLue ≠ « . » faire (*2*)
        si chLue = « + » alors (*3a*)
            x ← opérandeRécupérer (p)
            y ← opérandeRécupérer (p)
            empiler (p, x + y)
        sinon (*3s*)
            si chLue = « - » alors (*4a*)
                x ← opérandeRécupérer (p)
                y ← opérandeRécupérer (p)
                empiler (p, y - x)
            sinon (*4*)
                opérande ← chaineEntierConvertir (chLue)
                empiler (p, opérande)
            fsi (*4*)
        fsi (*3*)
        chLue ← lire( )
    ftant (*2*)
    si non estVidePile (p) alors (*7a*)
        valeur ← sommet (p)
    fsi (*7*)
    retourne valeur
    fin (*1*)
Lexique
    p : Pile (entier)
    valeur : entier, valeur de l'expression
    chLue : chaîne, chaîne lue (un opérande, un opérateur ou « . »)
    x : entier, un opérande de la pile
    y : entier, un opérande de la pile
    opérande : entier, un opérande lue

fonction opérandeRécupérer (p InOut : Pile (entier)) : entier
    début (*1*)
    si non estVidePile (p) alors (*2a*)
        x ← sommet (p)
        dépiler (p)
    fsi (*2*)
    retourne x
    fin (*1*)
Lexique
    p : Pile (entier)
    x : entier, un opérande de la pile
```

Question 2 :

```
fonction sommet (p : PileEntier) : entier
    début
    valeur ← p.tab[p. sommet]
    retourne valeur
    fin

fonction pileVide ( ) : PileEntier
    début
    p.sommet ← 0
    retourne p
    fin

fonction estVidePile (p : PileEntier) : booléen
    début
    test ← (p.sommet = 0)
    retourne test
    fin
```

```
fonction empiler (p InOut : PileEntier, v : entier)
    début
    si p. sommet < MAXTAB alors (* test facultatif : ce cas ne devrait pas arriver*)
          p.sommet ← p.sommet + 1
          p.tab [p.sommet] ← v
    fsi
    fin

fonction dépiler (p InOut : PileEntier)
    début
    p.sommet ← p.sommet - 1
    fin
```

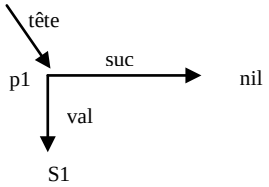
4.2 Les files

Une **file** est une liste dans laquelle toutes les adjonctions se font en queue et toutes les suppressions en tête. Autrement dit, on ne sait ajouter des éléments qu'en queue et le seul élément qu'on puisse supprimer est le plus anciennement entré. Par analogie avec les files d'attente, on dit que l'élément présent depuis le plus longtemps est le premier, on dit aussi qu'il est en tête. Une file a une structure "FIFO" (First In, First Out) c'est-à-dire « premier entré premier sorti ».

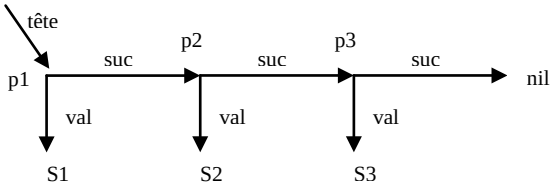
Exemple : file d'attente, par exemple queue (disciplinée) au téléski

Le matin :

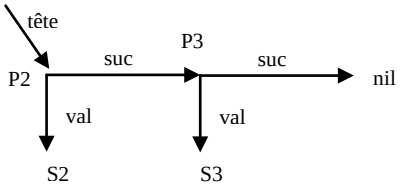
Arrivée du skieur S1



Arrivée des skieurs S2 puis S3



Montée d'un skieur : c'est S1. Il est supprimé de la file.



etc.

Les opérations sur une file sont :

- tester si une file est vide (*estVideFile*) ;
- accéder au premier élément de la file (*premier*);
- ajouter un élément dans la file (*adjfil*);
- retirer le premier élément de la file (*supfil*),
- créer une file vide (*fileVide*).

Définition abstraite du type file :

Soit *Valeur* un ensemble de valeurs (par exemple des entiers). On appelle type **file de Valeur** et on note **File(Valeur)** l'ensemble des files dont les valeurs sont des éléments de *Valeur*.

Ensembles définis et utilisés : *File*, Valeur, booléen

Description fonctionnelle des opérations :

-	fileVide :		→	File (Valeur)
-	premier :	File (Valeur)- {fileVide( )}	→	Valeur
-	estVideFile :	File (Valeur)	→	booléen
-	adjfil :	File (Valeur) x Valeur	→	
-	supfil :	File (Valeur)-{fileVide( )}	→	

Les opérations *adjfil* et *supfil* modifient la file donnée en paramètre. L'opération *adjfil* est parfois appelée *enfiler* et l'opération *supfil défiler*.

Utilité :

Le modèle de file est très utilisé en informatique. On le retrouve dans de nombreuses situations, comme, par exemple, dans la file d'attente d'un gestionnaire d'impression d'un système d'exploitation.

Représentations :

On peut utiliser pour implémenter les files toutes les représentations étudiées pour les listes. Mais pour limiter la complexité, on a intérêt, pour les files, à gérer un indicateur de tête. Une représentation souvent satisfaisante est la représentation contiguë dans un tableau avec tête mobile. Dans ce cas, la file est représentée par un triplet (tableau, tête, nombre d'éléments). Lorsqu'on arrive en fin de tableau et non en fin de file, on continue le parcours en se plaçant au début du tableau. Le nombre d'éléments de la file est bien sûr limité à la taille du tableau.

Exemple :
Soit une file d'attente de skieurs contenant 5 éléments. Elle peut être représentée de la manière suivante :

	1	2	3	4	5	6	7	8	9	10
f.tab :	s4	s5						s1	s2	s3

f.tête : 8

f.nb : 5

Exercice :

Ecrire les algorithmes de programmation des fonctions *fileVide*, *premier*, *adjfil*, *supfil* et *estVideFile* dans le cas d'une file d'entiers représentée de manière contiguë à l'aide d'un triplé (tableau, tête, nombre d'éléments) comme proposée dans l'exemple précédent. On suppose que la taille du tableau est suffisante et donnée par la constante MAXTAB.

Corrigé

```
- fonction fileVide ( ) : FileEntier
      début
      f.nb ← 0
      f.tête ← 1 (* pour éviter le cas particulier de l'adjonction dans une file vide *)
      retourne f
      fin
  Lexique
    FileEntier = <tab : tableau entier [1..MAXTAB], tête : entier, nb : entier >
    f : FileEntier

- fonction premier (f : FileEntier) : entier
      début
      retourne f.tab [f.tête]
      fin
  Lexique
    f : FileEntier
```

- fonction **adjfil** (f InOut: FileEntier, v : entier)
  début
  indice ← (f.tête + f.nb – 1) mod MAXTAB + 1
  /*ou : si f.tete + f.nb > MAXTAB alors indice ← f.tete + f.nb - MAXTAB sinon indice ← f.tete + f.nb fsi */
  f.tab [indice] ← v
  f.nb ← f.nb + 1
  fin
  Lexique
  f : FileEntier
  v : entier, valeur à ajouter dans la file
  indice : entier, indice de *v* dans *f.tab*

- fonction **supfil** (f InOut : FileEntier)
  début
  f.tête ← f.tête mod MAXTAB + 1
  /*ou : si f.tete = MAXTAB alors f.tete ← 1 sinon f.tete ← f.tete + 1 fsi */
  f.nb ← f.nb - 1
  fin
  Lexique
  f : FileEntier

- fonction **estVidefile** (f : FileEntier) : booléen
  début
  retourne (f.nb = 0)
  fin
  Lexique
  f : FileEntier

4.3 Les files avec priorité

Les files avec priorité remettent en question le modèle FIFO des files ordinaires. Avec ces files, l'ordre d'arrivée des éléments n'est plus respecté. Les éléments sont munis d'une priorité et ceux qui possèdent les priorités les plus fortes sont traités en premier.

Les opérations sur une file avec priorité sont :
- tester si la file est vide (*estVidefp)* ;
- accéder à l'élément le plus prioritaire de la file (*premierfp*);
- ajouter un élément et sa priorité dans la file (*adjfp*);
- retirer l'élément le plus prioritaire de la file (*supfp*),
- créer une file vide (*fpVide*).

Définition abstraite du type filePriorité :

Soit *Valeur* un ensemble de valeurs (par exemple des entiers), munies d'une priorité prise dans un ensemble notée *Priorité* qui est pourvu d'une relation d'ordre total permettant d'ordonner les éléments du plus prioritaire au moins prioritaire. On appelle type **FilePriorité de Valeur** et on note **FilePriorité(Valeur)** l'ensemble des files avec priorité dont les valeurs sont des éléments de *Valeur*.

Ensembles définis et utilisés : *FilePriorité*, Valeur, Priorité, booléen

Description fonctionnelle des opérations :

-	fpVide :		→	FilePriorité (Valeur)
-	premierfp :	FilePriorité (Valeur) -{fpVide( )}	→	Valeur
-	estVidefp :	FilePriorité (Valeur)	→	booléen
-	adjfp :	FilePriorité (Valeur) x Valeur x Priorité	→	
-	supfp :	FilePriorité (Valeur)-{fpVide( )}	→	

Les opérations *adjfp* et *supfp* modifient la file avec priorité donnée en paramètre

Utilité :

Les systèmes d'exploitation utilisent fréquemment les files avec priorité, par exemple, pour gérer l'accès des travaux d'impression à une imprimante, ou encore l'accès des processus au processeur.

Représentations :

On peut utiliser pour implémenter les files avec priorité toutes les représentations étudiées pour les listes. Si on choisit une liste non ordonnée, l'adjonction peut se faire en tête de liste. Les opérations *premierfp* et *supfp* nécessitent alors une recherche linéaire de l'élément le plus prioritaire. Cette recherche peut demander un parcours complet de la liste. Si on choisit une liste ordonnée, on peut placer les éléments par ordre de priorité décroissante. C'est alors l'opération d'adjonction qui peut nécessiter un parcours complet de la liste.

Exercice :

On s'intéresse à la gestion des travaux en attente d'impression. Ils sont munis d'une priorité dépendant des utilisateurs. Cette priorité est exprimée sous forme d'un entier de 1 pour les plus prioritaires à 5 pour les moins prioritaires. Les travaux sont placés dans une file avec priorité. On choisit une représentation contiguë dans un tableau avec tête mobile sans ordonner les éléments. Un élément de la file est un couple (nom du fichier à imprimer, sa priorité).

Exemple :

	1	2	3	4	5	6	7	8	9	10	
f.tab :	LS_pho	FP_fic	AN_fac					LS_let	CD_ess	AN_pai	nom
	2	5	1					2	4	1	priorité

f.tête :	8
f.nb :	6

Question 1 : Ecrire l'algorithme de programmation de la fonction *premierfp*. On suppose que la taille du tableau est suffisante et donnée par la constante MAXTAB.

```
Corrigé
fonction premierfp (f : FilePrioritéTravaux) : chaîne
début (*1*)
    maxPr ← f.tab [f.tête].priorité
    nomFichPr ←f.tab [f.tête].nom
    i ← f.tête mod  MAXTAB + 1
    (* ou : si f.tete = MAXTAB alors i ← 1 sinon i ← f.tete + 1 fsi *)
    nbParcouru ← 1
    arrêt ← maxPr = 1
    tantque(*2*) non arrêt et nbParcouru ≤ f.nb faire
        si f.tab [i].priorité < maxPr alors (*3a*)
                maxPr ← f.tab [i].priorité
                nomFichPr ← f.tab[i].nom
        fsi(*3*)
        nbParcouru ← nbParcouru + 1
        i ← i mod  MAXTAB + 1
        (* ou bien :   si i = MAXTAB alors i ← 1 sinon i ← i + 1 fsi  *)
        arrêt ← (maxPr = 1)
    fintantque(*2*)
    nomFich ← nomFichPr
    retourne nomFich
fin (*1*)
Lexique
      FilePrioritéTravaux = < tab : tableau Travail [1..MAXTAB], tête : entier, nb : entier >
      Travail = < nom : chaîne, priorité : entier >
      f : FilePrioritéTravaux
      nomFich : chaîne, nom du fichier à imprimer
      maxPr : entier, la plus grande priorité rencontrée dans la file à un instant du parcours
      nomFichPr : chaîne, nom du fichier ayant cette priorité
      i : entier, indice de parcours dans f.tab
      arrêt : booléen, à vrai dès qu'on rencontre un travail de priorité 1
      nbParcouru : entier, nombre d'éléments parcourus au rang i
```

Question 2 : Même question en supposant que les éléments sont ordonnés par priorité dans la file.

Exemple :

	1	2	3	4	5	6	7	8	9	10	
f.tab :	LS_pho	CD_ess	FP_fic					AN_pai	AN_fac	LS_let	nom
	2	4	5					1	1	2	priorité

f.tête :	8
f.nb :	6

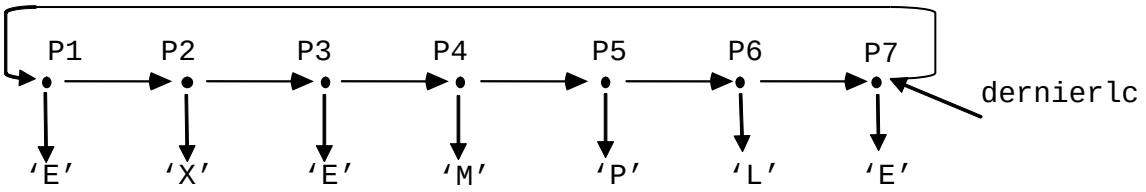
Corrigé

```
fonction premierfp (f : FilePrioritéTravaux) : chaîne
début (*1*)
    nomFich ← f.tab [f.tête].nom
    retourne nomFich
fin (*1*)
Lexique
      f : FilePrioritéTravaux
      nomfich : chaîne, nom du fichier à imprimer
```

5 Les listes circulaires et les listes symétriques

5.1 Les listes circulaires

Une liste circulaire est une liste telle que le dernier élément de la liste a pour successeur le premier. On peut ainsi parcourir toute la liste à partir de n'importe quel élément. Il faut pouvoir identifier la tête de liste. Mais il est plus avantageux de remplacer l'indication sur le premier élément par une indication sur le dernier, ce qui donne facilement accès au dernier et au premier qui est le suivant du dernier.



Définition abstraite :

Soit *Valeur* un ensemble de valeurs (par exemple des entiers). On appelle type **Liste Circulaire de Valeur** et on note **ListeCirc (Valeur)** l'ensemble des listes circulaires dont les valeurs sont des éléments de *Valeur*.

Ensembles définis et utilisés : *ListeCirc*, Valeur, Place, booléen

Description fonctionnelle des opérations :

-	dernierlc :	ListeCirc (Valeur)	→	Place
-	vallc :	ListeCirc (Valeur) x Place-{nil}	→	Valeur
-	suclc :	ListeCirc (Valeur) x Place-{nil}	→	Place
-	estvidelc :	ListeCirc (Valeur)	→	booléen
-	lcvide :		→	ListeCirc (Valeur)
-	adjtlc :	ListeCirc (Valeur) x Valeur	→	
-	suptlc :	ListeCirc (Valeur)-{lcvide( )}	→	
-	adjqlc :	ListeCirc (Valeur) x Valeur	→	
-	supqlc :	ListeCirc (Valeur)-{lcvide( )}	→	
-	adjlc :	(ListeCirc (Valeur)-{lcvide( )})x(Place-{nil}) x Valeur	→	
-	suplc :	(ListeCirc (Valeur)-{lcvide( )})x(Place-{nil})	→	
-	chglc :	(ListeCirc (Valeur)-{lcvide( )})x(Place-{nil}) x Valeur	→	

Les opérations *adjtlc*, *suptlc*, *adjqlc*, *supqlc* , *adjlc*, *suplc* et *chglc* modifient la liste circulaire donnée en paramètre. L'opération *estvidelc* rend vrai si la liste circulaire est vide. Si la liste contient un seul élément, celui-ci est son propre successeur.

Exercice :

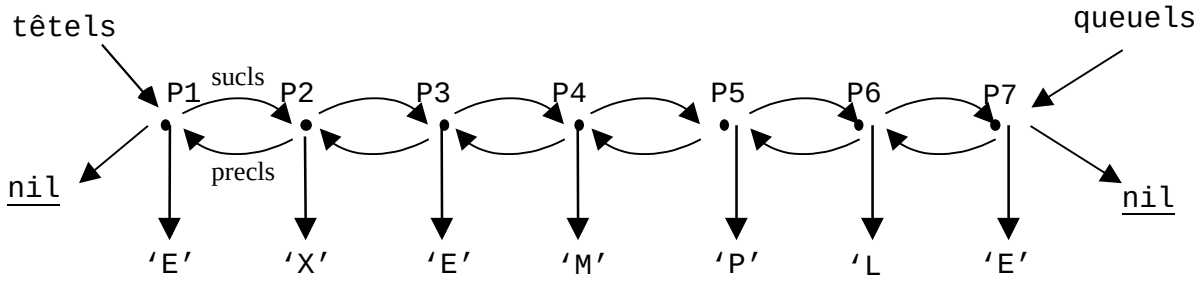
Pour jouer à AM-STRAM-GRAM, des enfants forment une ronde, choisissent l'un d'eux comme le premier, commencent à compter à partir de celui-ci et décident que le kième enfant doit quitter la ronde. Ils recommencent ensuite en comptant à partir de l'enfant qui suivait celui qui est sorti, et ainsi de suite. En représentant la ronde des enfants par une liste circulaire, écrire l'algorithme logique de la fonction qui permet d'imprimer la suite des prénoms des enfants dans l'ordre où ils sont sortis de la ronde. On suppose que la ronde est non vide au départ et que les enfants sont représentés par leurs prénoms.

```
Corrigé
fonction AM-STRAM-GRAM (ronde InOut : Ronde, nomPremier : chaîne, k : entier)
    début (*1*)
    p ← rondeChercherDépart (ronde, nomPremier)
    tantque(*2*) non estvidelc (ronde) faire
            // on compte jusqu'au k^ième
            pour i de 1 à k-1 faire (*3a*)
                    p ← suclc (ronde, p)
            fpour
            // l'enfant à la place pSup quitte la ronde
            pSup ← p
            écrire (vallc (ronde, pSup))
            p ← suclc (ronde, p)
            suplc (ronde, pSup)
    fintantque(*2*)
    fin (*1*)
Lexique
  Ronde = ListeCirc (chaîne)
  ronde : Ronde
  nomPremier : chaîne, nom de l'enfant à partir duquel on commence à compter
  k : entier
  p : Place, place courante dans la liste
  i : entier, compteur
  psup : Place, place à supprimer

fonction rondeChercherDépart (ronde : Ronde, nom : chaîne) : Place
    début (*1*)
    p ← dernierlc (ronde)
    arrêt ← faux
    tantque(*2*) non arrêt faire
        si vallc (ronde, p) = nom alors (*3a*)
                arrêt ← vrai
        sinon (*3s*)
                p ← suclc (ronde, p)
                si  p = dernierlc (ronde)  alors (*4a*)
                        arrêt ← vrai
                fsi(*4*)
        fsi(*3*)
    fintantque(*2*)
    retourne p
    (* si le nom n'a pas été trouvé dans la ronde, on commence par le dernier élément de la liste circulaire*)
    fin (*1*)
Lexique
  ronde : Ronde, liste circulaire non vide
  nom : chaîne, nom de l'enfant à partir duquel on commencera à compter
  p : Place, place courante dans la liste puis place de l'enfant cherché s'il existe dans la liste, sinon place du dernier
  arrêt : booléen, à vrai si on a trouvé l'enfant à partir duquel on commencera à compter ou si on a parcouru la liste
      en entier
```

Attention: On mémorise une place dans une variable pour y accéder après une suppression dans la liste. Une représentation contiguë ne sera donc pas possible pour cette liste car les places seraient modifiées lors de la suppression et la place mémorisée ne correspondrait plus à celle souhaitée.

5.2 Les listes symétriques

Une liste symétrique est une liste telle que chaque élément désigne l'élément suivant et l'élément précédent.



L'intérêt des listes symétriques réside dans le fait qu'il est facile d'extraire un élément à partir de sa place. Il n'est pas nécessaire de parcourir la liste pour retrouver le précédent.

Définition abstraite :

Soit *Valeur* un ensemble de valeurs (par exemple des entiers). On appelle type **Liste Symétrique de Valeur** et on note **ListeSym (Valeur)** l'ensemble des listes symétriques dont les valeurs sont des éléments de *Valeur*.

Ensembles définis et utilisés : *ListeSym*, Valeur, Place, booléen

Description fonctionnelle des opérations :

-	têtels :	ListeSym (Valeur)	→	Place
-	queuels :	ListeSym (Valeur)	→	Place
-	valls :	ListeSym (Valeur) x Place-{nil}	→	Valeur
-	sucls :	ListeSym (Valeur) x Place-{nil}	→	Place
-	precls :	ListeSym (Valeur) x Place-{nil}	→	Place
-	finls :	ListeSym (Valeur) x Place	→	booléen
-	lsvide :		→	ListeSym (Valeur)
-	adjtls :	ListeSym (Valeur) x Valeur	→	
-	suptls :	ListeSym (Valeur)-{lsvide( )}	→	
-	adjqls :	ListeSym (Valeur) x Valeur	→	
-	supqls :	ListeSym (Valeur)-{lsvide( )}	→	
-	adjls :	(ListeSym (Valeur)-{lsvide( )})x(Place-{nil})xValeur	→	
-	supls :	(ListeSym (Valeur)-{lsvide( )})x(Place-{nil})	→	
-	chgls :	(ListeSym (Valeur)-{lsvide( )})x(Place-{nil})xValeur	→	

Les opérations *adjtls*, *suptls*, *adjqls*, *supqls* , *adjls*, *supls* et *chgls* modifient la liste symétrique donnée en paramètre.

Exercice :

On appelle palindrome un mot qui se lit de la même façon de gauche à droite et de droite à gauche. Par exemple, les mots « elle » et « radar » sont des palindromes. Un mot étant représenté par une liste symétrique de caractères, écrire l'algorithme logique de la fonction qui indique si un mot est un palindrome. On suppose que le mot à analyser comprend au moins 2 lettres.

Corrigé

```
Fonction motChercherPalindrome (mot : Mot) : booléen
        début (*1*)
        pDeb ← têtels (mot)
        pFin ← queuels (mot)
        palindrome ← vrai
        arrêt ← faux
        tant que non arrêt faire (*2*)
            vDeb ← valls ( mot, pDeb)
```

```
                vFin ← valls ( mot, pFin)
                si vDeb ≠ vFin alors (*3a*)
                        arrêt ← vrai
                        palindrome ← faux
                sinon (*3s*)
                        pDeb ← sucls (mot, pDeb)
                        si pDeb = pFin alors (*4a*)
                                arrêt ← vrai
                        sinon (*4s*)
                                pFin ← precls (mot, pFin)
                                si pDeb = pFin alors (*5a*)
                                        arrêt ← vrai
                                fsi (*5*)
                        fsi (*4*)
                fsi (*3*)
        ftant (*2*)
        retourne palindrome
        fin (*1*)
Lexique
        Mot = ListeSym (caractère)
        mot : Mot
        pDeb :  Place, place courante à partir du début
        pFin : Place, place courante à partir de la fin
        vDeb : caractère, caractère courant à partir du début
        vFin : caractère, caractère courant à partir de la fin
        arrêt : booléen , à vrai lorsque le mot a été parcouru en entier ou dès qu'il ne peut plus être un palindrome
        palindrome : booléen, à vrai si le mot est un palindrome
```

6 Exercices récapitulatifs

Exercice 1 : liste d'admissions

Reprenons l'exemple de la liste d'admissions des étudiants dans une école. Rappelons qu'elle est triée par note décroissante et qu'elle contient pour chaque étudiant son nom et une note.
1) Ecrire l'algorithme logique de la fonction de recherche de la place d'un étudiant à l'aide de son nom dans la liste des admissions.
2) Même question pour l'adjonction d'un couple (nom, note) dans la liste, supposée non vide au départ.
3) Même question pour la suppression dans la liste d'un étudiant donné par son nom. On suppose la liste non vide au départ.

```
Correction question 1 :
Fonction lEtudChercherElément (listeEtudiant : Liste(Etudiant), nomEtudiant : chaîne) : Place
        début (*1*)
        placeElémEtudiant ← tete (listeEtudiant)
        trouve ← faux
        tant que non finliste (listeEtudiant, placeElémEtudiant) et non trouve faire (*2*)
            élémEtudiant ← val (listeEtudiant, placeElémEtudiant)
            si élémEtudiant.nom = nomEtudiant
                    alors (*3a*) trouve ← vrai
                    sinon (*3s*) placeElémEtudiant ← suc (listeEtudiant, placeElémEtudiant)
            fsi (*3*)
        ftant (*2*)
        retourne placeElémEtudiant
        fin (*1*)
Lexique
        listeEtudiant : Liste(Etudiant)
        nomEtudiant : chaîne, nom de l'étudiant dont on cherche la place dans listeEtudiant
        placeElémEtudiant : Place, place de l'étudiant cherché ou nil s'il n'y est pas
        élémEtudiant : Etudiant
        trouve : booléen, à vrai si l'étudiant cherché est dans la liste
```

Correction question 2 :

```
Fonction lEtudInsérerEtudiant (listeEtudiant InOut : Liste(Etudiant), étudiant : Etudiant)
    début(*1*)
        placeCourante ← tête (listeEtudiant)
        étudiantCourant ← val (listeEtudiant, placeCourante)
        si étudiant.note > étudiantCourant.note alors(*2a*) (*adjonction en tête*) adjtlis (listeEtudiant, étudiant)
        sinon(*2s*) (*cas général*)
                arrêt ← faux
                placePrécédente ← placeCourante
                placeCourante ← suc (listeEtudiant, placeCourante)
                tantque(*3*) non finliste (listeEtudiant, placeCourante) et non arrêt faire
                        étudiantCourant ← val (listeEtudiant, placeCourante)
                        si étudiant.note > étudiantCourant.note
                                alors (*4a*)
                                        arrêt ← vrai
                                sinon (*4s*)
                                        placePrécédente ← placeCourante
                                        placeCourante ← suc (listeEtudiant, placeCourante)
                        fsi(*4*)
                ftantque(*3*)
                adjlis (listeEtudiant, placePrécédente, étudiant)
        fsi(*2*)
    fin(*1*)
```

Lexique

Etudiant = <nom : chaîne, note : réel>
listeEtudiant : Liste(Etudiant), liste d'admission
étudiant : Etudiant, étudiant et note à ajouter
placeCourante : Place, suite des places de *listeEtudiant*
étudiantCourant : Etudiant, suite des valeurs de *listeEtudiant*
arrêt : booléen, à vrai lorsqu'on a trouvé l'endroit où insérer
placePrécédente : Place, place précédant la place *placeCourante*. Valeur finale : place après laquelle il faut insérer.

Correction question 3 :

```
Fonction lEtudSupprimerEtudiant (listeEtudiant InOut : Liste(Etudiant), nomEtudiant : chaîne)
    début(*1*)
    place ← lEtudChercherElément (listeEtudiant, nomEtudiant)
    si non finliste (listeEtudiant, place) alors (*2a*)
            suplis (listeEtudiant, place)
    fsi(*2*)
    fin(*1*)
```

Lexique

listeEtudiant : Liste(Etudiant)
nomEtudiant : chaîne, nom de l'étudiant à supprimer
place : Place, place de l'étudiant à supprimer ou nil
fonction lEtudChercherElément (listeEtudiant : Liste(Etudiant), nomEtudiant : chaîne) : Place, recherche de la place d'un étudiant à l'aide de son nom dans la liste des admissions

Exercice 2 : permis de conduire

On dispose d'une liste existante de candidats se présentant au permis de conduire; chaque candidat, désigné par son nom, obtient trois notes: conduite en ville, conduite sur route et manœuvre. Un candidat est reçu si la somme des 3 notes est >= 92. Ecrire l'algorithme logique de la fonction qui affiche le nom et la somme des 3 notes des candidats reçus puis le nom et la somme des 3 notes des candidats ayant échoué.
Remarque : on ne veut pas faire 2 parcours de la liste des candidats.

Correction Exercice 2 : Permis de conduire

```
Fonction imprimerRésultatsPermis (lCand : Liste (Candidat))
    début(*1*)
    lInter ← lisvide( )
    placeLCand ← tête(lCand)
    tantque(*2*) non finliste (lCand, placeLCand) faire
        candidat ← val (lCand, placeLCand)
        somme ← candidat.noteVille + candidat.noteRoute + candidat.noteManoeuvre
```

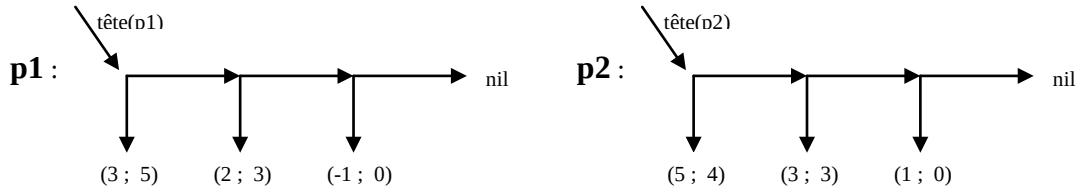
```
        si somme ≥ 92 alors(*3a*)
                écrire (candidat.nom, somme)
        sinon(*3s*)
                candidatEchoué ← (candidat.nom, somme)
                adjqlis (lInter, candidatEchoué)
        fsi(*3*)
        placeLcand ← suc (lcand, placeLcand)
    ftantque(*2*)
    placeLInter ← tête (lInter)
    tantque(*4*) non finliste (lInter, placeLInter) faire
        écrire (val (lInter, placeLInter))
        placeLInter ← suc (lInter, placeLInter)
    ftantque(*4*)
    fin(*1*)
Lexique
    Candidat =< nom : chaîne, noteVille : entier, noteRoute : entier, noteManoeuvre : entier>
    CandidatEchoué = < nomCandidat : chaîne, noteCandidat : entier >
    lCand : Liste (Candidat), liste de tous les candidats
    lInter : Liste (CandidatEchoué), liste des candidats ayant échoué
    placeLCand : Place, place courante de lCand
    candidat : Candidat, élément courant de lCand
    somme : entier, somme des notes du candidat courant
    placeLInter : Place, place courante de lInter
    candidatEchoué : CandidatEchoué, élément courant de lInter
```

Exercice 3 : les polynômes

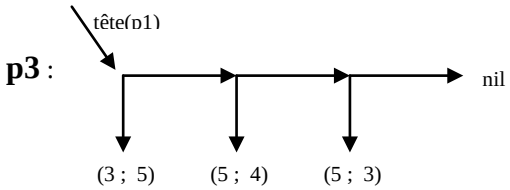
Il s'agit de mémoriser des polynômes d'une variable réelle et de réaliser des opérations sur ces polynômes. Un polynôme sera représenté par une liste de monômes, un monôme étant un couple (coefficient, exposant). Les monômes sont rangés dans la liste par exposant décroissant.

Exemple :

Soit les polynômes p1 = $3x^5+2x^3-1$ et p2 = $5x^4+3x^3+1$, ils seront représentés comme suit :



La valeur de p1 est 4 pour x = 1. Soit p3 le polynôme résultant de l'addition de p1 et p2, on a p3 = $3x^5+5x^4+5x^3$.



Ecrire les algorithmes logiques des fonctions permettant de répondre aux questions suivantes.

Question 1 : Calculer la valeur d'un polynôme pour une valeur de x donnée. On suppose disposer d'une fonction *puissanceEntièreCalculer* qui, à partir d'un réel x et d'un entier y, retourne x^y.

Question 2 : Réaliser l'addition de 2 polynômes.

Corrigé question 1 :

```
Fonction polynômeCalculerValeur (polynôme : Polynôme, x : réel) : réel
    début (*1*)
    p ← tête (polynôme)
    vp ← 0
    tant que non finliste (polynôme, p) faire (*2*)
        v ← val (polynôme, p)
        vp ← vp  + v.coef * puissanceEntièreCalculer (x, v.exp)
        p ← suc (polynôme, p)
    ftant (*2*)
    retourne vp
    fin (*1*)
```

Lexique

Monôme = <coef : entier, exp : entier>
polynôme : Liste (Monôme)
x : réel
vp : réel, valeur du polynôme pour *x*
p : Place, place courante dans le polynôme
v : Monôme, valeur associée à la place *p*
fonction puissanceEntièreCalculer (x : réel, exp : entier) : réel, retourne x^{exp}

Corrigé question 2 :

```
fonction polynômeAddition (polynôme1 : Liste (Monôme), polynôme2 : Liste (Monôme)): Liste (Monôme)
    début (*1*)
    polynôme3 ← lisvide( )
    place1 ← tête (polynôme1)
    place2 ← tête (polynôme2)
    tant que non finliste (polynôme1, place1) et non finliste (polynôme2, place2) faire (*2*)
        v1 ← val (polynôme1, place1)
        v2 ← val (polynôme2, place2)
        si v1.exp > v2.exp alors (*3a*)   adjqlis (polynôme3, v1)
                                          place1 ← suc (polynôme1, place1)
        sinon (*3s*)
            si v1.exp <  v2.exp alors (*4a*)   adjqlis (polynôme3, v2)
                                               place2 ← suc (polynôme2, place2)
            sinon (*4s*)  v3.coef ← v1.coef + v2.coef
                          si v3.coef ≠ 0 alors (*5a*)    v3.exp ← v1.exp
                                                         adjqlis (polynôme3, v3)
                          fsi (*5*)
                          place1 ← suc (polynôme1, place1)
                          place2 ← suc (polynôme2, place2)
            fsi (*4*)
        fsi (*3*)
    ftant (*2*)
    si finliste (polynôme1, place1)  alors (*4a*) polynômeCopierFin (polynôme2, place2, polynôme3)
                                     sinon (*4s*) polynômeCopierFin (polynôme1, place1, polynôme3)
    fsi (*4*)
    retourne polynôme3
    fin (*1*)
```

Lexique

```
    polynôme1 : Liste (Monôme)
    polynôme2 : Liste (Monôme)
    polynôme3 : Liste (Monôme), somme de polynôme1 et polynôme2
    place1 : Place, dans polynôme1
    place2 : Place, dans polynôme2
    v1 : Monôme, élément de polynôme1
    v2 : Monôme, élément de polynôme2
    v3 : Monôme, élément de polynôme3
    fonction polynômeCopierFin (polynômeA : Liste (Monôme), placeDeb : Place, polynômeB InOut : Liste (Monôme))
```

Algorithme de la fonction polynômeCopierFin

Il est identique, au type près, à celui de la fonction *lentierCopierFinListe* écrite pour l'interclassement de 2 listes triées paragraphe 1.4 exercice 5.

Exercice 4 : Simulation du jeu de la bataille

Règle du jeu :
La bataille est un jeu de cartes qui se joue à deux joueurs. Chaque joueur dispose initialement d'un paquet de cartes qu'il place à l'envers devant lui. Pour jouer une carte, un joueur doit obligatoirement prendre celle-ci au-dessus de son paquet.

Tant qu'il lui reste des cartes, chaque joueur retourne une carte devant lui :

- Le joueur ayant joué la carte de plus grande valeur ramasse les deux cartes retournées et les place en dessous de son propre paquet en commençant par celle qu'il a jouée.

- Si les deux cartes sont de même valeur, chaque joueur prend une carte de son paquet et la place à l'envers sur la carte qu'il a précédemment jouée. Puis, il retourne une nouvelle carte. Le joueur ayant joué la carte de plus grande valeur gagne toutes les cartes jouées (y compris celles qui sont à l'envers). Il les place alors en-dessous de son paquet en commençant par dépiler le tas qu'il a joué puis celui de son adversaire. Dans le cas où les cartes retournées sont toujours de valeur égale, alors on continue de la même façon jusqu'à ce que l'un des joueurs retourne une carte supérieure à celle de son adversaire.

Le jeu se termine lorsque le paquet de l'un des joueurs devient vide; celui-ci a alors perdu la partie.

Questions:
On désire simuler sur une machine le déroulement d'une partie de ce jeu.
1) Définir les structures de données logiques nécessaires.
2) Ecrire l'algorithme logique de la fonction qui, étant donnés les deux paquets initiaux, détermine le joueur gagnant.

Corrigé :
1)Le paquet de cartes de chaque joueur sera représenté par une file. Les cartes jouées par chaque joueur seront représentées par une pile.

2) données: le paquet de cartes de chacun des 2 joueurs : paquetJoueur1 et paquetJoueur2
résultat : un booléen indiquant le joueur gagnant

```
algorithme de la fonction jeuDeLaBataille
fonction jeuDeLaBataille (paquetJoueur1 : InOut PaquetJoueur, paquetJoueur2 : InOut  PaquetJoueur) : booléen
        début (*1*)
        jeuJoueur1 ← pileVide( )
        jeuJoueur2 ← pileVide( )
        tant que non estVideFile (paquet_joueur1) et non estVideFile (paquet_joueur2) faire (*2*)
            jouerCarte (jeuJoueur1, paquetJoueur1)
            jouerCarte (jeuJoueur2, paquetJoueur2)
            si sommet (jeuJoueur1) = sommet (jeuJoueur2) alors (*3a*)
                    si non estVideFile (paquetJoueur1) et non estVideFile (paquetJoueur2) alors (*4a*)
                    (* les joueurs jouent la carte à l'envers*)
                            jouerCarte (jeuJoueur1, paquetJoueur1)
                            jouerCarte (jeuJoueur2, paquetJoueur2)
                    fsi (*4*)
            sinon (*3s*)
                    si sommet(jeuJoueur1) > sommet(jeuJoueur2) alors (*5a*)
                            ramasserCarte(jeuJoueur1, paquetJoueur1)
                            ramasserCarte(jeuJoueur2, paquetJoueur1)
                    sinon (*5s*)
                            ramasserCarte(jeuJoueur2, paquetJoueur2)
                            ramasserCarte(jeuJoueur1, paquetJoueur2)
                    fsi (*5*)
            fsi (*3*)
        ftant (*2*)
        gagnéJoueur1 ← non estVideFile (paquetJoueur1)
        retourne gagnéJoueur1
        fin (*1*)
lexique
        Carte = entier
        PaquetJoueur = File (Carte)
```

```
JeuJoueur = Pile (Carte)
paquetJoueur1 : PaquetJoueur
paquetJoueur2 : PaquetJoueur
gagnéJoueur1 : booléen, à vrai si le joueur 1 a gagné
jeuJoueur1 : JeuJoueur, cartes jouées par le joueur 1
jeuJoueur2 : JeuJoueur, cartes jouées par le joueur 2
fonction jouerCarte (jeuJoueur InOut : JeuJoueur, paquetJoueur InOut : PaquetJoueur) le joueur prend la tête de son
    paquet et l'empile sur son jeu
fonction ramasserCarte (jeuJoueur_a InOut : JeuJoueur, paquetJoueur_b InOut : PaquetJoueur) le joueur a  dépile le
    jeu de b et enfile chaque carte dans son paquet

algorithme de la fonction jouerCarte
fonction jouerCarte (jeuJoueur InOut : JeuJoueur, paquetJoueur InOut : PaquetJoueur)
    début (*1*)
    empiler (jeuJoueur, premier (paquetJoueur))
    défiler (paquetJoueur)  /* ou supfil (paquetJoueur) */
    fin (*1*)
lexique :
    paquetJoueur : PaquetJoueur
    jeuJoueur : JeuJoueur

algorithme de la fonction ramasserCarte
fonction ramasserCarte (jeuJoueurA InOut : JeuJoueur, paquetJoueurB InOut : PaquetJoueur)
    début (*1*)
    tant que non estVidePile (jeuJoueurA) faire (*2*)
        enfiler (paquetJoueurB, sommet (jeuJoueurA))  /* adjfil (paquetJoueurB, sommet (jeuJoueurA)) */
        dépiler (jeuJoueurA)
    ftant (*2*)
    fin (*1*)
lexique :
    paquetJoueurB : PaquetJoueur
    jeuJoueurA : JeuJoueur
```

Exercice 5 : recherche d'ouvrages par mots clés

On souhaite faire une recherche par mots clés sur les références d'une liste d'ouvrages. Pour cela, on dispose d'une liste (*lOuvrage*) contenant, pour chaque ouvrage : une référence (*entier*), un titre (*chaîne*) et une liste de mots clés (un mot clé est une *chaîne*).

Question 1: Ecrire l'algorithme logique de la fonction qui, à partir d'un mot clé, affiche les références et titres des ouvrages correspondants au mot clé donné.

Question 2: Proposez une représentation physique pour la liste *louvrage* sachant que chaque ouvrage dispose d'une liste de 5 mots clés.

```
Correction Exercice 5
Question 1 : Algorithme de la fonction louvrageRechMotCle
    fonction lOuvrageRechmotCle (lOuvrage : Liste(Ouvrage), motCléCherché : chaîne)
    début(*1*)
        pLOuv ← tête(lOuvrage )
        tantque(*2*) non finliste (lOuvrage, pLOuv) faire
                ouvrage ← val (lOuvrage , pLOuv)
                lMotClé ← ouvrage.motsClés
                pLMot ← tête(lMotClé)
                arrêt ← faux
                tantque(*3*) non finliste (lMotClé, pLMot) et non arrêt faire
                        motCléLliste ← val (lMotClé, pLMot)
                        si (motCléListe = motCléCherché)
                                alors (*4a*) arrêt ← vrai
                                sinon (*4s*) plmot ← suc (lMotClé, pLMot)
                        fsi(*4*)
                fintantque(*3*)
```

```
                si (arrêt = vrai)
                        alors (*5a*) écrire (ouvrage.ref, ouvrage.titre)
                fsi(*5*)
                pLOuv ← suc (lOuvrage , pLOuv)
            fintantque(*2*)
        fin(*1*)
Lexique
        Ouvrage = < réf : entier, titre : chaîne, motsClés : liste (chaîne) >
        lOuvrage : Liste(Ouvrage)
        motCléCherché : chaîne, mot clé recherché
        pLOuv : Place, ième place dans la liste lOuvrage
        ouvrage : Ouvrage, ième ouvrage dans la liste lOuvrage
        lMotClé: Liste (chaîne) liste des mots clés de l'ouvrage courant
        pLMot : Place, ième place dans la liste lMotClé
        arrêt : booléen, à vrai si on trouve le mot clé dans lMotClé
        motCléListe : chaîne, mot clé courant de l'ouvrage courant
```

Question 2 :

La liste des ouvrages sera représentée de manière contiguë dans un fichier séquentiel pour ne pas perdre les informations. Comme il y a toujours 5 mots clés par ouvrage, la liste des mots clés sera représentée de manière contiguë dans un tableau de 5 éléments (pas besoin de mémoriser le nombre d'éléments, le tableau suffit) :

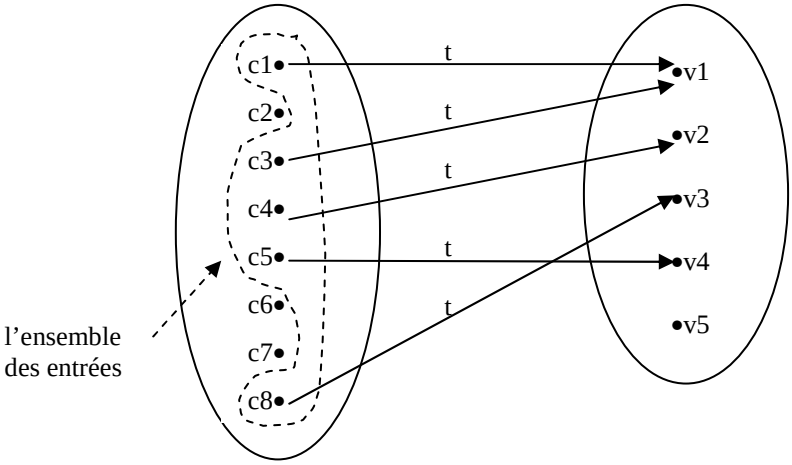
```
ListeOuvrage = fichier (Ouvrage)
Ouvrage = < réf : entier, titre : chaîne, motsClés : tableau chaîne [1..5] >
```

LES TABLES

La conservation de l'information sous des formes diverses, que ce soit en mémoire centrale ou en mémoire secondaire, et la recherche d'informations à partir de critères spécifiques est une activité très courante en informatique. Nous appellerons ***table*** la structure qui permet de conserver des éléments de nature quelconque et d'y accéder à partir d'une clé. La clé permet d'identifier un élément de manière unique. Par exemple, si une table conserve des informations sur des personnes, on pourra choisir comme clé le numéro INSEE de chaque individu.

Une table peut être considérée comme une fonction qui, à une clé, associe une valeur .

Exemple de table t :



L'ensemble *Clé* est appelé ensemble des ***clés*** de la table (ou ensemble des ***indicatifs***). Les clés pour lesquelles la fonction est définie sont les ***entrées*** de la table: on suppose leur nombre fini. Les éléments de *Valeur* sont appelés les ***valeurs*** de la table, ce sont les informations spécifiques de l'application pour une clé donnée.

La façon de représenter une table aura une grande incidence sur la complexité des algorithmes de manipulation de table. Ces algorithmes s'appuient essentiellement sur des recherches basées sur des comparaisons entre clés. Nous commencerons tout d'abord par décrire formellement la notion de table.

1 Les tables et leurs opérations

1.1 Définition abstraite

Soit *Valeur* un ensemble de valeurs et *Clé* un ensemble de clés. On appelle type **Table de Clé dans Valeur** et on note

T = **table [Clé → Valeur]**

l'ensemble des tables de *Clé* dans *Valeur*.

Ensembles définis et utilisés : *T*, Valeur U { ω }, Clé (où ω est la valeur indéfinie)

Description fonctionnelle des opérations :

- tabvide :		→	T	
- dom :	T	→	ENS (Clé)	(voir remarque ci-après)
- accèstab :	T x Clé	→	Valeur U { ω }	
- adjtab :	T x (Clé-dom (T)) x Valeur	→		
- suptab :	T x dom (T)	→		
- chgtab :	T x dom (T) x Valeur	→		

Les opérations *adjtab*, *suptab* et *chgtab* modifient la table donnée en paramètre.

Remarque sur le type ensemble :

ENS(Clé) est l'ensemble des entrées de la table.

Itération sur les éléments d'un ensemble:

Soit *ensemble* une variable de type ENS(V) où V est un type quelconque. On écrit comme suit l'itération sur les éléments de *ensemble*:

```
pour chaque élément dans ensemble [*jusqu'à condition*]  faire
        traitement de élément
fpour
```

Appartenance :

Une fonction utile sur les ensembles est *estDans (ensemble, el)* qui rend vrai si et seulement si *el* est un élément de *ensemble*.

1.2 Description informelle des opérations

Nous expliquons ici le rôle de chaque opération.

tabvide :

création d'une table vide (l'ensemble de ses entrées est vide)

dom :

fournit l'ensemble des entrées de la table (appelé aussi le domaine de la table)

accèstab :

accès à la valeur associée à une clé ; si elle n'existe pas, la fonction rend **ω**.

adjtab :

adjonction d'un couple (c, v) dans une table (c ne doit pas déjà être une entrée de la table ; il le devient après l'adjonction)

suptab :

suppression d'une entrée dans la table

chgtab :

changement de la valeur associée à une entrée

1.3 Exemples

Exemple 1 : origine des habitants de Nancy

En consultant les numéros de Sécurité Sociale des habitants de Nancy, on veut compter combien sont nés dans l'Ain, combien dans l'Aisne, ... combien dans le Val d'Oise (code 95) ; rappelons que ce sont les 6ème et 7ème chiffres du numéro de Sécurité Sociale qui l'indiquent. On veut écrire l'algorithme qui fournit ce résultat. On choisit de donner aux numéros de Sécurité Sociale un type "composite" pour les besoins du problème, par exemple :

NuméroSécu = < avant : entier, dép : entier, après : réel >

Ainsi pour le numéro suivant 2 45 10 38 153 227, le champ *avant* vaut 24510, le champ *dép* 38 et le champ *après* 153227.

Données :
le nombre de numéro de sécurité sociale à traiter
les numéros de sécurité sociale sous forme de triplet
Résultats :
création de la table qui associe au numéro de département le nombre de Nancéiens qui en sont originaires

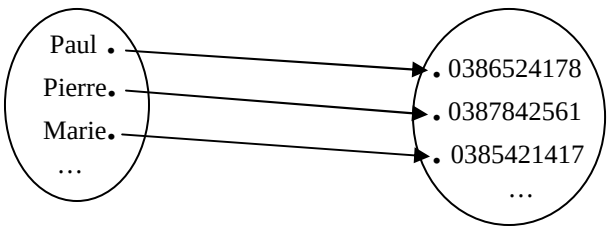
Algorithme logique
```
Fonction créerTableOrigineNancéiens ( ) : TableOrigine
    début ( *1* )
    ( *création d'une table dont toutes les valeurs sont 0* )
    tabDépOrigine ← tabvide( )
    pour ( *2* ) clé de 1 à 95 faire
            adjtab (tabDépOrigine, clé, 0 )
    fpour ( *2* )
    ( *modification de cette table* )
    nb ← lire ( )
    pour k de 1 à nb faire  ( *3* )
            numéroSécu ← lire ( )
            ancienneValeur ← accèstab (tabDépOrigine, numéroSécu.dép)
            chgtab (tabDépOrigine, numéroSécu.dép, ancienneValeur +1 )
    fpour ( *3* )
    retourne tabDépOrigine
    fin ( *1* )
```
Lexique
- TableOrigine = table [[1..95] → entier]
- tabDépOrigine : TableOrigine, table des origines qui associe au numéro de département le nombre de Nancéiens qui en sont originaires parmi les k premiers.
- numéroSécu : NuméroSécu, kème numéro de Sécurité Sociale, donné sous forme d'un triplé.
- clé : entier, clé courante
- nb : entier, le nombre de numéro de sécurité sociale à traiter
- k : entier, indice d'itération
- ancienneValeur : entier, nombre de Nancéiens originaires du département de la kième donnée avant prise en compte de celle-ci.

Exemple 2 : annuaire

Dans une entreprise, on dispose, pour chaque employé, de son numéro de téléphone. On souhaite avoir une fonction qui, étant donné un nom d'employé, donne son numéro de téléphone. On suppose qu'il n'y a pas 2 employés de même nom.

On stocke les informations dans une table associant un numéro de téléphone à un nom d'employé .

Exemple:



On utilise la fonction *accèstab* pour obtenir le numéro de téléphone de l'employé dont on a le nom.

Exemple 3 : liaisons aériennes

On souhaite écrire un algorithme fournissant, pour une ville de départ donnée et une ville d'arrivée donnée, une liaison aérienne les reliant avec au plus une escale et d'un prix minimum.

On supposera qu'il existe toujours une liaison (directe ou avec une escale) d'une ville à une autre. On fait abstraction des cohérences horaires. On connaît l'ensemble des villes. On dispose d'une table *tarif* qui, à un couple de villes, associe le prix de la liaison directe les reliant si elle existe et sinon associe une valeur particulière (par exemple –1, cette valeur correspond au ω dans les définitions abstraites des opérations).

Exemple de table *tarif* :

départ \ arrivée	PARIS	NANCY	LYON	AGEN		
PARIS	0	850	700	1600	. . . .	-1
NANCY	845	0	1500	-1		
LYON	700	1500	0	-1		
AGEN	1600	-1	-1	0		
. . . .	. . . .	. . . .			. . . .	. . . .
	-1					0

Algorithme logique (On suppose que 100 000 est un majorant des prix des liaisons)

```
Fonction imprimerLiaisonAerienne (villeArrivée : chaîne, villeDépart : chaîne, ensVille : ENS(chaîne), tarif : table [CoupleVille → réel])
    début(*1*)
    prixMin ← 100 000
    pour chaque(*2*) ville dans ensVille faire /*i*/
        prixVilleDépart ← accèstab (tarif, (villeDépart,ville))
        prixVilleArrivée ← accèstab (tarif, (ville, villeArrivée))
        si prixVilleDépart ≠ -1 et prixVilleArrivée ≠-1 et prixVilleDépart + prixVilleArrivée < prixMin alors(*3a*)
            villeInter ← ville
            prixMin ← prixVilleDépart + prixVilleArrivée
        fsi(*3*)
    fpour(*2*)
    si villeInter = villeDépart ou villeInter = villeArrivée
        alors(*4a*)
            écrire (« liaison directe, prix: », prixMin)
        sinon(4s*)
            écrire (« escale: », villeInter, « prix: », prixMin)
    fsi(*4*)
    fin(*1*)
```

Lexique

- CoupleVille = <vDep : chaîne, vArr : chaîne>
- villeArrivée : chaîne, ville d'arrivée
- villeDépart : chaîne, ville de départ
- villeInter : chaîne, ville escale retenue parmi les i premières
- ville : chaîne, ième ville
- ensVille : ENS(chaîne), ensemble des villes
- tarif : table [CoupleVille → réel], table des tarifs
- prixMin : réel, prix de la liaison minimum parmi les i premières
- prixVilleDépart : réel, prix de la liaison de *villeDépart* à la ième ville
- prixVilleArrivée : réel, prix de la liaison de la ième ville à *villeArrivée*

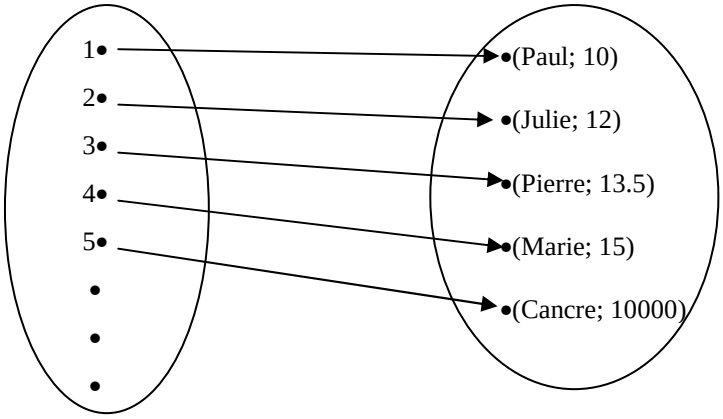
Exemple 4 : arrivée d'une course contre la montre

On veut écrire un algorithme qui, lors d'une compétition, affiche le nouveau classement après l'arrivée de chaque coureur. Le nombre de coureurs est donné et chaque coureur est donné par son nom et son temps.
Le résultat est l'écriture d'une suite de classements. Chaque classement peut être représenté par une table de coureurs triée dans l'ordre des temps croissants. Les clés de la table sont les rangs de classement des coureurs, et les valeurs sont formées des noms et temps des coureurs. Le passage d'un classement au suivant correspond à l'arrivée d'un coureur qui doit être intercalé à la place correspondant à son temps. Cette place, qu'on désignera par *rang* dans l'algorithme, est déterminée par une recherche dans la table des temps des coureurs déjà arrivés. Pour intercaler un coureur dans la table, il faut l'insérer dans la place laissée libre par le décalage d'un rang de tous les coureurs ayant un moins bon temps que lui.

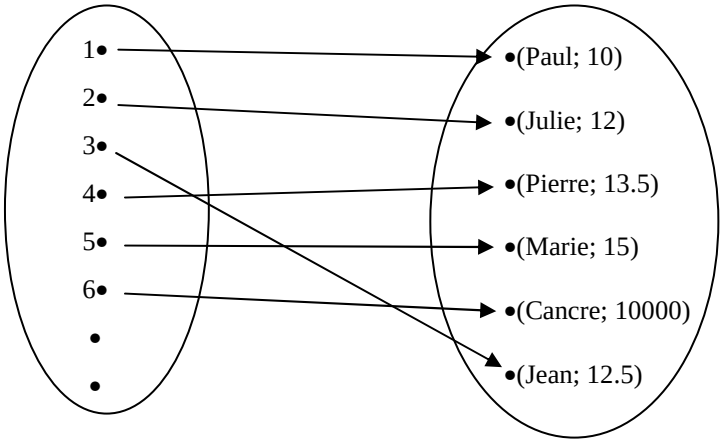
Pour éviter de faire des cas particuliers du premier coureur ou d'un arrivant qui aurait le moins bon temps, une solution classique consiste à introduire artificiellement un "cancre" qui initialise le classement et sera toujours le dernier grâce au temps prohibitif qu'on lui attribue. Bien entendu, on ne le fera pas figurer dans l'écriture du classement.

Exemple

A un instant donné, on a la table suivante :



Le coureur Jean arrive avec un temps de 12.5 ; la table est modifiée comme suit :



Algorithme logique

```
début ( *1* )
nbCour ← lire( )
tabClassement ← tabvide( )
adjtab  (tabClassement, 1, (« CANCRE », 10000) )
pour ( *2* ) i de 1 à  nbCour faire
    coureur ← lire( )
    ( *recherche du rang* )
    k ← 1
    tantque ( *3* ) coureur.temps ≥ accèstab (tabClassement, k) .temps faire
            k ← k+1
    ftant ( *3* )
    rang ← k
     ( *décalage* )
    adjtab  (tabClassement, i+1, accestab (tabClassement, i))    ( *du dernier* )
    pour ( *4* ) j décroissant de i-1 à rang faire ( *des autres* )
            chgtab  (tabClassement, j+1,accèstab (tabClassement, j))
    fpour ( *4* )
    chgtab  (tabClassement, rang, coureur) ( *insertion* )
    écrire (tabClassement) (* attention : lors de la programmation, ne pas écrire le dernier *)
fpour ( *2* )
suptab  (tabClassement, nbcour + 1)
fin ( *1* )
```

Lexique
- Coureur = < nom : chaîne, temps : réel >
- TabClassement = table [[1.. nbCour+1] → Coureur >]
- tabClassement : TabClassement, classement consécutif à l'arrivée du iéme coureur
- nbCour : entier, nombre de coureurs
- coureur : Coureur, ième arrivant
- rang : entier, rang où il faut placer le ième arrivant
- k : entier, place dans *tabClassement* des coureurs meilleurs que le ième arrivant
- i : entier, indice d'itération sur les arrivants
- j : entier, suite des places des coureurs moins bons que le ième arrivant.

1.4 Exercices

Exercice 1

20 joueurs participent à une loterie : 20 petits papiers portant les noms des joueurs sont placés dans un chapeau et une main innocente tire un à un ces 20 papiers. Le joueur dont le papier est tiré le premier gagne 1000€, le deuxième 250€, ... le kième gagne $1000/k^2$,…. On désire construire la table *tabGain* qui, à chaque joueur, associe son gain. La donnée est la suite des 20 noms sortis dans l'ordre du tirage. Ecrire l'algorithme logique de la fonction de création de la table *tabGain*. On suppose qu'il n'y a pas 2 joueurs de même nom.
Le principe est le suivant : si le premier nom tiré est n, on doit en conclure que le gain de n vaut 1000; si le deuxième est m, que gain de m vaut 250 ; etc.

Correction Exercice 1

```
Fonction tabGainCréer( ) : TabGain
    début ( *1* )
    tabGain ←  tabvide( )
    pour ( *2* ) étape de 1 à 20 faire
            gain ← 1000/étape^2
            tirage ← lire( )
            adjtab  (tabGain, tirage, gain)
    fpour ( *2* )
    retourne tabGain
    fin ( *1* )
```

Lexique
- TabGain = table [chaîne → réel]
- tabGain : TabGain, table associant aux joueurs déjà tirés leur gain. Le gain du joueur de nom i a pour clé i.
- tirage : chaîne, ième nom sorti
- gain : réel, gain du joueur de nom *tirage*
- étape : entier, numéro d'étape du tirage (c'est i)

Exercice 2

Une assemblée vote en choisissant un candidat. On connaît le nombre de candidats (*nc*) et le nombre de votants (*nv*). On lit les votes de chaque membre de l'assemblée. On suppose que les candidats sont représentés par des numéros de 1 à *nc*. On demande de créer la table associant à chaque candidat le nombre de voix qu'il a obtenu. Ecrire l'algorithme logique de la fonction de création qui a comme paramètres *nc* et *nv*.

Correction Exercice 2

```
Fonction tabVoteCréer (nc : entier, nv : entier ) : TabVote
    début ( *1* )
        tabVote ←  tabvide( )
        pour ( *2* ) i de 1 à nc faire
                adjtab (tabVote, i, 0)
        fpour ( *2* )
        pour ( *3* ) j de 1 à nv faire
                vote ←  lire( )
                chgtab (tabVote, vote, accèstab  (tabVote, vote) + 1)
        fpour
        retourne tabVote
    fin ( *1* )
```

Lexique
- TabVote = table [[1..nc] → entier]
- tabVote : TabVote, associe à *i* le nombre de personnes ayant voté pour *i*.
- nc : entier, nombre de candidats
- vote : entier, numéro du candidat du jème vote
- i : entier, indice d'itération sur les candidats
- nv : entier, nombre de votants
- j : entier, indice d'itération sur les votes

Exercice 3 :

Reprenons l'exemple 2. Dans une entreprise, on dispose d'une table qui associe à chaque employé son numéro de téléphone. On souhaite construire l'annuaire inversé, il associe à chaque numéro de téléphone le nom de l'employé correspondant. Ecrire l'algorithme logique de la fonction de création de cet annuaire inversé.

Correction Exercice 3

```
Fonction annuaireInverséCréer( tabNomTel : TabNomTel ) : TabTelNom
        début ( *1* )
        tabTelNom ← tabvide( )
        pour chaque nom dans dom (tabNomTel) faire ( *2* )
                tel ← accèstab (tabNomTel, nom)
                adjtab (tabTelNom, tel, nom)
        fpour ( *2* )
        retourne tabTelNom
        fin ( *1* )
```

Lexique
- TabNomTel= table [chaîne → entier]
- TabTelNom = table [entier → chaîne]
- tabNomTel : TabNomTel, table associant à un nom d'employé son numéro de téléphone
- tabTelNom : TabTelNom, table associant à un numéro de téléphone le nom de l'employé correspondant
- tel : entier, numéro de téléphone de l'employé de nom *nom*
- nom : chaîne, nom de l'employé courant

Exercice 4

n (n ≤ 1000) candidats participent à un brevet. Ils doivent répondre par un mot à chacune des vingt questions posées (numérotées de 1 à 20). Chaque bonne réponse donne un nombre de points égal au numéro de la question. On dispose d'une table *tabRéponse* associant à chaque question le mot correspondant à la réponse exacte. Pour réussir le brevet, un candidat doit obtenir au moins *nbMin* points.
On demande de créer une table associant, à chaque rang dans le classement (de 1 à n), les renseignements sur le candidat ayant ce rang : son nom, le nombre de points qu'il a obtenus et un libellé indiquant s'il est admis ou non. Cette table est donc triée par ordre décroissant des nombres de points et permet d'accéder directement au candidat se trouvant à un rang donné.
Ecrire l'algorithme logique de la fonction qui crée cette table. Cette fonction a pour paramètres *n*, *tabRéponse* et *nbMin*. Elle lit pour chaque candidat, son nom suivi de ses vingt réponses données dans l'ordre des questions.

Correction Exercice 4

```
Fonction tabCandidatCréer (n : entier, tabRéponse : table[ [ 1..20 ] →  chaîne ], nbMin : entier ) : TabCand
        début  ( *1* )
        tabCand ←   tabvide( )
        adjtab (tabCand, 1, ("",-1,""))
        pour ( *2* ) i de 1 à n faire  ( *traitement du ième candidat* )
                nomCand ← lire( )
                ( *calcul de son total* )
                totalCand ← 0
                pour ( *3* ) j de 1 à 20 faire
                        réponseCand ← lire( )
                        si réponseCand = accèstab (tabRéponse, j)
                                alors  ( *4a* ) totalCand ← totalCand+j
                        fsi ( *4* )
                fpour ( *3* )
                si totalCand ≥ nbMin
                        alors ( *5a* ) libelléCand ← « admis »
                        sinon ( *5s* ) libelléCand ← « refusé »
```

```
                fsi ( *5* )
                ( *recherche du rang où insérer le candidat* )
                k ← 1
                tantque ( *6* ) totalCand ≤ accèstab (tabCand,k).tot faire
                        k ← k+1
                ftantque ( *6* )
                rang ← k
                adjtab (tabCand, i+1, accèstab (tabCand, i))  ( *décalage du dernier* )
                pour ( *9* ) l décroissant de i-1 à rang faire  (* des autres*)
                        chgtab (tabCand, l+1, accèstab (tabCand, l))
                fpour ( *9* )
                ( *insertion* )
                chgtab (tabCand, rang,  (nomCand, totalCand, libelléCand))
        fpour ( *2* )
        suptab (tabCand, n+1)
        retourne tabCand
        fin ( *1* )
```

Lexique

- n : entier, nombre de candidats
- tabRéponse : table[[1..20] → chaîne], table associant à chaque question le mot correspondant à la réponse exacte
- nbMin : entier, nombre minimum de points pour réussir le brevet
- Candidat = <nom : chaîne, tot : entier, lib : chaîne >
- TabCand = table [[1..n+1] → Candidat]
- tabCand : TabCand, table des candidats
- nomCand : chaîne, nom du ième candidat
- réponseCand : chaîne, réponse du candidat i à la jème question
- totalCand : entier, nombre de points du candidat i
- libelléCand : chaîne, libellé du candidat i
- k : entier, clé courante dans *tabCand*
- rang : entier, rang où il faut placer le candidat i
- i : entier, indice d'itération sur les candidats
- j : entier, indice d'itération sur les réponses du ième candidat
- l : entier, indice d'itération pour le décalage des candidats qui se trouveront après le ième candidat

2 Etude du cas particulier des tables « simples »

Lorsque l'ensemble des clés est un intervalle d'entiers ou une "composition" d'intervalles d'entiers, nous qualifions les ***tables*** de ***simples***, car elle se représentent facilement par un tableau ou un fichier relatif. Il y a ***correspondance immédiate entre clés et indices dans un tableau ou entre clé et rang dans un fichier.***

Soit une table simple *t* de type

table [[CLEMIN..CLEMAX] → TypeEl]

où *TypeEl* est le type des valeurs de la table (*entier* ou *réel* ou type composite ou …), *CLEMIN* la plus petite valeur possible pour les clés et *CLEMAX* la plus grande possible. Choisissons une représentation en mémoire centrale à l'aide d'un tableau. La table *t* est alors représentée par le type tableau suivant :

TabTypeEl = tableau TypeEl [CLEMIN..CLEMAX]

Voici un exemple de table t pour laquelle les valeurs sont des réels positifs, *CLEMIN* vaut –2 et *CLEMAX* 4 . Elle peut être simplement représentée par un tableau :

Niveau logique

-2	→	2.3
0	→	89.56
1	→	6.32
3	→	65
4	→	45.2

Niveau physique

-2	2.3
-1	-1
0	89.56
1	6.32
2	-1
3	65
4	45.2

Voici, en tenant compte de ces choix, les algorithmes de programmation des fonctions logiques :

```
fonction tabvide ( ) : TabTypeEl
  début
  pour i de CLEMIN à CLEMAX faire
        t[i] ← vp          où vp est la représentation de ω, c'est une valeur particulière différente de toutes les
                           valeurs possibles (dans l'exemple précédent, c'est –1)
  fpour
  retourne t
  fin

fonction accèstab (t : TabTypeEl, c : entier) : TypeEl
  début
  retourne t[c]
  fin

fonction adjtab (t InOut : TabTypeEl, c : entier, v : TypeEl)
  début
  t [c] ← v
  fin

fonction suptab (t InOut : TabTypeEl, c : entier)
  début
  t [c] ← vp
  fin

fonction chgtab (t InOut : TabTypeEl, c : entier, v : TypeEl)
  début
  t [c] ← v
  fin

fonction dom (t : TabTypeEl) : ENS (entier)      (* l'ensemble peut, par exemple, être représenté par un tableau *)
  début
  ensClé ← ensVide( )
  (* ensVide( ) crée un ensemble vide *)
  pour i de CLEMIN à CLEMAX faire
        si t[i] ≠ vp alors
              adjens (ensClé, i)
              (* adjens ajoute l'élément i à l'ensemble ensClé *)
        fsi
  fpour
  retourne ensClé
  fin
```

Exemples

Reprenons l'exemple de l'arrivée d'une course. L'ensemble des clés est un intervalle d'entiers, on peut donc très facilement représenter la table des classements par un tableau de type tableau Coureur [1.. nbCour+1].

Il en est de même pour la table *tabDépOrigine* qui associe au numéro de département le nombre de Nancéiens qui en sont originaires : tableau entier [1.. 95].

Exercice

Dans le but de faciliter la gestion de sa commune, le maire de *Simpleville* souhaite regrouper dans une table les nom, prénom et adresse de chaque habitant. Il attribue donc à chaque habitant un numéro arbitraire (entier, compris entre 1 et le nombre d'habitants) et constitue une table, *tabNomAdresse*, qui à chacun de ces numéros associe le nom, le prénom et l'adresse de l'habitant concerné.

a) Proposer une représentation logique permettant de minimiser les redondances.

b) Evaluer le gain de place réalisé grâce à l'accès indirect aux noms des rues en le comparant à la solution qui consiste à répéter les noms des rues pour chaque habitant.

On suppose que:
- la commune contient 20 000 habitants et 50 rues,
- les noms des rues ne dépassent pas 40 caractères,
- les entiers sont codés sur 4 caractères.

c) Proposer une représentation physique.

d) Ecrire l'algorithme logique de la fonction de création de la table des rues à l'aide des données suivantes :
- le nombre de rues,
- les noms des rues par ordre alphabétique.

e) Même question pour la table des noms, prénoms et adresses à partir des données suivantes :
- le nombre d'habitants,
- les nom, prénom et adresse de chaque habitant par ordre alphabétique.

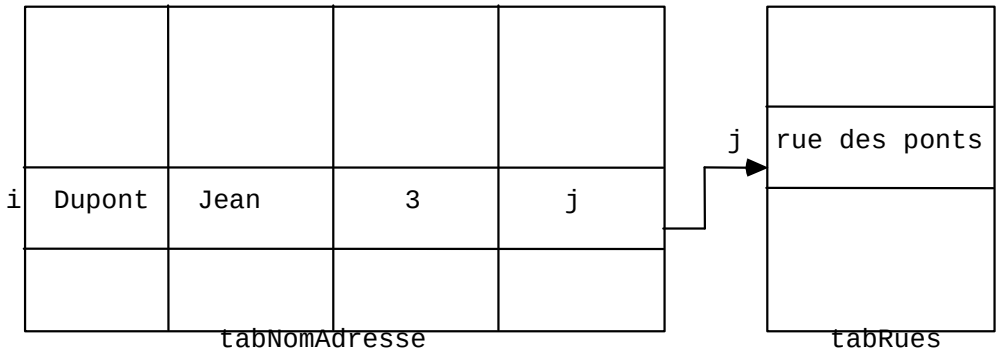
Correction Exercice

a) Comme les noms des rues se répètent un grand nombre de fois pour les habitants de la commune, on choisit d'attribuer à chaque rue un numéro arbitraire (entier, compris entre 1 et le nombre de rues) et de créer une table, *tabRues*, des rues qui associe à chacun de ces numéros le nom de la rue concernée.

Ainsi, dans la table *tabNomAdresse*, l'adresse est remplacée par un couple formé du numéro dans la rue et du numéro de la rue. On obtient donc le nom de la rue à partir de la table *tabNomAdresse* par un *accès* appelé *indirect*.

table des noms et des adresses : tabNomAdresse : table [entier → NomAdr]
table des rues : tabRues : table [entier → chaîne]
NomAdr = < nom : chaîne, prénom : chaîne, numRue : entier, codeRue : entier >

Exemple : Dupont Jean, 3 rue des Ponts



On peut choisir de trier la table des noms et adresses et la table des rues pour accélérer la recherche d'un élément (par dichotomie).

b) Calcul du coût en place mémoire pour les noms des rues
accès direct :
dans tabNomAdresse : 20 000 x 40 = 800 000 caractères

accès indirect :
dans tabNomAdresse : 20 000 x 4 = 80 000
dans tabRues: 50 x 40 = 2 000
total = 82 000 caractères

gain :
800 000 - 82 000 = 718 000 caractères
donc on a réduit la place de 90 % (718 000 / 800 000)

c) Représentation physique : un fichier direct pour la table tabNomAdresse et un tableau pour la table tabRues, (pour conserver les informations, on peut mémoriser le tableau tabRues dans un fichier).

d) Algorithme logique

```
Fonction créerTabRues ( ) : TabRues
    début  ( *1* )
        nbRues ← lire( )
        tabRues ←  tabvide( )
        pour ( *2* ) k de 1 à nbRues faire
                nomRueHab ← lire( )
                adjtab (tabRues, k, nomRueHab )
        fpour ( *2* )
        retourne tabRues
    fin (* 1 *)
```

Lexique

- TabRues = table [[1..nbRues] → chaîne]
- tabRues : TabRues, table des rues triée par ordre alphabétique des noms de rues
- nbRues : entier, nombre de rues
- nomRueHab : chaîne, kème nom de rue
- k : entier, clé courante de *tabRues*

e) Algorithme logique

```
Fonction créerTabNomAdresse (tabRues : TabRues, nbRues : entier ) : TabNomAdr
    début  ( *1* )
        nbHab ← lire( )
        tabNomAdresse ← tabvide( )
        pour ( *2* ) i de 1 à nbHab faire
                nomHab ← lire( )
                prénomHab ← lire( )
                numRueHab ← lire( )
                nomRueHab ← lire( )
                j ← tableRuesRechercherRue (tabRues, nomRueHab, nbRues)
                adjtab (tabNomAdresse, i, (nomHab, prénomHab, numRueHab, j))
        fpour ( *2* )
        retourne tabNomAdresse
    fin ( *1* )
```

Lexique

- TabRues = table [[1..nbRues] → chaîne]
- TabNomAdr = table [[1..nbHab]→ NomAdr]
- NomAdr = < nom : chaîne, prénom : chaîne, numRue : entier, codeRue : entier >
- tabNomAdresse : TabNomAdr , table des noms et des adresses triée par ordre alphabétique des noms des habitants
- tabRues : TabRues, table des rues triée par ordre alphabétique des noms de rues
- nbRues : entier, nombre de rues
- nomRueHab : chaîne, nom de rue du ième habitant
- nbHab : entier, nombre d'habitants
- i : entier, clé courante dans *tabNomAdresse*
- nomHab : chaîne, nom du ième habitant
- prénomHab : chaîne, prénom du ième habitant
- numRueHab : entier, numéro dans la rue du ième habitant
- j : entier, clé de *nomRueHab* dans *tabRues*

Fonction recherche

```
    fonction tableRuesRechercherRue (t : TableRues, ch : chaîne, n : entier) : entier
     ( *recherche ch dans t définie sur  [ 1..n ] et retourne la clé correspondante ou 0 si ch n'existe pas dans t *)
    début ( *1* )
        bInf← 1
        bSup← n
         ( * recherche dichotomique *)
        tantque ( *2* ) bInf <  bSup faire
                milieu ← (bInf + bSup)÷2
                si ch ≤ accèstab (t, milieu)
                        alors ( *3a* )  bSup ← milieu
                        sinon ( *3s* ) bInf ← milieu+1
                fsi ( *3* )
        ftantque ( *2* )
```

```
        si accèstab (t, bInf) = ch
                alors ( *4a* )  clé ← bInf
                sinon ( *4s* ) clé ← 0
        fsi ( *4* )
        retourne clé
    fin ( *1* )
```

Lexique de la fonction

- TableRues = table [[1.. n] → chaîne]
- t : TableRues
- n : entier, taille de la table
- clé : entier, clé de *ch* dans *t* ou 0
- ch : chaîne, chaîne dont on cherche la clé dans *t*
- bInf : entier, borne inférieure de l'intervalle de recherche courant
- bSup : entier, borne supérieure de l'intervalle de recherche courant
- milieu : entier, milieu de l'intervalle de recherche courant

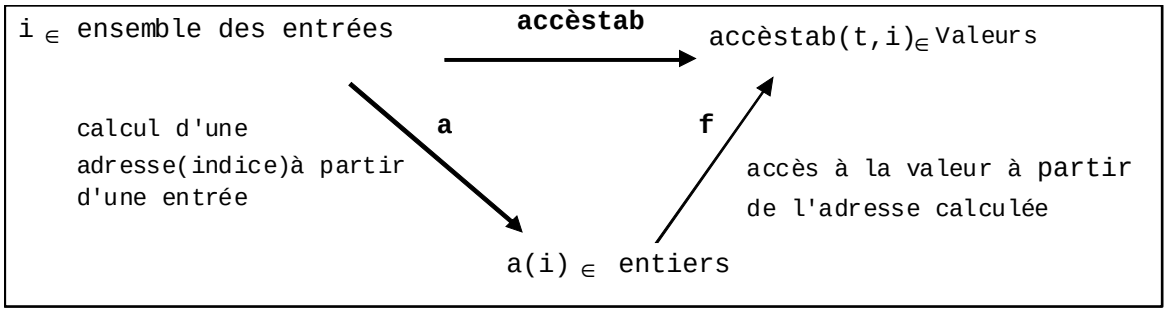
3 Représentation des tables " non simples "

Lorsque les clés ne sont pas des entiers ou une composition d'entiers, il est impossible d'accéder directement à la valeur associée. La correspondance entre clés et indices dans un tableau ou entre clé et rang dans un fichier devra faire l'objet d'un traitement dont nous parlons ci-dessous.

Soit une table t. On souhaite stocker en mémoire les valeurs associées aux entrées. On va représenter les valeurs dans un tableau ou un fichier relatif (appelons-le *Valeurs*).

Dans tous les exemples et/ou exercices de ce paragraphe, **le choix tableau peut être remplacé par fichier relatif et inversement** selon que l'on souhaite travailler en mémoire centrale ou secondaire.

La fonction d'accès se décompose de la façon suivante : **accèstab = f o a**
où **a** est une "fonction d'adressage", **f** une fonction d'accès à la valeur dans *Valeurs*.

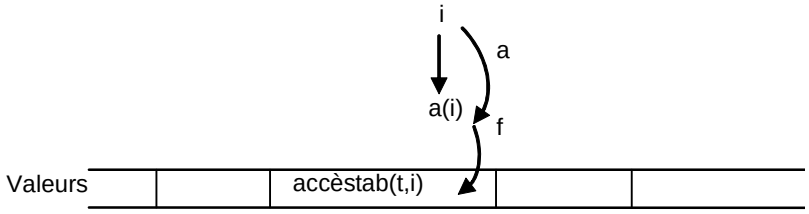


Remarque :

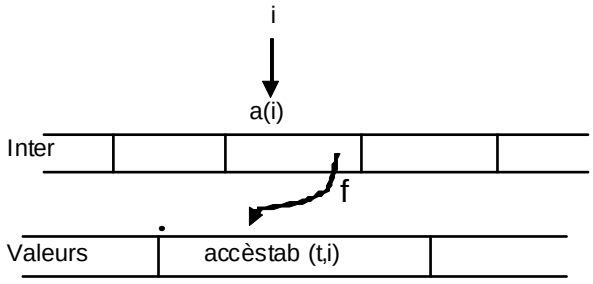
le cas des tables simples répond à ce schéma, avec a qui est la fonction identité.

f peut prendre deux formes :

- **f est un accès direct :** l'indice a(i) désigne directement la valeur dans *Valeurs*. La valeur accèstab (t,i) est Valeurs [a(i)] si *Valeurs* est un tableau.



- **f est un accès indirect :** a(i) est un indice dans un tableau ou un fichier relatif intermédiaire *Inter*. L'indice a(i) désigne dans *Inter* l'élément contenant l'indice de la valeur dans *Valeurs*. La valeur accèstab(t,i) est Valeurs [Inter [a(i)]] si *Valeurs* est un tableau.



Remarque

On peut imaginer d'autres formes d'accès indirect.

Pour choisir une représentation, on doit avant tout tenir compte de la nature des clés : ensemble d'entiers avec plus ou moins de trous, ensemble de noms plus ou moins disparates, ensemble de codes significatifs, ...Cette nature va amener à choisir une fonction d'adressage **a,** et donc une correspondance entre clés et emplacements, c'est-à-dire une façon de ranger les valeurs, un rangement de ces valeurs. **C'est ce rangement qui caractérise presque complètement la représentation**.

3.1 Adressage (et rangement) calculé

Définition

On dit que la représentation d'une table se fait par adressage calculé lorsque l'adresse *a(clé)* d'une clé est calculée à l'aide d'une fonction injective (clé1 ≠ clé2 => a (clé1) ≠ a (clé2)).

Remarque : les clés ne sont pas stockées.

Exemple

Une entreprise de distribution possède un catalogue qui permet de retrouver le prix de chacun des produits qu'elle vend. Un produit est identifié par un code de la forme $i_1\ i_2$ où i_1 est le numéro d'entrepôt où est stocké le produit (de 1 à 4) et i_2 le numéro du produit dans l'entrepôt (sur trois chiffres à partir de 000). Chaque entrepôt contient au plus 150 articles, les codes vont donc de 1000 à 4149 au plus.

Structure logique

Un tel catalogue correspond à une table associant à un code de produit un prix :

tabCatalogue : table [entier → réel]

Représentation

Première solution : on choisit de représenter le catalogue par un tableau dont les indices sont égaux aux codes des produits (cas d'une table simple) ; on réserve donc une zone de 3 150 places (4149 - 1000 + 1), par exemple un tableau défini sur l'intervalle [1000 .. 4149], pour stocker au plus 4x150 = 600 articles. La perte de place est considérable (au moins 2 550 places inutilisées).

Deuxième solution : pour ne réserver que les 600 places qui sont nécessaires, on peut choisir comme représentation un tableau défini sur l'intervalle [0..599]. La correspondance entre clés (code de produit) et indices est alors calculée par la fonction suivante :

$$a(x) = 150\ (i_1 - 1) + i_2 \qquad \text{pour} \quad x = i_1\ i_2$$

exemples de correspondance entre clés et indices à l'aide de cette fonction:

1000 → 0
1001 → 1
...
1149 →149
2000 →150
2001 →151
...
2149 →299
3000 →300
...
4149 →599

Dans l'adressage calculé, une place est réservée dans le tableau ou le fichier *Valeurs* pour la valeur associée à chaque clé. Or, certaines clés ne sont pas des entrées, aucune valeur ne leur est alors associée au niveau logique. Au niveau physique, il faudra donc introduire une valeur particulière (différente de toutes les valeurs possibles) qui indiquera que la clé correspondante n'est pas une entrée (il s'agit du ω des définitions abstraites).

Exercice

Une entreprise possède un annuaire des adresses et numéros de téléphone de ses clients. Un client est identifié par une référence composée de la première lettre de son nom et d'un numéro d'ordre (de 1 à 100).

1) Quelle est la structure logique de cet annuaire ?

2) Proposez une représentation physique en mémoire centrale pour cette structure.

3) Traduisez les opérations tabvide, adjtab, accèstab, suptab et chgtab sur les tables en fonction de cette représentation.

On suppose disposer d'une fonction *rang* qui renvoie le rang dans l'alphabet de la lettre donnée en paramètre.

Correction :
1) L'annuaire des clients correspond à une table tabClients de type :
TabClients = table [Référence → AdrNum]
avec Référence = <lettre : caractère, num : entier>
AdrNum = <adresse : chaîne, numTel : entier>.

L'ensemble des clés est le suivant :
{ (A, 1), (A, 2),...,(A, 100), (B, 1),..., (B, 100), ..., (Z, 1),..., (Z, 100) }

Exemple de table logique tabClients :
(A, 10) ----------> (Nancy, 0383832020)
(M, 2) ----------> (Vandœuvre, 0383549671)
(P, 3) ----------> (Villers, 0383405611)
(P, 10) ----------> (Nancy, 0383601045)

2) On peut représenter la table tabClients par le tableau tabClients. La correspondance entre clés et indices est obtenue par une fonction d'adressage a.

On réserve une place pour chaque clé possible. Il faut trouver a tel que :
(A, 1) ----------> 1
(A, 2) ----------> 2
...
(A, 100) ---------> 100
(B, 1) ----------> 101
...
(Z, 100) ---------> 2600

Une solution possible pour a est la suivante :

a(x) = (rang (x.lettre) - 1) ∗100 + x.num, où x : Référence.

La fonction rang fournit le rang d'une lettre dans l'alphabet.

1	(« », 0)
	...
10	(Nancy, 0383832020)
	...
1202	(Vandœuvre, 0383549671)
	...
1503	(Villers, 0383405611)
	...
1510	(Nancy, 0383601045)
	...
2600	(« », 0)

tabClients : TabClients = tableau AdrNum [1..2600]

Si le nombre de clients effectifs (soit le nombre d'entrées) est beaucoup moins important que le nombre de clients possibles (soit le nombre d'indicatifs, ici 26∗100 = 2600), il y a une grande perte de place.

3)
```
fonction tabvide ( ) : TabClients
  début
  pour i de 1 à 2600 faire
                t [i] ← (« »,0)
  fpour
  retourne t
  fin

fonction adjtab (tabClients InOut : TabClients, x : Référence, v : AdrNum)
  début
  tabClients [a(x)] ← v
  fin

fonction accèstab (tabClients : TabClients, x : Référence) : AdrNum
  début
  retourne tabClients [a(x)]
  fin

fonction suptab (tabClients InOut : TabClients, x : Référence)
  début
  tabClients [a(x)] ← («», 0)
  fin

fonction chgtab (tabClients InOut : TabClients, x : Référence, v : AdrNum)
  début
  tabClients [a(x)] ← v
  fin
```

Avantages et inconvénients

Avantages

L'accès et le changement d'une valeur associée à une entrée sont rapides. L'adjonction d'une clé à l'ensemble des entrées ne pose pas de problème puisqu'à la construction de la table, la place a été réservée pour cette nouvelle

entrée. Pour supprimer une entrée i, il suffit de remplacer sa valeur par une valeur particulière (différente de toutes les valeurs possibles).

Inconvénients

On perd de la place lorsque le nombre d'entrées est plus petit que le nombre de clés. Cette solution est impossible quand l'ensemble des clés est infini (par exemple si les clés sont des chaînes de longueur non limitée ou des réels). Enfin, on ne trouve pas toujours une fonction injective qui permet d'associer à toute clé un entier compris entre 1 et le nombre maximum de clés.

3.2 Adressage (et rangement) associatif

(toujours possible mais pas toujours très efficace)

Définition

Dans un rangement associatif, on ne représente pas toutes les clés possibles. On constitue une liste de toutes les entrées accompagnées de leur valeur ou de l'adresse à laquelle se trouve cette valeur. On l'appelle **liste des entrées**. Si i est une entrée, a(i) est définie comme étant la place p où se trouve i dans la liste des entrées.

La table est représentée par une liste : la liste des entrées qui peut être ordonnée ou non.

On parle d'adressage associatif car, à chaque accès, on doit rechercher le couple qui a cet indicatif dans la liste des entrées. Cette liste des entrées peut être représentée de manière chaînée ou contiguë, triée dans un certain ordre ou non triée.

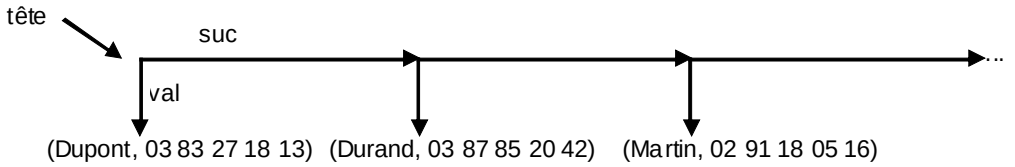
Exemple

On cherche une représentation pour une table qui, à un nom d'employé associe son numéro de téléphone. Soit *tabNumTél* cette table :

Dupont ----> 03 83 27 18 13
Durand ----> 03 87 85 20 42
Martin ----> 02 91 18 05 16
... ...

Les clés sont des chaînes de caractères. L'accès calculé n'est donc pas conseillé. On choisit l'adressage associatif.

La table *tabNumTél* est représentée par la liste des entrées :



Il existe plusieurs possibilités pour représenter cette liste des entrées.

Solution 1 : liste des entrées représentée de façon contiguë et triée

Cette solution est intéressante lorsqu'il y a très peu de modifications (départs ou arrivées d'employés dans l'entreprise). En effet, les adjonctions et suppressions sont lentes car elles nécessitent des décalages. Par contre, la recherche est rapide car elle peut être dichotomique.

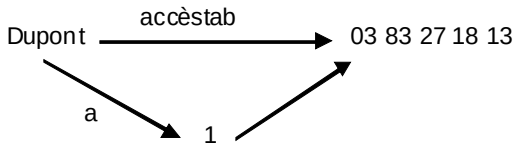
	nom	numéro de téléphone
1	Dupont	03 83 27 18 13
2	Durand	03 87 85 20 42
3	Martin	02 91 18 05 16
	.	
	.	
	.	

tableau (ou fichier relatif) tabNumTél

On trouve l'entier correspondant à un nom par accès associatif. La fonction d'adressage a est la suivante :

a : Dupont ----> 1
Durand ----> 2
Martin ----> 3
...

La recherche des informations relatives à Dupont se décompose de la manière suivante :



Solution 2 : liste des entrées représentée de façon chaînée et triée

Cette solution est intéressante lorsqu'il y a beaucoup de modifications. En effet, le chaînage facilite les adjonctions et suppressions. Par contre, la recherche est séquentielle donc plus lente que dans le cas précédent.

		val		suc	
	nom	**numéro de téléphone**		**lien**	
0				1	Tête
1	Dupont	03 83 27 18 13		20	
2	Perrin	03 87 87 01 19		3	
3	Valentin	03 83 55 76 88		0	
4	Martin	02 91 18 05 16		2	
	.				
	.				
	.				
20	Durand	03 83 85 20 42		4	
	.				
	.				
	.				
n					

Plusieurs solutions sont possibles pour la tête de liste. Nous avons choisi de la ranger à l'indice 0 dans la partie lien.

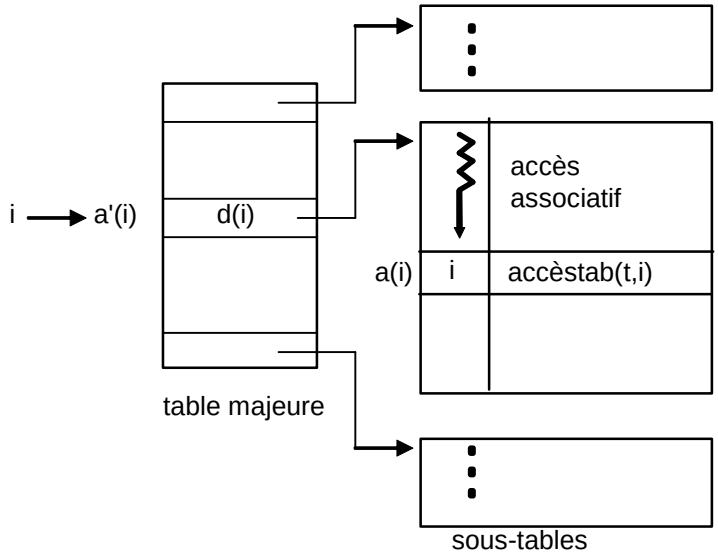
On trouve l'entier correspondant à un nom par accès associatif.
D'où
a : Dupont ----> 1
Durand ----> 20
Martin ----> 4
Perrin ----> 2
Valentin ----> 3

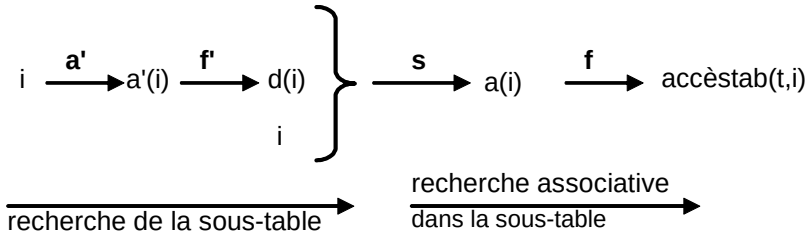
3.3 Partage (découpage) de la table

On veut garder l'adressage associatif mais diminuer l'intervalle de recherche. Pour cela, on va partager l'ensemble des clés en sous-ensembles et donc avoir des sous-tables. L'adressage va se faire en deux phases : retrouver la sous-table dans laquelle se trouve la clé puis rechercher la clé dans cette sous-table. Cela se fait en général à l'aide d'une *table majeure* qui associe à chaque entrée *i* l'adresse *d(i)* de la sous-table à laquelle appartient *i* ; la table majeure joue en quelque sorte un rôle d'aiguillage.

La recherche d'une entrée se décompose alors en 2 :

- détermination de la sous-table où elle se trouve
- recherche associative dans cette sous-table





s représente la recherche (associative) de l'élément contenant i dans la sous-table
(son adresse, résultat de la recherche, est a(i))

remarque1 :

afin d'avoir des temps d'accès du même ordre dans les différentes sous-tables, il vaut mieux que celles-ci soient approximativement de même taille. C'est un critère important pour le choix d'un partage.

remarque2 :

un avantage du partage en sous-tables est qu'il permet de n'avoir, à chaque instant, qu'une seule sous-table (et la table majeure) en mémoire centrale. Ceci peut justifier un partage même si on fait de l'adressage calculé dans chaque sous-table.

Nous allons étudier les différentes représentations possibles de a', c'est-à-dire les différentes façons de découper l'ensemble des entrées en sous-tables.

Adressage (rangement) partitionné

Principe

On regroupe dans une même sous-table les clés *i* ayant une certaine propriété *p(i)* commune (par exemple un même préfixe). La priorité est portée sur le regroupement des clés, à priori significatif.

Définition

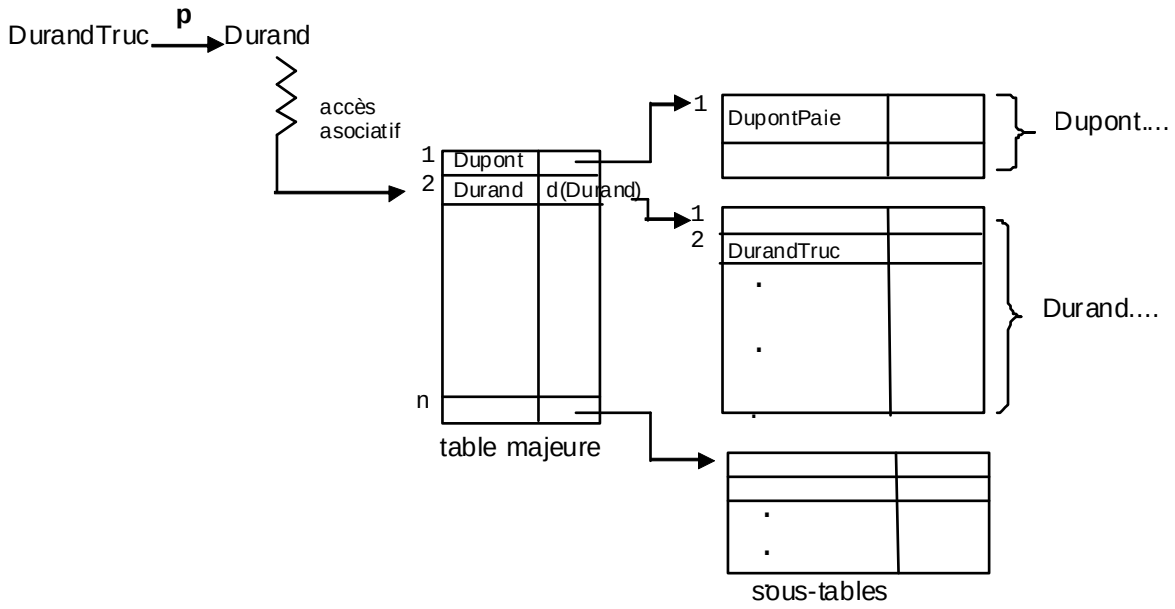
a'(i) est obtenu par un adressage associatif portant sur la valeur *p(i)* obtenue à partir de *i* et commune à tous les éléments d'une même sous-table.

Exemple : catalogue de fichiers

table : nom de fichier → renseignements sur ce fichier

Chaque nom de fichier est formé du nom de son propriétaire suivi d'un titre, par exemple DupontPaie.

Le partage peut se faire selon le nom du propriétaire (c'est une valeur obtenue à partir de la clé et commune à tous les éléments d'une même sous-table): ici p (DupontPaie) = Dupont.



La table majeure d'une part et les sous-tables d'autre part sont représentées par des listes, qui à leur tour peuvent être représentées de manière contiguë ou non, triée ou non.

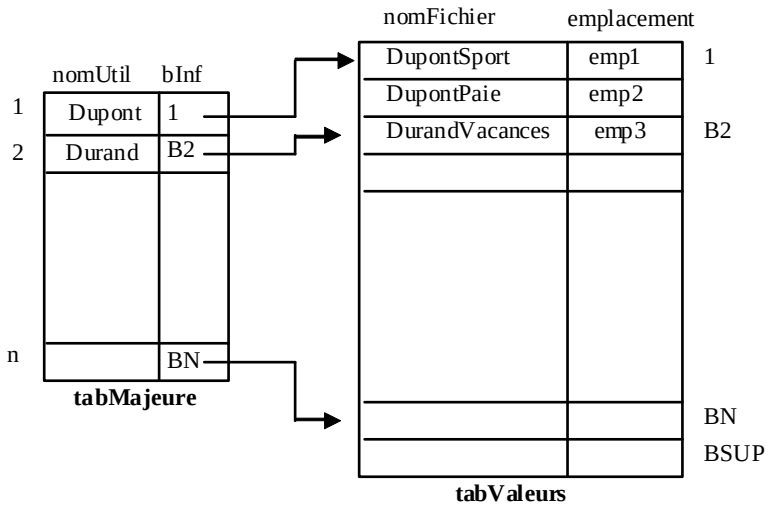
Exercice

Soit *tabFichier* la table décrite dans l'exemple ci-dessus. Ecrire, pour l'instruction logique suivante, l'algorithme de programmation lié à la représentation choisie :

renseignement ← accèstab (tabFichier, nomFich)

sachant que *tabFichier* est partitionné et que la table majeure et les sous-tables sont représentées de manière contiguë dans deux tableaux :

- *tabMajeure* le tableau contenant *n* couples (*n* donné) composés du nom de l'utilisateur et de la borne inférieure de la sous-table associée. *tabMajeure* est trié par nom.
- *tabValeurs* contenant toutes les sous-tables.



On suppose que les renseignements sur un fichier correspondent à une variable composite de type ***Renseignement*** dont le premier champ contient la taille du fichier.

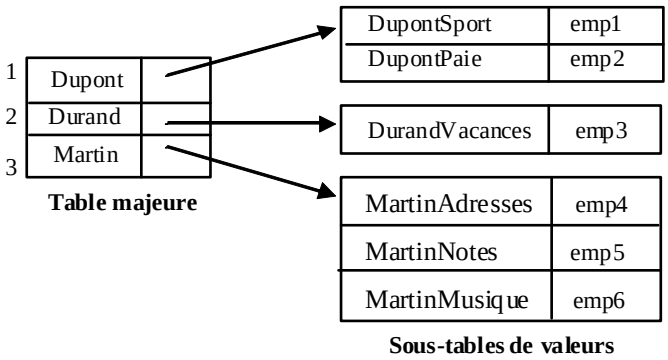
On dispose des fonctions suivantes :
- fonction fichPréfixe (nomFich) : chaîne, extrait le préfixe du nom de fichier.
- fonction fichRechDicho (tab, bi, bs, nom): entier, recherche dichotomique de *nom* dans le tableau *tab* entre les bornes *bi* et *bs*, rend l'indice où se trouve le nom ou 0 s'il n'y est pas.
- fonction fichRechSeq (sstab, taille, bi, nom) : entier, recherche séquentielle de *nom* dans le tableau *sstab* de borne supérieure *taille* à partir de *bi*, rend l'indice où se trouve le nom ou 0 s'il n'y est pas.

Illustration des différents niveaux de choix sur un exemple :

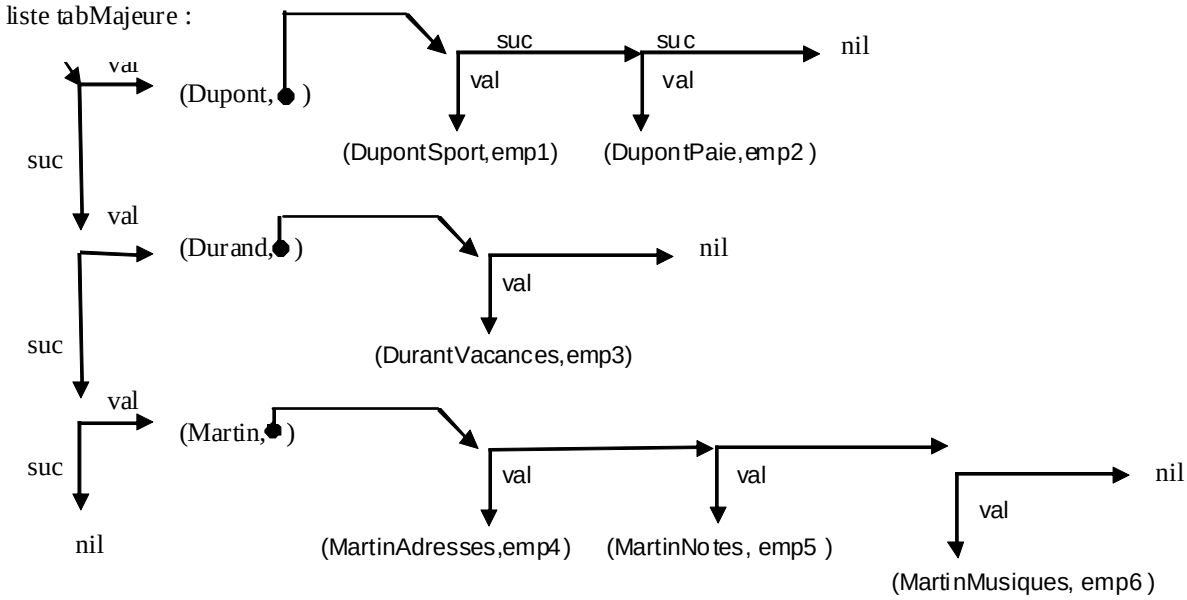
- niveau logique :

  table tabFichier : DupontSport → emp1
  DupontPaie → emp2
  DurandVacances → emp3
  MartinAdresses → emp4
  MartinNotes → emp5
  MartinMusique → emp6

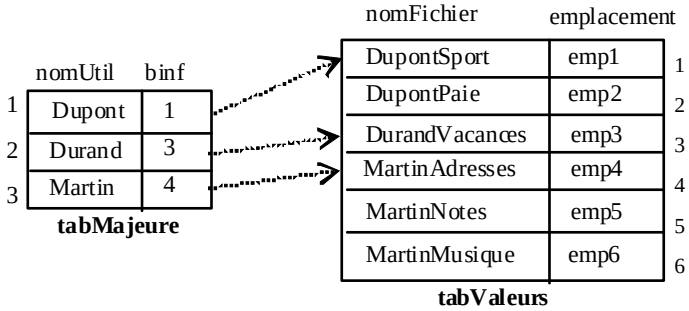
- choix 1 : découpage et adressage partitionné



- choix 2 : 2 choix séparables
  table majeure : rangement associatif (par liste des entrées) : liste tabMajeure
  sous-tables : idem



- choix 3 : représentation contiguë dans un tableau pour la liste table majeure (tabMajeure), représentation contiguë dans un seul et même tableau pour toutes les listes des sous-tables (tabValeurs).



Algorithme de programmation

```
Fonction accèstab (tabFichier : TabFichier, nomFich : chaîne) : Renseignement
    début
        préfixe ← fichPréfixe (nomFich)
        indiceUtil ← fichRechDicho (tabFichier.tMaj, 1, tabFichier.tMajSup, préfixe)
        indiceSousTable ← tabFichier.tMaj [indiceUtil].bInf
        si indiceUtil < tabFichier.tMajSup  alors bs ← tabFichier.tMaj [indiceUtil+1].bInf -1
                                            sinon bs ← tabFichier.tValSup
        fsi
        indiceFichier ←fichRechSeq (tabFichier.tVal, bs, indiceSousTable, nomFich)
        si indiceFichier = 0  alors renseignement.taille ← -1  (* valeur particulière correspondant à ω *)
                              sinon renseignement ← tabFichier. tval [indiceFichier].emplacement
        fsi
        retourne renseignement
    fin
```

Lexique

- tabFichier : TabFichier, variable composite regroupant la table majeure et les tables de valeurs ainsi que les bornes supérieures de leur intervalle de définition
- TabFichier = <tMaj : TMaj, tMajSup : entier, tVal : TVal, tValSup : entier>
- TMaj = tableau <nomUtil : chaîne, bInf : entier> [1..tabFichier.tMajSup]
- TVal = tableau <nomFichier : chaîne, emplacement : Renseignement> [1.. tabFichier.tValSup]
- préfixe : chaîne, nom de l'utilisateur

- nomFich : chaîne, nom du fichier
- indiceUtil : entier, indice du propriétaire dans la table majeure
- indiceSousTable : entier, indice du début de la sous-table associée au propriétaire
- indiceFichier : entier, indice du fichier cherché dans tabFichier.tVal
- bs : entier, borne supérieure de la sous-table associée au propriétaire
- renseignement : Renseignement, renseignement sur le fichier cherché ou valeur particulière s'il n'y est pas
- Renseignement = <taille : entier, …>

Rangement indexé

Principe

On est dans le cas où l'ensemble C des clés est totalement ordonné et borné supérieurement; le partage en sous-tables se fait en partageant C en intervalles.

Définition

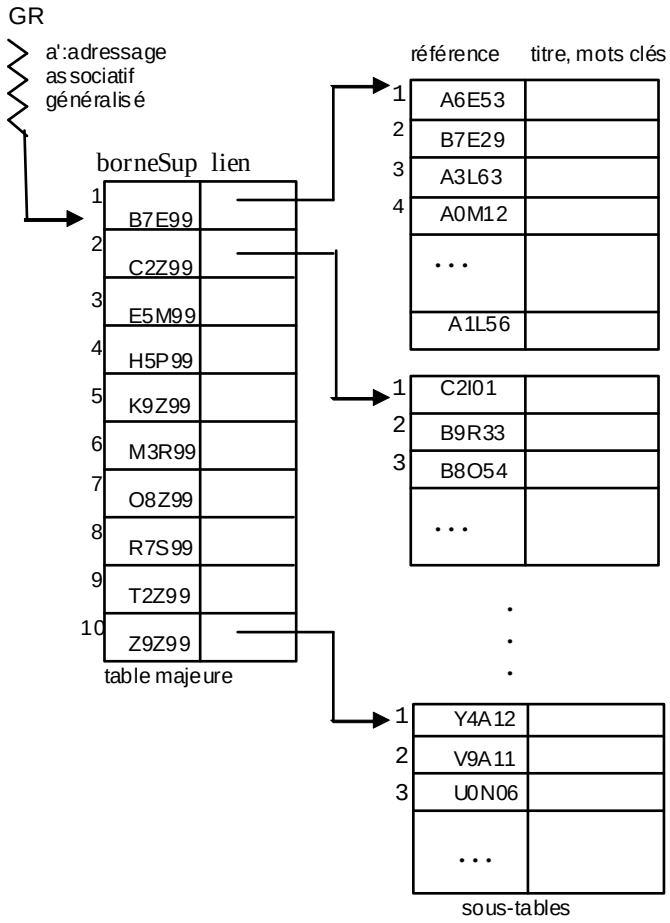
La table majeure contient les bornes supérieures des intervalles. a' est un adressage associatif généralisé, donnant, pour une clé i, l'élément (indic, adr) de la table majeure représentatif de son intervalle : c'est le premier élément dont indic ≥ i. La table majeure est triée, il est intéressant de la représenter de manière contiguë, ce qui permet une recherche dichotomique et est sans inconvénient car elle ne varie pas. Les sous-tables peuvent être représentées de manière contiguë ou non, triées ou non.

Exemple

Soit une table d'ouvrages (*tOuvrage*), associant à chaque référence d'ouvrage (c'est une chaîne de 5 caractères) : un titre (*chaîne*) et un tableau de 5 mots clés (un mot clé est une *chaîne*) :

tOuvrage : référence → titre, tableau de mots clés

On suppose qu'il y a au maximum 10000 ouvrages. On choisit de découper l'ensemble des références en 10 intervalles et donc de répartir les ouvrages dans 10 sous-tables en fonction de la valeur de leur référence. La table majeure contient pour chaque sous-table son indicatif (référence) maximum.



La recherche de la valeur (titre, mots clés) associée à une clé (référence de l'ouvrage, notons la *ref*) se fait en 2 étapes :

- Recherche dans la table majeure du premier élément dont la valeur du champ *borneSup* est supérieure ou égale à *ref*, soit *el* cet élément
- Recherche dans la sous-table désignée par le lien associé à *el* de l'élément de référence *ref* par un parcours séquentiel.

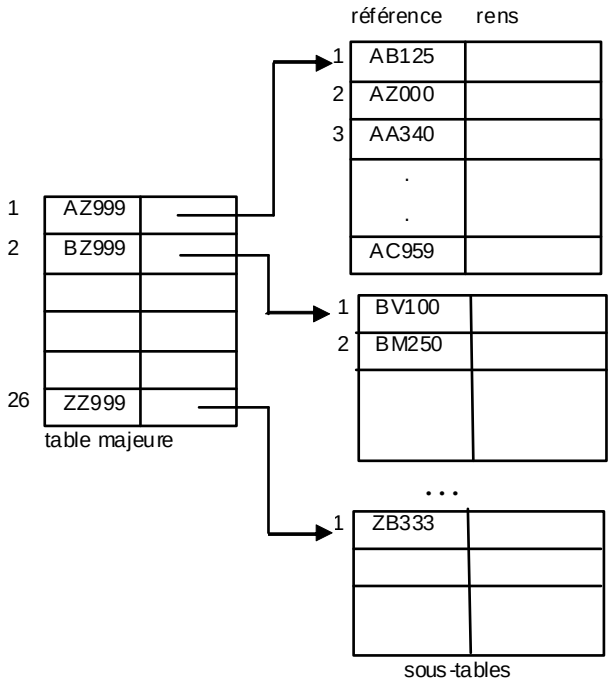
Cette technique est utilisée pour le rangement indexé sur disque. Les fichiers indexés seront étudiés en Cobol et la recherche à partir d'une clé se fera à l'aide d'une instruction Cobol. C'est cette instruction qui réalisera la recherche d'abord dans la table majeure puis dans une sous-table.

Exercice

Une entreprise possède un annuaire des adresses et numéros de téléphone de ses clients. Il y a au maximum 300 000 clients. Un client est identifié par une référence composée de 2 lettres et d'un numéro (de 0 à 999).

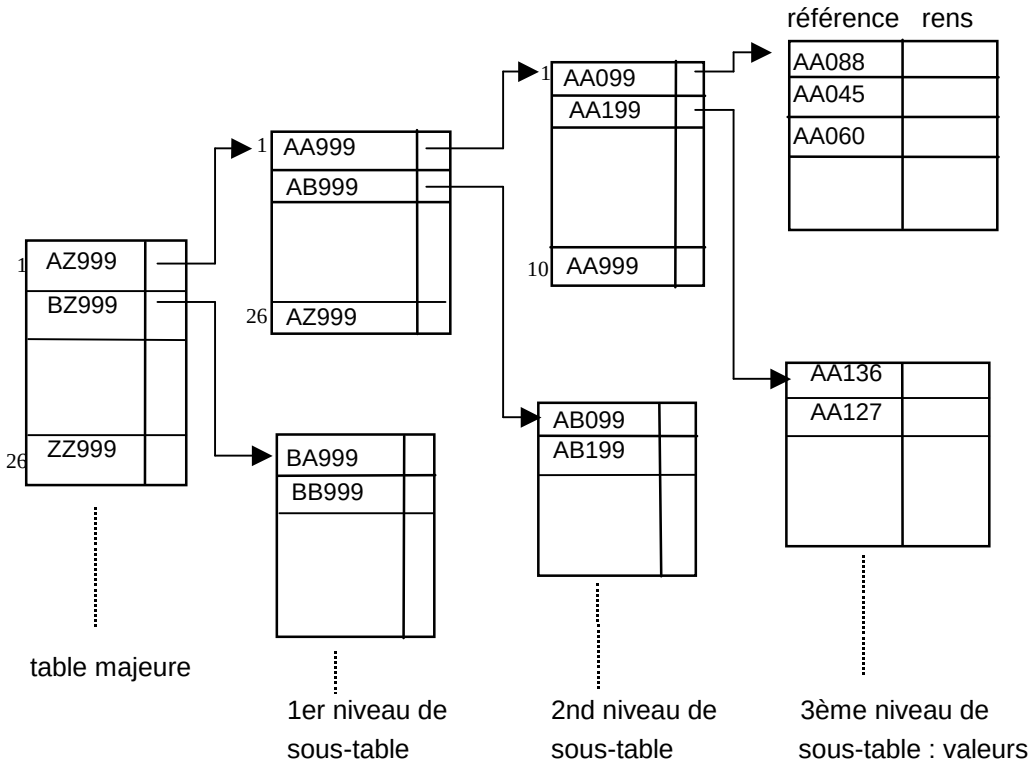
On a choisi une représentation indexée pour cet annuaire.

(nombre de clés : 676000 (26 X 26 X 1000))



1) Sachant que le découpage de la table se fait sur la première lettre de la référence, et qu'il n'y a qu'un niveau de sous-table (on a donc 26 sous-tables), calculer le nombre moyen de tests pour trouver les informations associées à une référence donnée.

2) Les fichiers séquentiels indexés admettent plusieurs niveaux d'index. En supposant que la représentation choisie intègre 3 niveaux de sous-tables décrits dans le schéma ci-après, calculer le nombre moyen de tests pour retrouver une information. Evaluer le gain par rapport à la première représentation.



3) Donner l'algorithme de programmation de l'opération logique *accèstab* avec la représentation à 3 niveaux.

On suppose que :
- ω est représenté par le couple (« », 0),
- la table majeure et les sous-tables de niveau 1 et 2 sont représentées dans un seul et même tableau appelé *tabIndex* (pour "table d'index"). L'adresse d'une sous-table est donnée par son indice de début dans *tabIndex*. Ce tableau est mémorisé dans un fichier.
- les sous-tables de valeurs (dernier niveau de sous-tables) sont représentées de manière chaînée dans un même fichier relatif *tabValeurs*. L'adresse d'une sous-table est donnée par le rang de son premier élément.
- on dispose des fonctions *clientRechDicho*, *clientRechSeq* et *fichierExtraireElément* :

  fonction clientRechDicho (t, début, fin, élément) : entier
  effectue une recherche dichotomique généralisée de *élément* dans le tableau *t* entre les indices *début* et *fin* et rend l'indice où devrait se trouver *élément*.

  fonction clientRechSeq (f, début, élément) : entier
  effectue une recherche séquentielle de *élément* dans le fichier *f* à partir du rang *début* en parcourant les liens de chaînage et rend le rang où il se trouve ou 0 s'il n'y est pas.

  fonction fichierExtraireElément (f, r) : ….
  extrait du fichier *f* l'élément qui se trouve au rang *r*.

Solution proposée :
Remarque :

En cas de rangement associatif simple : 300 000 éléments au maximum, donc 150 000 tests en moyenne (= 300 000 / 2).

1) * Recherche dans la table majeure :
si recherche séquentielle : 13 tests en moyenne
si recherche dichotomique : $\log_2 26 = 4{,}7$ tests

* Recherche de la référence dans la sous-table des valeurs :
Une sous-table contient au maximum 26000 éléments.
recherche séquentielle : 13000 tests en moyenne

Coût total de la recherche :
13013 tests si recherche séquentielle
13005 tests si recherche dichotomique quand c'est possible

2) * Recherche dans la table majeure :
si recherche séquentielle : 13 tests en moyenne
si recherche dichotomique : $\log_2 26 = 4{,}7$ tests

* Recherche dans la sous-table de premier niveau :
Une sous-table de premier niveau contient 26 éléments.
si recherche séquentielle : 13 tests en moyenne
si recherche dichotomique : $\log_2 26 = 4{,}7$ tests

* Recherche dans la sous-table de second niveau :
Une sous-table de second niveau contient 10 éléments.
si recherche séquentielle : 5 tests en moyenne
si recherche dichotomique : $\log_2 10 = 3{,}3$ tests

* Recherche séquentielle dans la sous-table des valeurs.
Une telle sous-table contient au pire 100 éléments.
Nombre de tests en moyenne : 50

au total :
81 tests en moyenne si recherches séquentielles (13 + 13 + 5 + 50)
62,7 tests en moyenne si recherches dichotomiques quand c'est possible (4,7 + 4,7 + 3,3 + 50)

Remarque :
Calcul de la taille de la table majeure et des sous-tables d'index (niveau 1 et 2) :
26+26*26+26*26*10 = 26+676+6760 = 7462
Calcul de la taille de la sous table de valeurs (niveau 3) :
= 300 000 (nombre maximum de clients)
D'où au total 307 462 places.
Un rangement calculé est possible mais nécessiterait 676 000 places (26 x 26 x 1000).

3) Algorithme de programmation

```
Fonction accèstab (tabAnnuaire : tabAnnuaire, réfClient : chaîne) : Rens
début
   (*recherche dans la table majeure*)
   indTabMaj ← clientRechDicho (tabAnnuaire.tabIndex, 1, 26, réfClient)
   indiceDébutSst1 ← tabAnnuaire.tabIndex [indTabMaj].indiceDébut
   (*recherche dans la sous-table de niveau 1*)
   indSst1 ← clientRechDicho (tabAnnuaire.tabIndex, indiceDébutSst1, indiceDébutSst1+25, réfClient)
   indiceDébutSst2 ← tabAnnuaire.tabIndex [indSst1].indiceDébut
   (*recherche dans la sous-table de niveau 2*)
   indSst2 ← clientRechDicho (tabAnnuaire.tabIndex, indiceDébutSst2, indiceDébutSst2+9, réfClient)
   indiceDébutSstval ← tabAnnuaire.tabIndex [indSst2] . indiceDébut
   (*recherche dans la sous-table de valeurs*)
   indTabValeurs ← clientRechSeq (tabAnnuaire.tabValeurs, indiceDébutSstval, réfClient)
   si indTabValeurs ≠ 0 alors
                élemTabValeurs ← fichierExtraireElément (tabAnnuaire.tabValeurs, indTabValeurs)
                adresseTéléphone ← élemTabValeurs. rens
   sinon   adresseTéléphone ← (« », 0)
   fsi
   retourne adresseTéléphone
fin
```

Lexique

TabAnnuaire =< tabIndex : TabIndex, tabValeurs : TabValeurs>
TabIndex = tableau <borneSup : chaîne, indiceDébut : entier> [1..7462]
TabValeurs = fichier ElemTabValeurs
ElemTabValeurs = <référence: chaîne, rens: Rens, lien: entier>
Rens = <adresse : chaîne, téléphone : entier> type des informations pour un client
tabAnnuaire : TabAnnuaire, le champ tabIndex est un tableau contenant la table majeure et les sous-tables de niveau 1 et 2, le champ tabValeurs est un fichier contenant la table des valeurs.
réfClient : chaîne, référence du client recherché
indTabMaj : entier, indice dans la table majeure de la sous-table de premier niveau cherchée
indiceDébutSst1 : entier, indice de début de la sous-table de niveau 1 dans *tabAnnuaire.tabIndex*
indSst1 : entier, indice dans une sous-table de niveau 1 de la sous-table de niveau 2 cherchée
indiceDébutSst2 : entier, indice de début de la sous-table de niveau 2 dans *tabAnnuaire.tabIndex*

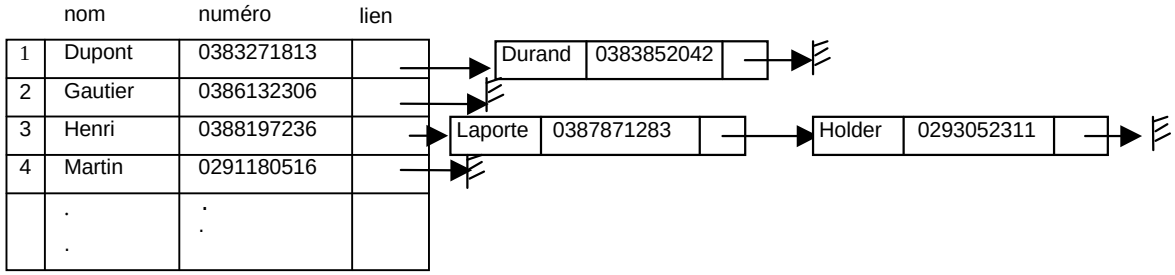
indSst2 : entier, indice dans une sous-table de niveau 2 de la sous-table de valeurs contenant *réfClient*
indiceDébutSstval : entier, rang de début de la sous-table de valeurs dans *tabAnnuaire.tabValeurs*
indTabValeurs : entier, rang de *réfClient* dans *tabAnnuaire.tabValeurs* ou 0
élemTabValeurs : ElemTabValeurs, valeur de l'élément de *tabAnnuaire.tabValeurs* de référence *réfClients*
adresseTéléphone : Rens, information sur le client de référence *réfClient*

Remarque :

On peut arriver à ce type de rangement en faisant du rangement associatif dans lequel on gère des **zones de débordement**. Dans l'exemple d'adressage (et rangement) associatif (2. b), on pourrait envisager la solution 3 suivante :

Solution mixte

La liste des couples est triée ; elle est contiguë au départ (table principale) puis les entrées ajoutées sont chaînées, la recherche dichotomique est possible. Cette solution est intéressante lorsque les modifications (départs ou arrivées d'employés) sont peu nombreuses. Elle évite des décalages de la solution 1 mais permet une recherche encore rapide.



Rangement dispersé ("hashing" en anglais)

Principe

On cherche une fonction non injective h(i) qui permet de découper l'ensemble des clés : celles qui donnent le même résultat appartiendront à la même sous-table. Le but, ici, est de moduler souplement le nombre (donc la taille) des sous-tables et l'équilibrage entre elles.

Définition

Un rangement dispersé est un rangement partagé dans lequel l'adresse a'(i) d'une clé est calculée à l'aide d'une fonction non injective.

a' s'appelle **fonction d'adressage dispersé** ou **hashcode**.

On peut distinguer deux cas :

Rangement dispersé indirect

La fonction de hashcode fournit une entrée dans la table majeure. L'accès aux sous-tables se fait par indirection via la table majeure.

Exemple 1 : table : nom d'employé → numéro de téléphone

h1 - Un exemple de fonction de hashcode possible :

appelons b_k le $k^{ème}$ caractère du nom et *nc* le nombre de classes désiré, c'est-à-dire le nombre d'entrées dans la table majeure ;

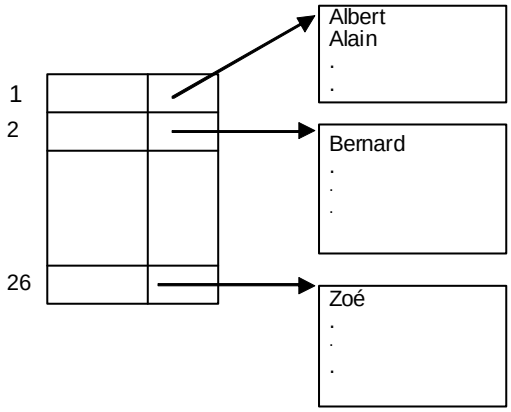
$h(nom) = (\sum_k k * num(b_k) * num(b_k)) \bmod nc + 1$,

où *num* est une fonction qui, à un caractère, associe un entier.

Le premier membre est un calcul portant sur les caractères de l'entrée, et *modulo nc* permet d'obtenir les *nc* classes.

h2 - Un autre exemple :

L'adresse a' (le résultat de la fonction de hashcode) est donnée par le rang de la première lettre du nom de l'employé. La table majeure contient 26 entrées et permet d'accéder aux 26 sous-tables.



Rangement dispersé direct

On ne stocke pas la table majeure, a'(i) donne directement l'adresse de début de la sous-table : la table majeure est remplacée par une fonction de calcul d'adresse. Dans ce mode de représentation, une seule zone contiguë de mémoire est suffisante, toutes les sous-tables sont dans la même zone.

Remarque

Cette méthode peut être vue comme une extension du rangement calculé : on calcule l'adresse de l'entrée considérée mais celle-ci peut ne pas s'y trouver parce qu'un "synonyme" (même valeur de la fonction a') a déjà pris la place ; il faudra la chercher en suivant une liste. Il existe diverses variantes d'adressage dispersé direct.

L'exemple 1 (h2) devient :

		nom	n° tél	lien	(n° sous-table)	
	1	André	...	30	(1)	
les 26	2	Blaise	...	28	(2)	
débuts de	3				(3)	
sous-tables	4	Dupont	...	27	(4)	
	.					
	26	Zaroli	...	0	(26)	
	27	Durand	...	29	(4)	zone de
	28	Bastien	...	31	(2)	débordement
	29	Didier	...	0	(4)	
	30	Alexandre	...	0	(1)	
	31	Bernard	...	0	(2)	
	.					
	.					
	.					
	n					

↑ indice du suivant ayant la même initiale

Les numéros de sous-tables rappellent la sous-table à laquelle chaque élément appartient. La 1ère sous-table contient 2 éléments, la 2ème 3 éléments, la 3ème aucun élément (mais une place est réservée pour son 1er élément) et la 4ème 3 éléments. Il y a 26 sous-tables ; chaque sous-table a une place qui lui est réservée parmi les 26 premières ; les autres places sont à chercher dans la zone de débordement commune à l'ensemble des sous-tables.

Exemple 2 : table : nom d'employé → numéro de téléphone

a' (i) = 1 + S mod T

avec S : somme des rangs, dans l'alphabet, de chacun des caractères de l'entrée
T : taille de la table (prenons 100)

Cette formule donne pour a' des valeurs comprises entre 1 et 100. On peut ainsi espérer une répartition équitable des adresses et avoir en moyenne un élément par sous-table.

	nom	n° tél	lien
1	Massinet	------	0
2	Sartini	------	0
.			
.			
.			
91	Dupont	------	2
.			
100			

a'(Massinet) = 1
a'(Dupont) = 91
a'(Sartini) = 91

Il y a au maximum 100 sous-tables ; une sous-table a son 1er élément qui se trouve au rang donné par a', les autres sont placés à des endroits libres dans la table. Les liens de chaînage permettent d'accéder à l'ensemble des éléments d'une sous-table à partir du premier. La particularité, ici, est que le nombre de classes est fixé à la valeur de la taille de la table, si bien qu'en moyenne la taille de chaque sous-liste est de 1.

Exercice

Pour faciliter la gestion des utilisateurs d'un système d'exploitation, on dispose d'une table *tabUtil* qui, à un nom de « login » (supposé unique), associe les informations suivantes : nom, prénom, numéro de bureau, numéro de téléphone et adresse électronique de l'utilisateur. On choisit pour représenter cette table un rangement dispersé direct. Les sous-tables sont représentées de manière chaînée dans un tableau *tabUtil*. La fonction h(i) est le calcul du rang de la première lettre de i. Les 26 premières cases de *tabUtil* sont occupées par des éléments pour lesquels h(i) correspond à l'indice de rangement, il s'agit des premiers éléments de chaque sous-table. Les autres places forment la zone de débordement et contiennent les autres éléments des sous-tables. Le type de *tabUtil* est le suivant :

TabUtil = tableau ElemTabUtil [1..MAX]
ElemTabUtil = <nomLogin : chaîne, infoUtil : InfoUtil, suiv : entier>
InfoUtil = <nom : chaîne, prénom : chaîne, bureau : entier, tél : entier, mél : chaîne>

On demande d'écrire les algorithmes de programmation de *accèstab*, *adjtab* et *suptab*. On suppose disposer d'une fonction *rang* qui renvoie le rang dans l'alphabet de la lettre donnée en paramètre.

Convention
on initialise le tableau *tabUtil* en mettant dans chaque case un élément dont le nom de « login » est la chaîne vide. Cette convention permettra de détecter les places libres et de faciliter les suppressions.

Correction :
Fonction de recherche
Elle recherche les informations associées à un utilisateur à l'aide de la fonction prédéfinie *rang* appliquée au premier caractère de l'entrée ; si la place désignée par cette adresse n'est pas vide, la recherche se fait en parcourant les liens qui unissent des éléments de même adresse.

Algorithme

```
Fonction accètab (tabUtil: TabUtil, nomLogin : chaîne) : InfoUtil
        début(*1*)
        indiceUtil ← rang (ième (nomLogin,1))
        élémTabUtil ← tabUtil [indiceUtil]
        si élémTabUtil . nomLogin ≠ «» alors (*2a*)
                trouvéUtil ← faux
                tantque (*3*) indiceUtil≠ 0 et non trouvéUtil faire
                        élémTabUtil ← tabUtil [indiceUtil]
                        si élémTabUtil.nomLogin ≠ nomLogin
                                alors (*4a*) indiceUtil ← élémTabUtil .suiv
                                sinon (*4s*) trouvéUtil ← vrai
                        fsi (*4*)
                ftantque (*3*)
```

```
        sinon (*2s*)
            indiceUtil ← 0
        fsi (*2*)
        si indiceUtil = 0 (*5*) alors
            res  ← (« », « », 0, 0, « »)  // ou res.nom ← «»
        sinon
            res ← tabUtil[indiceUtil] . infoUtil
        fsi (*5*)
        retourne res
        fin (*1*)
```

Lexique

indiceUtil : entier, indice dans *tabUtil*
trouvéUtil : booléen, à vrai si on trouve la clé cherchée
tabUtil : TabUtil
nomLogin : chaîne, entrée recherchée
élémTabUtil : ElémTabUtil, élément courant de *tabUtil*
res : InfoUtil, les informations associées à l'utilisateur ou une valeur particulière représentant ω
fonction rang (l : caractère) : entier, retourne le rang dans l'alphabet de la lettre *l*

Fonction d'insertion

On détermine l'adresse de l'élément à insérer. Si la place désignée par celle-ci est libre, l'insertion est immédiate ; on positionne alors le lien associé à cet élément à zéro. Si ce n'est pas le cas, on recherche dans la zone de débordement la première place libre *placeLibre* dans laquelle, si elle existe, on range l'élément traité. On modifie les liens de telle sorte que le nouvel élément soit le 2ème de la liste des utilisateurs de même hashcode que lui.

Algorithme

```
Fonction adjtab (tabUtil InOut : TabUtil, nomLogin : chaîne, infoUtil : InfoUtil )
        début (*1*
        adresseUtil ← rang (ième (nomLogin,1))
        premierUtil ← tabUtil[adresseUtil]
        si premierUtil . nomLogin  = «» alors (*2a*)
                    tabUtil[adresseUtil] ←  (nomLogin, infoUtil,0)
        sinon (*2s)
                    placeLibre ← tabUtilRechercherPlaceLibre (tabUtil, MAX)
                    si placeLibre ≠0 alors (*3a*)
                                tabUtil [placeLibre] ← (nomLogin, infoUtil, premierUtil.suiv)
                                tabutil [adresseUtil].suiv ←  placeLibre
                    fsi(*3*)
        fsi (*2*)
        fin (*1*)
```

Lexique

tabUtil : TabUtil
nomLogin : chaîne, entrée de l'élément à insérer
infoUtil : InfoUtil, valeur de l'élément à insérer
adresseUtil : entier, hashcode de l'élément à insérer
premierUtil : ElemTabUtil, premier utilisateur de même hashcode que celui à insérer
placeLibre : entier, indice de la première place libre dans la zone de débordement ou 0
fonction tabUtilRechercherPlaceLibre (tabUtil : TabUtil, bs : entier) : entier, rend l'indice d'une place libre dans la zone de débordement, de l'indice 27 à bs, ou 0 s'il n'y en a pas
fonction rang (l : caractère) : entier, retourne le rang dans l'alphabet de la lettre *l*

Fonction de suppression

On recherche l'élément dans la table :

- s'il est parmi les 26 premiers (soit s'il se trouve à la place *indAdresse* indiquée par son adresse) et si son lien désigne un élément dans la zone de débordement, on ramène cet élément à la place *indAdresse* et on le supprime dans la zone de débordement ; si l'élément à supprimer est le seul ayant cette adresse dans la table, on le supprime directement.
- sinon (soit s'il se trouve dans la zone de débordement), on relie l'élément qui le désigne à celui que lui-même désigne (existant ou non) et on le supprime.

```
Algorithme
fonction suptab (tabUtil InOut : TabUtil, nomLogin : chaîne)
    début (*1*)
    indAdresse ← rang (ième (nomLogin, 1))
    premierUtil ← tabUtil[indAdresse]
    si premierUtil.nomLogin ≠ «» alors (*2a*)  //facultatif
        indSup ← indAdresse
        présent ← faux
        tantque (*3*) indSup ≠ 0 et non présent faire
            si tabUtil [indSup].nomLogin ≠ nomLogin alors (*4a*)
                indPréc ← indSup
                indSup ← tabUtil [indSup].suiv
            sinon (*4s*)
                présent ← vrai
            fsi (*4*)
        ftantque (*3*)
        si présent alors (*5a*)   //facultatif
            si indSup ≤ 26 alors (*6a*)
                indSuiv ← tabUtil [indSup].suiv
                si indSuiv ≠ 0 alors (*7a*)
                    tabUtil [indSup] ← tabUtil [indSuiv]
                    tabUtil [indSuiv].nomLogin ← «»
                sinon (*7s*)
                    tabUtil[indSup].nomLogin ← «»
                fsi (*7*)
            sinon (*6s*)
                tabUtil [indPréc].suiv ← tabUtil [indSup].suiv
                tabUtil [indSup].nomLogin ← «»
            fsi (*6*)
        fsi (*5*)
    fsi (*2*)
    fin (*1*)
Lexique
    indAdresse : entier, adresse de l'élément à supprimer
    présent : booléen, indique si l'élément à supprimer existe dans la table
    indSup : entier, indice de l'élément à supprimer
    tabUtil : TabUtil
    nomLogin : chaîne, entrée de l'élément à supprimer
    premierUtil : ElemTabUtil, premier utilisateur de même hashcode que celui à supprimer
    indPréc : entier, suite d'indices, le dernier terme désignera la place de l'élément précédent celui à supprimer (dans la
        liste des éléments de même adresse)
    indSuiv : entier, place de l'élément qui suit celui à supprimer ou 0
    fonction rang (l : caractère) : entier, retourne le rang dans l'alphabet de la lettre l
```

3.4 Résumé

Comment représenter une table dont l'ensemble des clés n'est pas un intervalle d'entiers ?

Accès calculé (le plus efficace mais pas toujours possible)

Il faut trouver une fonction injective a : C → [1 . . nb] où nb = nombre de clés.
recommandé quand le nombre de clés n'est pas très grand par rapport au nombre d'entrées et quand la fonction est trouvable.

Accès associatif (toujours possible mais pas toujours très efficace)

On stocke les couples (indicatif, valeur) : c'est la liste des entrées.
Inconvénient : l'accès n'est pas direct.

Partage de la table

Une table majeure (ou un calcul) permet de définir dans quelle sous-table chercher. L'accès n'est pas direct mais tout de même meilleur que l'accès associatif car l'intervalle de recherche est plus petit.

Trois modes de rangement pour le partage de la table :

- **rangement partitionné**

lorsqu'on est capable de regrouper les clés en classes (partition) grâce à une propriété commune, et que l'accès dans la table majeure est associatif.

- **rangement indexé**

lorsque les clés sont ordonnés et qu'on définit des intervalles sur C.

- **rangement dispersé ou « hashcode » (direct ou indirect)**

lorsqu'on applique une fonction de calcul non injective sur les clés et que l'accès dans la table majeure ou aux sous-tables est direct.

EXERCICES SUR LES LISTES ET LES TABLES

1 Exercice 1

On s'intéresse à la gestion de l'organisation du brevet des collèges pour un collège qui est centre d'examen. Ce collège accueille ses élèves de 3ème ainsi que ceux d'autres collèges. L'examen comporte 3 épreuves écrites: français, mathématiques et histoire/géographie. La note finale du brevet est obtenue, pour chaque élève, à partir des notes aux 3 épreuves ainsi que d'une note de contrôle continu calculée à l'aide des notes obtenues en classe de 4ème et 3ème. On dispose des informations suivantes :

- la liste des enseignants retenus pour la correction des copies (de nom *lEnsCor* et de type *LEnsCor*) contenant pour chaque enseignant, son nom, son prénom et la matière qu'il corrigera.

- la table des élèves (de nom *tabEl* et de type *TabEl*) qui associe à un numéro d'élève les informations suivantes sur cet élève: nom, prénom, adresse et nom du collège d'origine.

- la table des notes (de nom *tabNotes* et de type *TabNotes*) qui associe à un numéro d'élève les notes de cet élève: note de français, note de mathématiques, note d'histoire/géographie, note de contrôle continu.

- la table des copies corrigées par un enseignant (de nom *tabEnsCop* et de type *TabEnsCop*) qui associe à un triplet (nom, prénom, matière corrigée) caractérisant un enseignant, la liste contenant pour chaque copie corrigée, le numéro de l'élève et la note attribuée.

Les types des différentes structures de données sont précisés ci-après :

LEnsCor = liste (Enseignant)
Enseignant = <nom : chaîne, prénom : chaîne, matière : chaîne>

TabEl = table [NumEl ⟶ ElCol]
ElCol = <élève : Elève, collège : chaîne>
Elève = <nom : chaîne, prénom : chaîne, adresse : chaîne>
NumEl = entier

TabNotes = table [NumEl ⟶ Notes]
Notes = <français : entier, math : entier, histGéo : entier, contrôleCont : entier>

TabEnsCop = table [Enseignant ⟶ LCopies]
LCopies = liste (Copie)
Copie = < numEl : NumEl, note : entier>

Question 1

Ecrire **l'algorithme logique de la fonction** qui recherche dans la liste *lEnsCor* la matière corrigée par un enseignant dont on connaît le nom et le prénom. On suppose qu'il n'y a pas 2 enseignants de même nom et prénom.

Question 2

La table *tabEl* existe, on souhaite la compléter en ajoutant les élèves d'un collège donné. Ecrire **l'algorithme logique de la fonction** qui réalise cette mise à jour. On dispose :

- de la liste *lElUnCol* (de type LElUnCol = liste (Elève)) fournie par le collège donné,
- du nom de ce collège,
- et du dernier numéro d'élève attribué.

Le numéro affecté à un élève est obtenu par incrémentation du dernier numéro attribué.

Question 3

La table *tabNotes* a été créée, avant la correction des copies, en associant à chaque numéro d'élève 0 comme note de français, de mathématiques, d'histoire/géographie et la note de contrôle continu fournie par son collège d'origine pour le champ *contrôleCont*. On souhaite la compléter en ajoutant les notes de français, mathématiques et histoire/géographie obtenues au brevet et mémorisées dans la table *tabEnsCop*. Ecrire **l'algorithme logique de la fonction** réalisant ce traitement.

Question 4

Proposer et justifier en peu de lignes **une représentation en mémoire secondaire** (sur disques, disquettes,…) avec partage (découpage) pour la table *tabEnsCop*. On demande le principe, l'implantation physique et les définitions lexicales des variables nécessaires à cette représentation.

On suppose que :

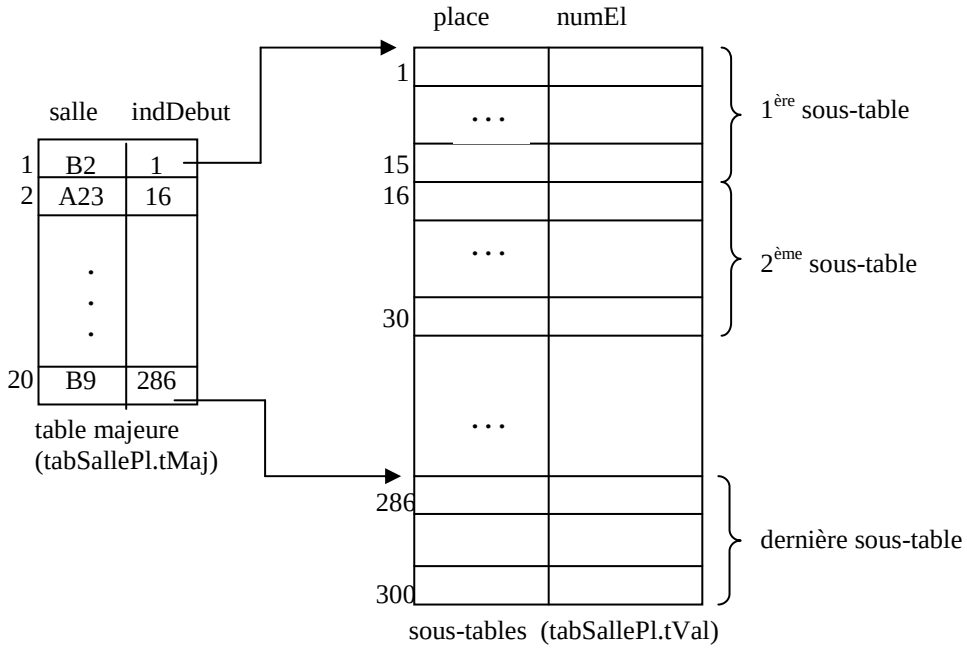
- chaque enseignant corrige une seule matière (rappel : il y a en tout 3 matières à corriger),
- il n'y a pas 2 enseignants de même nom et prénom,
- chaque enseignant corrige au maximum 50 copies.

Question 5

On dispose de la table des salles et des places (de nom *tabSallePl* et de type *TabSallePl*) qui associe à un couple (nom de salle, numéro de place) un numéro d'élève. Voici la définition de type correspondante :

TabSallePl = table [SallePlace⟶ NumEl]
SallePlace = < salle : chaîne, place : entier>.

On choisit pour cette table *tabSallePl* une représentation basée sur un rangement partitionné en mémoire centrale. Sachant qu'il y a 20 salles de 15 places au maximum, que les salles sont repérées par leur nom et les places par des entiers, la table majeure est représentée de manière contiguë dans un tableau et les sous-tables également de manière contiguë mais dans un autre tableau :



Le type de *tabSallePl* est le suivant :

TabSallePl = < tMaj : TMaj, tVal : TVal >
TMaj = tableau ElTMaj [1..20]
ElTMaj = < salle : chaîne, indDebut : entier >
TVal = tableau ElTVal [1..300]
ElTVal = < place : entier, numEl : NumEl>

Ecrire l'algorithme de programmation de la fonction logique chgtab avec les paramètres tabSallePl, (salle, place) de type SallePlace et numEl de type entier sachant qu'on dispose :

- d'une fonction tMajRechSalle qui recherche une salle dans la table majeure et rend l'indice correspondant,
- d'une fonction tValRechPlace qui recherche une place dans une sous-table et rend l'indice correspondant.

On ne vous demande pas d'écrire les algorithmes de ces 2 fonctions dont voici les entêtes:

- fonction tMajRechSalle (tMaj : TMaj, salle : chaîne) : entier,
  retourne l'indice (compris entre 1 et 20) où se trouve la salle recherchée dans *tMaj*,
- fonction tValRechPlace (tVal : TVal, place : entier, indSst : entier) : entier,
  retourne un indice de *tVal*, c'est l'indice où se trouve la place recherchée dans la sous-table débutant à l'indice *indSst*.

Corrigé :
Question 1

```
fonction  lEnsCorTrouver (lEnsCor : LEnsCor, nom : chaîne, prénom : chaîne) : chaîne
        début (*1*)
        place ← tête (lEnsCor)
        trouve ← faux
        tant que  non finliste (lEnsCor, place) et non trouve faire (*2*)
                enseignant ← val (lEnsCor, place)
                si enseignant.nom = nom et enseignant.prénom = prénom (*3*)
                        alors trouve ←vrai
                        sinon place ←suc(lEnsCor, place)
                fsi (*3*)
        ftantque (*2*)
        si non trouve (*3*)
            alors matière ← « »
            sinon matière ← enseignant.matière
        fsi (*3*)
        retourne matière
        fin (*1*)
```

Lexique

lEnsCor : LensCor, liste des enseignants
nom : chaîne, nom de l'enseignant recherché dans la liste *lEnsCor*
prénom : chaîne, prénom de l'enseignant recherché dans la liste lEnsCor
place : Place, place courante dans la liste
enseignant : Enseignant, élément courant de la liste *lEnsCor*
trouve : booléen, à vrai si l'enseignant cherché est dans la liste *lEnsCor*
matière : chaîne, matière cherchée ou la chaîne vide si on n'a pas trouvé l'enseignant

Question 2

```
fonction  tabElCompleter (tabEl InOut : TabEl, collège : chaîne, lElUnCol: LElUnCol, numéroCourant InOut: entier)
        début (*1*)
        place  ← tête (lElUnCol)
        tant que non  finliste  (lElUnCol, place)  faire (*2*)
                élève ← val (lElUnCol, place)
                numéroCourant ← numéroCourant + 1
                adjtab (tabEl, numéroCourant, (élève, collège))
                place ← suc (lElUnCol, place)
        fintantque (*2*)
        fin (*1*)
```

Lexique
tabEl : TabEl, table d'élèves
lElUnCol: LelUnCol, liste d'élèves fournie par un collège
collège : chaîne, nom donné du collège
numéroCourant : entier, dernier numéro d'élève attribué
place : Place, place courant de la liste *lElUnCol*
élève : Elève, élève courant de la liste *lElUnCol*

Question 3

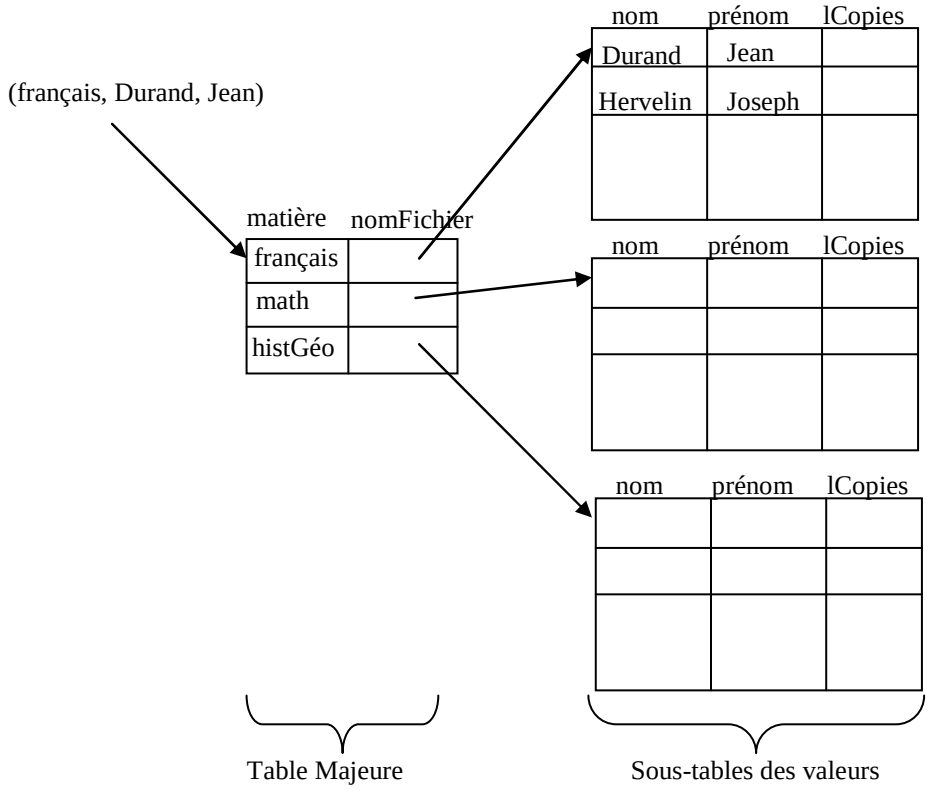
```
Fonction tabNotesMiseAJour (tabNotes InOut: TabNotes, tabEnsCop : TabEnsCop)
    début (*1*)
    Pour chaque enseignant dans dom (tabEnsCop) faire (*2*)
        (* on récupère la liste des copies corrigées par l'enseignant *)
        listeCopies ← accèstab (tabEnsCop, enseignant)
        (* parcours de la liste pour la maj de la table de notes *)
        place ← tête (listeCopies)
        tant que non finliste (listeCopies, place) faire (*3*)
            (* on récupère les informations liées à la copie courante *)
            copie ← val (listeCopies, place)
            (* on récupère les notes de l'étudiant dont la copie est traitée *)
            notes ← accèstab (tabNotes, copie.numEl)
            (* traitement selon la matière de l'enseignant *)
            si enseignant.matière = «francais» (*4*)
                alors notes.francais ← copie.note
                sinon si enseignant.matière = «math» (*5*)
                        alors notes.math ← copie.note
                        sinon notes.histGéo ← copie.note
                      fsi (*5*)
            fsi (*4*)
            (* mise à jour *)
            chgtab (tabNotes, copie.numEl, notes)
            (* copie suivante *)
            place ← suc (listeCopies, place)
        ftantque (*3*)
    fpour (*2*)
    fin (*1*)
```

Lexique
tabNotes : TabNotes, table contenant les notes de tous les élèves
tabEnsCop : TabEnsCop, table contenant la liste des copies corrigées par les enseignants correcteurs.
enseignant : Enseignant, enseignant courant dont on va parcourir la liste des copies corrigées
listeCopies: LCopies, liste des copies corrigées par l'enseignant courant de la table *tabEnsCop*
place : Place, place courante dans la liste des copies *listeCopies*
copie : Copie, copie courante dans la liste des copies *listeCopies*
notes : Notes, notes de l'élève courant (celui à qui appartient la copie courante) dans *tabNotes*

Question 4
Principe
Nous avons choisi un adressage partitionné car il est intéressant ici de regrouper les enseignants par une propriété commune p(indicatif) qui est la matière corrigée. Nous sommes dans le cas d'un rangement partitionné avec un accès associatif, donc il faut stocker les entrées.

Les sous-tables des valeurs regroupent les notes de tous les enseignants qui corrigent la même matière. Chaque sous-table est représentée de manière contiguë et triée sur le nom.

La table majeure est composée des trois matières français, math et histoire/géographie. Elle pointe sur le début de chaque sous-table de valeurs qui contient les enseignants (nom, prénom) ainsi que la liste des notes. On fait un accès associatif sur la table majeure pour trouver l'indice correspondant à la matière dans la table majeure, on fait ensuite un accès associatif sur la sous-table correspondante pour trouver l'indice correspondant dans la sous-table. On peut alors accéder à la liste des copies.



Implantation

Chaque sous-table de valeurs est stockée dans un fichier à accès séquentiel. Les données ne sont pas souvent modifiées. On choisit de les trier ce qui optimise la recherche sur le nom et le prénom.

La table majeure est stockée dans un fichier à accès séquentiel. Elle doit être conservée en mémoire secondaire d'après l'énoncé. Elle peut être chargée en mémoire centrale dans un tableau car sa taille est raisonnable. Vu le nombre d'éléments (3), le tri ne s'impose pas. Le pointeur sur la sous-table est en fait le nom physique du fichier qui contient la sous-table.

Définitions lexicales

Table Majeure :

TabEnsCop = fichier ElemTabMajeure

ElemTabMajeure = <matière : chaîne, nomFichier : chaîne>

Sous-tables :

SousTable : fichier ElemSousTable

ElemSousTable = <nom : chaîne, prénom : chaîne, listeNote : tableau Copie [1..50]>

Question 5

```
Fonction chgtab (tabSallePl InOut : TabSallePl, clé : SallePlace, numEl : entier)
    début (*1*)
        (* Accès à la table majeure *)
        indMaj ← tMajRechSalle (tabSallePl.tMaj, clé.salle)
        (* Indice de début de la sous-table *)
        indSst ← tabsallepl.tmaj [indMaj].indDebut
        (* Accès à la sous-table : recherche de l'indice de la place *)
        indPlace ← tvalRechPlace (tabSallePl.tVal, clé.place, indSst)
        (* mise à jour du numéro d'élève *)
        tabSallePl.tVal [indPlace].numEl ← numEl
    fin (*1*)
```

Lexique

tabSallePl : TabSallePl, table qui associe à une salle et une place, un numéro d'élève

clé : SallePlace, clé recherchée dans la table, pour laquelle on veut changer le numéro d'élève

numEl : entier, nouveau numéro d'élève

indMaj :entier, indice dans la table majeure correspondant à la salle recherchée

indSst : entier, indice dans le tableau des sous-tables correspondant au début de la liste des places de la salle recherchée

indPlace : entier, indice dans le tableau des sous-tables correspondant à la place recherchée dans la salle recherchée.

2 Exercice 2

Un club nautique organise des stages de croisière à la voile en France et désire gérer ses croisières et ses réservations sur plusieurs années. Voici les données à gérer :

- la table *tabEscales* des escales possibles (escale = port) qui associe à chaque nom de port (identifiant) les informations suivantes:

le code postal	*codePostal*
la difficulté d'entrée (de 1 à 5)	*difficulté*
le montant de la taxe à verser	*taxe*

- la collection *lesCroisières* des croisières proposées

Chaque croisière *croisière* est identifiée par un code *codeCroisière* constitué de :
l'an *an* (exemple : 1990)
suivie d'un numéro d'identification *numéro* qui est un entier entre 1 et 50 (le nombre de croisières varie d'une année à l'autre) ;
elle comprend aussi les renseignements suivants :

ville de départ et date	*départ*
ville d'arrivée et date	*arrivée*
liste *listeEscale* des noms d'escales intermédiaires avec les dates (au maximum 5 éléments)	
prix	*prix*
nombre de places disponibles (non réservées)	*nbPlaceDisponible*

Les dates ont la forme aaaammjj et sont de type réel.

- la liste *listeRéservation* des réservations pour les croisières.

Chaque réservation *réservation* (de type *Réservation*) comporte :
le nom *nom* et le prénom *prénom* du client
le code *codeCroisière* de la croisière choisie
la somme versée *accompte*
une observation éventuelle *observation* (exemple : «désire être avec Toto»)
le nombre *nbPlaces* de places à réserver.

Questions :

1) Proposez une structure logique pour *lesCroisières* (liste sur …, table de … dans …, …), et précisez complètement la structure des éléments.
Expliquez votre choix.

2) Représentation physique pour *lesCroisières* ; les représentations demandées seront données en 2 temps :
- principe ;
- implantation physique : tableau (précisez comment) ? fichier ? …

2-1) Proposez une représentation basée sur un rangement calculé et précisez (et justifiez) une représentation des "valeurs" des croisières. Quel est l'inconvénient de ce rangement ?
2-2) Proposez une représentation basée sur un rangement associatif ; on suppose que la collection *lesCroisières* est relativement stable.
2-3) Proposez une représentation basée sur un rangement partitionné.
2-4) Proposez une représentation basée sur un rangement dispersé, en mémoire centrale.

3) Ecrivez l'algorithme logique de la fonction qui permet d'imprimer les croisières pour lesquelles à la fois d'une part la difficulté maximale d'entrée dans les ports escales ne dépasse pas *difficultéMax* donné et d'autre part la somme totale à verser pour les escales (taxes) est inférieure à *taxeCroisièreMax* donné.

4) Ecrivez l'algorithme logique de la fonction qui permet d'enregistrer une inscription à une croisière et d'éditer une facture comportant le nom et le prénom du client, la ville et la date de départ, le prix et la liste des escales (nom du port et code postal). L'inscription est fournie sous la forme d'une variable de type Réservation.

Correction

1) Proposez une structure logique pour *lesCroisières* (liste sur …, table de … dans …, …), et précisez complètement la structure des éléments.

lesCroisières : LesCroisières
LesCroisières : table [CodeCroisière → Croisière]
CodeCroisière = <an : entier, numéro : entier>
Croisière = < départ : VilleDate, arrivée : VilleDate, listeEscale : liste (VilleDate), prix : réel, nbPlaceDisponible : entier >
VilleDate = <ville : chaîne, date : réel>

Expliquez votre choix : besoin d'un accès direct.

2) Représentation physique pour *lesCroisières* ; les représentations demandées seront données en 2 temps :
- principe ;
- implantation physique : tableau (précisez comment) ? fichier ? …

2-1) Proposez une représentation basée sur un rangement calculé et précisez (et justifiez) une représentation des valeurs des croisières. Quel est l'inconvénient de ce rangement ?

Principe : l'emplacement est donné par un "adressage calculé", c'est-à-dire à l'aide d'une fonction injective. On peut proposer :

$$a(i) = (i.an - anMin) * 50 + i.numéro$$

avec i : CodeCroisière, clé
anMin : entier, valeur minimum de l'année

Implantation physique :
Pour garder la table *lesCroisières* : fichier à accès direct.

Valeurs : type composite (cf question 1) ; avec pour *listeEscale* une représentation contiguë à l'aide d'un tableau à 5 éléments et d'une variable donnant le nombre d'escales.

2-2) Proposez une représentation basée sur un rangement associatif ; on suppose que la collection *lesCroisières* est relativement stable.

Rangement associatif <==> stockage de la liste des entrées :
lesCroisières stable ==> rangement contigu
On veut le conserver ==> fichier
Trié ? Pas indispensable mais trier par code croissant rend plus efficace l'accès.
Fonction f : direct car ni grande disparité de longueur, ni communauté de valeur entre éléments.

2-3) Proposez une représentation basée sur un rangement partitionné.

En voici un exemple :
Fonction p : codeCroisière → code_croisière.an
Table majeure : année → adresse de la sous-table contenant les croisières de l'année
Représentation logique de la table majeure : liste contiguë et triée de couples : an, tête
Représentation physique des sous-tables : un fichier par sous-table (séquentiel)
Représentation physique de table majeure : un autre fichier, séquentiel avec :
tête = nom du fichier correspondant à la sous-table (articles classés par années croissantes).

2-4) Proposez une représentation basée sur un rangement dispersé, en mémoire centrale.

Exemple de fonction hashcode (avec *n* classes) : h(i) = (i.an + i.numéro) mod n.
Donne directement accès à l'élément de la table majeure qui contient la tête de la sous-table voulue.
Table majeure : tableau à n éléments.
Sous-tables: dans un même tableau, soit chaînée, soit contiguë.
Si assez stable, plutôt contigu, mais il faut estimer le nombre d'éléments à attribuer à chaque sous-liste (place dans le tableau).

3) Ecrivez l'algorithme logique de la fonction qui permet d'imprimer les croisières pour lesquelles à la fois d'une part la difficulté maximale d'entrée dans les ports ne dépasse pas *difficultéMax* donné et d'autre part la somme totale à verser pour les escales (taxes) est inférieure à *taxeCroisièreMax* donné.

Algorithme logique

```
Fonction croisièresDiffTaxesImprimer (lesCroisières : LesCroisières, tabEscales : TabEscales, difficultéMax : entier, taxeCroisièreMax : réel)
    début (*1*)
    pour chaque (*2*) i dans dom (lesCroisières) faire
        croisière ← accèstab (lesCroisières, i)
        échec ← faux
        taxeCroisière ← 0
        place ← tête (croisière. listeEscale)
        tantque (*3*) non finliste (croisière. listeEscale, place) et non échec faire
                nomPort ← val (croisière. listeEscale, place) . ville
                valeurEscale ← accèstab (tabEscales, nomPort)
                si valeurEscale. difficulté > difficultéMax alors (4a*) échec ← vrai
                sinon (*4s)   taxeCroisière ← taxeCroisière + valeurEscale . taxe
                              place ← suc (croisière. listeEscale, place)
                fsi (*4*)
        ftantque (*3*)
        si non échec et taxeCroisière < taxeCroisièreMax
                alors (*5a*) écrire (i)
        fsi (*5*)
    fpourchaque(*2*)
    fin (*1*)
Lexique
    lesCroisières : LesCroisières
    difficultéMax : entier, difficulté maxi, donnée
    taxeCroisièreMax : réel, somme maxi, donnée
    ValeurEscale = < codePostal : chaîne, difficulté : entier, taxe : réel >
    tabEscales : TabEscales, table des escales possibles
    TabEscales = table [chaîne → ValeurEscale]
    i : CodeCroisière, entrée dans lesCroisières (identifiant de croisière)
    croisière : Croisière, renseignements sur la croisière i
    échec : booléen, « la croisière n'est pas à retenir car la difficulté est > à la difficulté demandée »
    taxeCroisière : réel, somme des taxes des premières escales de la croisière
    place : Place, place dans la liste des escales de la croisière courante
    nomPort : chaîne, nom du port de l'escale courante
    valeurEscale : ValeurEscale, renseignements sur l'escale courante
```

Remarque

La constitution du lexique au fur et à mesure est essentielle et guide fortement le bon emploi des listes, tables et variables composites. Tout ceci doit avoir une rigueur "algébrique".

4) Ecrivez l'algorithme logique de la fonction qui permet d'enregistrer une inscription à une croisière et d'éditer une facture comportant le nom et le prénom du client, la ville et la date de départ, le prix et la liste des escales (nom du port et code postal).

Algorithme logique

```
Fonction enregistrerInscriptionCroisière (lesCroisières : InOut LesCroisières, listeRéservation : InOut liste (Réservation), tabEscales : TabEscales, réservation : Réservation)
    début (*1*)
        croisière ← accèstab (lesCroisières, réservation . codeCroisière)
        si croisière . nbPlaceDisponible ≥ réservation .nbPlaces alors (*2*)
                croisière . nbPlaceDisponible ← croisière . nbPlaceDisponible - réservation .nbPlaces
                chgtab (lesCroisières, réservation . codeCroisière, croisière)
                adjqlis (listeRéservation, réservation)
                écrire (« Facture \n NOM = », réservation . nom)
                écrire (« PRENOM = », réservation . prénom )
                écrire (« DEPART = », croisière . départ )
                écrire (« PRIX = », croisière . prix * réservation .nbPlaces )
                écrire (« LISTE DES ESCALES »)
                place ← tête (croisière . listeEscale)
```

```
                tant que (*3*) non finliste (croisière . listeEscale, place) faire
                        nomEscale ← val (croisière . listeEscale, place). ville
                        écrire (nomEscale)
                        écrire (accèstab (tabEscales, nomEscale). codePostal)
                        place ← suc (croisière . listeEscale, place)
                ftantque(*3*)
        fsi (*2*)
    fin (*1*)
Lexique
    CodeCroisière = cf question 1
    Réservation = <nom : chaîne, prénom : chaîne, codeCroisière : CodeCroisière, acompte : réel, observation : chaîne,
        nbPlaces : entier>
    listeRéservation : liste (Réservation)
    réservation : Réservation, renseignement sur la réservation à ajouter
    croisière : Croisière, renseignements sur la croisière
    lesCroisières : LesCroisières
    place : Place, dans listeEscale
    tabEscales : TabEscales
    nomEscale : chaîne, nom de l'escale courante
```

ANNEXE 1

Conventions pour l'écriture des algorithmes

Un algorithme permet d'expliciter clairement les idées de solution d'un problème indépendamment d'un langage de programmation. Le «langage algorithmique» que nous utilisons est un compromis entre un langage naturel et un langage de programmation. Nous présentons les algorithmes comme une suite d'instructions dans l'ordre des traitements. Ils sont accompagnés d'un lexique qui indique, pour chaque variable, son type et son rôle. Un algorithme est délimité par les mots clés *début* et *fin*. Nous manipulerons les types couramment rencontrés dans les langages de programmation : entier, réel, booléen, caractère, chaîne, tableau et type composite.

1 Les principales instructions

- **Affectation** : ←

  On accepte, dans les algorithmes, les affectations (et même les opérations) globales sur tous les types où cela a un sens. Les différences viendront à la programmation.

- **Saisie d'une donnée** : v ← lire()

- **Affichage d'une valeur** : écrire (v)

- **Conditionnelle** : si *condition* alors *instructions1* sinon *instructions2* fsi

  La partie *sinon* est optionnelle.

- **Itérations** (les seuls en-têtes qui seront utilisés) :

  - pour *i* de *m* à *n* faire ... fpour

  - pour *i* décroissant de *n* à *m* faire ... fpour

  - tantque *condition* faire ... ftant

2 Le type tableau

Pour un tableau, on précise l'intervalle de définition et le type des éléments:

tableau typeDesEléments [borneInférieure .. borneSupérieure].

Par exemple, pour un tableau t de 10 entiers, on pourra écrire :

t : tableau entier [1..10].

3 Les fonctions

Les « morceaux » d'algorithmes indépendants sont présentés sous forme de fonctions dont les en-têtes précisent les paramètres donnés et le type du résultat, chaque paramètre est suivi de son type :

fonction *nomFonction* (liste des paramètres) : type du résultat

Le formalisme retenu est très largement inspiré du standard UML. Une fonction peut n'avoir aucun paramètre et/ou aucun résultat. Certains paramètres peuvent être modifiés par la fonction, ils sont alors paramètres données et résultats, ils seront signalés par la présence du mot clé *InOut*. Cette information est nécessaire pour les utilisateurs de la fonction et pour le codage dans un langage de programmation. La liste des paramètres contiendra donc, pour chaque paramètre, son nom, éventuellement le mot clé *InOut* et son type.

Exemple :

On souhaite écrire l'algorithme de la fonction qui insère un élément dans un tableau d'entiers trié par ordre croissant. La valeur retournée par la fonction est le rang où l'insertion a été réalisée. Le tableau passé en paramètre est modifié par la fonction.

```
fonction insérerElémentDansTableauCroissant (t InOut : Tab, n : entier, élément : entier) : entier
        début

        /* recherche de la place où insérer */
        rang ← 0
        trouve ←  faux
        tant que rang < n et non trouve faire
                si t[rang] < élément
                        alors rang ←  rang + 1
                        sinon trouve ←  vrai
                fsi
        ftant

        /* décalage des éléments plus grands que celui à insérer */
        pour j décroissant de n à rang + 1 faire
                t [j] ← t [j -1]
        fpour

        /* insertion de élément */
        t [rang] ← élément

        retourne rang
        fin
```

lexique

- Tab = tableau entier [0.. n], on a choisi de donner un nom au type utilisé, cela est courant pour les types structurés
- t : Tab, tableau d'entiers trié de manière croissante dans lequel on veut insérer un élément
- n : entier, nombre d'éléments du tableau avant insertion
- élément : entier, élément à insérer
- rang : entier, position à laquelle on insérera l'élément, résultat de la fonction
- j : entier, indice d'itération sur le tableau
- trouve : booléen, à vrai lorsque l'élément courant du tableau est plus grand que l'élément à insérer

4 Le type composite

On appelle *type composite* un type qui sert à regrouper les caractéristiques (éventuellement de types différents) d'une variable. Chaque constituant est appelé *champ* et est désigné par un *identificateur de champ*. Le type composite est parfois aussi appelé *structure*, *produit cartésien* ou *enregistrement*. On utilise chaque identificateur de champ comme sélecteur du champ correspondant.

Définition lexicale :

c : <chp1 : type1, ... , chpn : type n>

où *c* est une variable composite dont le type est décrit,
chp1, *chp2*, ..., *chpn* sont les noms des champs de la variable composite *c*,
type1 est le type du champ *chp1*,..., *typen* le type du champ *chpn*.

Sélection de champ :

Sélection du champ chpi de c : *c.chpi*

c.chpi est une variable de type *typei* et peut être utilisé comme toute variable de ce type. On utilise chaque identificateur de champ comme sélecteur du champ correspondant.

Construction d'une variable composite par la liste des valeurs des champs :

c ← (c1, c2, ..., cn) où *c1* : *type1* , *c2* : *type2*, ..., *cn* : *typen*.

La liste des valeurs des champs est donnée dans l'ordre de la description du type composite.

Modification de la valeur d'un champ :

c.chpi ← ci où *ci* est de type *typei*, la valeur *ci* est affectée au champ *chpi* de *c*.

Autres "opérations" :

c ← lire( )

écrire(c)

c1 ← c2 où c1 et c2 sont deux variables composites du même type.

Exemples :

Soit les deux types composites :
Date = <jour : entier , mois : chaîne , année : entier >
Personne = <nom : chaîne , prénom : chaîne , dateNaiss : Date , lieuNaiss : chaîne >

et les variables :
père, fils : Personne,
date : Date.

date.jour désigne une variable de type entier .
père.dateNaiss.année est aussi une variable de type entier.

On peut écrire : date ← (1, "janvier", 1995)
fils ← (père.nom, "Paul", (14, "mars", 1975), "Lyon")
date.jour ← 30

5 Les fichiers séquentiels

Les variables manipulées dans un programme sont placées en mémoire centrale et disparaissent donc à l'issue de l'exécution. Pour pouvoir manipuler des « informations » de taille supérieure à la mémoire centrale ou pour conserver des données après la fin de l'exécution d'un programme, une solution consiste à utiliser des *fichiers séquentiels*.

La communication entre un fichier et un programme se fait par l'intermédiaire de mécanismes d'entrées/sorties ou flux (en anglais *stream*). Un flux est un support de communication de données entre une source émettrice et une destination réceptrice. Ces deux extrémités sont de toute nature, par exemple fichier, mémoire centrale, programme local ou distant.

Un fichier peut être identifié de deux manières, soit par son nom externe ou **nom physique,** soit par sa référence interne ou **nom logique**.

Le nom physique du fichier est celui qui lui a été donné au moment de sa création par son propriétaire. Ce nom, constitué d'une chaîne de caractères (éventuellement, nom du support, et toujours, nom du fichier sur le support), référence le fichier d'une manière permanente. Le nom logique est celui choisi par le programmeur pour son application. Il désigne une variable qui contient des informations propres au fichier telles qu'un repère indiquant à quel emplacement on se trouve dans le fichier. Ce repère peut être appelé *pointeur* ou *tête de lecture/écriture*. La durée de vie du nom logique est liée à celle de l'exécution du programme qui le manipule.

Pour travailler avec un fichier, on doit préalablement le **créer** (on choisit alors le nom physique) ou l'**ouvrir** (on lie alors le nom logique au nom physique du fichier à traiter). Ceci permet ensuite soit de **lire** c'est-à-dire récupérer une ou plusieurs informations, soit d'**écrire** dans le fichier c'est-à-dire enregistrer ou stocker des informations. Après la dernière opération, il faut **fermer** le fichier.

Définition lexicale :

fich : fichier TypeElément, commentaire

où TypeElément est le type des éléments du fichier du nom logique *fich*. Pour un fichier texte, TypeElément est nécessairement caractère.

Ouverture :

fich ← fichierCréer (nomPhysique)

crée le fichier vide *fich* de nom physique *nomPhysique* et positionne le pointeur de fichier au début.

fich ← fichierOuvrir (nomPhysique)

ouvre le fichier de nom physique *nomPhysique*, l'associe à la variable *fich* et positionne le pointeur de fichier sur le premier élément.

Fermeture :

fichierFermer (fich)

ferme le fichier *fich*

Lecture :

élément ← fichierLire (fich)

lit dans le fichier *fich* l'élément désigné par le pointeur de fichier et range la valeur lue dans *élément*, avance le pointeur de fichier d'un élément.

Ecriture :

fichierEcrire (fich, élément)

écrit dans le fichier *fich* l'élément *élément* à la position courante donnée par le pointeur de fichier, avance le pointeur de fichier d'un élément.

Fin de fichier :

fichierFin (fich)

retourne vrai si la fin de fichier est atteinte (c'est-à-dire si le dernier élément lu est la marque de fin de fichier), faux sinon.

Exemple :

On dispose d'un fichier contenant le classement de skieurs à l'issue d'une course de ski. On souhaite écrire une fonction permettant d'imprimer les éléments de ce fichier.

```
algorithme
    fonction imprimerFichierSkieurs (nomPhysique : chaîne)
        début
        fSki ← fichierOuvrir (nomPhysique )
        nom ← fichierLire (fSki)
        tant que non fichierFin (fSki) faire
                écrire (nom)
                nom ← fichierLire (fSki)
        ftant
        fichierFermer (fski)
        fin

lexique
    nomPhysique : chaîne, nom physique du fichier des skieurs
    fSki : fichier chaîne, fichier contenant les noms des skieurs d'après leur ordre d'arrivée
    nom : chaîne, nom du skieur courant
```

Annexe 2

Expression des fonctions logiques attachées à la structure de liste, lorsqu'on utilise une représentation contiguë dans un tableau.

- fonction **supqlis** (lAdmis InOut : Liste(Etudiant))
  début
  /* Il suffit de mettre le nombre d'éléments à jour :*/
  lAdmis.nb ← lAdmis.nb-1
  fin

- fonction **chglis** (lAdmis InOut : Liste(Etudiant), place : entier, étudiant : Etudiant)
  début
  lAdmis.tab[place] ← étudiant
  fin

- fonction **suplis** (lAdmis InOut : Liste(Etudiant), place : entier)
  début
  /* Il faut décaler tous les éléments qui suivent la place p, d'un rang vers le début du tableau :*/
  pour j de p à lAdmis.nb-1 faire
  lAdmis.tab[j] ← lAdmis.tab[j+1]
  fpour
  lAdmis.nb ← lAdmis.nb-1
  fin

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

Annexe 3

Gestion de l'espace libre avec marquage et récupération des places libres

Initialisation de l'espace libre

```
Fonction  initEspaceLibre (xl InOut : ListeChainéeTableau, xTailleTableau : entier)
début
    pour k de 1 à xTailleTableau faire
            xl [k].suc ← -1
    fpour
fin
Lexique
xl : ListeChainéeTableau, tableau
k : entier,
xTailleTableau : entier
```

Recherche d'une place libre pl

```
Fonction  rechercherPlaceLlibre (xl : ListeChainéeTableau, xTailleTableau: entier) : entier
début
    j ← 1
    xTrouvePlaceLibre ←  faux
    tantque (j< xTailleTableau et non xTrouvePlaceLibre)  faire
            xTrouvePlaceLibre ← (xl[j] .suc = -1)
            j ← j + 1
    ftantque
    si  xTrouvePlaceLibre  alors
            xPlaceLibre ← j - 1
    sinon
            xPlaceLibre ← -10
    fsi
    retourne xPlaceLibre
fin

Lexique
xl : ListeChainéeTableau, tableau
xTailleTableau : entier, nombres de places dans xl'
xTrouvePlaceLibre : booléen, « on a trouvé une place libre »
xPlaceLibre : entier, place libre trouvée ou –10 (valeur choisie arbitrairement, elle doit être négative) si plus de place
    libre
j : entier, place courante
```

Restitution de la place libre lors de la suppression de l'élément d'indice i

```
l[i].suc ← -1
```