R101-c
Structures de données

Étienne ANDRÉ
Yolande BELAÏD
Vincent THOMAS

www.loria.science/andre/enseignement/R101c/

Version : 8 décembre 2021

Table des matières

1 Les types abstraits **1**
- 1.1 Définition d'un type abstrait . . . 1
- 1.2 Implantation d'un type abstrait . . . 2
- 1.3 Utilisation du type abstrait . . . 2
- 1.4 Généricité . . . 2

2 Les structures linéaires **3**
- 2.1 Les listes . . . 3
  - 2.1.1 Définition abstraite . . . 3
  - 2.1.2 Description informelle des opérations . . . 4
  - 2.1.3 Parcours de listes . . . 8
  - 2.1.4 Exercices : algorithmes logiques sur les listes . . . 12
- 2.2 Représentation contiguë des listes . . . 13
  - 2.2.1 Représentation contiguë dans un tableau . . . 13
  - 2.2.2 Représentation contiguë dans un fichier séquentiel . . . 15
  - 2.2.3 Conclusion . . . 16
- 2.3 Représentation chaînée des listes . . . 16
  - 2.3.1 Généralités . . . 16
  - 2.3.2 Représentation chaînée dans un tableau ou fichier direct . . . 17
  - 2.3.3 Représentation chaînée à l'aide de pointeurs . . . 24
- 2.4 Les piles et les files . . . 25
  - 2.4.1 Les piles . . . 25
  - 2.4.2 Les files . . . 28
- 2.5 Les listes symétriques . . . 30
  - 2.5.1 Définition abstraite du type liste symétrique . . . 30
  - 2.5.2 Exercice sur les listes symétriques . . . 31
- 2.6 Exercices récapitulatifs . . . 31

3 Préparation SAÉ et entraînement **34**
- 3.1 Préparation SAÉ 1.02 : listes triées . . . 34
- 3.2 Exercice d'entraînement : listes de films . . . 34
  - 3.2.1 Structure de données . . . 34
  - 3.2.2 Algorithmes logiques . . . 35
  - 3.2.3 Algorithmes de programmation . . . 35

A Conventions pour l'écriture des algorithmes **36**
- A.1 Les mots-clés . . . 36
- A.2 Les principales instructions . . . 36
- A.3 Le type tableau . . . 37
- A.4 Les fonctions . . . 37
- A.5 Le type composite . . . 39
  - A.5.1 Définition lexicale . . . 39

A.5.2 Sélection de champ . . . 39
A.5.3 Construction d'une variable composite par la liste des valeurs des champs 39
A.5.4 Modification de la valeur d'un champ . . . 39
A.5.5 Autres « opérations » . . . 39
A.5.6 Exemples . . . 40
A.6 Les fichiers séquentiels . . . 40

B Fonctions logiques pour représentation contiguë dans un tableau 42

C Gestion de l'espace libre avec marquage et récupération des places libres 44
C.1 Initialisation de l'espace libre . . . 44
C.2 Recherche d'une place libre pL . . . 45
C.3 Restitution de la place libre lors de la suppression de l'élément d'indice i . . . 45

D Bilans récapitulatifs 46
D.1 Bilan de la section 2.1 . . . 46

E Crédits 47
E.1 Texte . . . 47
E.2 Images . . . 47

Chapitre 1

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.

À 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 : listes, piles, files. (Par ailleurs, les tables seront vraisemblablement vues par la suite, dans une autre matière.)

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

1.1 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 1

Pour le type abstrait **Ensemble** :

— ensembles définis et utilisés : **Ensemble**, Élément, **booléen** ;

— description fonctionnelle des opérations :

êtreVide	: **Ensemble**(Élément)		→ **booléen**
appartenir	: **Ensemble**(Élément) × Élément		→ **booléen**
ajouter	: **Ensemble**(Élément) × Élément		→
enlever	: **Ensemble**(Élément) × Élément		→
union	: **Ensemble**(Élément) × **Ensemble**(Élément)		→ **Ensemble**(Élément)

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

1.2 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*.

1.3 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.

1.4 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++, Java, OCaml, ...) 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**.

Chapitre 2

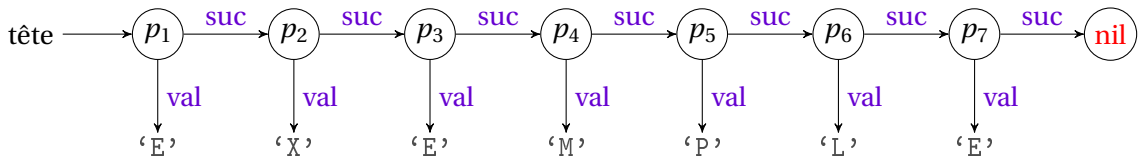
Les structures linéaires

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.

2.1 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 1

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

2.1.1 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), **booléen**.

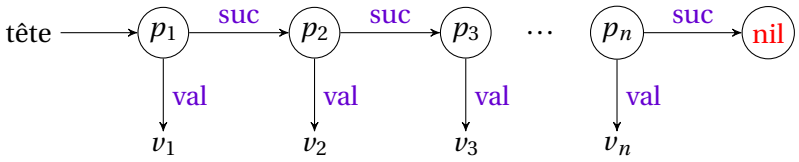
Description fonctionnelle des opérations :

tête	: **Liste**(**Valeur**)	→ **Place**
val	: **Liste**(**Valeur**) × (**Place** \ {nil})	→ **Valeur**
suc	: **Liste**(**Valeur**) × (**Place** \ {nil})	→ **Place**
finliste	: **Liste**(**Valeur**) × **Place**	→ **booléen**
lisvide	:	→ **Liste**(**Valeur**)
adjtlis	: **Liste**(**Valeur**) × **Valeur**	→
suptlis	: **Liste**(**Valeur**) \ {lisvide()}	→
adjqlis	: **Liste**(**Valeur**) × **Valeur**	→
supqlis	: **Liste**(**Valeur**) \ {lisvide()}	→
adjlis	: (**Liste**(**Valeur**) \ {lisvide()}) × (**Place** \ {nil}) × **Valeur**	→
suplis	: (**Liste**(**Valeur**) \ {lisvide()}) × (**Place** \ {nil})	→
chglis	: (**Liste**(**Valeur**) \ {lisvide()}) × (**Place** \ {nil}) × **Valeur**	→

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

2.1.2 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 v_1, v_2, v_3, v_n. On peut représenter la liste ℓ par le schéma suivant :



L'ensemble des places est $\{p_1, p_2, p_3, \cdots, p_n, \text{nil}\}$.

2.1.2.1 Les opérations de parcours

tête La fonction tête désigne la place du premier élément de la liste. Par exemple, dans l'exemple ci-dessus, tête(ℓ) rend p_1.

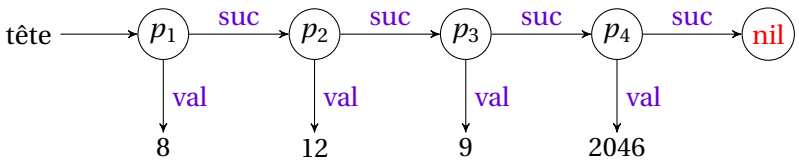
val La fonction val désigne la valeur associée à une place. Par exemple, val(ℓ, p_3) rend v_3.

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, p_2) rend p_3.

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

2.1.2.2 Exemples d'utilisations des opérations de parcours

Exercice 2.1. Soit la liste *lEntier* schématisée ci-après :



Question 1 : Évaluer les expressions suivantes :

1. val(*lEntier*, tête(*lEntier*))
2. val(*lEntier*, suc(*lEntier*, suc(*lEntier*, tête(*lEntier*))))
3. finliste(*lEntier*, suc(*lEntier*, suc(*lEntier*, suc(*lEntier*, tête(*lEntier*)))))
4. finliste(*lEntier*, suc(*lEntier*, suc(*lEntier*, suc(*lEntier*, suc(*lEntier*, tête(*lEntier*))))))

Question 2 : Quelle est la valeur affichée par l'algorithme logique suivant, si l'on lit la liste *lEntier* définie ci-dessus ?

Algorithme 1 :

```
1 lEntier ← lire()
2 p ← tête(lEntier)
  /* on suppose ici que la liste contient au moins 4 éléments       */
3 pour i de 1 à 3 faire
4 |  p ← suc(lEntier, p)
5 fin pour
6 écrire(val(lEntier, p))
```

Lexique

p	: **Place**	i^e place de la liste
i	: **entier**	variable d'itération
lEntier	: **Liste**(**entier**)	liste donnée

Exercice 2.2. Soit la liste *lÉliminés* contenant les nom, prénom et note des candidat·e·s ayant échoué à l'épreuve du permis de conduire. Écrire 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.

Correction

Algorithme 2 :

```
1  Algorithme logique
2  fonction imprimerPremierEliminé(lÉliminés : Liste(Candidat))
3  début
4  |  placeTête ← tête(lÉliminés)
5  |  si finliste(lÉliminés, placeTête) alors
6  |  |  écrire(« pas d'échec »)
7  |  sinon
8  |  |  candidatÉliminé ← val(lÉliminés, placeTête)
9  |  |  écrire(candidatÉliminé.nom, candidatÉliminé.prénom)
10 |  fin si
11 fin
```

Lexique

Candidat = ⟨*nom* : **chaîne**, *prénom* : **chaîne**, *note* : **entier**⟩

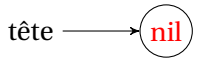
lÉliminés	: **Liste**(**Candidat**)	liste des candidat·e·s éliminé·e·s
placeTête	: **Place**	place du premier élément de la liste s'il existe
candidatÉliminé	: **Candidat**	premier·e candidat·e éliminé·e (si existe)

Notons que le type **Candidat** est un type composite [1].

1. Le type composite est rappelé en annexe A.5.

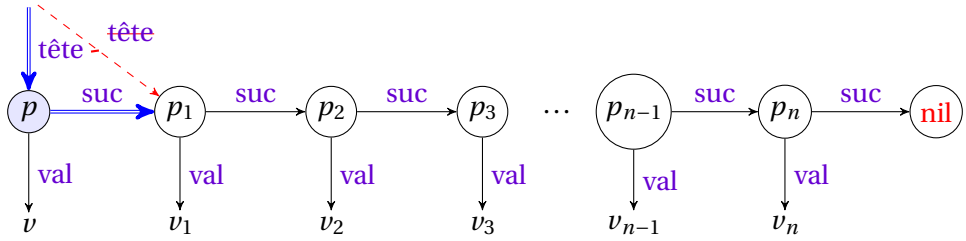
2.1.2.3 Les opérations de construction et de mises à jour

lisvide Construction d'une liste vide c'est-à-dire d'une liste ne contenant aucun élément.
$\ell \leftarrow$ lisvide() : création de la liste vide ℓ :



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

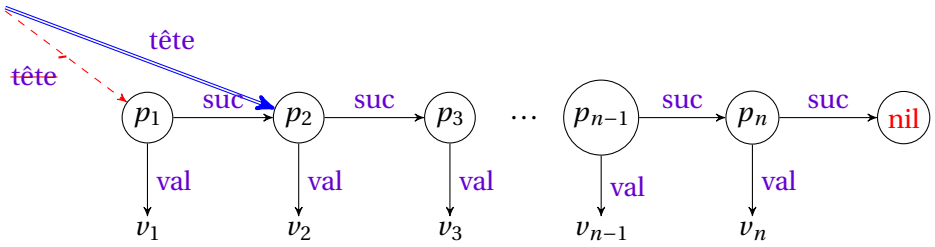
adjtlis Adjonction en tête de la liste ; c'est le cas où l'on insère un élément avant tous les autres ; adjtlis(ℓ, v) ajoute en tête de ℓ un élément de valeur v :



Sur le schéma :

1. nous dessinons la liste initiale,
2. nous barrons (en rouge) les liens qui disparaissent lors de la modification,
3. nous notons en double (en bleu) ceux qui apparaissent.

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

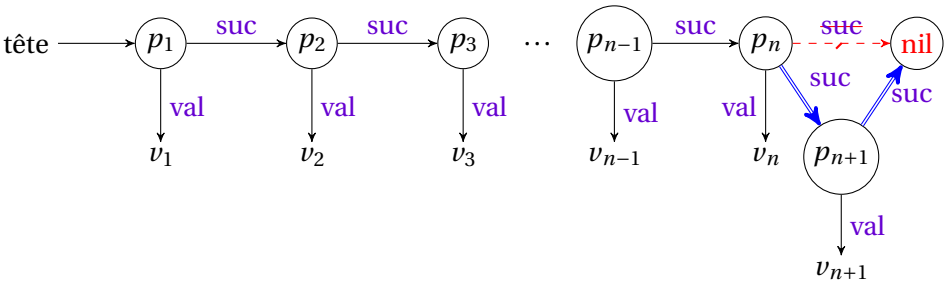


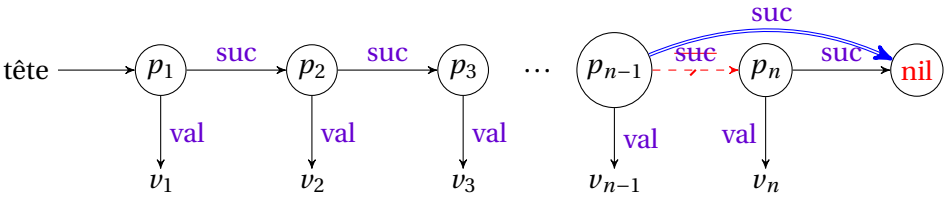
adjqlis adjonction en queue de liste ; adjqlis(ℓ, v) : adjonction d'un élément de valeur v en queue de la liste ℓ :

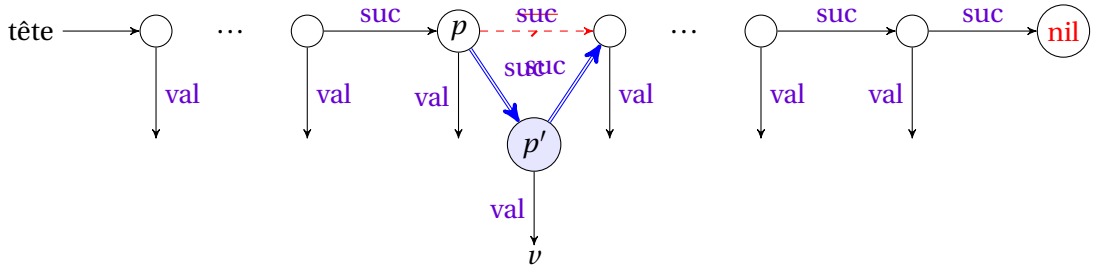
supqlis suppression en queue ; supqlis(ℓ) : c'est le cas où le dernier élément de la liste est enlevé :

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. Dans ce qui suit, 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.





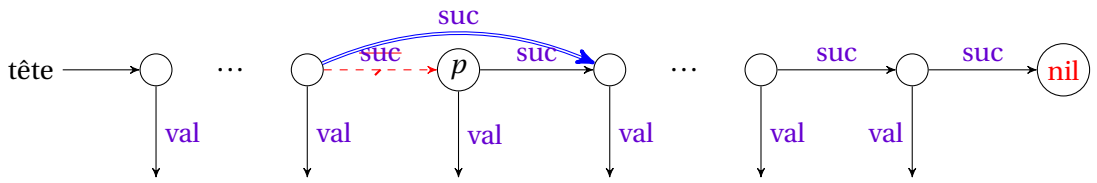


Remarque 2

adjlis ne permet pas de faire une adjonction en tête de liste.

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.

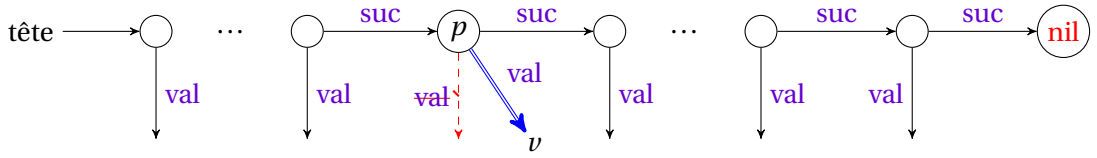


Remarque 3

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, et remplacement par la nouvelle valeur v.



Remarque importante 1

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.

2.1.2.4 Exemple de création de liste

Prenons l'exemple de la liste d'admission des étudiant·e·s 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

1. *nombreDossiers*
2. *nombreDossiers* fois (*nom* ET *note*)

Résultats liste d'admission

```
1  Algorithme logique
2  fonction lÉtudSaisir() : Liste(Étudiant)
3  début
4      lAdmis ← lisvide()
5      nombreDossiers ← lire()
6      pour i de 1 à nombreDossiers faire
7          étudiant ← lire()
8          adjqlis(lAdmis, étudiant)
9      fin pour
10     retourner lAdmis
11 fin
```

Lexique

Étudiant = ⟨*nom* : **chaîne**, *note* : **réel**⟩

lAdmis	: **Liste**(Étudiant)	liste d'admission
nombreDossiers	: **entier**	nombre de dossiers
etudiant	: Étudiant	nom et note contenus dans le i^e dossier
i	: **entier**	indice d'itération sur les dossiers

2.1.3 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.

2.1.3.1 Exemples de parcours complet

Exemple 2

Soit une liste *lEntier* d'entiers. Calculer la moyenne des éléments de cette liste.

Correction

```
1  Algorithme logique
2  fonction entierCalculerMoyenne(lEntier : Liste(entier)) : réel
3  début
4      somme ← 0
5      nombre ← 0
6      p ← tête(lEntier)
7      tant que non finliste(lEntier, p) faire
8          valeur ← val(lEntier, p)
9          somme ← somme + valeur
10         nombre ← nombre + 1
11         p ← suc(lEntier, p)
12     fin tq
13     si nombre ≠ 0 alors
14         moyenne ← somme / nombre
15     sinon
16         moyenne ← 0
17     fin si
18     retourner moyenne
19 fin
```

Lexique

lEntier	: **Liste**(**entier**)	liste des entiers
moyenne	: **réel**	moyenne des éléments
p	: **Place**	place courante
somme	: **entier**	somme des premiers entiers
nombre	: **entier**	nombre courant de valeurs lues
valeur	: **entier**	valeur courante de la liste *lEntier*

Exemple 3

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

Correction

```
1  Algorithme logique
2  fonction lpersonneImprimerMajeure(lPersonne :
     Liste(Personne), annéeCourante : entier)
3  début
4      p ← tête(lPersonne)
5      tant que non finliste(lPersonne, p) faire
6          personne ← val(lPersonne, p)
7          si annéeCourante − personne.naissance ≥ 18 alors
8              écrire(personne.nom, personne.adresse)
9          fin si
10         p ← suc(lPersonne, p)
11     fin tq
12 fin
```

Lexique

Personne = ⟨*nom* : **chaîne**, *adresse* : **chaîne**, *naissance* : **entier**⟩

lPersonne	: **Liste**(**Personne**)	liste des personnes de la commune
p	: **Place**	place courante
annéeCourante	: **entier**	année en cours
personne	: **Personne**	personne courante de la liste

Remarque 4

Des parcours complets de listes seront réalisés dans les exercices 2.3, 2.4 et 2.8 (parmi d'autres).

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

Reprenons l'exemple 3 ci-dessus en supposant que les personnes sont classées *par année de naissance croissante* (année de naissance croissante signifie donc âge décroissant). Ainsi on arrête le parcours dès que l'on rencontre une personne mineure.

```
1  Algorithme logique
2  fonction
     lpersonneImprimerMajeure(lPersonne : Liste(Personne), annéeCourante : entier)
3  début
4      p ← tête(lPersonne)
5      mineur ← faux
6      tant que non finliste(lPersonne, p) et non mineur faire
7          personne ← val(lPersonne, p)
8          si annéeCourante − personne.naissance < 18 alors
9              mineur ← vrai
10         sinon
11             écrire(personne.nom, personne.adresse)
12             p ← suc(lPersonne, p)
13         fin si
14     fin tq
15 fin
```

Lexique

Personne = ⟨*nom* : **chaîne**, *adresse* : **chaîne**, *naissance* : **entier**⟩

lPersonne	: **Liste**(**Personne**)	liste des personnes de la commune
p	: **Place**	place courante
annéeCourante	: **entier**	année en cours
personne	: **Personne**	personne courante de la liste
mineur	: **booléen**	à vrai quand la personne courante est mineure

Remarque 5

Des parcours de listes jusqu'à une condition d'arrêt seront réalisés à l'exercice 2.5 (parmi d'autres).

2.1.3.3 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 :

C_1 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 « Marie »).

C_2 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 « Martin »).

C_3 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 7e étudiant).

Remarque 6

Des parcours de listes à partir d'un certain élément seront réalisés à l'exercice 2.6 (parmi d'autres).

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

```
1  ...
2  [p ← tête(ℓ)]
3  [trouvé ← faux]
4  tant que non finliste(ℓ, p) [et non trouvé] faire
5  |  valeur ← val(ℓ, p)
6  |  ...
7  |  si [condition sur valeur] alors
8  |  |  [trouvé ← vrai]
9  |  sinon
10 |  |  p ← suc(ℓ, p)
11 |  fin si
12 fin tq
```

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

2.1.4 Exercices : algorithmes logiques sur les listes

Exercice 2.3 (cardinalité d'une liste)**.** Donner l'algorithme logique de la fonction *calculerNb* qui prend en paramètre une liste, et retourne le nombre d'éléments de cette liste.

fonction *calculerNb*(ℓ : **Liste**(**Valeur**)) : **entier**

Exercice 2.4 (liste de températures)**.** On souhaite créer une liste de températures et calculer la moyenne des températures d'une liste. Écrire les algorithmes logiques des fonctions suivantes :

Question 1 : **fonction** *lTempSaisir*() : **Liste**(**réel**)
saisit un nombre de températures puis les températures et les range dans une liste

> **Remarque 7**
>
> Dans cet exercice (et dans la suite), on lit d'abord le nombre de températures *nbTemp*, puis on saisit *nbTemp* fois une température. Une autre méthode serait de lire directement les températures une par une, jusqu'à ce que l'on lise une température avec une valeur spéciale (par exemple −1000) indiquant la fin de la lecture. Il faudrait alors utiliser une boucle **tant que** au lieu d'une boucle **pour**.

Question 2 : **fonction** *lTempMoyenne*(*lTemp* : **Liste**(**réel**)) : **réel**
calcule la moyenne des températures de la liste *lTemp*

Exercice 2.5 (bibliothèque)**.** Soit la liste des ouvrages (auteur-titre) d'une bibliothèque classée par ordre alphabétique des noms d'auteurs ou d'autrices. Écrire l'algorithme logique de la fonction *lLivreCréerListeTitre* qui crée la liste de tous les titres d'un auteur ou d'une autrice donné·e.

Exercice 2.6 (course cycliste)**.** Soit *lCycliste*, la liste des noms des cyclistes d'une course dans l'ordre d'arrivée.

Question 1 : Écrire l'algorithme logique de la fonction *lCyclisteImprimerAprès10* qui permet d'imprimer les noms des cyclistes se trouvant (strictement) après la dixième place, c'est-à-dire à partir de la onzième place.

Question 2 : Écrire l'algorithme logique de la fonction *lCyclisteImprimer3Après* qui permet d'imprimer les noms des 3 cyclistes arrivé·e·s après celle ou celui dont le nom est donné en paramètre, ou un message indiquant qu'aucun·e cycliste de ce nom n'est dans la liste.

Exercice 2.7 (éléments en nombre impair)**.** Soit une suite de valeurs entières (que l'on peut supposer être 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. Écrire l'algorithme logique de la fonction *lEntierCréer* réalisant cette construction. On lit le nombre de valeurs puis chaque valeur.

Exemple Supposons que l'on lit (avec lire()) le nombre de valeurs « 8 », puis que l'on lit (toujours avec lire()) les valeurs suivantes : 1, 3, 2, 2, 4, 1, 3, 3. Alors la liste retournée par l'algorithme est $[3,4]$.

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, puisque adjlis prend en paramètre la place *après* laquelle on souhaite insérer un élément.

Exercice 2.8 (interclassement de listes)**.** Écrire l'algorithme logique de la fonction *lEntierInterclasser* qui interclasse deux listes d'entiers triées par ordre croissant.

Exemple Soit $\ell_1 = [1,3,5,2022,2046]$. Soit $\ell_2 = [4,8,17,1980]$. Alors *lEntierInterclasser*(ℓ_1,ℓ_2) va retourner la liste $[1,3,4,5,8,17,1980,2022,2046]$.

Principe

— on compare les premiers éléments des deux 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

— ...

Il pourra être utile de factoriser une partie de l'algorithme à l'aide d'une fonction annexe.

2.2 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.

2.2.1 Représentation contiguë dans un tableau

Les éléments de la liste sont représentés dans un tableau [2] contenant les valeurs rangées successivement à partir du début. Une place dans la liste est *représentée* par un indice d'accès dans le tableau.

$$\text{place } p \iff \text{indice } p$$

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).

2. Le type tableau est rappelé en annexe A.3.

Remarque 8

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 9

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.

2.2.1.1 Exemple

Reprenons, comme exemple, la liste des admissions (section 2.1.2.4). 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 [3] suivant :

ListeÉtudiant = ⟨*tab* : **tableau Étudiant**[1..*MAXNBETUDIANT*], *nb* : **entier**⟩

(rappel : **Étudiant** = ⟨*nom* : **chaîne**, *note* : **réel**⟩)

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

Exemple de représentation de la liste *lAdmis* :

lAdmis.tab =

	nom	note
1	Irène	18
2	Ibrahim	15
3	Agathe	14
4	Dylan	13
5		
6		

lAdmis.nb = 4

Lorsque l'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.

```
1 fonction tête(lAdmis : ListeÉtudiant) : entier
2 début
3 |   retourner 1
4 fin
```

```
1 fonction val(lAdmis : ListeÉtudiant, p : entier) : Étudiant
2 début
3 |   retourner lAdmis.tab[p]
4 fin
```

3. Le type composite est rappelé en annexe A.5.

```
1 fonction suc(lAdmis : ListeÉtudiant, p : entier) : entier
2 début
3 |   retourner p + 1
4 fin
```

```
1 fonction finliste(lAdmis : ListeÉtudiant, p : entier) : booléen
2 début
3 |   fin ← (p = lAdmis.nb + 1)
4 |   retourner fin
5 fin
```

```
1 fonction lisvide() : ListeÉtudiant
2 début
3 |   lAdmis.nb ← 0
4 |   retourner lAdmis
5 fin
```

```
1 fonction adjqlis(lAdmis InOut : ListeÉtudiant, étudiant : Étudiant)
2 début
3 |   lAdmis.nb ← lAdmis.nb + 1
4 |   lAdmis.tab[lAdmis.nb] ← étudiant
5 fin
```

2.2.1.2 Exercices

Exercice 2.9. Écrire les algorithmes de programmation des fonctions suptlis et adjlis.

Les autres fonctions (supqlis, chglis, suplis, adjtlis) sont données en annexe B.

Exercice 2.10 (cardinalité d'une liste)**.** Dans l'exercice 2.3, nous avions rédigé l'algorithme logique de la fonction *calculerNb* qui prend en paramètre une liste, et retourne le nombre d'éléments de cette liste.

Donner l'algorithme de programmation de la fonction *calculerNb* qui prend en paramètre une liste, et retourne le nombre d'éléments de cette liste, dans le cadre d'une représentation contiguë dans un tableau.

2.2.2 Représentation contiguë dans un fichier séquentiel

Les éléments de la liste sont stockés dans un fichier séquentiel [4] conservé en mémoire secondaire (sur disque dur, bande magnétique...).

Reprenons comme exemple la liste des admissions (section 2.1.2.4). Elle est représentée par : *lAdmis* : **fichier Étudiant**. 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.

4. Les fichiers séquentiels sont rappelés en annexe A.6.

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.

2.2.3 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.

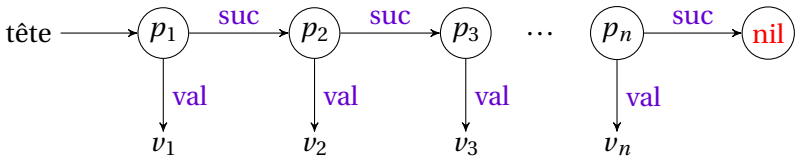
2.3 Représentation chaînée des listes

2.3.1 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).

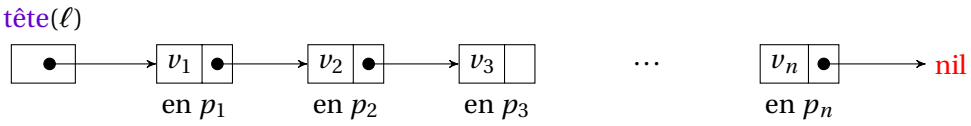
Dans la représentation contiguë, la fonction suc était représentée *implicitement*. Nous allons maintenant la représenter *explicitement* : chaque élément va comporter l'indication de son successeur. Nous verrons deux façons différentes de le faire. Chaque élément est un couple formé de sa valeur et de l'indication de la place du successeur. 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 10



La flèche vers une place p est appelée « indicateur de place » ; nil doit être considéré comme une place fictive et son indicateur.

2.3.2 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.

Les éléments de la liste sont représentés dans 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. La liste sera représentée par un couple contenant ce tableau et l'indice du premier élément. Souvent, nil est représenté par 0.

Exemple 4

La liste chaînée de la figure 2.1 peut être représentée de manière chaînée par une variable composite à 2 champs :

— *tab* : le tableau qui mémorise les couples (valeur, successeur),

— *tête* : l'indice du 1er élément de la liste dans le tableau.

$\ell.tête = 3$

$\ell.tab =$

indices	1	2	3	4	5	6	7		
val	'C'	'T'	'I'		'U'			⋯	
suc	0	1	5		2			⋯	

où une case grisée ( ) représente une place libre.

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.

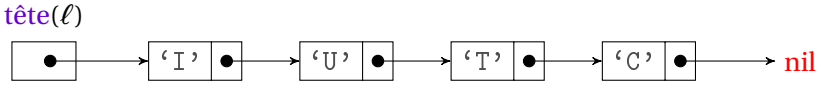


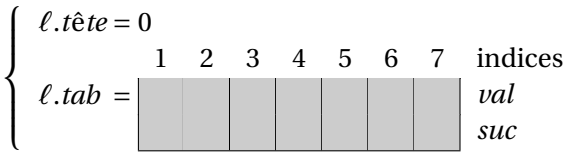
FIGURE 2.1 – Une liste chaînée

Exercice 2.11 (jouons à la machine)**.** Simuler l'évolution de la liste d'admissions lors de sa création à partir des données suivantes (l'ajout s'effectuant de haut en bas), sachant que cette liste est représentée de manière chaînée dans le tableau *lAdmis* :

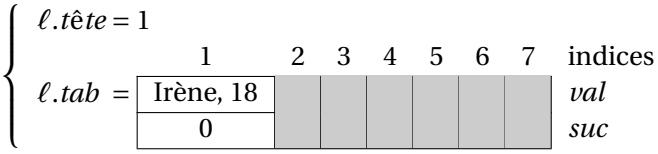
Nom	Note
Irène	18
Agathe	14
Ibrahim	15
Dylan	13
Lucas	19

Rappel : les éléments doivent être rangés dans la liste par note décroissante.

Initialement



Après le 1er ajout



2.3.2.1 Expression des fonctions logiques dans le cas d'une représentation chaînée dans 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** = $\langle$*tête*: **entier**, *tab*: **tableau**$\langle$*val*: **Élément**, *suc*: **entier**$\rangle[1..b^{sup}]\rangle$

où **Élément** est le type des éléments de la liste.

Type

p est une place $\iff$ p est un indice (**entier**)

suc(ℓ, p)

```
1 fonction suc(ℓ : ListeChainéeTableau, p : entier) : entier
2 début
3 |   retourner ℓ.tab[p].suc
4 fin
```

val(ℓ, p)

```
1 fonction val(ℓ : ListeChainéeTableau, p : entier) : Élément
2 début
3 |   retourner ℓ.tab[p].val
4 fin
```

tête(ℓ)

```
1 fonction tête(ℓ : ListeChainéeTableau) : entier
2 début
3 |   retourner ℓ.tête
4 fin
```

finliste(ℓ, p)

```
1 fonction finliste(ℓ : ListeChainéeTableau, p : entier) : booléen
2 début
3 |   retourner p = 0
4 fin
```

$\ell \leftarrow$ **lisvide**()

```
1 fonction lisvide() : ListeChainéeTableau
2 début
3 |   ℓ.tête ← 0
  |   /* Initialisation de l'espace libre                    */
4 |   retourner ℓ
5 fin
```

La partie « initialisation de l'espace libre » sera discutée plus tard dans le cours.

adjtlis(ℓ, v)

```
1 fonction adjtlis(ℓ InOut : ListeChainéeTableau, v : Élément)
2 début
  |   /* recherche d'une place libre pL                     */
3 |   ℓ.tab[pL].val ← v
4 |   ℓ.tab[pL].suc ← ℓ.tête
5 |   ℓ.tête ← pL
6 fin
```

La partie « recherche d'une place libre » sera discutée plus tard dans le cours.

suptlis(ℓ)

```
1 fonction suptlis(ℓ InOut : ListeChainéeTableau)
2 début
3 |   ℓ.tête ← ℓ.tab[ℓ.tête].suc
  |   /* restitution de la place libre                       */
4 fin
```

La partie « restitution de la place libre » sera discutée plus tard dans le cours.

adjqlis(ℓ, v)

```
1  fonction adjqlis(ℓ InOut : ListeChainéeTableau, v : Élément)
2  début
       /* recherche d'une place libre pL                              */
3      ℓ.tab[pL].val ← v
4      ℓ.tab[pL].suc ← 0
5      si ℓ.tête = 0 alors /* la liste était vide                     */
6
7          ℓ.tête ← pL
8      sinon /* la liste n'était pas vide                             */
9
10         p ← ℓ.tête
           /* On parcourt la liste jusqu'à la dernière place          */
11         tant que ℓ.tab[p].suc ≠ 0 faire
12             p ← ℓ.tab[p].suc
13         fin tq
           /* p : place après laquelle se fait l'adjonction           */
14         ℓ.tab[p].suc ← pL
15     fin si
16 fin
```

Exercice 2.12. Écrire les algorithmes de programmation des autres fonctions (supqlis, adjlis, suplis et chglis).

2.3.2.2 Exercice

Exercice 2.13 (cardinalité d'une liste)**.** Dans l'exercice 2.3, nous avions rédigé l'algorithme logique de la fonction *calculerNb* qui prend en paramètre une liste, et retourne le nombre d'éléments de cette liste.

Donner l'algorithme de programmation de la fonction *calculerNb* qui prend en paramètre une liste, et retourne le nombre d'éléments de cette liste, dans le cadre d'une représentation chaînée dans un tableau.

2.3.2.3 Gestion de l'espace libre sans récupération des places libérées

Motivation Les cases grisées des figures peuvent être trompeuse. En réalité, par défaut, il n'y a aucun moyen « simple » de savoir si une case est libre. Par exemple, dans la liste ci-dessous, quelle est la première case libre ?

$\ell.tête = 4$

$\ell.tab =$

	1	2	3	4	5	6	7	8	9	10	11	
	'L'	'A'	'E'	'A'	'R'	'O'	'G'	'W'	'B'	'U'	'L'	*val*
	7	6	9	1	10	0	6	0	8	2	4	*suc*

En réalité, il n'y a aucun moyen de le savoir… sauf à parcourir la liste depuis la tête, et appliquer itérativement l'opération suc jusqu'à atteindre la fin de la liste. Si une case du tableau n'apparaît pas dans ce parcours… elle est libre !

Mais il est en général très fastidieux de faire cette opération de parcours à *chaque* recherche d'une place libre.

Dans ce qui suit, nous allons donc voir des méthodes de gestion de l'espace libre plus efficaces.

Gestion sans récupération des places libérées Dans ce mode de gestion de l'espace libre, 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 pL 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 un nouveau champ, noté *libre*, de la liste. La liste sera alors représentée par une variable composite à 3 champs.

Exemple 5

$\ell.tête = 3$
$\ell.libre = 6$

$\ell.tab =$

	1	2	3	4	5	**6**	7				
	'C'	'T'	'I'		'U'					⋯	
val											
suc	0	1	5		2					⋯	

pL

Voyons comment se modifient les expressions des fonctions logiques :

Initialisation de l'espace libre

```
1 ℓ.libre ← 1
```

Recherche d'une place pL libre

```
1 pL ← ℓ.libre
2 ℓ.libre ← ℓ.libre + 1
```

Remarque 11

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.

Exemple 6

Considérons la liste suivante, où le marqueur de place libre a atteint la fin du tableau :

$\ell.tête = 3$
$\ell.libre = 12$

$\ell.tab =$

	1	2	3	4	5	6	7	8	9	10	11
		'N'	'N'		'A'				'C'		'Y'
val											
suc		9	5		2				11		0

pL

Après application du ramasse-miettes, on obtient :

$\ell.tête = 2$
$\ell.libre = 6$

	1	2	3	4	5	6	7	8	9	10	11	
$\ell.tab$ =	'N'	'N'	'A'	'C'	'Y'							*val*
	4	3	1	5	0							*suc*
					pL							

2.3.2.4 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 que l'on supprime un élément, on indique que la place qu'il occupait est libre en mettant -1 dans le champ *suc*. Chaque fois que l'on veut ajouter un élément, on cherche dans le tableau une case dont le champ *suc* est marqué à -1.

Exemple 7

$\ell.tête = 3$

	1	2	3	4	5	6	7				
$\ell.tab$ =	'C'	'T'	'I'		'U'					⋯		*val*
	0	1	5	-1	2	-1	-1	-1	-1	⋯	-1	*suc*

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

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 que l'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 : ℓ_{occup}.

2.3.2.5 Gestion de l'espace libre à l'aide d'une liste (« liste libre »)

Le tableau $\ell.tab$ contient deux listes (dans le même tableau) :

1. la liste sur laquelle on travaille (liste de travail) ;
2. la liste des places inoccupées (dite *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. Lorsque l'on supprime un élément dans la liste de travail, on ajoute sa place dans la liste libre (en tête).

La *tête de la liste libre* peut être mémorisée dans un nouveau champ, noté *tLibre*, de la liste. La liste sera alors représentée par une variable composite à 3 champs.

Nous retiendrons cette hypothèse.

Exemple 8

$\ell.tLibre = 4$
$\ell.tête = 3$

	1	2	3	4	5	6	7	8	9	10	11	indices
$\ell.tab =$	'C'	'T'	'I'		'U'							*val*
	0	1	5	6	2	7	8	9	10	11	0	*suc*

Remarque 12

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

Voici comment compléter les expressions des fonctions logiques :

Déclaration

$$\ell : \langle tLibre : \textbf{entier}, tête : \textbf{entier}, tab : \textbf{tableau}\langle val : \textbf{Élément}, suc : \textbf{entier}\rangle[1..b^{sup}]\rangle$$

où **Élément** est le type des éléments de la liste.

Initialisation de l'espace libre

```
1 ℓ.tLibre ← 1
2 pour i de 1 à b^sup − 1 faire
3 |   ℓ.tab[i].suc ← i + 1
4 fin pour
5 ℓ.tab[b^sup].suc ← 0
```

Exemple 9

$\ell.tLibre = 1$
$\ell.tête = 0$

	1	2	3	4	5	6	7	8	9	10	11	indices
$\ell.tab =$												*val*
	2	3	4	5	6	7	8	9	10	11	0	*suc*

Recherche d'une place libre pL (en tête) (on suppose qu'il y en a)

```
1 pL ← ℓ.tLibre
2 ℓ.tLibre ← ℓ.tab[pL].suc
```

Exemple 10

Supposons la liste suivante :

$$\begin{cases} \ell.tLibre = 4 \\ \ell.tête = 3 \\ \ell.tab = \end{cases}$$

	1	2	3	4	5	6	7	8	9	10	11	indices
	'C'	'T'	'I'		'U'							*val*
	0	1	5	6	2	7	8	9	10	11	0	*suc*

La fonction de recherche d'une place libre pL retourne « 4 », et la liste devient :

$$\begin{cases} \ell.tLibre = 6 \\ \ell.tête = 3 \\ \ell.tab = \end{cases}$$

	1	2	3	4	5	6	7	8	9	10	11	indices
	'C'	'T'	'I'		'U'							*val*
	0	1	5	6	2	7	8	9	10	11	0	*suc*

Restitution d'une place libre pL (en tête)

```
1 ℓ.tab[pL].suc ← ℓ.tLibre
2 ℓ.tLibre ← pL
```

Exemple 11

Supposons la liste suivante :

$$\begin{cases} \ell.tLibre = 4 \\ \ell.tête = 3 \\ \ell.tab = \end{cases}$$

	1	2	3	4	5	6	7	8	9	10	11	indices
	'C'	'T'	'I'		'U'							*val*
	0	0	5	6	2	7	8	9	10	11	0	*suc*

Si l'on libère la place 1, la liste devient :

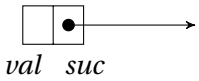
$$\begin{cases} \ell.tLibre = 1 \\ \ell.tête = 3 \\ \ell.tab = \end{cases}$$

	1	2	3	4	5	6	7	8	9	10	11	indices
	'C'	'T'	'I'		'U'							*val*
	4	0	5	6	2	7	8	9	10	11	0	*suc*

2.3.3 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, successeur) 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 dans le langage C.

2.4 Les piles et les files

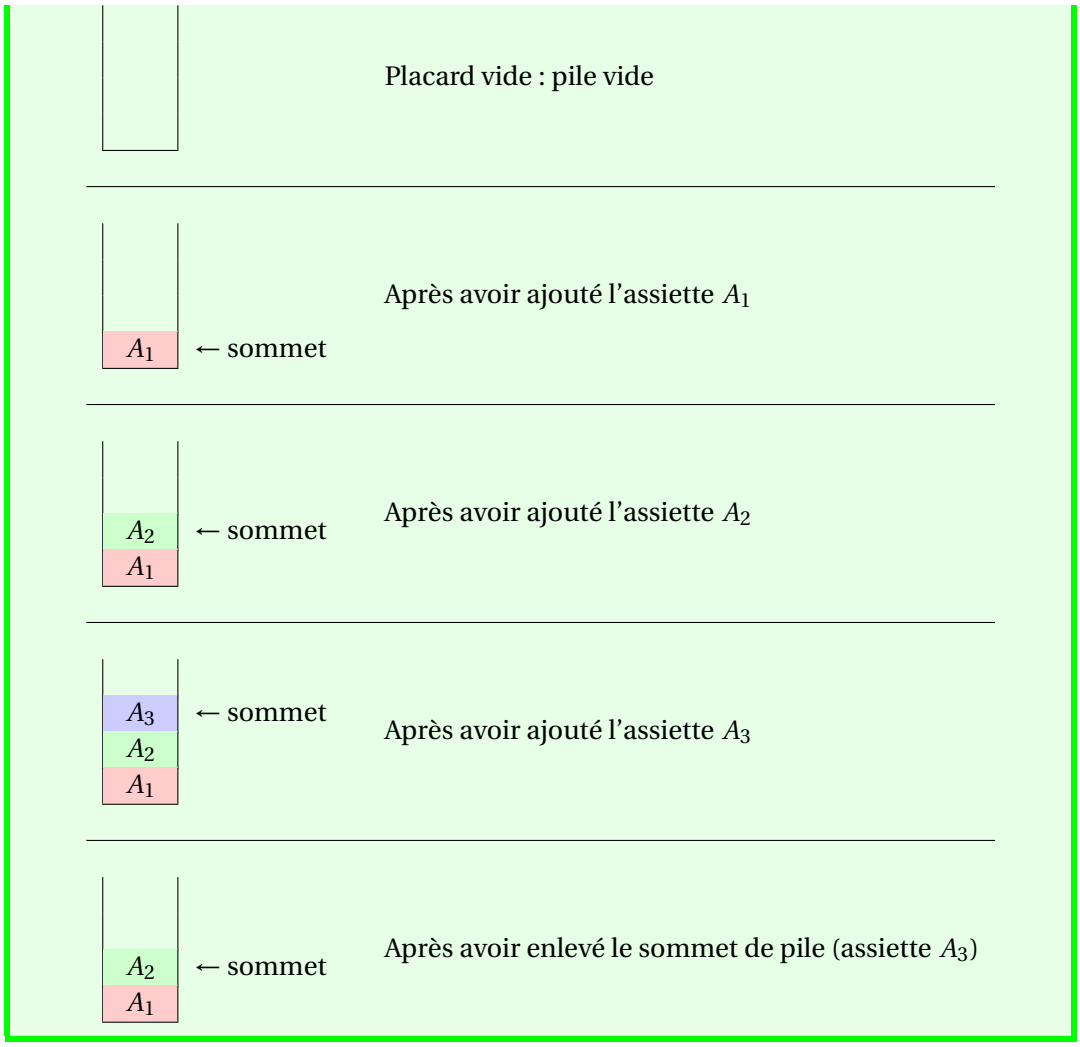
Pour beaucoup d'applications, les seules opérations à effectuer sur les listes sont des insertions et des suppressions aux extrémités.

2.4.1 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 que l'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 12: la pile d'assiettes

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



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).

Notons qu'accéder au sommet (sommet) n'ôte pas ledit élément de la pile. Pour cela, il faut utiliser dépiler.

2.4.1.1 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**) × **Valeur** →

dépiler : **Pile**(**Valeur**) \ {pileVide()} →

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

2.4.1.2 Utilité

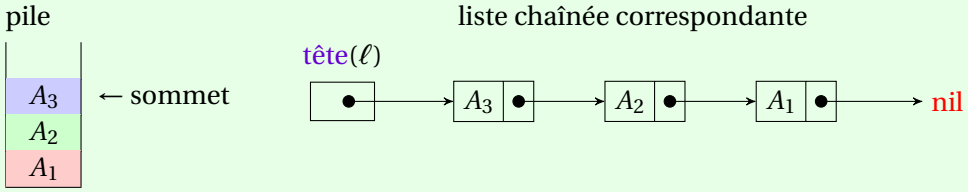
Les piles sont des structures fondamentales, et leur emploi dans les programmes informatiques est très fréquent. Le mécanisme d'appel des sous-programmes suit ce modèle de pile. Les logiciels qui proposent une fonction « Annuler » se servent également d'une pile pour défaire, en ordre inverse, les dernières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

2.4.1.3 Représentation

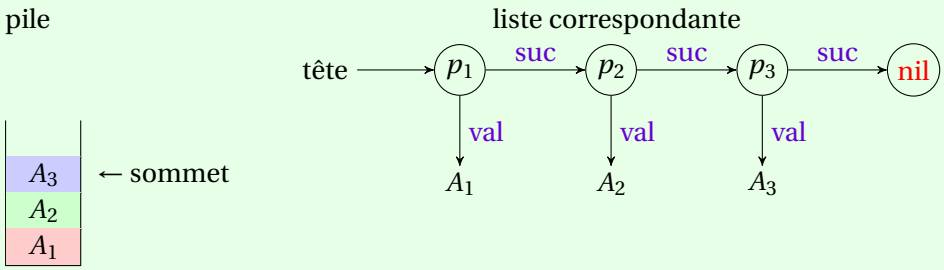
On peut utiliser pour implémenter les piles toutes les représentations étudiées pour les listes. Les opérations empiler et dépiler travailleront soit sur la tête de liste (dans le cas des représentations chaînées), soit sur la queue de liste (dans le cas des représentations contiguës) pour limiter la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

Exemple 13

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 2.14 (expressions postfixées)**.** 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 lu 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 : Écrire l'algorithme logique de la fonction d'évaluation (que l'on nommera *ExpPostfEvaluer*) d'une expression postfixée supposée syntaxiquement correcte. On suppose disposer d'une fonction *chaineEntierConvertir* qui convertit en **entier** une chaîne de caractères représentant un entier. Les opérateurs et les opérandes sont séparés par des blancs et l'expression est suivie d'un « . ».

Donc l'exemple précédent serait lu ainsi :

```
7 12 + 5 3 - + .
```

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 :

$$\textbf{PileEntier} = \langle tab : \textbf{tableau entier}[1..MAXTAB], sommet : \textbf{entier} \rangle$$

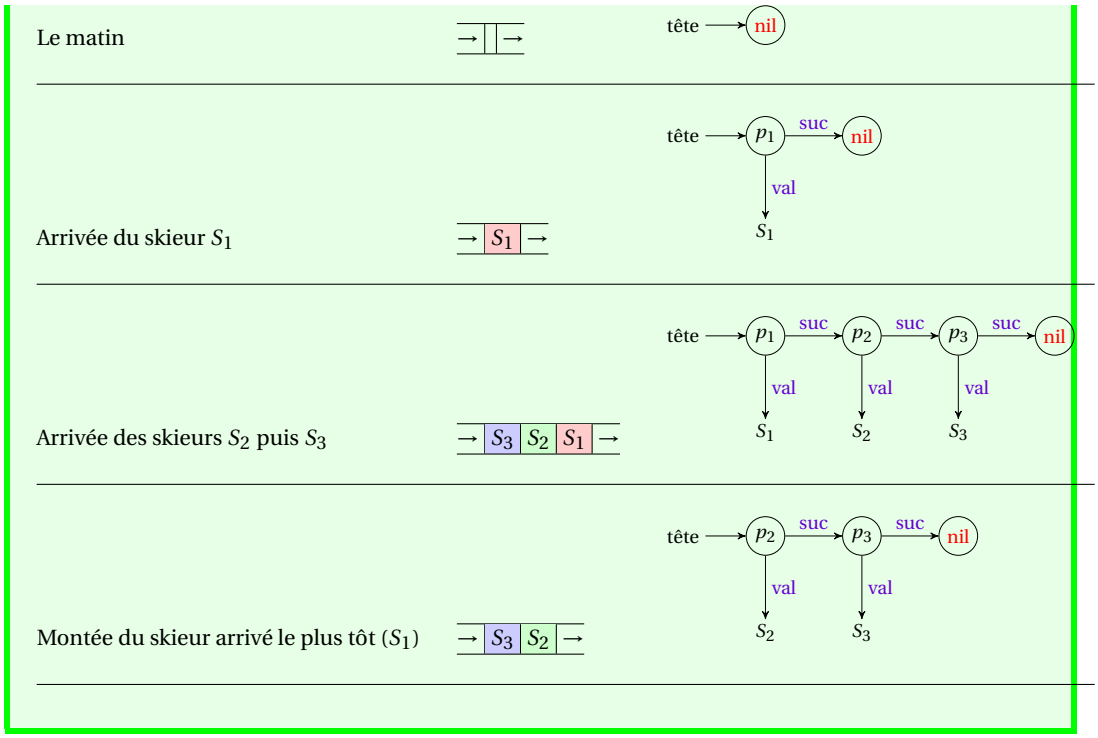
On suppose que la taille du tableau est suffisante et donnée par la constante *MAXTAB*. Écrire les algorithmes de programmation des opérations sur les piles.

2.4.2 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 que l'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 14: file d'attente, par exemple queue (disciplinée) au téléski

On considère des skieurs faisant la queue pour un télésiège. Voici une évolution possible :



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).

2.4.2.1 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**) × **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 est parfois appelée défiler.

2.4.2.2 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 du gestionnaire d'impression d'un système d'exploitation, la file d'attente des paquets à traiter dans le tampon d'un équipement réseau, etc.

2.4.2.3 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). Lorsque l'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 15: les skieurs (suite)

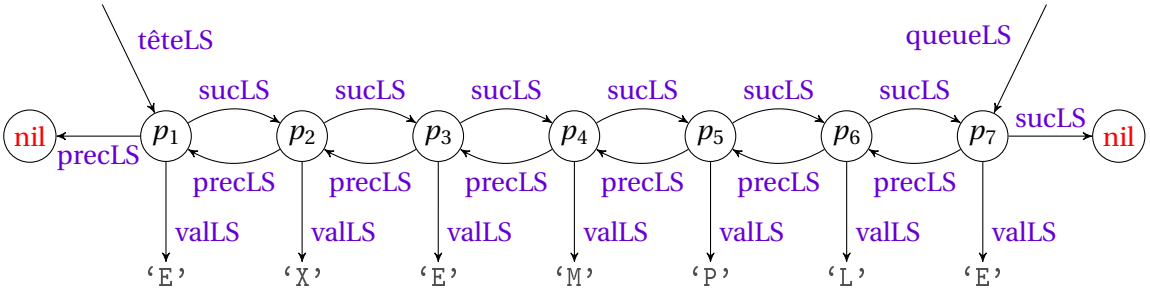
Soit une file d'attente de skieurs contenant 5 éléments. Elle peut être représentée de la manière suivante :

$$\begin{cases} f.tab = \begin{array}{|c|c|c|c|c|c|c|c|c|c|} \hline 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\ \hline s_4 & s_5 & & & & & & s_1 & s_2 & s_3 \\ \hline \end{array} \\ f.tête = 8 \\ f.nb = 5 \end{cases}$$

Exercice 2.15 (algorithmes de programmation pour les files)**.** Écrire 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 15 précédent. On suppose que la taille du tableau est suffisante et donnée par la constante *MAXTAB*.

2.5 Les listes symétriques

Une liste symétrique est une liste dans laquelle chaque élément pointe vers l'élément suivant *et* vers 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.

2.5.1 Définition abstraite du type liste symétrique

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** (ensemble des places, y compris nil qui est une place fictive), **booléen**.

Description fonctionnelle des opérations :

têteLS	: **ListeSym**(**Valeur**)	→ **Place**
queueLS	: **ListeSym**(**Valeur**)	→ **Place**
valLS	: **ListeSym**(**Valeur**) × (**Place** \ {nil})	→ **Valeur**
sucLS	: **ListeSym**(**Valeur**) × (**Place** \ {nil})	→ **Place**
precLS	: **ListeSym**(**Valeur**) × (**Place** \ {nil})	→ **Place**
finLS	: **ListeSym**(**Valeur**) × **Place**	→ **booléen**
LSvide	:	→ **ListeSym**(**Valeur**)
adjtLS	: **ListeSym**(**Valeur**) × **Valeur**	→
suptLS	: **ListeSym**(**Valeur**) \ {lisvide()}	→
adjqLS	: **ListeSym**(**Valeur**) × **Valeur**	→
supqLS	: **ListeSym**(**Valeur**) \ {lisvide()}	→
adjLS	: (**ListeSym**(**Valeur**) \ {lisvide()}) × (**Place** \ {nil}) × **Valeur**	→
supLS	: (**ListeSym**(**Valeur**) \ {lisvide()}) × (**Place** \ {nil})	→
chgLS	: (**ListeSym**(**Valeur**) \ {lisvide()}) × (**Place** \ {nil}) × **Valeur**	→

Les opérations adjtLS, suptLS, adjqLS, supqLS, adjLS, supLS, chgLS modifient la liste symétrique donnée en paramètre.

2.5.2 Exercice sur les listes symétriques

Exercice 2.16 (les palindromes)**.** Un palindrome est un mot qui se lit de la même façon de gauche à droite et de droite à gauche. Par exemple, les mots « elle », « radar », « PHP », « kayak » ou encore « ressasser » (le plus long palindrome en langue française) sont des palindromes.

Un mot étant représenté par une liste symétrique de caractères, écrire l'algorithme logique de la fonction *motChercherPalindrome* qui indique si un mot donné en paramètre est un palindrome. On suppose que le mot à analyser comprend au moins 2 lettres.

2.6 Exercices récapitulatifs

Exercice 2.17 (liste d'admissions)**.** Reprenons l'exemple de la liste d'admissions des étudiant·e·s dans une école (section 2.1.2.4). Rappelons qu'elle est triée par note décroissante et qu'elle contient pour chaque étudiant·e son nom et une note.

Question 1 : Écrire l'algorithme logique de la fonction de recherche de la place d'un·e étudiant·e à l'aide de son nom dans la liste des admissions. Si plusieurs personnes portent le même nom, la fonction retourne la place de la première personne avec ce nom.

```
1 fonction lEtudChercherÉlément(listeÉtudiant : Liste(Étudiant), nomÉtudiant :
    chaîne) : Place
```

Question 2 : Écrire l'algorithme logique de la fonction d'insertion d'un objet de type **Étudiant** (donc un type composite avec nom et note) dans la liste, supposée non vide au départ.

```
1 fonction
    lEtudInsérerÉtudiant(listeÉtudiant InOut : Liste(Étudiant), étudiant : Étudiant)
```

Question 3 : Écrire l'algorithme logique de la fonction *lEtudSupprimerÉtudiant* de suppression dans la liste d'un·e étudiant·e donné·e par son nom. On suppose la liste non vide au départ ; on suppose par ailleurs que le nom appartient bien à la liste. Si plusieurs personnes portent le même nom, on supprime la première uniquement.

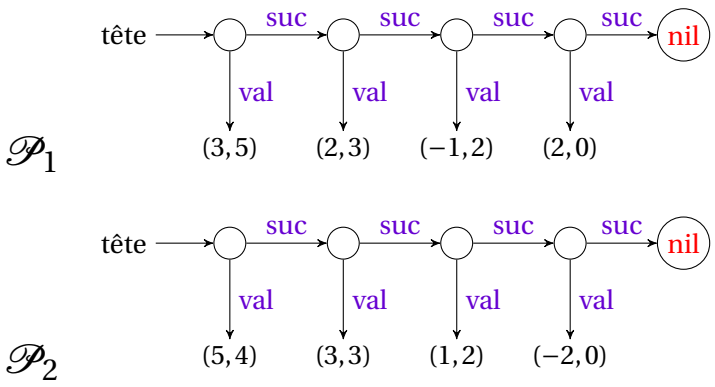
Exercice 2.18 (permis de conduire)**.** On dispose d'une liste existante de candidat·e·s se présentant au permis de conduire ; chaque candidat·e, désigné·e par son nom, obtient trois notes : conduite en ville, conduite sur route et manœuvre. Un·e candidat·e est reçu·e si la somme des 3 notes est ≥ 92.

Écrire l'algorithme logique de la fonction *imprimerRésultatsPermis* qui affiche le nom et la somme des 3 notes des candidat·e·s reçu·e·s puis le nom et la somme des 3 notes des candidat·e·s ayant échoué.

Remarque : on ne veut pas faire deux parcours de la liste des candidat·e·s.

Exercice 2.19 (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 $\mathscr{P}_1 = 3x^5 + 2x^3 - x^2 + 2$ et $\mathscr{P}_2 = 5x^4 + 3x^3 + x^2 - 2$. Ils seront représentés comme suit :

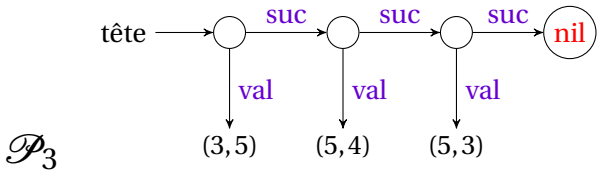


Question 1 : Écrire l'algorithme logique de la fonction *polynômeCalculerValeur* prenant en arguments un polynôme $\mathscr{P}$ et une valeur réelle de x, et calculant la valeur de $\mathscr{P}$ pour x. Par exemple, la valeur de $\mathscr{P}_1$ pour $x = 1$ est $3 \times 1^5 + 2 \times 1^3 - 1^2 + 2 = 6$. Par exemple, la valeur de $\mathscr{P}_2$ pour $x = 2$ est $5 \times 2^4 + 3 \times 2^3 + 2^2 - 2 = 80 + 24 + 4 - 2 = 106$.

On suppose disposer d'une fonction *puissanceEntièreCalculer* qui, à partir d'un réel x et d'un entier y, retourne le résultat de x^y.

Question 2 : Écrire l'algorithme logique de la fonction *polynômeAddition* prenant en arguments deux polynômes $\mathscr{P}_1$ et $\mathscr{P}_2$, et calculant leur addition $\mathscr{P}_1 + \mathscr{P}_2$.

Par exemple, soit $\mathscr{P}_3$ le polynôme résultant de l'addition de $\mathscr{P}_1$ et $\mathscr{P}_2$ (donnés ci-dessus) ; on a $\mathscr{P}_3 = 3x^5 + 5x^4 + 5x^3$.



Exercice 2.20 (simulation du jeu de la bataille)**.** 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.

Objectif On désire simuler sur une machine le déroulement d'une partie de ce jeu.

Question 1 : Définir les structures de données logiques nécessaires.

Question 2 : Écrire l'algorithme logique de la fonction qui, étant donnés les deux paquets initiaux, détermine le joueur gagnant.

Exercice 2.21 (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 : Écrire l'algorithme logique de la fonction *louvrageRechMotCle* 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.

Chapitre 3

Préparation SAÉ et entraînement

3.1 Préparation SAÉ 1.02 : listes triées

On considère ici des listes de chaînes de caractères triées par ordre alphabétique croissant. L'objectif sera de proposer des algorithmes logiques pour ces listes, qui utiliseront les fonctions existantes sur les listes (par exemple lisvide, adjlis, suplis, etc.).

Exercice 3.1 (listes triées de chaînes)**.**

Question 1 : Donner l'algorithme logique de la fonction adjlisT qui prend en paramètres une liste de chaînes triée et une chaîne de caractères, et insère la chaîne de caractères entre les places adéquates dans la liste.

Par exemple, soit ℓ = [“Basselin”,“Mari”,“Royer”,“Ventura”].

Considérons l'appel adjlisT(ℓ,“Kremer”).

La liste devient alors ℓ = [“Basselin”,“Kremer”,“Mari”,“Royer”,“Ventura”].

Question 2 : Donner l'algorithme logique de la fonction suplisT qui prend en paramètres une liste de chaînes triée et une chaîne de caractères, et supprime la place ayant pour valeur cette chaîne de caractères, si elle existe. (Si plusieurs places ont comme valeur cette chaîne, seule la première est supprimée.)

Question 3 : Donner l'algorithme logique de la fonction memlisT qui prend en paramètres une liste de chaînes triée et une chaîne de caractères, et rend vrai si et seulement si (au moins) une place a pour valeur la chaîne de caractères passée en paramètre.

3.2 Exercice d'entraînement : listes de films

On considère qu'un film est caractérisé par cinq champs : son année de réalisation, son titre, son type (une **chaîne** pouvant être par exemple `fiction` ou `documentaire`), le nom de son réalisateur, et enfin la liste des noms de ses acteurs.

3.2.1 Structure de données

Exercice 3.2.

Question 1 : Proposer une structure de données (par exemple un type composite) pour décrire un film.

On considère maintenant, et dans le reste de ces exercices, une liste de films triée par ordre croissant des années.

3.2.2 Algorithmes logiques

Exercice 3.3.

Question 1 : Écrire l'algorithme logique de la fonction *afficherFilmsRéalisateur* qui prend en argument la liste triée des films, et un réalisateur ou une réalisatrice identifié·e par son nom (type **chaîne**), et qui affiche l'ensemble de tous les films réalisés par ce réalisateur ou réalisatrice.

Question 2 : Écrire l'algorithme logique de la fonction *calculerFilmsAnnée* qui prend en argument la liste triée des films, et une année, et rend une liste qui contient l'ensemble de tous les films réalisés cette année-là.

Question 3 : Écrire l'algorithme logique de la fonction *ajouterFilm* qui ajoute un film à la liste. (La liste doit évidemment rester bien ordonnée.)

Question 4 : Écrire un algorithme logique *afficherFilms* qui affiche à l'écran le titre et l'année de réalisation de tous les films dont le type est `fiction` (classés de la même façon que dans la liste d'origine) puis le titre et l'année de réalisation de tous les films dont le type n'est *pas* `fiction`. On effectuera un seul parcours de la liste.

Question 5 : Écrire un algorithme logique *supprimerFilm* qui prend en paramètre la liste triée des films, et un nom d'acteur (type **chaîne**), et supprime de la liste tous les films dans lesquels cet acteur a joué (c'est-à-dire tous les films dont la liste des acteurs contient ce nom d'acteur).

Question 6 : Que faut-il modifier à la fonction précédente si la liste des acteurs est triée par ordre alphabétique ?

3.2.3 Algorithmes de programmation

Exercice 3.4. Nous cherchons maintenant à définir une représentation concrète de la liste de films.

On considère que l'on travaille sur un support (disque dur) où le temps d'écriture (modification de la mémoire) est beaucoup plus important que le temps de lecture.

Question 1 : Quelle représentation concrète de la liste de films (triée par année de réalisation) proposez-vous s'il est important que l'opération de suppression d'un film soit la plus efficace possible, c'est-à-dire afin que le nombre d'écritures en cas de suppression d'un film soit minimal ? Justifier.

Question 2 : Écrire pour cette représentation concrète l'algorithme de programmation *supprimerFilm* qui prend en argument une liste de films triée par année croissante et un titre de film (type : **chaîne**) et supprime le film de la liste. Si plusieurs films ont le même titre, seul le premier est supprimé. Si aucun film n'a ce titre, un message d'erreur « `Film non trouvé` » doit être affiché.

Annexe A

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** (voir annexe A.3) et type composite (voir annexe A.5).

A.1 Les mots-clés

Les mots-clés sont généralement en **gras** dans ce document. Lors de l'écriture manuscrite sur papier (TD, contrôle...), il est impératif d'utiliser du souligné, le **gras** n'étant bien sûr pas possible dans le contexte manuscrit. Une mise en couleur particulière au lieu du souligné peut être envisagée, mais n'est pas recommandée en contrôle sur table (manque de temps).

L'ensemble des mots-clés (types : **entier**, **Liste**... ; structures de contrôles : **de**, **pour**, ... ; **InOut** ...) devront donc être mis en valeur en les soulignant.

> **Remarque 13**
>
> L'usage du souligné doit être proscrit dans les documents numériques (sauf rares exceptions), selon les règles typographiques. En effet, de meilleures options d'emphase (**gras**, *italique*...) existent, et améliorent grandement la lisibilité et la qualité de la présentation.

A.2 Les principales instructions

Affectation $\leftarrow$

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.

Exemple : $i \leftarrow 1$

Saisie d'une donnée $v \leftarrow$ lire()

Affichage d'une valeur à l'écran écrire(v)

Conditionnelle La conditionnelle permet d'exécuter des instructions en fonction d'une condition.

```
1 si condition alors
2 |  instructions si la condition est vraie
3 sinon
4 |  instructions si la condition est fausse
5 fin si
  /* La partie « sinon » est optionnelle:                    */
6 si condition alors
7 |  instructions si la condition est vraie
8 fin si
```

Itération Les seuls en-têtes qui seront utilisés :

```
  /* Itérations croissantes                                  */
1 pour i de 1 à n faire
2 |  instruction à répéter
3 fin pour
  /* Itérations décroissantes                                */
4 pour i décroissant n à m faire
5 |  instruction à répéter
6 fin pour

  /* Itération tant qu'une condition est vérifiée            */
7 tant que condition faire
8 |  instruction à répéter
9 fin tq
```

A.3 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]

Par exemple, pour un tableau t' de films (type **Film**) numérotés de 100 à 199, on pourra écrire :

t' : **tableau Film**[100..199]

A.4 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 :

```
1 fonction nomFonction (liste des paramètres) :type du résultat
2 début
  | /* corps de la fonction                                    */
3 fin
```

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 16

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.

```
1  fonction insérerÉlémentDansTableauCroissant(t InOut : Tab, n :
     entier, élément : entier) : entier
2  début
   |  /* recherche de la place où insérer                      */
3  |  rang ← 0
4  |  trouvé ← faux
5  |  tant que rang < n et non trouvé faire
6  |  |  si t[rang] < élément alors
7  |  |  |  rang ← rang + 1
8  |  |  sinon
9  |  |  |  trouvé ← vrai
10 |  |  fin si
11 |  fin tq
   |  /* décalage des éléments plus grands que celui à insérer
   |     */
12 |  pour j décroissant de n à rang + 1 faire
13 |  |  t[j] ← t[j − 1]
14 |  fin pour
   |  /* insertion de élément                                  */
15 |  t[rang] ← élément
16 |  retourner rang
17 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
trouvé	: **booléen**	à vrai lorsque l'élément courant du tableau est plus grand que l'élément à insérer

A.5 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.

A.5.1 Définition lexicale

$$c : \langle champ_1 : \textbf{type}_1, champ_2 : \textbf{type}_2, \cdots, champ_n : \textbf{type}_n \rangle$$

où

— c est une variable composite dont le type est décrit,

— $champ_1, champ_2, \ldots, champ_n$ sont les noms des champs de la variable composite c,

— $\textbf{type}_1$ est le type du champ $champ_1$, …, $\textbf{type}_n$ le type du champ $champ_n$.

A.5.2 Sélection de champ

Sélection du champ $champ_i$ de c : $c.champ_i$

$c.champ_i$ est une variable de type $\textbf{type}_i$ et peut être utilisé comme toute variable de ce type. On utilise chaque identificateur de champ comme sélecteur du champ correspondant.

A.5.3 Construction d'une variable composite par la liste des valeurs des champs

$c \leftarrow (ch_1, ch_2, \cdots, ch_n)$ où $ch_1 : \textbf{type}_1, ch_2 : \textbf{type}_2, \cdots, ch_n : \textbf{type}_n$.

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

A.5.4 Modification de la valeur d'un champ

$c.champ_i \leftarrow v_i$, où v_i est de type $\textbf{type}_i$, la valeur v_i est affectée au champ $c.champ_i$ de c.

A.5.5 Autres « opérations »

$c \leftarrow$ lire()
écrire(c)
$c_1 \leftarrow c_2$ où c_1 et c_2 sont deux variables composites du même type.

A.5.6 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 », 2005)

— *fils* ← (*père.nom*, « Paul », (14, « mars », 1975), « Laxou »)

— *date.jour* ← 30

A.6 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 TypeÉlement**

où **TypeÉlement** est le type des éléments du fichier du nom logique *fich*. Pour un fichier texte, **TypeÉlement** 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, range la valeur lue dans *élément*, et avance le pointeur de fichier d'un élément.

Écriture fichierÉcrire(*fich*, *élément*) Écrit dans le fichier *fich* l'élément *élément* à la position courante donnée par le pointeur de fichier, et 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 17: skieurs

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.

```
1  fonction imprimerFichierSkieurs(nomPhysique : chaîne)
2  début
3      fSki ← fichierOuvrir(nomPhysique)
4      nom ← fichierLire(fSki)
5      tant que non fichierFin(fSki) faire
6          écrire(nom)
7          nom ← fichierLire(fSki)
8      fin tq
9      fichierFermer(fSki)
10 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 B

Fonctions logiques pour représentation contiguë dans un tableau

Nous donnons ici l'expression des fonctions logiques attachées à la structure de liste lorsque l'on utilise une représentation contiguë dans un tableau.

```
1 fonction supqlis(lAdmis InOut : ListeÉtudiant)
2 début
  |  /* Il suffit de mettre le nombre d'éléments à jour            */
3 |  lAdmis.nb ← lAdmis.nb − 1
4 fin
```

```
1 fonction chglis(lAdmis InOut : ListeÉtudiant, p : entier, étudiant : Étudiant)
2 début
3 |  lAdmis.tab[p] ← étudiant
4 fin
```

```
1 fonction suplis(lAdmis InOut : ListeÉtudiant, p : entier)
2 début
  |  /* Il faut décaler tous les éléments qui suivent la place p, d'un
  |     rang vers le début du tableau                               */
3 |  pour i de p à lAdmis.nb − 1 faire
4 |  |  lAdmis.tab[i] ← lAdmis.tab[i + 1]
5 |  fin pour
6 |  lAdmis.nb ← lAdmis.nb − 1
7 fin
```

```
1 fonction adjtlis(lAdmis InOut : ListeÉtudiant, étudiant : Étudiant)
2 début
  |  /* Il faut libérer la première place en décalant tous les
  |     éléments vers la fin du tableau                          */
3 |  pour i décroissant de lAdmis.nb + 1 à 2 faire
4 |  |  lAdmis.tab[i] ← lAdmis.tab[i − 1]
5 |  fin pour
6 |  lAdmis.tab[1] ← étudiant
7 |  lAdmis.nb ← lAdmis.nb + 1
8 fin
```

Annexe C

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

C.1 Initialisation de l'espace libre

```
1 fonction initEspaceLibre(ℓ InOut : ListeChainéeTableau, tailleTableau : entier)
2 début
3 |   pour i de 1 à tailleTableau faire
4 |   |   ℓ.tab[i].suc ← −1
5 |   fin pour
6 fin
```

Lexique		
ℓ	: **ListeChainéeTableau**	liste représentée de manière chaînée à l'aide d'un tableau
i	: **entier**	indice d'itération
tailleTableau	: **entier**	taille du tableau $\ell.tab$

C.2 Recherche d'une place libre pL

```
1  fonction
     rechercherPlaceLlibre(ℓ InOut : ListeChainéeTableau, tailleTableau : entier) : entier
2  début
3  |  i ← 1
4  |  trouvéPlaceLibre ← faux
5  |  tant que i < tailleTableau et non trouvéPlaceLibre faire
6  |  |  trouvéPlaceLibre ← (ℓ.tab[i].suc = −1)
7  |  |  i ← i + 1
8  |  fin tq
9  |  si trouvéPlaceLibre alors
10 |  |  pL ← i − 1
11 |  sinon
12 |  |  pL ← −999
13 |  fin si
14 |  retourner pL
15 fin
```

Lexique

ℓ	: **ListeChainéeTableau**	liste représentée de manière chaînée à l'aide d'un tableau
tailleTableau	: **entier**	nombre de places dans $\ell.tab$
trouvéPlaceLibre	: **booléen**	à vrai dès que l'on a trouvé une place libre
pL	: **entier**	place libre trouvée, ou –999 (valeur choisie arbitrairement, elle doit être négative) si plus de place libre
i	: **entier**	place courante

Remarque 14

Il est évidemment prohibé d'indiquer directement -999 dans une implémentation. Au strict minimum, il convient d'avoir une constante globale. L'idéal est sinon de gérer cette valeur de meilleure manière (mécanisme d'exception, type `option` en OCaml…).

C.3 Restitution de la place libre lors de la suppression de l'élément d'indice i

```
1  ℓ.tab[i].suc ← −1
```

Annexe D

Bilans récapitulatifs

D.1 Bilan de la section 2.1

À l'issue de la section 2.1, vous devez savoir manipuler des TAD liste et

- **comprendre les descriptifs** des fonctions associées au liste (section 2.1.1)
- faire des **parcours complets** de listes (section 2.1.3.1)
  - exercice 2.3 : taille d'une liste
  - exemple 2 et exercice 2.4 : moyenne d'une liste d'entiers
  - exemple 3 : imprimer personnes majeures
- faire des **parcours partiels** de liste (section 2.1.3.4)
  - parcours avec condition d'arrêt (section 2.1.3.2 : imprimer personnes majeures dans liste triée)
  - recherche d'un élément dans une liste
  - recherche d'un élément dans une liste triée (exercices 2.5 et 2.7)
  - parcours partiel avec condition de départ (section 2.1.3.3 et exercice 2.6)
- **Créer et manipuler** du contenu de listes
  - créer une liste à partir de valeurs lues (exercice 2.4)
  - créer une liste triée à partir de valeurs lues (exercice 2.7)
  - manipuler une liste : ajouts et suppressions dans une liste (exercice 2.7)
- **Interclasser** deux listes triées (exercice 2.8)

Annexe E

Crédits

E.1 Texte

Cours de Yolande BELAÏD (IUT Charlemagne) converti en LaTeX par Étienne ANDRÉ en août 2021 et amélioré depuis.

E.2 Images

Image	Auteur	Source	Licence
	frankes	https://openclipart.org/detail/175643/exercise-book	CC0
	libberry	https://commons.wikimedia.org/wiki/File:Memo_icon.svg	CC0
	libberry + Étienne André	https://commons.wikimedia.org/wiki/File:Memo_icon.svg	CC0
	Jakub Steiner	https://commons.wikimedia.org/wiki/File:Application-pgp-encrypted.svg	CC BY SA
	Icons8	https://commons.wikimedia.org/wiki/File:Icons8_flat_inspection.svg	MIT
	Kelvinsong	https://commons.wikimedia.org/wiki/File:Permanent_protect.svg	CC0
	Creative Commons	https://commons.wikimedia.org/wiki/File:Cc.logo.circle.svg	Public domain
	Sting	https://commons.wikimedia.org/wiki/File:Cc-by_new_white.svg	Public domain
	Rafał Pocztarski	https://commons.wikimedia.org/wiki/File:Cc-sa_white.svg	Public domain
	Creative Commons	https://commons.wikimedia.org/wiki/File:Cc-zero.svg	Public domain
	(multiples)	https://commons.wikimedia.org/wiki/File:PD-icon.svg	Public domain
	Massachusetts Institute of Technology	https://commons.wikimedia.org/wiki/File:PD-icon.svg	Public domain

Polycopié compilé par LaTeX le 8 décembre 2021