Avant-propos

Cet ouvrage intitulé « Algorithmique et structures de données » est le résultat de 8 années d'enseignements à l'ISET de Ksar-Hellal et dans d'autres institutions universitaires. De ce fait, il a bénéficié des questions et problèmes posés par les étudiants durant les cours et travaux pratiques de programmation.

L'objectif de l'ouvrage est d'expliquer en termes simples et à travers de multiples exemples et exercices corrigés comment apprendre à programmer tout en évitant les contraintes spécifiques imposées par les langages de programmation.

La première partie du livre traite les notions fondamentales de l'algorithmique : types de base, instructions simples, structures conditionnelles et itératives, procédures et fonctions, etc.

La seconde partie est consacrée aux structures de données composées : tableaux, enregistrements, fichiers, structures dynamiques : listes chaînées, piles, files et arbres.

Pour être complet, le livre contient une leçon sur la récursivité. De même, plusieurs problèmes ont été traités de façon itérative puis récursive afin de mettre en évidence les points forts et les points faibles de chaque approche.

Le livre contient environ une centaine d'exercices souvent conçus comme une application du cours à des situations de la vie professionnelle (calcul mathématique, problèmes de gestion, jeux, etc.). La solution proposée à chaque exercice n'est pas unique et dans plusieurs cas elle n'est optimale car on a toujours privilégié l'apport pédagogique et la simplicité.

Je tiens à remercier toutes les personnes qui m'ont aidé à l'élaboration de cet ouvrage et, en particuler, Messieurs Samir CHEBILI, Technologue à l'ISET de Sousse et Youssef GAMHA, Assistant Technologue à l'ISET de Ksar-Hellal pour leur active et aimable participation.

Leçon 1 : Introduction à la programmation

Objectifs

- Connaître le vocabulaire de base en programmation
- Comprendre la démarche de programmation

I. Notion de programme

Le mot programmation recouvre l'ensemble des activités qui font passer d'un problème à un programme.

Un **programme** est un texte constitué d'un ensemble de directives, appelées *instructions*, qui spécifient :

- les opérations élémentaires à exécuter
- la façon dont elles s'enchaînent.

Supposons qu'un enseignant dispose d'un ordinateur et d'un programme de calcul de la moyenne des étudiants. Pour s'exécuter, ce programme nécessite qu'on lui fournisse pour chaque étudiant, les notes de contrôle continu, de T.P et de l'examen final : ce sont ***les données***. En retour, le programme va fournir la moyenne cherchée : c'est ***le résultat***.

De la même manière, un programme de paie nécessite des informations données : noms des employés, grades, numéros de sécurité sociale, situations familiales, nombre d'heures supplémentaires, etc. Les résultats seront imprimés sur les différents bulletins de paie : identification de l'employé, salaire brut, retenue sécurité sociale, etc.

Pour éviter de saisir à chaque fois les mêmes données, certaines informations doivent être "**archivées**" sur une mémoire de masse. Le programme y accèdera alors directement (figure 1).

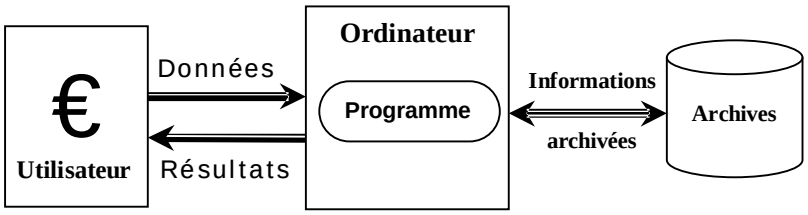


Figure 1. Schéma d'exécution d'un programme

II. Interprétation et compilation

L'ordinateur ne sait exécuter qu'un nombre limité d'opérations élémentaires dictées par des instructions de programme et codées en binaire (**langage machine**). Obligatoirement, les premiers programmes étaient écrits en binaire. C'était une tâche difficile et exposée aux erreurs car il fallait aligner des séquences de bits dont la signification n'est pas évidente.

Par la suite, pour faciliter le travail, les programmes ont été écrits en désignant les opérations par des codes mnémoniques faciles à retenir (ADD, DIV, SUB, MOVE, etc.). Les adresses des instructions et les variables pouvaient aussi être données sous forme symbolique. Pour pouvoir utiliser effectivement tous ces symboles, il fallait trouver le moyen de les convertir en langage machine ; ce fut réalisé par un programme appelé **assembleur**. Le langage d'assemblage est toujours utilisé, car c'est le seul langage qui permet d'exploiter au maximum les ressources de la machine.

L'écriture de programmes en langage d'assemblage reste une tâche fastidieuse. De plus, ces programmes présentent un problème de *portabilité* étant donné qu'ils restent dépendants de la machine sur laquelle ils ont été développés. L'apparition des **langages évolués**, tels que Fortran et Pascal, a apporté une solution à ces problèmes. Les programmes écrits en langage évolué doivent également être convertis en langage machine pour être exécutés. Cette conversion peut s'effectuer de deux façons : par *compilation* ou par *interprétation.*

La **compilation** consiste à traduire dans un premier temps l'ensemble du programme en langage machine. Dans un deuxième temps, le programme est exécuté.

L'**interprétation** consiste à traduire chaque instruction du programme en langage machine avant de l'exécuter (figure 2).

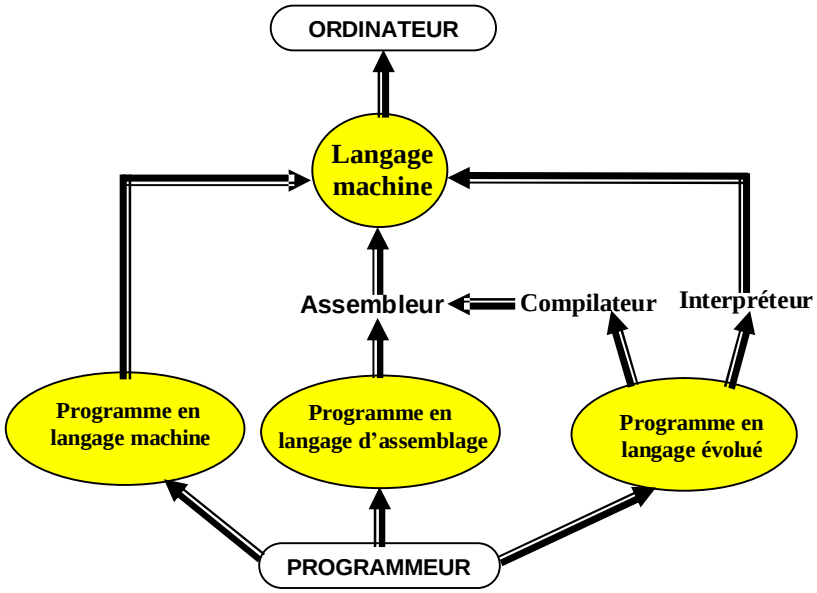


Figure 2. Différents niveaux de langages de programmation

III. Démarche de programmation

La démarche de programmation se déroule en deux phases (figure 3) :

1- Dans un premier temps, on identifie les données du problème, les résultats recherchés et par quel moyen on peut obtenir ces résultats à partir des données. C'est l'étape d'***analyse*** du problème qui aboutit à un procédé de résolution appelé **algorithme**. Une **approche modulaire** consiste à décomposer le problème initial en sous-problèmes plus simples à résoudre appelés **modules**. Par exemple, un programme de gestion de scolarité comportera un module inscription, un module examen, un module diplôme, etc.

2- Dans un deuxième temps, on traduit dans le langage de programmation choisi le résultat de la phase précédente. Si l'analyse a été menée convenablement, cette opération se réduit à une simple transcription systématique de l'algorithme selon la grammaire du langage.

Figure 3. Démarche de réalisation d'un programme

Lors de l'exécution, soit des erreurs syntaxiques sont signalées, ce qui entraîne des retours au programme et des corrections en général simples à effectuer, soit des erreurs sémantiques plus difficiles à déceler. Dans ce dernier cas, le programme produit des résultats qui ne correspondent pas à ceux escomptés : les retours vers l'analyse (l'algorithme) sont alors inévitables.

IV. Notion de variable

Malgré une apparente diversité, la plupart des langages sont basés sur la même technique fondamentale à savoir la « manipulation de valeurs contenues dans des variables ».

En programmation, une **variable** est un identificateur qui sert à repérer un emplacement donné de la mémoire centrale. Cette notion nous permet de manipuler des valeurs sans nous préoccuper de l'emplacement qu'elles occupent effectivement en mémoire.

La lisibilité des programmes dépend étroitement du choix des noms des variables qui doivent être simples et significatifs. Ainsi, Mont_Fac est un meilleur choix que x pour désigner le montant d'une facture.

Comme tout ***identificateur***, le nom d'une variable est formé d'une ou de plusieurs lettres ; les chiffres sont également autorisés, à condition de ne pas les mettre au début du nom (figure 4).

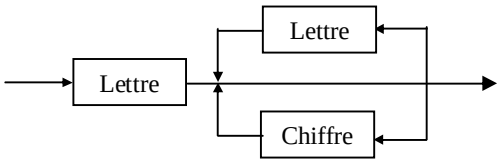


Figure 4. Règle de construction d'un identificateur

Ainsi : P, Quantite et f91 sont tous des noms corrects.

Notons enfin que certains langages comme C, C++ et Java font la différence entre lettres minuscules et majuscules alors que d'autres comme Turbo Pascal n'en font aucune distinction.

V. Notion de constante

Contrairement aux variables, les constantes sont des données dont la valeur reste fixe durant l'exécution du programme.

Exemples : Pi = 3.14 g = 9.80 etc.

VI. Notion de type

A chaque variable utilisée dans le programme, il faut associer un type qui permet de définir :

- l'ensemble des valeurs que peut prendre la variable
- l'ensemble des opérations qu'on peut appliquer sur la variable.

La syntaxe de l'action de déclaration est la suivante :

Variable 1, Variable 2, … : Type

Les principaux types utilisés en algorithmique sont :

- le type entier
- le type réel
- le type caractère
- le type chaîne de caractères
- le type logique ou booléen.

VI.1. le type entier

Une variable est dite entière si elle prend ses valeurs dans Z (ensemble des entiers relatifs) et qu'elle peut supporter les opérations suivantes :

Opération	Notation
Addition	+
Soustraction	-
Multiplication	*
Division entière	div
Modulo (reste de la division)	mod

Exemples

13 div 5 = 2
13 mod 5 = 3

L'ensemble de valeurs que peut prendre un entier varie selon le langage de programmation utilisé étant donné que le nombre de bits réservés pour une variable de ce type n'est pas le même. A titre d'exemple, en Turbo Pascal, les entiers varient entre –32768 et +32767.

VI.2. le type réel ou décimal

Il existe plusieurs types de réels représentant chacun un ensemble particulier de valeurs prises dans $\Re$ (ensemble des nombres réels). Ici encore, cette distinction se justifie par le mode de stockage des informations dans le langage de programmation.

Il existe deux formes de représentation des réels :

- la forme **usuelle** avec le point comme symbole décimal.

  Exemples : -3.2467 2 12.7 +36.49

- la notation **scientifique** selon le format **aEb**, où :

a est la mantisse, qui s'écrit sous une forme usuelle
b est l'exposant représentant un entier relatif.

Exemple : 347 = 3.47E2 = 0.347E+3 = 3470E-1 = ...

Les opérations définies sur les réels sont :

Opération	Notation
Addition	+
Soustraction	-
Multiplication	*
Division (réelle)	/

VI.3. le type caractère

Un caractère peut appartenir au domaine des chiffres de "0" à "9", des lettres (minuscules et majuscules) et des caractères spéciaux ("*", "/", "{", "$", "#", "%" …). Un caractère sera toujours noté entre des guillemets. Le caractère espace (blanc) sera noté " ".

Les opérateurs définis sur les données de type caractère sont :

Opération	Notation
Égal	=
Différent	#
Inférieur	<
Inférieur ou égal	<=
Supérieur	>
Supérieur ou égal	>=

La comparaison entre les caractères se fait selon leur codes ASCII (voir annexe 2).

Exemple :

" " < "0" < "1" < "A" < "B" < "a" < "b" < "{"

VI.4. le type logique ou booléen

Une variable logique ne peut prendre que les valeurs ”Vrai” ou ”Faux”. Elle intervient dans l'évaluation d'une condition.

Les principales opérations définies sur les variables de type logique sont : la négation (**NON**), l'intersection (**ET**) et l'union (**OU**).

L'application de ces opérateurs se fait conformémnet à la table de vérité suivante :

Table de vérité des opérateurs logiques

A	B	NON (A)	A ET B	A OU B
Vrai	Vrai	Faux	Vrai	Vrai
Vrai	Faux	Faux	Faux	Vrai
Faux	Vrai	Vrai	Faux	Vrai
Faux	Faux	Vrai	Faux	Faux

Remarque

En plus de ces types pré-définis, le programmeur a la possibilité de définir lui-même de nouveaux types en fonction de ses besoins.

Exemple

```
Types
        Saison = (”A”,”H”,”P”,”E”)
        Tnote = 0 .. 20
Variables
        s : Saison
        note : Tnote
```

ð La variable s de type saison ne peut prendre que les valeurs ”A”, ”H”, ”P” ou ”E”.

VII. Les expressions

Ce sont des combinaisons entre des variables et des constantes à l'aide d'opérateurs. Elles expriment un calcul (expressions arithmétiques) ou une relation (expressions logiques).

VII. 1. Les expressions arithmétiques

Exemple : x * 53.4 / (2 + pi)

L'ordre selon lequel se déroule chaque opération de calcul est important. Afin d'éviter les ambiguïtés dans l'écriture, on se sert des parenthèses et des relations de priorité entre les opérateurs arithmétiques :

Ordre de priorité des opérateurs arithmétiques

Priorité	Opérateurs	
1	-	signe négatif (oprérateur unaire)
2	( )	parenthèses
3	^	puissance
4	* et /	multiplication et division
5	+ et -	addition et soustraction

En cas de conflit entre deux opérateurs de même priorité, on commence par celui situé le plus à gauche.

VII. 2. Les expressions logiques

Ce sont des combinaisons entre des variables et des constantes à l'aide d'opérateurs relationnels (=, <, <=, >, >=, #) et/ou des combinaisons entre des variables et des constantes logiques à l'aide d'opérateurs logiques (NON , ET, OU, …).

Ici encore, on utilise les parenthèses et l'ordre de priorité entre les différents opérateurs pour résoudre les problèmes de conflits.

Opérateurs logiques

Priorité	Opérateur
1	NON
2	ET
3	OU

Opérateurs relationnels

Priorité	Opérateur
1	>
2	>=
3	<
4	<=
5	=
6	#

EXERCICES D'APPLICAION

Exercice 1.1

1- Quel est l'ordre de priorité des différents opérateurs de l'expression suivante :

((3 * a) – x ^ 2) – (((c – d) / (a / b)) / d)

2- Evaluer l'expression suivante :

5 + 2 * 6 – 4 + (8 + 2 ^ 3) / (2 – 4 + 5 * 2)

3- Ecrire la formule suivante sous forme d'une expression arithmétique :

$$\frac{(3-xy)^2 - 4ac}{2x - z}$$

Exercice 1.2

Sachant que a = 4, b = 5, c = -1 et d = 0, évaluer les expressions logiques suivantes :

1- (a < b) ET (c >= d)

2- NON (a < b) OU (c # b)

3- NON((a # b ^ 2) OU (a*c < d))

SOLUTIONS DES EXERCICES

Ecercice 1.1

1- ((3 * a) – x ^ 2) – (((c – d) / (a / b)) / d)

2- 5 + 2 * 6 – 4 + (8 + 2 ^ 3) / (2 – 4 + 5 * 2) = 15

3- ((3 – x * y) ^ 2 – 4 * a * c) / (2 * x – z)

Ecercice 1.2

1- Faux
2- Vrai
3- Faux

ð Le résultat d'une expression logique est toujours Vrai ou Faux.

Leçon 2 : Les instructions simples

Objectifs

- Comprendre les actions algorithmiques simples
- Connaître la structure générale d'un algorithme

I. L'instruction d'affectation

Le rôle de cette instruction consiste à placer une valeur dans une variable. Sa syntaxe générale est de la forme :

Variable ← Expression

Ainsi, l'instruction :

A ← 6

signifie « mettre la valeur 6 dans la case mémoire identifiée par A ».

Alors que l'instruction :

B ← (A + 4) Mod 3

range dans B la valeur 1 (A toujours égale à 6).

Remarque

La valeur ou le résultat de l'expression à droite du signe d'affectation doit être de même type ou de type compatible avec celui de la variable à gauche.

Exemple 1

Quelles seront les valeurs des variables A, B et C après l'exécution des instructions suivantes :

A ← 5
B ← 3
C ← A + B
A ← 2
C ← B – A

Trace d'exécution

Instruction	Contenu des variables		
	A	**B**	**C**
(1) A ← 5	5	-	-
(2) B ← 3	5	3	-
(3) C ← A + B	5	3	8
(4) A ← 2	2	3	8
(5) C ← B – A	**2**	**3**	**1**

Exemple 2

Quelles seront les valeurs des variables A et B après l'exécution des instructions suivantes :

A ← 5
B ← 7
A ← A + B
B ← A - B
A ← A – B

Trace d'exécution

Instruction	Contenu des variables	
	A	**B**
(1) A ← 5	5	-
(2) B ← 7	5	7
(3) A ← A + B	12	7
(4) B ← A – B	12	5
(5) A ← A – B	**7**	**5**

Cette séquence d'instructions permet de permuter les valeurs de A et B.

Une deuxième façon de permuter A et B consiste à utiliser une variable intermédiaire X :

X ← A
A ← B
B ← X

Exercice

Soient 3 variables A, B et C. Ecrire une séquence d'instructions permettant de faire une permutation circulaire de sorte que la valeur de A passe dans B, celle de B dans C et celle de C dans A. On utilisera une seule variable supplémentaire.

Solution

```
X ← C
C ← B
B ← A
A ← X
```

II. L'instruction d'écriture *: communication Programme* ð *Utilisateur*

Pour qu'un programme présente un intérêt pratique, il devra pouvoir nous communiquer un certain nombre d'informations (résultats) par l'intermédiaire d'un périphérique de sortie comme l'écran ou l'imprimante. C'est le rôle de l'instruction d'écriture.

La forme générale de cette instruction est :

Ecrire(expression1, expression2, ...)

Exemple

Qu'obtient t-on à l'écran après l'exécution des instructions suivantes :

```
x ← 4
Ville ← "Tunis"
Ecrire(x)
Ecrire(x*x)
Ecrire("Ville")
Ecrire("Ville = ",Ville)
```

Solution

```
4
16
Ville
Ville = Tunis
```

III. L'instruction de lecture : *communication Utilisateur* ð *Programme*

Dans certains cas, nous pourrons être amenés à transmettre des informations (données) à notre programme par l'intermédiaire d'un périphérique d'entrée comme le clavier. Cela sera réalisé par l'instruction de lecture.

La forme générale de cette instruction est :

Lire(Variable1, Variable2, ...)

Ainsi, l'instruction :

Lire(A)

signifie « lire une valeur à partir du périphérique d'entrée, qui est le clavier par défaut, et la ranger dans la variable identifiée par A »

IV. Structure générale d'un algorithme

```
Algorithme Nom_algorithme
Constantes
        C1 = Valeur1
        …
Types
        Type1 = définition du type
        …
Variables
        V1 : Type1
        …
Début
        Instruction1
        …
Fin.
```

Remarques

1. Les nouveaux types doivent être déclarés dans la partie «Types».
2. Toute variable utilisée dans l'algorithme doit être préalablement déclarée.
3. Les instructions placées entre «Début» et «Fin» forment le *corps principal* de l'algorithme.
4. Seules les parties « Entête » et «Corps » de l'algorithme sont obligatoires.
5. L'indentation qui consiste à insérer des tabulations avant les objets déclarés et les instructions offre une meilleure lisibilité de l'algorithme et le rend, par conséquent, plus facile à comprendre et à corriger.

EXERCICES D'APPLICATION

Exercice 2.1

Donner toutes les raisons pour lesquelles l'algorithme suivant est incorrect :

```
Algoritme Incorrect
        x, y : Entier
        z : Réel
Début
        z ← x + 2
        y ← z
        x*2 ← 3 + z
        y ← 5y + 3
Fin.
```

Exercice 2.2

Ecrire un algorithme qui lit deux entiers au clavier et qui affiche ensuite leur somme et leur produit.

Exercice 2.3

Ecrire un algorithme qui lit le rayon d'un cercle et qui affiche ensuite son périmètre et sa surface.

Exercice 2.4

Ecrire un algorithme qui calcule et affiche la résistance d'un composant électrique en utilisant la loi d'Ohm :

$$U = R \times I \qquad \text{avec} \begin{cases} \text{U : Tension en V} \\ \text{R : Résistance en } \Omega \\ \text{I : Intensité en A.} \end{cases}$$

SOLUTIONS DES EXERCICES

Exercice 2.1

```
1. Algoritme Incorrect
2.       x, y : Entier
3.       z : Réel
4. Début
5.       z ← x + 2
6.       y ← z
7.       x*2 ← 3 + z
8.       y ← 5y + 3
9. Fin.
```

Cet algorithme est incorrect pour plusieurs raisons :

- ligne 1 : Le mot Algorithme s'écrit avec un « h » au milieu
- ligne 2 : La déclaration des variables commence par le mot « Variables »
- ligne 5 : La valeur de x est indéterminée
- Ligne 6 : incompatibilité de type (un résultat réel affecté à une variable de type entier)
- Ligne 7 : Le membre gauche d'une affectation doit être une variable
- Ligne 8 : Il faut écrire 5*y et non 5y.

Exercice 2.2 : calcul de la somme et du produit de deux entiers

```
Algorithme Som_Prod
Variables
        a, b, s, p : Entier
Début
        Ecrire("Entrer la valeur de a : ")   Lire(a)
        Ecrire("Entrer la valeur de b : ")   Lire(b)
        s ← a + b
        p ← a * b
        Ecrire ("Somme = ", s)
        Ecrire ("Produit = ", p)
Fin.
```

Exercice 2.3 : calcul du périmètre et de la surface d'un cercle

```
Algorithme Cercle
Constantes
        pi = 3.14
Variables
        r, p, s : Réel
Début
        Ecrire(”Entrer le rayon du cercle : ”)   Lire(r)
        p ← 2 * pi * r
        s ← pi * r ^ 2
        Ecrire (”Périmètre = ”, p)
        Ecrire (”Surface = ”, s)
Fin.
```

Exercice 2.4 : calcul de la résistance d'un composant électrique

```
Algorithme Résistance
Variables
        U, I, R : Réel
Début
        Ecrire(”Entrer la tension : ”)   Lire(U)
        Ecrire(”Entrer l'intensité : ”)   Lire(I)
        R ← U/I          (* on suppose toujours I # 0 *)
        Ecrire (”Résitance = ”, R, ”  Ohms” )
Fin.
```

Leçon 3 : les structures conditionnelles

Objectif

Construire des algorithmes comportant des traitements conditionnels.

I. Introduction

En programmation, on est souvent confronté à des situations où on a besoin de choisir entre deux ou plusieurs traitements selon la réalisation ou non d'une certaine condition ; d'où la notion de *traitement conditionnel*.

On distingue deux structures de traitement conditionnel à savoir :

- La **structure de sélection simple** dans laquelle on a à choisir entre deux traitements au plus ;
- La **structure de sélection multiple** dans laquelle on a la possibilité de choisir un traitement parmi plusieurs.

II. Structure de sélection simple

II.1. Forme simple

```
Si <Condition> Alors
        <Séquence d'instructions>
FinSi
```

Cette primitive a pour effet d'exécuter la séquence d'instructions si et seulement si la condition est vérifiée.

L'exécution de cette instruction se déroule selon l'organigramme suivant (figure 5) :

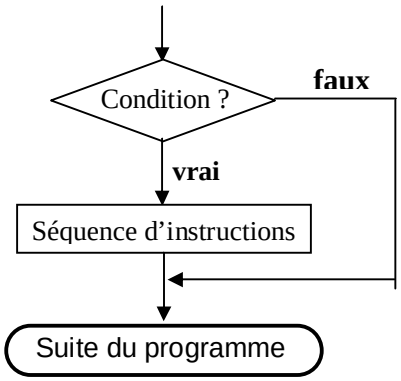


Figure 5. Schéma d'exécution d'une instruction conditionnelle simple

Exemple

Ecrire un algorithme qui calcule le salaire d'un employé à partir du nombre d'heures travaillées, du taux horaire et du nombre d'années de service. Les employés ayant une ancienneté de plus de 10 ans bénéficient d'une allocation supplémentaire de 35 D.

Analyse du problème

1- Données du problème

- nh : nombre d'heures travaillées
- th : taux horaire (en Dinar)
- anc : ancienneté (en Année)

2- Résultat recherché

- salaire : salaire net de l'employé (en Dinar)

3- Comment faire ?

```
Salaire ← nh * th
Si (anc > 10) Alors
        salaire ← salaire + 35
FinSi
```

Solution

Algorithme Calc_Salaire

```
Variables
        nh, th, anc, salaire : Réel
Début
        Ecrire(”Nombre d’heures travaillées : ”)  Lire(nh)
        Ecrire(”Taux horaire : ”)   Lire(th)
        Ecrire(”Ancienneté : ”)   Lire(anc)
        salaire ← nh * th
        Si (anc > 10) Alors
                salaire ← salaire + 35
        FinSi
        Ecrire (”Salaire de l’employé = ”, salaire)
Fin.
```

II.2 Forme alternative

```
Si <Condition> Alors
        <Séquence d’instructions 1>
Sinon
        <Séquence d’instructions 2>
FinSi
```

Cette primitive a pour effet d’exécuter la première séquence d’instructions si la condition est vérifiée ou bien la deuxième séquence d’instructions dans le cas contraire.
L’exécution de cette instruction se déroule selon l’organigramme suivant (figure 6) :

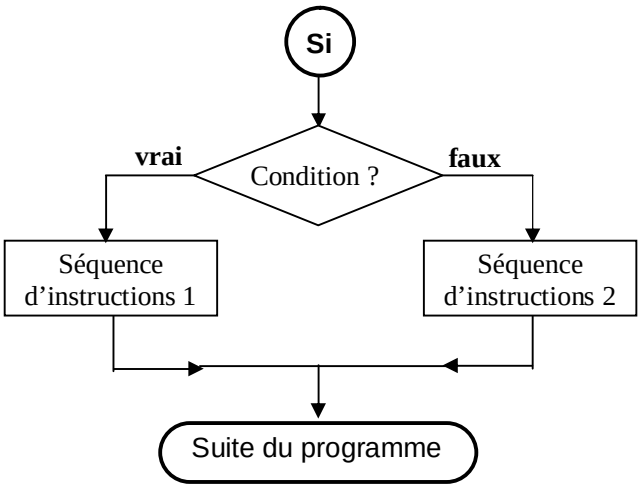


Figure 6. Schéma d'exécution d'une instruction conditionnelle alternative

Exemple 1

Ecrire un algorithme qui calcule et affiche la valeur absolue d'un entier quelconque lu au clavier.

Solution

```
Algorithme Val_Abs
Variables
        x, va : Entier
Début
        Ecrire("Entrer un entier : ")   Lire(x)
        Si (x >= 0) Alors
                va ← x
        Sinon
                va ← -x
        FinSi
        Ecrire ("|",x, "| = ",va)
Fin.
```

Exemple 2

Ecrire un algorithme qui lit un entier et affiche ensuite s'il est pair ou impair.

Solution

```
Algorithme pair_impair
Variables
        x : Entier
Début
        Ecrire("Entrer un entier : ")   Lire(x)
        Si (x Mod 2 = 0) Alors
                Ecrire("c'est un entier pair")
        Sinon
                Ecrire("c'est un entier impair")
        FinSi
Fin.
```

III. Structure de sélection multiple

Syntaxe générale

```
Selon <sélecteur> Faire
        Liste de valeurs 1 : < Séquence d'instructions 1>
        Liste de valeurs 2 : < Séquence d'instructions 2>
        …
        Liste de valeurs n : < Séquence d'instructions n>
Sinon
        <Autre Séquence d'instructions>
FinSelon
```

La séquence d'instructions numéro i sera exécuté si la valeur du sélecteur appartient à la $i^{ème}$ liste de valeurs.
Le sélecteur est une variable ou une expression de type **scalaire** (le résultat est un entier ou un caractère).

Exercice

Ecrire un algorithme qui permet de lire un numéro compris entre 1 et 12 et d'afficher le nom du mois correspondant. Si le numéro entré est en dehors de cet intervalle, un message d'erreur doit être affiché.
Solution

```
Algorithme mois
Variables
        n : Entier
Début
        Ecrire(”Entrer le numéro du mois : ”)   Lire(n)
        Selon n Faire
                1 : Ecrire(”janvier”)
                2 : Ecrire(”février”)
                …
                12 : Ecrire(”décembre”)
        Sinon
                Ecrire(”numéro de mois erroné... ”)
        FinSelon
Fin.
```

Remarque

L'exercice précédent peut être résolu en utilisant plusieurs instructions « Si » imbriquées, mais l'algorithme sera très lourd :

```
Algorithme mois     (* version Si *)
Variables
        n : Entier
Début
        Ecrire(”Entrer le numéro du mois : ”)   Lire(n)
        Si (n = 1) Alors
                Ecrire(”janvier”)
        Sinon
                Si (n = 2) Alors
                        Ecrire(”février”)
                Sinon
                        Si (n = 3) Alors
                                Ecrire(”mars”)
                        Sinon
                                Si (n = 4) Alors
                                …

Fin.
```

EXERCICES D'APPLICATION

Exercice 3.1

Ecrire un algorithme permettant de résoudre dans $\Re$ une équation du second degré de la forme $\mathbf{ax^2+bx+c = 0}$.

Exercice 3.2

Ecrire un algorithme pemettant de simuler une calculatrice à 4 opérations (+, -, *, et /). Utiliser la structure « selon » pour le choix de l'opération à effectuer.

Exercice 3.3

Ecrire un algorithme qui lit un caractère au clavier puis affiche s'il s'agit d'une lettre minuscule, d'une lettre majuscule, d'un chiffre ou d'un caractère spécial.

SOLUTIONS DES EXERCICES

Exercice 3.1 : résolution d'une équation du second degré

```
Algorithme equa2d
Variables
    a, b, c, delta : Réel
Début
    Ecrire("Entrer la valeur de a (non nulle) : ")   Lire(a)
    Ecrire("Entrer la valeur de b : ")   Lire(b)
    Ecrire("Entrer la valeur de c : ")   Lire(c)
    delta ← b^2 – 4*a*c
    Si (delta < 0) Alors
            Ecrire("pas de solution dans R")
    Sinon
            Si (delta = 0) Alors
                    Ecrire("x1 = x2 = ",-b/(2*a))
            Sinon
                    Ecrire("x1 =",(-b-racine(delta))/(2*a))
                    Ecrire("x2 =",(-b+racine(delta))/(2*a))
            FinSi
    FinSi
Fin.
```

Exercice 3.2 : simulation d'une calculatrice à 4 opérations

```
Algorithme calculatrice
Variables
    val1, val2 : Réel
    opération : Caractère
Début
    Ecrire("Première opérande : ")   Lire(val1)
    Ecrire("Opération : ")   Lire(opération)
    Ecrire("Deuxième opérande : ")   Lire(val2)
    Selon opération Faire
            "+" :   Ecrire("Résultat = ", val1 + val2)
            "-" :   Ecrire("Résultat = ", val1 - val2)
            "*" :   Ecrire("Résultat = ", val1 * val2)
            "/" :   Si (b # 0) Alors
                            Ecrire("Résultat = ", val1/val2)
```

```
                    Sinon
                        Ecrire(”Division par zéro !”)
                    FinSi
        Sinon
            Ecrire(”opérateur erroné... ”)
        FinSelon
Fin.
```

Exercice 3.3 : nature d’un caractère

```
Algorithme Nature_car
Variables
        c : Caractère
Début
        Ecrire(”Entrer un caractère : ”)   Lire(c)
        Selon c Faire
            ”a”..”z” : Ecrire(”c’est une lettre minuscule”)
            ”A”..”Z” : Ecrire(”c’est une lettre majuscule”)
            ”0”..”9” : Ecrire(”c’est un chiffre”)
        Sinon
            Ecrire(”c’est un caractère spécial”)
        FinSelon
Fin.
```

Leçon 4 : Les structures itératives

Objectif

Construire des algorithmes comportant des traitements itératifs.

I. Introduction

La notion d'itération est une des notions fondamentales de l'algorithmique. On l'utilise souvent quand on doit exécuter le même traitement un certain nombre de fois qui peut être connu à l'avance ou non. Dans ce dernier cas, l'arrêt de l'itération est déclenché par une condition sur l'état des variables dans le programme.

II. La structure « Pour … Faire »

Syntaxe générale

```
Pour compteur De valeur initiale A valeur finale Faire
        <Séquence d'instructions>
FinPour
```

Principe de fonctionnement

Le compteur (variable de contrôle) prend la valeur initiale au moment d'accès à la boucle puis, à chaque parcours, il passe **automatiquement** à la valeur suivante dans son domaine jusqu'à atteindre la valeur finale (figure 7) :

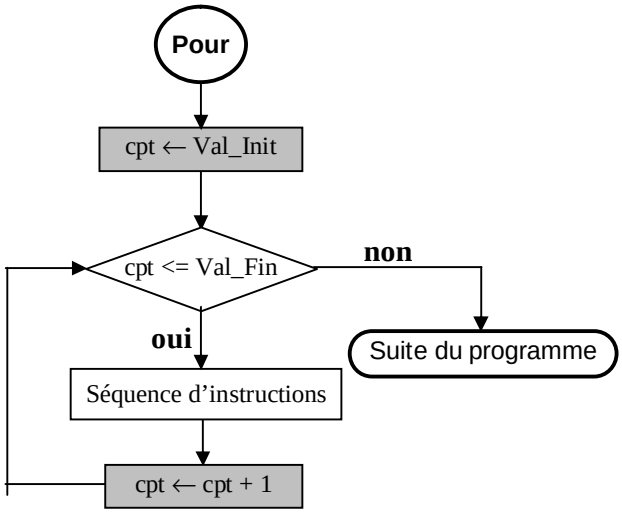


Figure 7. Schéma d'exécution d'une boucle « pour »

Exemple

```
Pour i de 1 à 5 Faire
        Ecrire(i*10)
FinPour
```

Cette boucle affiche respectivement les nombres 10, 20, 30, 40 et 50.

Remarques

1- Une boucle « pour » peut être exécutée 0, 1 ou n fois
2- On ne peut utiliser la boucle « pour » que si on connaît au préalable combien de fois le traitement sera exécuté (valeur finale – valeur initiale + 1).
3- Dans une boucle « pour », l'évolution du compteur peut se faire dans le sens **décroissant** comme dans l'exemple suivant :

```
Pour i de 5 à 1 [pas = -1] Faire
        Ecrire(i*10)
FinPour
```

Dans ce cas, le compteur i sera **décrémenté** après chaque parcours.

Exercice

Ecrire un algorithme qui lit un entier positif n puis affiche tous ses diviseurs.

Solution

```
Algorithme Diviseurs     (* version pour *)
Variables
        n, i : Entier
Début
        Ecrire(”Entrer un entier positif : ”)   Lire(n)
        Pour i de 1 à n Faire
                Si (n Mod i = 0) Alors
                        Ecrire(i)
                FinSi
        FinPour
Fin.
```

Exercice

Ecrire un algorithme qui lit un entier positif n puis calcule et affiche son factoriel selon la formule $\mathbf{n! = 1 \times 2 \times \ldots \times n}$.

Solution

```
Algorithme Facto     (* version pour *)
Variables
        n, i, f : Entier
Début
        Ecrire(”Entrer un entier positif : ”)   Lire(n)
        f ← 1
        Pour i de 2 à n Faire
                f ← f * i
        FinPour
        Ecrire(n,”! = ”,f)
Fin.
```

III. La structure « Répéter … Jusqu’à »

Syntaxe générale

Répéter
    <Séquence d'instructions>
Jusqu'à <condition>

Principe de fonctionnement

La séquence d'instructions est exécutée une première fois, puis l'exécution se répète jusqu'à ce que la condition de sortie soit vérifiée.

Une boucle « répéter » s'exécute toujours au moins une fois (figure 8).

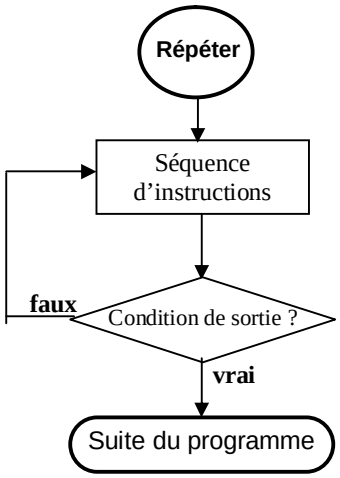


Figure 8. Schéma d'exécution d'une boucle « répéter »

Exemple

```
i ← 1
Répéter
        Ecrire(i*10)
        i ← i + 1
Jusqu'à (i > 5)
```

Remarques

1- Contrairement à une boucle « pour », dans une boucle « répéter », l'initialisation et l'avancement du compteur doivent être gérés manuellement par le programmeur.

2- Dans une boucle « répéter », il faut toujours s'assurer que la condition de sortie sera vérifiée après un nombre fini de parcours. Sinon, c'est une *boucle infinie* comme dans le cas suivant :

```
c ← "A"
Répéter
        Ecrire(c)
Jusqu'à (c > "Z")
```

Exercice

Réécrire l'algorithme diviseurs en remplaçant la boucle « pour » par une boucle « répéter »

Solution

```
Algorithme Diviseurs     (* version répéter *)
Variables
        n, i : Entier
Début
        Ecrire("Entrer un entier positif : ")   Lire(n)
        i ← 1
        Répéter
                Si (n Mod i = 0) Alors
                        Ecrire(i)
                FinSi
                i ← i + 1
        Jusqu'à (i > n)
Fin.
```

Exercice

Réécrire l'algorithme facto en remplaçant la boucle « pour » par une boucle « répéter »

Solution

```
Algorithme Facto     (* version répéter *)
Variables
        n, f, i : Entier
```

```
Début
        Ecrire(”Entrer un entier positif : ”)   Lire(n)
        f ← 1
        i ← 1
        Répéter
                f ← f * i
                i ← i + 1
        Jusqu’à (i > n)
        Ecrire(n,”! = ”,f)
Fin.
```

IV. La structure « TantQue … Faire »

Syntaxe générale

```
TantQue <condition> Faire
        <Séquence d’instructions>
FinTQ
```

Principe de fonctionnement

Le traitement est exécuté aussi longtemps que la condition est vérifiée. Si dès le début cette condition est fausse, le traitement ne sera exécuté aucune fois.

ð Une boucle « tantque » peut s’exécuter 0, 1 ou n fois (figure 9).

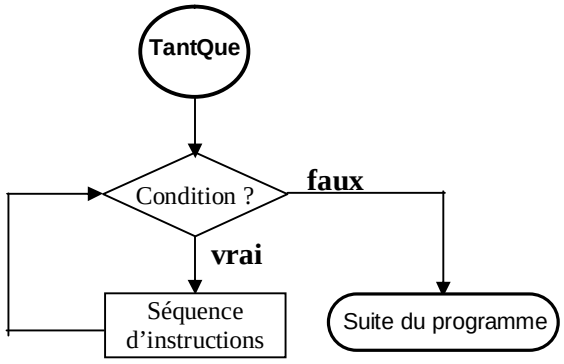


Figure 9. Schéma d'exécution d'une boucle « tantque »

Exemple

```
i ← 1
TantQue (i<=5) Faire
        Ecrire(i*10)
        i ← i + 1
FinTQ
```

Exercice

Réécrire l'algorithme diviseurs en remplaçant la boucle « répéter » par une boucle « tantque »

Solution

```
Algorithme Diviseurs     (* version tantque *)
Variables
        n, i : Entier
Début
        Ecrire("Entrer un entier positif : ")   Lire(n)
        i ← 1
        TantQue (i <= n) Faire
                Si (n Mod i = 0) Alors
                        Ecrire(i)
                FinSi
                i ← i + 1
        FinTQ
Fin.
```

Exercice

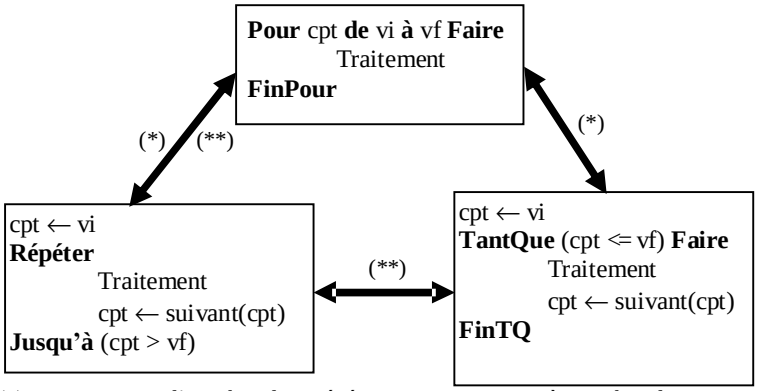
Réécrire l'algorithme facto en remplaçant la boucle « répéter » par une boucle « tantque ».

Solution

```
Algorithme Facto     (* version tantque *)
Variables
        n, f, i : Entier
Début
        Ecrire("Entrer un entier positif : ")   Lire(n)
        f ← 1
        i ← 2
        TantQue (i <= n) Faire
                f ← f * i
                i ← i + 1
        FinTQ
        Ecrire(n,"! = ",f)
Fin.
```

V. Synthèse

V.1. Passage d'une structure itérative à une autre



(*) : Le passage d'une boucle « répéter » ou « tantque » à une boucle « pour » n'est possible que si le nombre de parcours est connu à l'avance.

(**) : Lors du passage d'une boucle « pour » ou « tantque » à une boucle « répéter », faire attention aux cas particuliers (le traitement sera toujours exécuté au moins une fois).

V.2. Choix de la structure itérative

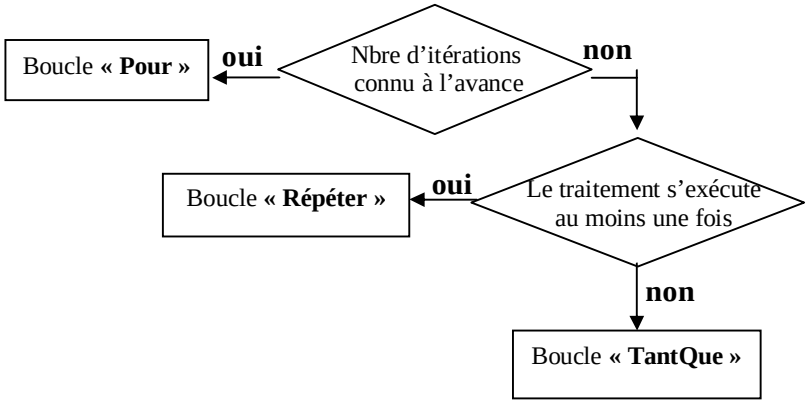


Figure 10. Critères de choix de la structure itérative adéquate

EXERCICES D'APPLICATION

Exercice 4.1

Ecrire un algorithme permettant de :

- Lire un nombre fini de notes comprises entre 0 et 20
- Afficher la meilleure note, la mauvaise note et la moyenne de toutes les notes.

Exercice 4.2

Calculer a^b avec a réel et b entier par multiplications successives.

Exercice 4.3

Ecrire un algorithme qui lit un entier positif et vérifie si ce nombre est premier ou non.

Remarque : un nombre premier n'est divisible que par 1 ou par lui-même.

Exercice 4.4

Ecrire un algorithme qui détermine tous les nombres premiers inférieurs à une valeur donnée.

Exercice 4.5

Ecrire un algorithme qui lit deux entiers A et B puis calcule et affiche leur PGCD en utilisant la méthode suivante :

- Si A = B ; PGCD(A,B) = A
- Si A > B ; PGCD(A,B) = PGCD(A–B,B)
- Si B > A ; PGCD(A,B) = PGCD(A,B-A)

Exemple : PGCD(18,45)=PGCD(18,27)=PGCD(18,9)=PGCD(9,9)=9

Exercice 4.6

Ecrire un algorithme qui calcule le PPCM (Plus Petit Commun Multiple) de 2 entiers A et B en utilisant la méthode suivante :

- Permuter, si nécessaire, les données de façon à ranger dans A le plus grand des 2 entiers ;
- Chercher le plus petit multiple de A qui est aussi multiple de B.

Exemple : PPCM(6,8) = PPCM(8,6) = 24.

Exercice 4.7

Ecrire un algorithme qui calcule et affiche les 10 premiers termes de la suite de Fibonacci.

La suite de Fibonacci est définie par :

- $F_0 = 1$
- $F_1 = 1$
- $F_n = F_{n-2} + F_{n-1}$ pour n >1

Exercice 4.8

Ecrire un algorithme qui calcule la somme harmonique $s = \sum_{i=1}^{n} \frac{1}{i}$; n est un entier positif lu à partir du clavier.

Exemple : Pour n = 3, s = 1 + 1/2 + 1/3 = 1.83.

Exercice 4.9 : nombres cubiques

Parmi tous les entiers supérieurs à 1, seuls 4 peuvent être représentés par la somme des cubes de leurs chiffres.

Ainsi, par exemple : $153 = 1^3 + 5^3 + 3^3$ est un nombre cubique

Ecrire un algorithme permettant de déterminer les 3 autres.

Note : les 4 nombres sont compris entre 150 et 410.

Exercice 4.10 : nombres parfaits

Un nombre parfait est un nombre présentant la particularité d'être égal à la somme de tous ses diviseurs, excepté lui-même.
Le premier nombre parfait est 6 = 3 + 2 + 1.

Ecrire un algorithme qui affiche tous les nombres parfaits inférieurs à 1000.

Exercice 4.11

Ecrire un algorithme qui simule le jeu suivant :

a- A tour de rôle, l'ordinateur et le joueur choisissent un nombre qui ne peut prendre que 3 valeurs : 0, 1 ou 2.
   ð L'instruction : N ← Random(3) réalise le choix de l'ordinateur

b- Si la différence entre les nombres choisis vaut :
   - 2, le joueur qui a proposé le plus grand nombre gagne un point
   - 1, le joueur qui a proposé le plus petit nombre gagne un point
   - 0, aucun point n'est marqué

c- Le jeu se termine quand l'un des joueurs totalise 10 points.

SOLUTIONS DES EXERCICES

Exercice 4.1

```
Algorithme Notes
Variables
        n, i : Entier
        note, min, max, s : Réel
Début
        Ecrire(”Entrer le nombre de notes : ”)
        Lire(n)     (*On suppose que n est toujours supérieur à zéro *)
        s ← 0
        max ← 0
        min ← 20
        Pour i de 1 à n Faire
                Ecrire(”Entrer une note : ”)   Lire(note)
                s ← s + note
                Si (note > max) Alors
                        max ← note
                FinSi
                Si (note < min) Alors
                        min ← note
                FinSi
        FinPour
        Ecrire(”Meilleure note = ”,max)
        Ecrire(”Mauvaise note = ”,min)
        Ecrire(”Moyenne des notes = ”,s/n)
Fin.
```

Exercice 4.2 : calcul de a^b par multiplications successives

```
Algorithme Puissance
Variables
        a, c : Réel
        b, i : Entier
Début
        Ecrire(”Entrer la valeur de a : ”)   Lire(a)
        Ecrire(”Entrer la valeur de b : ”)   Lire(b)
        c ← 1
        Pour i de 1 à  Abs(b) Faire
```

```
            c ← c * a
        FinPour
        Si (b < 0) Alors
            c ← 1 / c
        FinSi
        Ecrire(a," à la puissance ",b," = ",c)
Fin.
```

Exercice 4.3 : vérifier si un entier n est premier ou non

```
Algorithme Premier
Variables
        n, i, nb_div : Entier
Début
        Ecrire("Entrer un entier positif : ")   Lire(n)
        nb_div ← 0
        i ← 1
        TantQue (i <= n) Faire
            Si (n Mod i = 0) Alors
                nb_div ← nb_div + 1
            FinSi
            i ← i + 1
        FinTQ
        Si (nb_div <= 2) Alors
            Ecrire("C'est un nombre premier")
        Sinon
            Ecrire("Ce n'est pas un nombre premier")
        FinSi
Fin.
```

Exercice 4.4 : déterminer les nombres premiers inférieurs à un entier n

```
Algorithme Premiers
Variables
        n, i, j, nb_div : Entier
Début
        Ecrire("Entrer la valeur de n : ")   Lire(n)
        Pour i de 1 à n Faire
            nb_div ← 0
            Pour j de 1 à i Faire
```

```
                    Si (i Mod j = 0) Alors
                        nb_div ← nb_div + 1
                    FinSi
                FinPour
                Si (nb_div <= 2) Alors
                    Ecrire(i)
                FinSi
        FinPour
Fin.
```

Exercice 4.5 : calcul du PGCD de 2 entiers a et b

```
Algorithme PGCD
Variables
        a, b : Entier
Début
        Ecrire(”Entrer la valeur de a : ”)   Lire(a)
        Ecrire(”Entrer la valeur de b : ”)   Lire(b)
        Répéter
                Si (a > b) Alors
                        a ← a – b
                FinSi
                Si (b > a) Alors
                        b ← b - a
                FinSi
        Jusqu’à (a = b)
        Ecrire(”PGCD = ”, a)
Fin.
```

Exercice 4.6 : calcul du PPCM de 2 entiers a et b

```
Algorithme PPCM
Variables
        a, b, i, x : Entier
Début
        Ecrire(”Entrer la valeur de a : ”)
        Lire(a)
        Ecrire(”Entrer la valeur de b : ”)
        Lire(b)     (* On suppose que b est toujours supérieur à zéro *)
        Si (a < b) Alors
```

```
                x ← a
                a ← b
                b ← x
        FinSi
        i ← 1
        TantQue (((i*a) Mod b) # 0) Faire
                i ← i + 1
        FinTQ
        Ecrire(”PPCM = ”, i*a)
Fin.
```

Exercice 4.7 : calcul des n premiers termes de la suite de Fibonacci

```
Algorithme Fibo
Variables
        f0, f1, f, n, i : Entier
Début
        Ecrire(”Nombre de termes à calculer : ”)   Lire(n)
        f0 ← 1
        Ecrire(”f0 = ”,f0)
        f1 ← 1
        Ecrire(”f1 = ”,f1)
        Pour i de 2 à (n-1) Faire
                f ← f0 + f1
                Ecrire(”f”,i,” = ”,f)
                f0 ← f1
                f1 ← f
        FinPour
Fin.
```

Exercice 4.8 : calcul de la somme harmonique $s = \sum_{i=1}^{n} \frac{1}{i}$

```
Algorithme Somme
Variables
        n, i : Entier
        s : Réel
Début
        Ecrire(”Entrer la valeur de n : ”)   Lire(n)
        s ← 0
        Pour i de 1 à n Faire
                s ← s + 1/i
        FinPour
        Ecrire(”s = ”,s)
Fin.
```

Exercice 4.9 : nombres cubiques

```
Algorithme cubiques
Variables
        i, centaine, dizaine, unite : Entier
Début
        Pour i de 150 à 410 Faire
                centaine ← i Div 100
                dizaine ← (i Mod 100) Div 10
                unite ← i Mod 10
                Si ((centaine^3 + dizaine^3 + unite^3) = i)
                Alors
                        Ecrire(i,” est un nombre cubique”)
                FinSi
        FinPour
Fin.
```

Remarque : les nombres cubiques sont : 153, 370, 371 et 407

Exercice 4.10 : nombres parfaits inférieurs à 1000

```
Algorithme parfaits
Variables
        i, n, s, j : Entier
Début
        Pour i de 1 à 1000 Faire
                s ← 0
                Pour j de 1 à (i Div 2) Faire
                        Si ((i Mod j) = 0) Alors
                                s ← s + j
                        FinSi
                FinPour
                Si (s = i) Alors
                        Ecrire(i," est un nombre parfait")
                FinSi
        FinPour
Fin.
```

Remarque : les nombres parfaits inférieurs à 1000 sont : 6, 28 et 496

Exercice 4.11 : simulation d'un jeu

```
Algorithme jeu
Variables
        nb_j, nb_ord, tot_j, tot_ord : Entier
Début
        tot_j ← 0
        tot_ord ← 0
        TantQue (tot_j < 10) ET (tot_ord < 10) Faire
                nb_ord = random(3)
                Ecrire("Entrer un entier (0/1/2) : ")   Lire(nb_j)
                Si (Abs(nb_ord – nb_j) = 2) Alors
                        Si (nb_ord > nb_j) Alors
                                tot_ord ← tot_ord + 1
                        Sinon
                                tot_j ← tot_j + 1
                        FinSi
                FinSi
                Si (Abs(nb_ord – nb_j) = 1) Alors
```

```
                    Si (nb_ord < nb_j) Alors
                        tot_ord ← tot_ord + 1
                    Sinon
                        tot_j ← tot_j + 1
                    FinSi
                FinSi
            FinTQ
            Ecrire(”Total ordinateur = ”,tot_ord)
            Ecrire(”Total joueur = ”,tot_j)
Fin.
```

Leçon 5 : les chaînes de caractères

Objectif

Construire des algorithmes qui traitent des caractères et des chaînes de caractères.

I. Le type caractère

I.1. Définition

Ce type s'applique à tous les caractères du code ASCII (American Standard Code for Information Interchange). La liste comprend :

- Les lettres : "A" .. "Z","a".."z"
- Les chiffres : "0".."9"
- Les caractères spéciaux : "/" ; "*" ; "?" ; "&" ; etc.
- Les caractères de contrôle : <Retour Chariot> ; <Echap> ; etc.

Chaque caractère est défini par son numéro d'ordre unique compris entre 0 et 255 (voir annexe 2).

I.2 Fonctions standards sur les caractères

Fonction	Rôle	Exemple
Asc(c)	retourne le code ASCII du caractère c	i ← Asc("A") ð i contiendra 65
Car(i)	retourne le caractère dont le code ASCII est égal à i.	c ← Car(65) ðc contiendra "A"
Succ(c)	retourne le successeur du caractère c	c ← Succ("a") ð c contiendra "b"
Pred(c)	retourne le prédécesseur du caractère c	c ← Pred("b") ð c contiendra "a"
Majus(c)	retourne le majuscule du caractère c	c ← Majus("a") ðc contiendra "A"

Exemple

Ecrire un algorithme qui lit un caractère au clavier puis affiche son prédécesseur, son successeur et le code ASCII de son équivalent en majuscule.

Solution

```
Algorithme Caract
Variables
        c : Caractère
Début
        Ecrire(”Entrer un  caractère: ”)   Lire(c)
        Ecrire(pred(c))
        Ecrire(succ(c))
        Ecrire(asc(majus(c)))
 Fin.
```

Exercice 1

Que fait l’algorithme suivant :

```
Algorithme Upcase
Variables
        c1, c2 : Caractère
Début
        Ecrire(”Entrer un  caractère: ”)   Lire(c1)
        Si (asc(c1) >= 97) et (asc(c1) <= 122) Alors
        c2 ← car(asc(c1) - 32)
        Sinon
                c2 ← c1
        FinSi
        Ecrire(c2)
Fin.
```

Solution

Cet algorithme lit un caractère puis affiche son équivalent en majuscule.

Exercice 2
Ecrire un algorithme qui affiche une table ASCII des lettres minuscules sous la forme suivante :

```
Le code ASCII de a est 97
Le code ASCII de b est 98
                …
Le code ASCII de z est 122
```

Solution

```
Algorithme table_minus
Variables
        c : Caractère
Début
        Pour c de "a" à "z" Faire
                Ecrire("Le code ASCII de ",c, " est ",asc(c))
        FinPour
Fin.
```

Exercice 3
Ecrire un algorithme qui lit une lettre au clavier puis affiche s'il s'agit d'une consonne ou d'une voyelle.

Remarque : les voyelles sont : "A" ; "a" ; "E" ; "e" ; "I" ; "i"; "O" ; "o" ; "U" ; "u" ; "Y" ; "y".

Solution

```
Algorithme Cons_Voy
Variables
        c : Caractère
Début
        Répéter
                Ecrire("Entrer un caractère : ")        Lire(c)
        Jusqu'à (c >= "A" et c <= "Z") ou (c >= "a" et c <= "z")
        Si (Majus(c)="A") ou (Majus(c)="E") ou (Majus(c)="I")
        ou (Majus(c)="O") ou (Majus(c)="U") ou (Majus(c)="Y")
        Alors
                Ecrire(c, " est une voyelle")
        Sinon
                Ecrire(c, " est une consonne")
        FinSi
Fin.
```

Dans cet algorithme, le but de la boucle répéter est d'obliger l'utilisateur à entrer une lettre.

II. Le type chaîne de caractères

Une chaîne est une suite de caractères. La chaîne ne contenant aucun caractère est appelée *chaîne vide*.

II. 1. Déclaration d'une chaîne

```
Variables
        c : Caractère
        ch : Chaîne
        chn : Chaîne[20]
```

ð La variable ch peut contenir jusqu'à 255 caractères alors que chn peut contenir au maximum 20.

II. 2 Opérations sur les chaînes de caractères

a- la concaténation

C'est l'assemblage de deux chaînes de caractères en utilisant l'opérateur « + ».

Exemple

```
chn1 ← "Turbo"
chn2 ← "Pascal"
chn3 ← chn1+" "+chn2
```

ð la variable chn3 contiendra "Turbo Pascal"

b- les opérateurs relationnels (>, >=, <, <=, =, #)

Il est possible d'effectuer une comparaison entre deux chaînes de caractères, le résultat est de type booléen. La comparaison se fait caractère par caractère de la gauche vers la droite selon le code ASCII.

Exemples

- L'expression (”a” > ”A”) est vraie puisque le code ASCII de ”a” (97) est supérieur à celui de ”A” (65)
- L'expression (”programme” < ”programmation”) est fausse puisque ”e” > ”a”
- L'expression (”” = ” ”) est fausse (le vide est différent du caractère espace).

c- accès à un caractère dans une chaîne

Pour accéder à un caractère de la chaîne, il suffit d'indiquer le nom de la chaîne suivi d'un entier entre crochets qui indique la position du caractère dans la chaîne.

Exemple

```
chn ← ”Turbo Pascal”
c ← chn[7]
```

ð la variable c contiendra le caractère ”P”.

En général, **ch[i]** désigne le $i^{ème}$ caractère de la chaîne ch.

III. Procédures et fonctions standards sur les chaînes

III.1. Procédures standards

Procédure	Rôle	Exemple
Efface(Chaîne, P, N)	Enlève N caractères de Chaîne à partir de la position P donnée.	chn ← ”Turbo Pascal” efface(chn,6,7) ð chn contiendra ”Turbo”
Insert(Ch1, Ch2, P)	Insère la chaîne Ch1 dans la chaîne Ch2 à partir de la position P.	ch1 ← ”D” ch2 ← ”AA” insert(ch1,ch2,2) ð ch2 contiendra ”ADA”
Convch(Nbr, Ch)	Converti le nombre Nbr en une chaîne de caractères Ch.	n = 1665 convch(n,chn) ð chn contiendra la chaîne ”1665”

III.2. Fonctions standards

Fonction	Rôle	Exemple
Long(Chaîne)	Retourne la longueur de la chaîne.	chn ← ”Turbo Pascal” n ← Long(chn) ð n contiendra 12
Copie(Chaîne, P, N)	Copie N caractères de Chaîne à partir de la position P donnée.	ch1 ← ”Turbo Pascal” ch2 ← Copy(ch1,7,6) ð ch2 contiendra ”Pascal”
Position(Ch1, Ch2)	Retourne la position de la première occurrence de la chaîne Ch1 dans la chaîne Ch2.	ch1 ← ”as” ch2 ← ”Turbo Pascal” n ← Position(ch1,ch2) ð n contiendra 8

Exercice

Ecrire un algorithme « Palind » qui lit une chaîne de caractères et vérifie si cette chaîne est un palindrome ou non.

Un palindrome est un mot qui peut être lu indifféremment de droite à gauche ou de gauche à droite (**Exemples :** ”AZIZA”, ”LAVAL”, ”RADAR”, ”2002”, etc.)

Correction

```
Algorithme Palind
Variables
        ch : Chaîne
        i, L : Entier
        Pal : Booléen
Début
        Ecrire(”Entrer une chaîne non vide : ”)  Lire(ch)
        L ← long(ch)
        Pal ← Vrai
        i ← 1
        TantQue (i <= L/2) et (Pal) Faire
                Si (ch[i] = ch[L-i+1]) Alors
                        i ← i + 1
                Sinon
                        Pal ← Faux
```

```
            FinSi
        FinTQ
        Si (Pal) Alors
            Ecrire(ch, " est un palindrome")
        Sinon
            Ecrire(ch, " n'est pas un palindrome")
        FinSi
Fin.
```

EXERCICES D'APPLICATION

Exercice 5.1

Ecrire un algorithme qui lit une chaîne de caractères puis affiche son inverse.

Exemple : Si la chaîne entrée est "algo", l'algorithme doit afficher "ogla".

Exercice 5.2

Ecrire un algorithme qui lit une chaîne de caractères et renvoie son équivalent en majuscules.

Exercice 5.3

Ecrire un algorithme qui permet de compter le nombre de mots dans une phrase.

La phrase commence obligatoirement par une lettre et les mots sont séparés par des espaces.

Exercice 5.4

Ecrire un algorithme qui détermine et affiche le mot le plus long dans une phrase donnée.

Exercice 5.5

Ecrire un algorithme qui lit :

- Un mot (chaîne de caractères formée uniquement de lettres)
- Une lettre

puis affiche le nombre d'apparitions de la lettre dans le mot.

Exercice 5. 6 : conversion décimal ð binaire

Ecrire un algorithme qui lit en entier positif puis affiche son équivalent en binaire (base 2).

Exemple : $(23)_{10} = (10111)_2$

SOLUTIONS DES EXERCICES

Exercice 5.1 : affichage de l'inverse d'une chaîne de caractères

```
Algorithme inverse
Variables
        i, L : Entier
        ch1, ch2 : Chaîne
Début
        Ecrire("Entrer une chaîne : ")   Lire(ch1)
        L ← Long(ch1)
        ch2 ← ""
        Pour i de 1 à L Faire
                ch2 ← ch1[i] + ch2
        FinPour
        Ecrire("Inverse de la chaîne = ",ch2)
Fin.
```

Exercice 5.2 : conversion d'une chaîne de caractères en majuscule

```
Algorithme Majuscule
Variables
        i, L : Entier
        ch1, ch2 : Chaîne
Début
        Ecrire("Entrer une chaîne : ")   Lire(ch1)
        L ← Long(ch1)
        ch2 ← ""
        Pour i de 1 à L Faire
                ch2 ← ch2 + Majus(ch1[i])
        FinPour
        Ecrire("Chaîne en majuscule = ",ch2)
Fin.
```

Exercice 5.3 : comptage du nombre de mots dans une phrase

```
Algorithme Comptage_Mots
Variables
        i, L, nb_mot : Entier
        phrase : Chaîne
```

```
Début
        Ecrire(”Entrer une phrase non vide : ”)   Lire(phrase)
        L ← Long(phrase)
        nb_mot ← 1
        Pour i de 1 à L Faire
                Si (phrase[i] = ” ”) Alors
                        nb_mot ← nb_mot + 1
                FinSi
        FinPour
        Ecrire(”Nombre de mots = ”,nb_mot)
Fin.
```

Exercice 5.4 : détermination du mot le plus long dans une phrase

```
Algorithme Plus_Long_Mot
Variables
        i, j, L : Entier
        phrase, mot, motpl : Chaîne
Début
        Ecrire(”Entrer une phrase : ”)   Lire(phrase)
        L ← Long(phrase)
        motpl ← ””
        i ← 1
        TantQue (i <= L) Faire
                mot ← ””
                j ← i
                TantQue ((j <= L) et (phrase[j] # ” ”)) Faire
                        mot ← mot + phrase[j]
                        j ← j + 1
                FinTQ
                Si (long(mot) > long(motpl)) Alors
                        motpl ← mot
                FinSi
                i ← j + 1
        FinTQ
        Ecrire(”Le mot le plus long est = ”,motpl)
Fin.
```

Exercice 5.5 : nombre d'occurrences d'une lettre dans un mot

```
Algorithme fréquence
Variables
        i, L, nb : Entier
        mot : Chaîne
        lettre : Caractère
Début
        Ecrire("Entrer un mot : ")   Lire(mot)
        Ecrire("Entrer une lettre : ")   Lire(lettre)
        L ← Long(mot)
        nb ← 0
        Pour i de 1 à L Faire
                Si (mot[i] = lettre) Alors
                        nb ← nb + 1
                FinSi
        FinPour
        Ecrire(lettre," apparaît ",nb, " fois dans ",mot)
Fin.
```

Exercice 5. 6 : conversion décimal ð binaire

```
Algorithme Decimal_Binaire
Variables
        i, n, quotient, reste : Entier
        c : Caractère
        equiv : Chaîne
Début
        Ecrire("Entrer votre entier : ")   Lire(n)
        equiv = ""
        TantQue (n # 0) Faire
                reste ← n Mod 2
                convch(reste,c)
                equiv ← c + equiv
                n ← n Div 2
        FinTQ
        Ecrire("Nombre binaire équivalent = ",equiv)
Fin.
```

Leçon 6 : Procédures et fonctions

Objectif

Appliquer la démarche de programmation modulaire pour construire des algorithmes structurés en procédures et fonctions.

I. Introduction

Pour maîtriser la complexité d'un problème, il est préférable de procéder à une conception modulaire qui consiste à le décomposer en plusieurs sous-problèmes. Ces sous-problèmes peuvent à leurs tours être décomposés jusqu'à aboutir à des traitements élémentaires simples. Ensuite, chaque sous-problème sera résolu par un sous-programme qui peut être une *procédure* ou une *fonction*.

A titre d'exemple, un programme de gestion de scolarité peut être découpé en plusieurs modules : inscription, suivi des absences, examens, diplômes, etc. (figure 11).

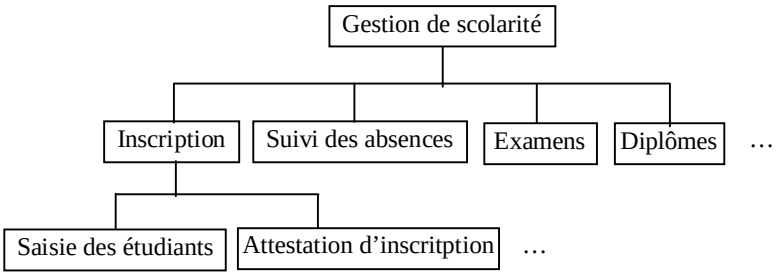


Figure 11. Décomposition d'un programme en sous-programmes

Les modules développés peuvent être réutilisés plusieurs fois dans le même programme ou dans d'autres programmes une fois intégrés à des bibliothèques.

II. Procédure simple

Une procédure est une structure de programme autonome qui permet d'effectuer une tâche bien précise et qui peut transmettre un certain nombre de résultats.

Exemple

```
Algorithme Proc     (* version 1 *)
Constantes
        titre = "Turbo Pascal"

Procédure trait
Variables
        i : Entier
Début
        Pour i de 1 à 12 Faire
                Ecrire("-")
        FinPour
Fin

Début
        Ecrire(titre)
        trait
Fin.
```

La procédure « trait » permet de souligner le titre "Turbo Pascal".

Remarques

1. La structure d'une procédure est anologue à celle d'un algorithme. Elle possède une entête, une partie déclarative et un corps.
2. Pour appeler une **procédure** simple, il suffit d'écrire le nom de cette procédure.
3. Une **ressource locale** (privée) est déclarée dans une procédure et ne peut être utilisée qu'à l'intérieur de celle-ci.

Exemple : i est une variable locale à la procédure trait.

4. Une **ressource globale** (publique) est déclarée au début de l'algorithme. Elle peut être utilisée dans le corps principal de l'algorithme ou par les différents modules.

Exemple : titre est une constante globale.

Il est fortement recommandé d'utiliser autant que possible des variables locales pour rendre les modules plus autonomes et par conséquent utilisables dans n'importe quel programme.

III. Procédure paramétrée

Lors de l'appel, il est souvent nécessaire d'échanger des informations entre la procédure et le programme ou le module appelant. C'est le rôle des *paramètres*.

A titre d'exemple, la procédure trait de l'algorithme suivant permet d'afficher un trait dont la longueur effective est précisée lors de l'appel, ce qui permet de souligner n'importe quel titre :

```
Algorithme Proc     (* version 2 *)
Variables
        titre : Chaîne
        L : Entier

Procédure Trait(n : entier)
Variables
        i : Entier
Début
        Pour i de 1 à n Faire
                Ecrire("-")
        FinPour
Fin

Début
        Ecrire("Entrer un titre : ")        Lire(titre)
        L ← Long(titre)
        Ecrire(titre)
        Trait(L)
Fin.
```

Remarques

1. L'entête d'une procédure paramétrée comporte, en plus du nom de la procédure, une liste de **paramètres formels** (fictifs) sous la forme suivante :

Procédure Nom_Proc(pf1 : type 1 ; pf2 : type 2 ; ...)

2. Pour appeler une procédure paramétrée, il faut écrire le nom de la procédure suivi d'une liste de **paramètres effectifs** (réels) sous la forme suivante :

Nom_Proc(pef1, pef2, ...)

Lors de l'appel, il faut respecter le nombre, l'ordre et le type des paramètres.

Exercice

Ecrire une procédure « ReptCar » qui permet d'afficher un caractère un certain nombre de fois.

Exemple : l'appel de procédure **ReptCar("*",5)** doit afficher "*****".

Solution

```
Procédure ReptCar(c : Caractère ; n : Entier)
Variables
        i : Entier
Début
        Pour i de 1 à n Faire
                Ecrire(c)
        FinPour
Fin
```

IV. Modes de passage de paramètres

La substitution des paramètres effectifs aux paramètres formels s'appelle *passage de paramètres*. Elle correspond à un transfert d'informations entre le programme ou le module appelant et la procédure appelée.

Notons d'abord que les paramètres d'une procédure peuvent être de type donnée, résultat ou donnée-résultat.

- Un **paramètre donnée** est une information nécessaire pour réaliser une tâche. Il ne subit aucun changement au cours de l'exécution de la procédure.

- Un **paramètre résultat** est un aboutissement de l'action, sa valeur n'est pas significative avant le début de l'exécution de la procédure.

- Un **paramètre donnée-résultat** est à la fois une donnée et un résultat, c'est à dire une information dont la valeur sera modifiée par la procédure.

Le tableau suivant indique pour chaque type de paramètre le sens de transfert et le mode de passage utilisé :

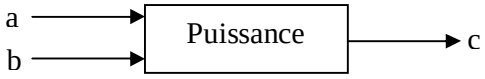
Type de paramètre	**Sens de transfert**	**Mode de passage**
Donnée	Programme ⇒ Procédure	Par valeur
Résultat	Programme ⇐ Procédure	Par variable (par adresse)
Donnée-Résultat	Programme ⇔ Procédure	

En algorithmique, le mot **Var** inséré devant un paramètre formel signifie que ce paramètre est passé par variable (paramètre de type résultat ou donnée-résultat).

Exemple 1

Ecrire une procédure « puissance » qui calcule $c = a^b = a \times a \times \ldots \times a$ (b fois) ; a et b étant des entiers positifs.

Solution



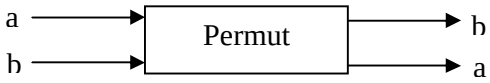

```
Procédure puissance(a, b : Entier ; Var c : Entier)
Variables
        i : Entier
Début
        c ← 1
        Pour i de 1 à b Faire
```

```
            c ← c * a
        FinPour
Fin.
```

Exemple 2

Ecrire une procédure « permut » qui permet d'échanger les valeurs de 2 entiers a et b.

Solution




```
Procédure permut(Var a, b : Entier)
Variables
        x : Entier
Début
        x ← a
        a ← b
        b ← x
Fin.
```

Remarque

Dans le mode de passage par valeur, les modifications effectuées sur le paramètre formel n'affectent pas la valeur du paramètre effectif (la procédure travaille sur une copie de la variable) alors que dans le mode de passage par variable, toute modification sur le paramètre formel affecte également le paramètre effectif correspondant (la procédure travaille sur la copie originale).

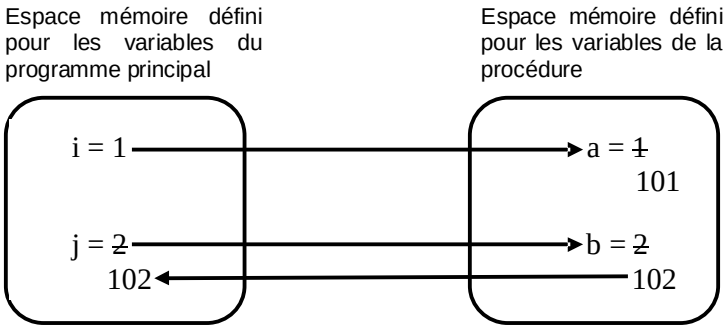
Exercice

Faire la trace d'exécution de l'algorithme suivant :

```
Algorithme Paramètre
Variables
        i, j : Entier
Procédure transmission(a : Entier ; Var b : Entier)
```

```
Début
        a ← a + 100
        b ← b + 100
        Ecrire(”a = ”,a, ”b = ”,b)
Fin
Début
        i ← 1
        j ← 2
        transmission(i,j)
        Ecrire(”i = ”,i, ”j = ”,j)
Fin.
```

Solution



A la fin de l'exécution de cet algorithme, on aura comme résultat :

a = 101	b = 102
i = 1	j = 102

V. Les fonctions

V.1. Définition

Une fonction est une procédure particulière qui admet un seul paramètre résultat de type simple (entier, réel, caractère, etc.).

Une fonction possède un type qui est celui de son résultat.

V.2. Structure d'une fonction

Fonction Nom_Fonc(<paramètres formels>) : Type

```
        <Déclarations>
Début
        <Séquence d'instructions>
        Nom_Fonc ← résultat
Fin
```

Remarques

1. Le nom de la fonction joue un double rôle, c'est à la fois l'identifiant de la fonction et une variable locale.
2. Dans une fonction, le passage de tous les paramètres se fait **par valeur** (tous les paramètres sont des données).

Exemple

Ecrire une fonction « min » qui retourne le minimum de 2 entiers a et b.

Solution

```
Fonction min(a, b : Entier) : Entier
Variables
        c : Entier
Début
        Si (a <= b) Alors
                c ← a
        Sinon
                c ← b
        FinSi
        min ← c
Fin
```

V.3. Appel d'une fonction

L'appel d'une fonction se fait toujours dans une expression sous la forme :

Nom_Fonc(<liste de paramètres effectifs>)

Exemples

- x ← 3
  y ← 5
  z ← min(a,b)
- z ← min(3,5)

- t ← (7 + min(x,y))/2
- Ecrire(min(x,y))
- Si (min(x,y) > 0) Alors …

Exercice

Comment utiliser la fonction min pour afficher le minimum de 3 entiers x, y et z en une seule instruction et sans utiliser de variables supplémentaires ?

Solution

Ecrire(min(min(x,y),z))

EXERCICES D'APPLICATION

Exercice 6.1

1. On appelle **bigramme** une suite de deux lettres. Ecrire une procédure qui calcule le nombre d'occurrences d'un bigramme dans une chaîne de caractères.

2. Peut-on transformer cette procédure en fonction ? si oui écrire cette fonction.

Exercice 6.2

Ecrire une fonction *Triangle* qui permet de vérifier si les 3 nombres a, b et c peuvent être les mesures des côtés d'un triangle rectangle.

Remarque : D'après le théorème de Pythagore, si a, b et c sont les mesures des côtés d'un triangle rectangle, alors $a^2 = b^2 + c^2$ ou $b^2 = a^2 + c^2$ ou $c^2 = a^2 + b^2$.

Exercice 6.3

Ecrire une fonction PGCD_Euc qui retourne le PGCD de 2 entiers a et b en utilisant l'algorithme d'Euclide :

L'algorithme d'Euclide consiste à répéter plusieurs fois le traitement :

PGCD(a,b) = PGCD(b,a Mod b)

jusqu'à obtenir PGCD(x,0). Le PGCD est alors x.

Exemple : PGCD(36,16) = PGCD(16,4) = PGCD(4,0) = 4.

SOLUTIONS DES EXERCICES

Exercice 6.1 : nombre d'occurrences d'un bigramme dans une chaîne

```
Procédure fréqunece(bigram:Chaîne[2] ; chn :Chaîne ; Var nb : Entier)
Variables
        i, L : Entier
Début
        L ← Long(chn)
        nb ← 0
        Pour i de 1 à (L-1) Faire
                Si (chn[i]=bigram[1]) ET (chn[i+1]=bigram[2])
        Alors
                        nb ← nb + 1
                FinSi
        FinPour
Fin
```

Cette procédure possède un seul paramètre résultat de type entier, donc elle peut être remplacée par une fonction.

```
Fonction fréqunece(bigram:Chaîne[2] ; chn :Chaîne) : Entier
Variables
        i, L : Entier
Début
        L ← Long(chn)
        nb ← 0
        Pour i de 1 à (L-1) Faire
                Si (chn[i]=bigram[1]) et (chn[i+1]=bigram[2])
        Alors
                        nb ← nb + 1
                FinSi
        FinPour
        fréquence ← nb
Fin
```

Exercice 6.2 : triangle rectangle

```
Fonction triangle (a, b, c : Réel) : Booléen
Début
        Si (a^2=b^2+c^2) OU(b^2=a^2+c^2) OU
        (c^2=a^2+b^2) Alors
                triange ← Vrai
        Sinon
                triangle ← Faux
        FinSi
Fin
```

Exercice 6.3 : calcul du PGCD(a,b) par la méthode d'Euclide

```
Fonction PGCD_Euc(a, b : Entier) : Entier
Variables
        r : Entier
Début
        TantQue (b # 0) Faire
                r ← a Mod b
                a ← b
                b ← r
        FinTQ
        PGCD_Euc ← a
Fin
```

Leçon 7 : Les tableaux

Objectif

Développer des algorithmes de traitement de tableaux.

I. Introduction

Supposons que nous avons à déterminer à partir de 30 notes fournies en entrée, le nombre d'étudiants qui ont une note supérieure à la moyenne de la classe.

Pour parvenir à un tel résultat, nous devons :

1. Lire les 30 notes
2. Déterminer la moyenne de la classe : m
3. Compter combien parmi les 30 notes sont supérieures à la moyenne m.

Il faut donc conserver les notes en mémoire afin qu'elles soient accessibles durant l'exécution du programme.

Solution 1 : utiliser 30 variables réelles nommées x1, x2, …, x30

Cette façon de faire présente deux inconvénients :

- il faut trouver un nom de variable par note ;
- il n'existe aucun lien entre ces différentes valeurs. Or, dans certains cas, on est appelé à appliquer le même traitement à l'ensemble ou à une partie de ces valeurs.

Solution 2 : utiliser la notion de **tableau** qui consiste à :

- attribuer un seul nom à l'ensemble des 30 notes, par exemple Tnote,
- repérer chaque note par ce nom suivi entre crochets d'un numéro entre 1 et 30 : Tnote[1], Tnote[2], … Tnote[30].

II. Tableaux unidimensionnels

Un tableau à une dimension, appelé aussi **vecteur**, est une structure de données constituée d'un nombre fini d'éléments *de même type* et directement accessibles par leurs indices ou indexes.

II.1. Déclaration d'un tableau

Pour définir une variable de type tableau, il faut préciser :

- le nom (identifiant du tableau)
- l'indice (généralement de type entier ou caractère)
- le type des éléments (entier, réel, caractère, etc.)

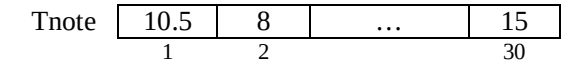
on note :

```
Variables
    Nom_tab : Tableau [PremInd..DernInd] de Type_éléments
```

Exemple

Tnote : Tableau[1..30] de Réel

Schématiquement, ce tableau peut être représenté comme suit :



Remarque

Il est également possible de définir un type tableau comme dans l'exemple suivant :

```
Constantes
        n = 10
Types
        Tab = Tableau[1..n] de Entier
Variables
        T : Tab
```

II.2. Identification d'un élément du tableau

Un élément dans un tableau est identifié de la façon suivante :

NomTab[position de l'élément]

ð Cela traduit bien l'accès direct aux éléments du tableau.

Ainsi, **Tnote[3]** désigne la note du 3ème étudiant et d'une façon générale, **T[i]** désigne le ième élément du tableau T.

L'indice i peut être une valeur, une variable ou une expression dont la valeur appartient à l'intervalle des indices.

II.3. Remplissage d'un tableau

Un tableau peut être rempli *élément par élément* à l'aide d'une série d'affectations :

```
T[1] ← Valeur 1
T[2] ← Valeur 2
…
T[n] ← Valeur n
```

Il est également possible de lire les éléments du tableau à partir du clavier grâce à une procédure :

```
Procédure remplir(Var T : tab)
Variables
        i : Entier
Début
        Pour i de 1 à n Faire
                Ecrire("Entrer un entier : ")
                Lire(T[i])
        FinPour
Fin
```

II.4. Affichage des éléments d'un tableau

L'affichage des éléments d'un tableau se fait également élément par élément. Seulement, le tableau constitue ici un paramètre donné et non pas un résultat comme dans la procédure de remplissage.

```
Procédure afficher(T : tab)
Variables
        i : Entier
Début
        Pour i de 1 à n Faire
                Ecrire(T[i])
        FinPour
Fin
```

Exercice

Soit T un tableau contenant n éléments de type entier. Ecrire une fonction MinTab qui retourne le plus petit élément de ce tableau.

Solution

On suppose initialement que le premier élément du tableau est le minimum puis on le compare à tous les autres éléments. Chaque fois qu'on trouve un élément qui lui est inférieur, ce dernier devient le minimum.

```
Fonction  MinTab(T : tab) : Entier
Variables
        m,i : Entier
Début
        m ← T[1]
        Pour i de 2 à n Faire
                Si (T[i] < m) Alors
                        m ← T[i]
                FinSi
        FinPour
        MinTab ← m
Fin
```

II.5. Recherche séquentielle d'un élément dans un tableau

Soit T un tableau contenant n éléments de type entier.

On veut écrire une procédure dont l'entête sera :

Procédure recherche(T : Tab, x : Entier)

Cette procédure affiche :

- l'indice de la première occurrence de x dans T si $x \in T$
- le message "élément introuvable…" si $x \notin T$

Principe

Comparer x aux différents éléments du tableau jusqu'à trouver x ou atteindre la fin du tableau.

```
Procédure Recherche(T : Tab ; x : Entier)
Variables
        i : Entier
Début
        i ← 0
        Répéter
                i ← i + 1
        Jusqu'à (T[i] = x) ou (i > n)
        Si (T[i] = x) Alors
                Ecrire("Indice = ",i)
        Sinon
                Ecrire("Elément introuvable…")
        FinSi
Fin
```

II.6. Algorithmes de tri

Il existe plusieurs méthodes de tri parmi lesquelles on peut citer :

- tri à bulle
- tri par sélection
- tri par insertion
- tri par comptage
- tri shell
- tri rapide (quick sort).

a- Tri à bulle

Soit T un tableau de n entiers. La méthode de tri à bulles nécessite deux étapes :

- Parcourir les éléments du tableau de 1 à (n–1) ; si l'élément i est supérieur à l'élément (i+1), alors on les permute.
- Le programme s'arrête lorsqu'aucune permutation n'est réalisable après un parcours complet du tabeau.

```
Procédure Tri_Bulle (Var T : Tab)
Variables
        i, x : Entier
        échange : Booléen
Début
        Répéter
                échange ← Faux
                Pour i de 1 à (n-1) Faire
                        Si (T[i] > T[i+1]) Alors
                                x ← T[i]
                                T[i] ← T[i+1]
                                T[i+1] ← x
                                échange ← Vrai
                        FinSi
                FinPour
        Jusqu'à (échange = Faux)
Fin
```

Trace d'éxécution

Tableau initial	6	4	3	5	2
Après la 1ère itération	4	3	5	2	6
Après la 2ème itération	3	4	2	5	6
Après la 3ème itération	3	2	4	5	6
Après la 4ème itération	2	3	4	5	6

b- Tri par sélection (par minimum)

C'est la méthode de tri la plus simple, elle consiste à :

- chercher l'indice du plus petit élément du tableau T[1..n] et permuter l'élément correspondant avec l'élément d'indice 1
- chercher l'indice du plus petit élément du tableau T[2..n] et permuter l'élément correspondant avec l'élément d'indice 2
- …
- chercher l'indice du plus petit élément du tableau T[n-1..n] et permuter l'élément correspondant avec l'élément d'indice (n-1).

```
Procédure Tri_Selection(Var T : Tab)
Variables
        i, j, x, indmin : Entier
Début
        Pour i de 1 à (n-1) Faire
                indmin ← i
                Pour j de (i+1) à n Faire
                        Si (T[j] < T[indmin]) Alors
                                indmin ← j
                        FinSi
                FinPour
                x ← T[i]
                T[i] ← T[indmin]
                T[indmin] ← x
        FinPour
Fin
```

Trace d'éxécution

Tableau initial	6	4	2	3	5

Après la 1ère itération	2	4	6	3	5
Après la 2ème itération	2	3	6	4	5
Après la 3ème itération	2	3	4	6	5
Après la 4ème itération	2	3	4	5	6

c- Tri par insertion

Cette méthode consiste à prendre les éléments de la liste un par un et insérer chacun dans sa bonne place de façon que les éléments traités forment une sous-liste triée.

Pour ce faire, on procède de la façon suivante :

- comparer et permuter si nécessaire T[1] et T[2] de façon à placer le plus petit dans la case d'indice 1
- comparer et permuter si nécessaire l'élément T[3] avec ceux qui le précèdent dans l'ordre (T[2] puis T[1]) afin de former une sous-liste triée T[1..3]
- …
- comparer et permuter si nécessaire l'élément T[n] avec ceux qui le précèdent dans l'ordre (T[n-1], T[n-2], …) afin d'obtenir un tableau trié.

```
Procédure Tri_Insertion(Var T : Tab)
Variables
        i, j, x, pos : Entier
Début
        Pour i de 2 à n Faire
                pos ← i - 1
                TantQue (pos>=1) et (T[pos]>T[i]) Faire
                        pos ← pos – 1
                FinTQ
                pos ← pos + 1
                x ← T[i]
                Pour j de (i-1) à pos [pas = -1] Faire
                        T[j+1] ← T[j]
```

```
                FinPour
                T[pos] ← x
            FinPour
Fin
```

[Pas = -1] signifie que le parcours se fait dans le sens décroissant.

Trace d'éxécution

Tableau initial	6	4	3	5	2
Après la 1ère itération	4	6	3	5	2
Après la 2ème itération	3	4	6	5	2
Après la 3ème itération	3	4	5	6	2
Après la 4ème itération	2	3	4	5	6

II.7. Recherche dichotomique

Soit T un tableau contenant n éléments triés dans le sens croissant :

T	1	3	5	5	8
	1	2	3	4	5

ð $\forall i \in [1, n-1], T[i] <= T[i+1]$

On veut écrire une procédure dont l'entête est de la forme :

Procédure Rechdicho(T : tab ; x : Entier)

et qui affiche l'indice de la première apparition de x dans le tableau s'il existe. Sinon, elle affiche le message "élément introuvable…".

Principe

Le but de la recherche dichotomique est de diviser l'intervalle de recherche par 2 à chaque itération. Pour cela, on procède de la façon suivante :

Soient premier et dernier les extrémités gauche et droite de l'intervalle dans lequel on cherche la valeur x, on calcule M, l'indice de l'élément médian :

ð M = (premier + dernier) div 2

Il y a 3 cas possibles :

- *x = T[M]* : l'élément de valeur x est trouvé, la recherche est terminée
- *x < T[M]* : l'élément x, s'il existe, se trouve dans l'intervalle [premier..M-1]
- *x > T[M]* : l'élément x, s'il existe, se trouve dans l'intervalle [M+1..dernier]

La recherche dichotomique consiste à itérer ce processus jusqu'à ce que l'on trouve x ou que l'intervalle de recherche soit vide.

```
Procédure Rechdicho(T : Tab , x : entier)
Variables
        premier, milieu, dernier : Entier
        trouve : Booléen
Début
        premier ← 1
        dernier ← n
        trouve ← Faux
        Répéter
                milieu ← (premier + dernier) div 2
                Si (x < T[milieu]) Alors
                        dernier ← milieu – 1
                Sinon
                        Si (x > T[milieu]) Alors
                                premier ← milieu + 1
                        Sinon
                                trouve ← Vrai
                        FinSi
                FinSi
        Jusqu'à (trouve = Vrai) ou (premier > dernier)
        Si (trouve = Vrai) Alors
```

```
                Ecrire(”Indice = ”,milieu)
        Sinon
                Ecrire(”Elément introuvable…”)
        FinSi
Fin.
```

Evaluation de l’algorithme

Dans le cas le plus défavorable, l’algorithme se termine lorsque le nombre d’éléments p présents dans l’intervalle [premier .. dernier] devient nul. A chaque itération, p est au moins divisé par 2. On est donc certain d’obtenir le résultat en au plus i itérations tel que $n = 2^i$. D’où $i=\log_2 n$ (fonction inverse).

À titre d’exemple, si le tableau trié contient 1024 éléments, il faudra au plus 1025 itérations avec l’algorithme séquentiel et au plus 10 avec l’algorithme dichotomique.

III. Tableaux multidimensionnels

Les tableaux multidimensionnels sont des tableaux qui contiennent des tableaux. Par exemple, le tableau bidimensionnel (3 lignes et 4 colonnes) suivant est en fait un tableau comportant 3 éléments, chacun d’entre eux étant un tableau de 4 éléments :

	1	**2**	**3**	**4**
1				
2				
3				

Cette matrice peut être définie de la façon suivante :

```
Types
        Mat : Tableau[1..3,1..4] de Réel
Variables
        Matrice : Mat
```

Chaque élément de la matrice est repéré par deux indices :

- le premier indique le numéro de la ligne
- le second indique le numéro de la colonne.

Ainsi, **Matrice[2,4]** désigne l'élément situé à la $2^{ème}$ ligne et la $4^{ème}$ colonne.

Remarque

Cette représentation est arbitraire, on a pu considérer que le premier indice désigne la colonne et le second désigne la ligne. Dans ce cas, l'élément Matrice[2,4] n'existe plus.

III.1. Remplissage d'un tableau à deux dimensions

Le remplissage d'un tableau bidimensionnel à n lignes et m colonnes se fait à peu près de la même façon qu'un tableau unidimensionnel. Seulement, il est nécessaire d'utiliser deux boucles imbriquées correspondant chacune à l'indice d'une dimension :

```
Procédure remplir(Var matrice : Mat)
Variables
        i, j : Entier
Début
        Pour i de 1 à n Faire
                Pour j de 1 à m Faire
                        Ecrire("Entrer un entier :")
                        Lire(Matrice[i,j])
                FinPour
        FinPour
Fin
```

III.2. Transposition d'une matrice carrée

Une matrice carrée est une matrice à n lignes et n colonnes.

L'opération de transposition consiste à inverser les lignes et les colonnes en effectuant une symétrie par rapport à la diagonale principale de la matrice.

Exemple

M

1	2	3
4	5	6
7	8	9

Devient

1	4	7
2	5	8
3	6	9

```
Procédure Transpose(Var M : Mat)
Variables
        i, j, x : Entier
Début
        Pour i de 1 à n Faire
                Pour j de (i+1) à n Faire
                        x ← M[i,j]
                        M[i,j] ← M[j,i]
                        M[j,i] ← x
                FinPour
        FinPour
Fin
```

III.3. Somme de deux matrices

Soient M1 et M2 deux matrices à n lignes et m colonnes.

On veut écrire une procédure qui calcule les éléments de la matrice M3=M1+M2

Exemple

M1

1	2	3
4	5	6

M2

2	5	3
3	0	1

donnent M3 =

3	7	6
7	5	7

```
Procédure SomMat(M1, M2 : Mat ; Var M3 : Mat)
```

```
Variables
        i, j : Entier
Début
        Pour i de 1 à n Faire
                Pour j de 1 à m Faire
                        M3[i,j] ← M1[i,j]+ M2[i,j]
                FinPour
        FinPour
Fin
```

III.4. Produit de deux matrices

Soient M1 une matrice ayant n lignes et m colonnes
M2 une matrice ayant m lignes et p colonnes

On veut écrire une procédure qui calcule les éléments de la matrice M3=M1×M2.

Notons d'abord que le nombre de colonnes de M1 doit être égal au nombre de lignes de M2.

Le produit M3 = M1 × M2 est défini comme une matrice ayant n lignes et p colonnes et dont les éléments sont calculés par la formule :

$$M3_{i,j} = M1_{i,1}M2_{1,j} + M1_{i,2}M2_{2,j} + \ldots + M1_{i,m}M2_{m,j}$$

soit $$M3_{i,j} = \sum_{k=1}^{m} M1_{i,k}M2_{K,j}$$

où $M1_{i,k}$, $M2_{k,j}$ et $M3_{i,j}$ sont respectivement les éléments des matrices M1, M2 et M3.

Exemple

M1

1	2	3
4	0	5

M2

2	1
3	0
1	4

donnent M3 =

11	13
13	24

```
Procédure ProdMat(M1 : Mat1; M2 : Mat2; Var M3 : Mat3)
```

```
Variables
        i, j, k : Entier
Début
        Pour i de 1 à n Faire
                Pour j de 1 à p Faire
                        M3[i,j] ← 0
                        Pour k de 1 à m Faire
                                M3[i,j]←M3[i,j]+M1[i,k]*M2[k,j]
                        FinPour
                FinPour
        FinPour
Fin
```

EXERCICES D'APPLICATION

Exercice 7.1

Soit T un tableau contenant n éléments de type entier et x un entier quelconque.

Ecrire une fonction **Fréquence(T : Tab ; x : Entier) : Entier** qui retourne le nombre d'apparitions de x dans le tableau T.

Exercice 7.2

Ecrire une procédure permettant d'éclater un tableau T d'entiers en deux tableaux :

- TP qui contiendra les éléments positifs de T
- TN qui contiendra les éléments négatifs de T.

Exercice 7.3

Un instituteur cherche à vérifier si ses élèves ont appris à réciter l'alphabet lettre par lettre dans l'ordre. Pour ceci, Il vous demande de lui développer l'algorithme d'un programme permettant d'évaluer chaque élève de la façon suivante :

1- Le programme demande à chaque élève de remplir un tableau nommé réponse par les lettres de l'alphabet dans l'ordre

2- Ensuite, le programme examine ce tableau élément par élément :

- Si la lettre est dans sa place, il l'accepte
- Sinon, il la remplace par la lettre adéquate et incrémente le nombre de fautes

3- Enfin, le programme affiche le nombre total de fautes.

Exercice 7.4 : produit scalaire de 2 vecteurs

Ecrire une fonction *ProdScal* qui calcule le produit scalaire de deux vecteurs U et V représentés par deux tableaux.

Le produit scalaire de deux vecteurs :

$U = (x_1, x_2,\ldots, x_n)$ et $V = (y_1, y_2,\ldots, y_n)$

est défini par :

$$U.V = x_1y_1 + x_2y_2 + \ldots + x_ny_n = \sum_{i=1}^{n} x_i y_i$$

Exercice 7.5 : norme d'un vecteur

Ecrire une fonction *NormVect* qui calcule la norme d'un vecteur et qui utilise la fonction *ProdScal* de l'exercice précédent.

La norme d'un vecteur $U = (x_1, x_2,\ldots, x_n)$

est défini par : $\|U\| = \sqrt{x_1^2 + x_2^2 + \ldots + x_n^2}$

Exercice 7.6

Ecrire une procédure qui permet de remplir un tableau Fib par les 10 premiers termes de la suite de Fibonacci.

La suite de Fibonacci est définie par :

- $F_0 = 1$
- $F_1 = 1$

- $F_n = F_{n-2} + F_{n-1}$ pour n >1

Exercice 7.7

Ecrire un algorithme permettant de trouver tous les nombres premiers inférieurs à 400 et qui utilise la méthode suivante :

- Créer un tableau T pouvant contenir 400 entiers
- Initialiser chaque élément du tableau à son indice c'est-à-dire T[1]=1 ; T[2]=2 ; … T[400]=400
- Remplacer tous les multiples de 2 par 0 sauf 2
- Chercher le prochain élément différent de 0 dans le tableau c'est à dire T[3] et remplacer tous les multiples de 3 par 0 sauf 3
- Continuer ce processus jusqu'à avoir T[i] ≥ 20 (20 étant la racine carrée de 400)
- Afficher tous les éléments non nuls de T.

Exercice 7.8 : compression d'un vecteur

Soit un vecteur U1 de composantes x_1, x_2, …, x_n non nulles et un vecteur L de même longueur dont les composantes sont 0 ou 1.

Ecrire une procédure qui fait la compression de U1 par L. Le résultat est un vecteur U2 dont les composantes sont, dans l'ordre, celles de U1 pour lesquelles la composante de L vaut 1.

Exemple

U1	1	2	3	4	5	6	7
L	0	1	1	0	1	0	1
U2	2	3	5	7	0	0	0

Exercice 7.9 : recherche de la plus grande monotonie dans un tableau

Soit un tableau T de n éléments, déterminer la longueur de la première plus longue séquence de nombres rangés par ordre croissant et le rang de son premier élément.

Exercice 7.10 : fusion de deux tableaux triés

Ecrire une procédure qui permet de fusionner deux tableaux triés A et B contenant respectivement n et m éléments. Le résultat est un tableau trié C à (n+m) éléments.

Exemple :

A	1	20	41		**B**	19	23	27	54	91

C	1	19	20	23	27	41	54	91

Exercice 7.11

Etant donné un tableau A de n nombres triés par ordre croissant. Ecrire un algorithme qui permet de lire un réel R et l'insérer dans sa bonne position. Le résultat sera un deuxième tableau B de longueur (n+1) et qui est également trié par ordre croissant.

Exemple :

A	1	2	5	8	10	25	R = 6

B	1	2	5	6	8	10	25

Exercice 7.12 : triangle de Pascal

Créer un tableau à deux dimensions qui contiendra les n premières lignes du triangle de Pascal.

Chaque élément du triangle de pascal est obtenu par la formule :

$$T[L,C] = T[L-1,C-1] + T[L-1,C]$$

A titre d'exemple, pour n = 6, le triangle de pascal est :

1	0	0	0	0	0
1	1	0	0	0	0
1	2	1	0	0	0
1	3	3	1	0	0
1	4	6	4	1	0
1	5	10	10	5	1

Exercice 7.13 : la tour sur un échiquier

Sur un échiquier, c'est-à-dire un tableau de 8 lignes et 8 colonnes, à partir d'une case quelconque, marquer toutes les cases susceptibles d'être atteintes en un coup par une tour. Au terme de l'exécution, les cases atteintes contiendront la valeur 1, les autres la valeur 0.

La tour peut se déplacer sur la ligne et sur la colonne correspondantes à la case où elle est située.

Exemple

Si la tour se trouve à la case (L=2 , C=3), le tableau final aura l'allure suivante :

0	0	1	0	0	0	0	0
1	1	1	1	1	1	1	1
0	0	1	0	0	0	0	0
0	0	1	0	0	0	0	0
0	0	1	0	0	0	0	0
0	0	1	0	0	0	0	0
0	0	1	0	0	0	0	0
0	0	1	0	0	0	0	0

Exercice 7.14 : le fou sur un échiquier

Sur un échiquier, c'est-à-dire un tableau de 8 lignes et 8 colonnes, à partir d'une case quelconque, marquer toutes les cases susceptibles d'être atteintes en un coup par un fou. Au terme de l'exécution, les cases atteintes contiendront la valeur 1, les autres la valeur 0.

Le fou peut se déplacer sur les diagonales issues de la case où il est situé.

Exemple

Si le fou se trouve à la case (L=4 , C=3), le tableau final aura l'allure suivante :

0	0	0	0	0	1	0	0
1	0	0	0	1	0	0	0
0	1	0	1	0	0	0	0
0	0	1	0	0	0	0	0
0	1	0	1	0	0	0	0
1	0	0	0	1	0	0	0
0	0	0	0	0	1	0	0
0	0	0	0	0	0	1	0

Remarque

Pour toutes les cases qui peuvent être atteintes par le fou, l'une des 2 relations suivantes est vérifiée :

- L – C = 4 - 3 = 1
- L + C = 4 + 3 = 7

Exercice 7.15 : le carré magique

Dans un tableau carré à N lignes et N colonnes (N impair) ranger les nombres entiers de 1 à N^2 de manière à ce que la somme des éléments de chaque ligne soit égale à la somme des éléments de chaque colonne et à la somme des éléments de chaque diagonale.

Exemple

T

6	1	8
7	5	3
2	9	4

est un carré magique dont somme S est 15

Le principe exposé ci-après est l'une des méthodes de création d'un carré magique :

- Pour commencer, 1 est placé dans la case centrale de la première ligne ;
- Ensuite, les entiers de 2 à N^2 sont placés les uns après les autres dans les cases d'une diagonale orientée nord-ouest.

Plusieurs cas particuliers peuvent se présenter :

- Si on tombe hors du carré en haut, le nombre est placé dans la dernière ligne sans changer de colonne ;
- Si on tombe hors du carré à gauche, le nombre est placé dans la colonne de droite sans changer de ligne ;
- Si on tombe sur une case déjà occupée, le nombre est placé sous la case précédemment remplie. A chaque fois qu'un multiple de N est placé dans une case, la case destinée au nombre suivant est normalement occupée.

SOLUTIONS DES EXERCICES

Exercice 7.1 : fréquence d'apparition de x dans le tableau T

```
 Fonction Fréquence(T : Tab ; x : Entier) : Entier
Variables
        i, cpt : Entier
Début
        cpt ← 0
        Pour i de 1 à n Faire
                Si (T[i] = x) Alors
                        cpt ← cpt + 1
                FinSi
        FinPour
        Fréquence ← cpt
Fin
```

Exercice 7.2 : éclatement d'un tableau

```
Procédure Eclater(T : Tab ; Var TP, TN : Tab)
Variables
        i, j, k : Entier
Début
        i ← 1   j ← 1   k ← 1
        Pour i de 1 à n Faire
                Si (T[i] > 0) Alors
                        TP[j] ← T[i]
                        j ← j + 1
                FinSi
                Si (T[i] < 0) Alors
                        TN[k] ← T[i]
                        k ← k + 1
                FinSi
                Si (T[i] = 0) Alors
                        TP[j] ← T[i]
                        j ← j + 1
                        TN[k] ← T[i]
```

```
                    k ← k + 1
                FinSi
            FinPour
Fin
```

Exercice 7.3

```
Algorithme Alphabet
Constantes
        n = 26
Types
        Tab = Tableau[1..n] de Caractère
Variables
        réponse : Tab
        i, nb_fautes : Entier

Procédure Remplir(Var T : Tab)
Début
        Pour i de 1 à n Faire
                Ecrire(”Entrer un caractère :”)   Lire(T[i])
        FinPour
Fin

Procédure Corriger(Var T : Tab ; Var nb : Entier)
Début
        nb ← 0
        Pour i de 1 à n Faire
                Si (Majus(T[i]) # Car(i+64)) Alors
                        T[i] ← Car(i+64)
                        nb ← nb + 1
                FinSi
        FinPour
Fin

Début
        remplir(réponse)
        corriger(réponse, nb_fautes)
        Ecrire(”Nombre de fautes : ”,nb_fautes)
Fin.
```

Exercice 7.4 : calcul du produit scalaire de deux vecteurs

```
Fonction ProdScal(U : Tab ; V : Tab) : Réel
Variables
        i : Entier
        ps : Réel
Début
        ps ← 0
        Pour i de 1 à n Faire
                ps ← ps + U[i] * V[i]
        FinPour
        ProdScal ← ps
Fin
```

Exercice 7.5 : calcul de la norme d'un vecteur

```
Fonction NormVect(U : Tab) : Réel
Début
        NormVect ← Racine(ProdScal(U,U))
Fin
```

Exercice 7.6 : les 10 premiers termes de la suite de Fibonacci (de 0 à 9)

```
Procédure Fibonacci(Var fib : Tab)
Variables
        i : Entier
Début
        fib[0] ← 1
        fib[1] ← 1
        Pour i de 2 à 9 Faire
                fib[i] ← fib[i-2] + fib[i-1]
        FinPour
Fin
```

Exercice 7.7 : nombres premiers inférieurs à 400

```
Algorithme Premiers
Constantes
        n = 400
Types
        Tab = Tableau[1..n] de Entier
Variables
        Prem : Tab
        i, j : Entier
Procédure initialiser(Var T : Tab)
Début
        Pour i de 1 à n Faire
                T[i] ← i
        FinPour
Fin
Procédure mise_a_zero(Var T : Tab)
Début
        Pour i de 1 à 20 Faire
                Si (T[i] # 0) Alors
                        Pour j de i+1 à n Faire
                                Si (T[j] Mod T[i] = 0) Alors
                                        T[j] ← 0
                                FinSi
                        FinPour
                FinSi
        FinPour
Fin
Procédure afficher(T : Tab)
Début
        Pour i de 1 à n Faire
                Si (T[i] # 0) Alors
                        Ecrire(T[i])
                FinSi
        FinPour
Fin
Début
        initialiser(Prem)
        mise_a_zero(Prem)
        afficher(Perm)
Fin.
```

Exercice 7.8 : compression d'un vecteur

```
Procédure Compression(U1, L : Tab ; Var U2 : Tab)
Variables
        i, j : Entier
Début
        j ← 1
        Pour i de 1 à n Faire
                Si (L[i] # 0) Alors
                        U2[j] ← U1[i]
                        j ← j+1
                FinSi
        FinPour
        Pour i de (j+1) à n Faire
                U2[i] ← 0
        FinPour
Fin.
```

Exercice 7.9 : recherche de la plus grande monotonie dans un tableau

```
Procédure Monotonie(T : Tab ; Var iplm, Lplm : Entier)
Variables
        i, j, L : Entier
Début
        i ← 1
        Lplm ← 1
        iplm ← 1
        TantQue (i <= n) Faire
                j ← i + 1
                TantQue (T[j] >= T[j-1]) Faire
                        j ← j + 1
                FinTQ
                L ← j – i + 1
                Si (L > Lplm) Alors
                        iplm ← i
                        Lplm ← L
                FinSi
                i ← j
        FinTQ
Fin
```

Exercice 7.10 : fusion de 2 tableaux triés

```
Procédure Fusion(A : Tab1 ; B : Tab2 ; Var C : Tab3)
```

```
Variables
        i, j, k : Entier
Début
        i ← 1   j ← 1   k ← 1
        TantQue (i <= n) et (j <= m) Faire
                Si (A[i] <=  B[j]) Alors
                        C[k] ← A[i]
                        i ← i + 1
                        k ← k + 1
                FinSi
                Si (B[j] <=  A[i]) Alors
                        C[k] ← B[j]
                        j ← j + 1
                        k ← k + 1
                FinSi
        FinTQ
        TantQue (i <= n) Faire
                C[k] ← A[i]
                i ← i + 1
                k ← k + 1
        FinTQ
        TantQue (j <= m) Faire
                C[k] ← B[j]
                j ← j + 1
                k ← k + 1
        FinTQ
Fin
```

Exercice 7.11 : insertion d'un élément dans un tableau trié

```
Procédure Insertion(A : Tab1; R : Entier; Var B : Tab2)
Variables
        i, j : Entier
Début
        i ← 1
        TantQue (A[i] <= R) et (i <= n) Faire
                B[i] ← A[i]
                i ← i + 1
        FinTQ
        B[i] ← R
        i ← i + 1
        TantQue (i <= n) Faire
                B[i] ← A[i]
                i ← i + 1
        FinTQ
Fin
```

Exercice 7.12 : triangle de Pascal

```
Procédure Pasc(n:Entier; Var T:Tableau[1..n,1..n] de Entier)
Variables
        i, j : Entier
Début
        (* Initialisation du tableau à zéro *)
        Pour i de 1 à n Faire
                Pour j de 1 à n Faire
                        T[i,j] ← 0
                FinPour
        FinPour

        (* Remplissage du tableau *)
        Pour i de 1 à n Faire
                T[i,1] ← 1
                Pour j de 2 à i Faire
                        T[i,j] ← T[i-1,j-1] + T[i-1,j]
                FinPour
        FinPour
Fin
```

Exercice 7.13 : la tour sur un échiquier

```
Procédure tour(L,C : Entier ; Var T : Tableau[1..8,1..8] de Entier)
Variables
        i, j : Entier
Début
        Pour i de 1 à 8 Faire
                Pour j de 1 à 8 Faire
                        Si (i = L) OU (j = C) Alors
                                T[i,j] ← 1
                        Sinon
                                T[i,j] ← 0
                        FinSi
                FinPour
        FinPour
Fin
```

Exercice 7.14 : le fou sur un échiquier

```
Procédure fou(L,C : Entier ; Var T : Tableau[1..8,1..8] de Entier)
Variables
        i, j : Entier
Début
        Pour i de 1 à 8 Faire
                Pour j de 1 à 8 Faire
                        Si (i-j = L-C) OU (i+j = L+C) Alors
                                T[i,j] ← 1
                        Sinon
                                T[i,j] ← 0
                        FinSi
                FinPour
        FinPour
Fin
```

Exercice 7.15 : le carré magique

```
Procédure carré(n : Entier; Var T: Tableau[1..n,1..n] de Entier)
Variables
        i, j, L, C : Entier
Début
        (* mettre 1 dans la case centrale de la 1ère ligne *)
        C ← (n + 1) Div 2
        T[1,C] ←  1

        (* remplissage du reste du tableau *)
        i ← 1
        L ← 1
        TantQue (i <= n^2) Faire
                Si (i Mod n = 0) Alors          (* case occupée *)
                        L  ← L + 1
                Sinon
                        Si (L =1) Alors   (* débordement de ligne *)
                                L  ← n
                        Sinon
                                L  ← L – 1
                        FinSi
                        Si (C = 1) Alors    (* débordement de colonne *)
                                C  ← n
                        Sinon
                                C  ← C – 1
                        FinSi
                FinSi
                i  ← i + 1
                T[L,C]  ← i
        FinTQ
Fin
```

Leçon 8 : Les enregistrements

Objectif

Manipuler correctement des variables de type enregistrement.

I. Notion d'enregistrement

Les variables que nous avons jusqu'à présent utilisées ne se constituent que d'un seul type de données (Entier, Réel, Caractère, etc.).

Les tableaux constituent une extension puisque nous y déclarons une variable composée de plusieurs éléments de même type.

Un **enregistrement** (ou **article**) permet de rassembler un ensemble d'éléments de types différents sous un nom unique. On définit ainsi un **type composé**.

A titre d'exemple, une date, une adresse ou nombre complexe peuvent être considérés comme des enregistrements.

II. Déclaration des variables de type enregistrement

Un enregistrement peut être défini par une déclaration de type contenant une liste de champs précédée par le mot *Struct* et terminée par *FinStruct*.

Exemples

- Définition d'une adresse :

```
Types
        Adresse = Struct
                        Rue  : Chaîne
                        Ville : Chaîne
                        CP   : Entier
                  FinStruct
Variables
        Adr : Adresse
```

- Définition d'un produit :

```
Types
        Produit = Struct
                        Codp  : Entier
                        Libellé : Chaîne
                        Prix_Unit : Réel
                        Stock : Réel
                  FinStruct
Variables
        P1, P2 : Produit
```

Exercice

Définir le type nombre complexe.

Solution

```
Types
        Complexe = Struct
                        Reel  : Réel
                        Imag : Réel
                  FinStruct
Variables
        C : Complexe
```

III. Manipulation des variables de type enregistrement

Les enregistrements ne peuvent être référencés globalement dans une instruction car ils sont composés d'éléments de types différents. Par contre, il est possible de faire référence à chaque élément d'un enregistrement.

Pour cela, il est nécessaire de préciser le nom (identificateur) de l'enregistrement, suivi d'un suffixe indiquant l'identificateur du champ concerné. Les deux identificateurs sont séparés par un point.

Exemple

```
Adr.Rue ← ”Ibn Rochd”
Adr.Ville ← ”Monastir”
Adr.CP ← 5000
```

Remarque

Il est possible d’utiliser l’opérateur d’affectation entre deux enregistrements de même type comme dans l’exemple suivant :

$$P1 \leftarrow P2 \quad \Leftrightarrow \quad \begin{cases} P1.Codp \leftarrow P2.Codp \\ P1.Libellé \leftarrow P2.Libellé \\ P1.Prix\_Unit \leftarrow P2.Prix\_Unit \\ P1.Stock \leftarrow P2.Stock \end{cases}$$

De même, on peut effectuer une comparaison égalitaire ou inégalitaire entre deux enregistrements de même type comme dans les 2 exemples suivants :

- Si (P1 = P2) Alors …
- TantQue (P1 # P2) Faire …

EXERCICES D’APPLICATION

Exercice 8.1

Ecrire un algorithme qui lit deux nombres complexes C1 et C2 et qui affiche ensuite leur somme et leur produit.

On utilisera les formules de calcul suivantes :

- $(a + bi) + (c + di) = (a + c) + (b + d)i$
- $(a + bi)(c + di) = (ac - bd) + (ad + bc)i$

Exercice 8.2

Créer un tableau TabEmp qui contiendra les informations sur les 50 employés d'une entreprise (Matricule, Nom, Salaire, Etat_Civil), le remplir puis afficher le nombre d'employés dont le salaire est compris entre 500 et 700 D.

SOLUTIONS DES EXERCICES

Exercice 8.1 : somme et produit de deux nombres complexes

```
Algorithme CalculComplexe
Types
        Complexe = Struct
                        Reel  : Réel
                        Imag : Réel
                   FinStruct
Variables
        C1, C2, S, P : Complexe
Début
        Ecrire(”Partie réelle du 1er nombre : ”)
        Lire(C1.Reel)
        Ecrire(”Partie imaginaire du 1er nombre : ”)
        Lire(C1.Imag)
        Ecrire(”Partie réelle du 2ème nombre : ”)
        Lire(C2.Reel)
        Ecrire(”Partie imaginaire du 2ème nombre : ”)
        Lire(C2.Imag)
        S.Reel ← C1.Reel + C2.Reel
        S.Imag ← C1.Imag + C2.Imag
        Ecrire(”Somme = ”,S.Reel, ”+”,S.Imag, ”i”)
        P.Reel ← (C1.Reel*C2.Reel) – (C1.Imag*C2.Imag)
        P.Imag ← (C1.Reel*C2.Imag) + (C1.Imag*C2.reel)
        Ecrire(”Produit = ”,P.Reel, ”+”,P.Imag, ”i”)
Fin.
```

Exercice 8.2 : tableau des employés

```
Algorithme Personnel
Constantes
        n = 50
Types
        Employé = Struct
                        Matricule : Entier
                        Nom : Chaîne
                        Sal : Réel
                        Etat_Civil : Caractère
                  FinStruct
        Tab = Tableau[1..n] de Employé
Variables
        TabEmp : Tab
```

```
        i, nb : Entier
Procédure remplir(Var T : Tab)
Début
        Pour i de 1 à n Faire
                Ecrire(”Matricule de l’employé : ”)
                Lire(T[i].Matricule)
                Ecrire(”Nom de l’employé : ”)
                Lire(T[i].Nom)
                Ecrire(”Salaire de l’employé : ”)
                Lire(T[i].Sal)
                Ecrire(”Etat civil de l’employé : ”)
        Lire(T[i].Etat_Civil)
        FinPour
Fin
Fonction compter(T : Tab) : Entier
Début
        Compter ← 0
        Pour i de 1 à n Faire
                Si (T[i].Sal >= 500)  ET  (T[i].sal <= 700)  Alors
                        Compter ← Compter + 1
                FinSi
        FinPour
Fin
Début
        Remplir(TabEmp)
        nb ← Compter(TabEmp)
        Ecrire(nb, ” employés touchent entre 500 et 700 D”)
Fin.
```

Leçon 9 : Les fichiers séquentiels

Objectifs

- Comprendre les concepts de base relatifs aux fichiers
- Manipuler des fichiers à organisation séquentielle.

I. Notion de fichier

Dans tous les programmes que nous avons jusqu'à présent développés le stockage des données se fait en mémoire vive qui est volatile et de capacité limitée. Or, la plupart des applications nécessitent une sauvegarde permanente des données.

Pour éviter la perte de ces informations au débranchement de l'ordinateur, on utilise des *fichiers*.

I.1 . Définition

Un fichier est une structure de données formée de cellules contiguës permettant l'implantation d'une suite de données en mémoire secondaire (disque, disquette, CD-ROM, bande magnétique, etc.)

Chaque élément de la suite est appelé *article* et correspond généralement à un enregistrement.

Exemples :

- liste des étudiants d'une institution
- état des produits stockés dans un magasin
- liste des employés d'une entreprise.

I.2. Eléments attachés à un fichier

- On appelle ***nom interne*** d'un ficher le nom sous lequel un fichier est identifié dans un programme.

- On appelle ***nom externe*** d'un fichier le nom sous lequel le fichier est identifié en mémoire secondaire. Ce nom est composé de trois parties:

    - l'identifiant du support
    - le nom du fichier proprement dit
    - une extension (ou suffixe) qui précise le genre du fichier (donnée, texte, programme, etc.)

Ainsi, « **A:nombres.DAT »** désigne un fichier de données stocké sur la disquette et qui s'appelle nombres.

- On appelle ***tampon*** ou ***buffer*** d'un fichier, une zone de la mémoire principale pouvant contenir un enregistrement du fichier. C'est une « fenêtre » à travers laquelle on « voit » le fichier.

- Un fichier possède toujours un enregistrement supplémentaire à la fin appelé ***marque de fin de fichier*** (FDF) permettant de le borner (fig. 12) :

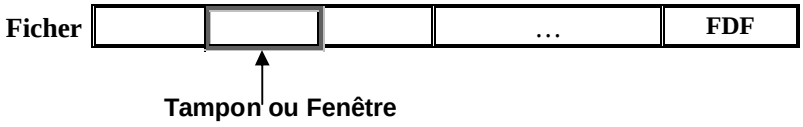


Figure 12. Structure d'un fichier

- Chaque fichier est caractérisé par :

    - un **mode d'organisation** : séquentielle, séquentielle indexée, relative ou sélective.

    - un **mode d'accès** : séquentiel ou direct

Un *fichier à organisation séquentielle* (F.O.S) ne permet que l'accès séquentiel : pour atteindre l'article de rang n, il est nécessaire de parcourir les (n-1) articles précédents.

L'accès direct se fait soit en utilisant le rang de l'enregistrement (cas de l'organisation relative) comme dans les tableaux, soit en utilisant une clé permettant d'identifier de façon unique chaque enregistrement (cas de l'organisation séquentielle indexée et sélective).

Remarque

Les caractéristiques d'un fichier sont étroitement liées aux langages de programmation qui offrent chacun différents types de fichiers.

II. Déclaration d'un fichier à organisation séquentielle

Pour déclarer une variable de type fichier, il faut spécifier :

- le nom du fichier
- le type des éléments du fichier

Exemple

```
Types
        Etudiant = Struct
                        Numéro : Entier
                        Nom : Chaîne[30]
                        Prénom : Chaîne[30]
                        Classe : Chaîne[5]
                FinStruct
        Fetud = Fichier de Etudiant
Variables
        Fe : Fetud
        et : Etudiant     (* variable tampon *)
```

III. Manipulation des fichiers à organisation séquentielle

Toute manipulation d'un fichier nécessite 3 phases :

1- Ouverture du fichier :

Ouvrir(NomFichier, mode)

Un fichier peut être ouvert en mode lecture (L) ou en mode écriture (E).

Après l'ouverture, le pointeur pointe sur le premier enregistrement du fichier.

2- Traitement du fichier :

- **Lire(NomFichier, fenêtre)**

Cette primitive a pour effet de copier l'enregistrement actuel dans la fenêtre du fichier. Après chaque opération de lecture, le pointeur passe à l'enregistrement suivant. Si on veut lire une information en amont du pointeur, il faut rouvrir le fichier et le lire jusqu'à l'information désirée.

- **Ecrire(NomFichier, fenêtre)**

Cette primitive a pour effet de copier le contenu de la fenêtre sur le fichier en mémoire secondaire. Dans un fichier à organisation séquentielle, l'ajout d'un nouveau article se fait toujours en fin de fichier.

3- Fermeture du fichier :

Fermer(NomFichier)

Remarque

La fonction booléenne **FDF(NomFichier)** permet de tester si la fin du fichier est atteinte. Sa valeur est déterminée par le dernier ordre de lecture exécuté.

III.1. Création d'un Fichier à organisation séquentielle

Ecrire une procédure permettant de créer et remplir le fichier des étudiants.

```
Procédure Création(Var fe : Fetud)
Variables
        et : Etudiant
Début
        Ouvrir(fe,E)
        Ecrire("Numéro de l'étudiant : ")
        Lire(et.Numéro)
        TantQue (et.Numéro # 0) Faire
                Ecrire("Nom de l'étudiant : ")
                Lire(et.Nom)
                Ecrire("Prénom de l'étudiant : ")
                Lire(et.Prénom)
                Ecrire("Classe de l'étudiant : ")
                Lire(et.Classe)
                Ecrire(fe,et)
                Ecrire("Numéro de l'étudiant : ")
                Lire(et.Numéro)
        FinTQ
        Fermer(fe)
Fin
```

Dans cette procédure, l'utilisateur doit entrer la valeur 0 pour le champ numéro pour indiquer la fin de la saisie étant donné que le nombre d'étudiants n'est pas connu à l'avance.

III.2. Parcours d'un fichier à organsation séquentielle

Ecrire une procédure permettant d'afficher la liste des étudiants à partir du fichier « fe ».

```
Procédure Consultation(fe : Fetud)
Variables
        et : Etudiant
Début
        Ouvrir(fe,L)
        Lire(fe,et)
        TantQue NON(FDF(fe)) Faire
                Ecrire(et.Numéro,et.Nom,et.Prénom,et.classe)
                Lire(fe,et)
        FinTQ
        Fermer(fe)
Fin
```

IV. Les fichiers de type texte

Les fichiers de texte sont des fichiers séquentiels qui contiennent des caractères organisés en lignes. La présentation sous forme de « ligne » est obtenue grâce à la présence des caractères de contrôle :

- retour chariot (noté souvent CR), de code ASCII 13
- saut de ligne (noté souvent LF) de code ASCII 10.

Un fichier texte peut être déclaré de 2 façons différentes :

```
Variables
        ftext : Fichier de Caractère
```

ou

```
Variables
        ftext : Fichier Texte
```

Remarques

1- Un fichier de type texte peut être traité ligne par ligne ou caractère par caractère

2- Dans un fichier de type texte, la primitive **Lire_Lig(NomFichier,Fenêtre)** permet de lire une ligne du fichier et la transférer dans la fenêtre.
3- De même, la fonction booléenne **FDL(NomFichier)** permet de vérifier si le pointeur a atteint la fin d'une ligne.

Exercice

Ecrire l'algorithme d'une procédure qui lit et affiche le contenu d'un fichier de type texte.

Solution

```
Procédure ParcoursFichText(ftext : Fichier texte)
Variables
        ligne : Chaîne
Début
        Ouvrir(ftext,L)
        Lire(ftext,ligne)
        TantQue Non(FDF(ftext)) Faire
                Ecrire(ligne)
                Lire(ftext,ligne)
        FinTQ
        Fermer(ftext)
Fin
```

EXERCICES D'APPLICATION

Exercice 9.1

Ecrire un algorithme permettant de :

- Créer et remplir un fichier « Fp » qui contient des informations sur le personnel d'une entreprise (matricule, nom, prénom, grade, salaire).

- Afficher la liste des employés de cette entreprise dont le salaire est compris entre 500 et 700 D.

Exercice 9.2

Ecrire une procédure permettant de rechercher un employé dans le fichier Fp à partir de son matricule.

- Si l'employé est trouvé, l'algorithme affiche son nom, son prénom et son grade
- Sinon, il affiche le message "ce matricule ne figure pas dans le fichier…"

Exercice 9.3

Ecrire un algorithme permettant de :

- Créer et remplir un fichier « fnotes » qui contient les notes de 30 étudiants
- Copier les notes dans un tableau Tnote
- Trier le tableau Tnote dans l'ordre croissant
- Copier les notes triées du tableau vers le fichier fnotes.

SOLUTIONS DES EXERCICES

Exercice 9.1 : fichier personnel

```
Algorithme Personnel
Types
        Employé = Struct
                        Matricule : Entier
                        Nom : Chaîne
                        Prénom : Chaîne
                        Grade : Caractère
                        Sal : Réel
                  FinStruct
        Fpers = Fichier de Employé
Variables
        Fp : Fpers
        emp : Employé

Procédure Création(Var f : Fpers)
Début
        Ouvrir(f,E)
        Ecrire(”Matricule : ”)    Lire(emp.Matricule)
        TantQue (emp.Matricule # 0) Faire
                Ecrire(”Nom : ”)          Lire(emp.Nom)
                Ecrire(”Prénom : ”)       Lire(emp.Prénom)
                Ecrire(”Grade : ”)        Lire(emp.Grade)
                Ecrire(”Salaire : ”)      Lire(emp.Sal)
                Ecrire(f,emp)
                Ecrire(”Matricule : ”)    Lire(emp.Matricule)
        FinTQ
        Fermer(f)
Fin

Procédure Consultation(f : Fpers)
Début
        Ouvrir(f,L)
        Lire(f,emp)
        TantQue NON(FDF(f)) Faire
                Si (emp.Sal>=500) ET (emp.Sal<=700) Alors
                     Ecrire(emp.Matricule, emp.Nom, emp.Sal)
```

```
                FinSi
                Lire(f,emp)
        FinTQ
        Fermer(f)
Fin

Début
        Création(Fp)
        Consultation(Fp)
Fin.
```

Exercice 9.2 : recherche d'un employé dans le fichier personnel

```
Procédure Recherche(Fp : Fpers ; x : Entier)
Variables
        emp : Employé
Début
        Ouvrir(Fp,L)
        Lire(Fp,emp)
        Trouve ← (emp.Matricule = x)
        TantQue (trouve = Faux) ET NON(FDF(Fp)) Faire
                Lire(Fp,emp)
                Trouve ← (emp.Matricule = x)
        FinTQ
        Si FDF(Fp) Alors
                Ecrire("Ce matricule ne figure pas dans le
                fichier…")
        Sinon
                Ecrire(emp.Nom, emp.Prénom, emp.Grade)
        FinSi
        Fermer(Fp)
Fin
```

Exercice 9.3 : tri du fichier fnotes

```
Algorithme Notes
Variables
        fnotes : Fichier de Réel
        Tnote : Tableau[1..30] de Réel
        x, note : Réel
        échange : Booléen
        i : Entier
Début
        (* Création du fichier fnotes *)
        Ouvrir(fnotes,E)
        Pour i de 1 à 30 Faire
                Ecrire("Entrer une note : ")      Lire(note)
                Ecrire(fnotes,note)
        FinPour
        Fermer(fnotes)

        (* Copie du fichier fnotes dans le tableau Tnote *)
        Ouvrir(fnotes,L)
        Pour i de 1 à 30 Faire
                Lire(fnotes,note)
                Tnote[i] ← note
        FinPour
        Fermer(fnotes)

        (* Tri du tableau Tnote *)
        Répéter
                échange ← Faux
                Pour i de 1 à 29 Faire
                        Si (Tnote[i] > Tnote[i+1]) Alors
                                x ← Tnote[i]
                                Tnote[i] ← Tnote[i+1]
                                Tnote[i+1] ← x
                                échange ← Vrai
                        FinSi
                FinPour
        Jusqu'à (échange = Faux)
        (* Copie du tableau Tnote dans le fichier fnotes *)
```

```
        Ouvrir(fnotes,E)
        Pour i de 1 à 30 Faire
                Ecrire(fnotes,Tnote[i])
        FinPour
        Fermer(fnotes)
Fin.
```

Leçon 10 : La récursivité

Objectifs

- Résoudre des problèmes récursifs
- Comparer les approches de programmation itérative et récursive.

I. Notion de récursivité

La récursivité est une technique de programmation alternative à l'itération qui permet de trouver, lorsqu'elle est bien utilisée, des solutions très élégantes à un certain nombre de problèmes.

Une procédure ou fonction est dite récursive si son corps contient un ou plusieurs appels à elle-même :

```
Procédure Precur(paramètres)
Début
        …
        Precur(valeurs)
        …
Fin
```

Utiliser la récursivité revient à définir :

- une solution pour un ensemble de cas de base (sans utiliser d'appels récursifs)
- une solution dans le cas général à travers une relation de récurrence avec des cas plus simples. Cette relation doit permettre d'arriver à un cas de base en un nombre fini d'étapes.

II. Etude d'un exemple : la fonction factorielle

- ***Solution itérative***

```
Fonction Facto( n : Entier) : Entier
Variables
        n, i, f : Entier
Début
        f ← 1
        Pour i de 2 à n Faire
                f ← f * i
        FinPour
        Facto ← f
Fin
```

- ***Solution récursive***

La relation de récurrence est définie par :

$0 ! = 1$
$n ! = n * (n-1)!$

Cette relation donne la fonction récursive suivante :

```
Fonction Facto( n : Entier) : Entier
Début
        Si (n = 0) Alors
                Facto ← 1
        Sinon
                Facto ← n * Facto(n-1)
        FinSi
Fin
```

III. Mécanisme de fonctionnement de la récursivité

Considérons le calcul de 4! par la fonction récursive définie ci-dessus :

Facto(4) renvoie 4 * Facto(3)
    Facto(3) renvoie 3 * Facto(2)
        Facto(2) renvoie 2 * Facto(1)
            Facto(1) renvoie 1 * Facto(0)
                Facto(0) renvoie 1 *(arrêt de la récursivité)*
            Facto(1) renvoie 1 * 1 = 1
        Facto(2) renvoie 2 * 1 = 2
    Facto(3) renvoie 3 * 2 = 6
Facto(4) renvoie 4 * 6 = 24

IV. Performances de la récursivité

Les problèmes de mémoire et de rapidité sont les critères qui permettent de faire le choix entre la solution itérative et la solution récursive.

Pour le calcul de la fonction factorielle, les différences entre les deux solutions ne se voient pas trop. Pour l'utilisation de la mémoire, il y aura dépassement dans les calculs avant la saturation de la mémoire.

Le calcul de la suite de fibonacci est le parfait contre-exemple de la solution récursive.

La relation de récurrence est définie par :

$f_0 = 1$
$f_1 = 1$
$f_n = f_{n-2} + f_{n-1}$ pour $n > 1$

La solution récursive vient immédiatement de cette relation :

```
Fonction Fibo( n : Entier) : Entier
Début
        Si (n = 0) ou (n = 1) Alors
                Fibo ← 1
        Sinon
                Fibo ←  Fibo(n - 2) + Fibo(n – 1)
        FinSi
Fin
```

On remarque que la fonction effectue deux appels, donc pour les valeurs importantes de n on aura des temps d'exécution importants (attente du résultat).

A titre d'exemple, voici le schéma de calcul de Fibo(4) (figure 13) :

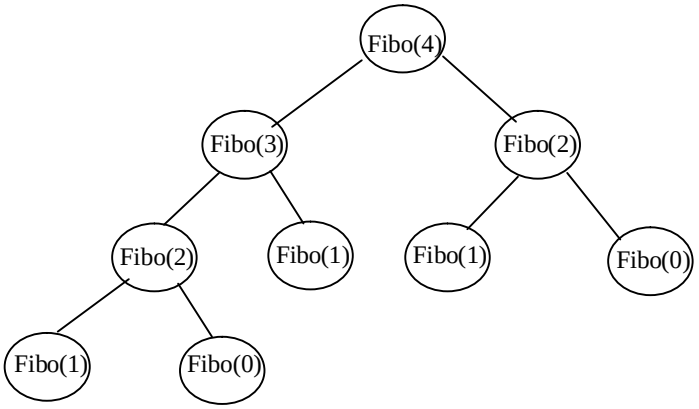


Figure 13. Appels récursifs pour le calcul de Fibo(4)

Fibo(4) = Fibo(3) + Fibo(2)
= (Fibo(2) + Fibo(1)) + Fibo(2)
= ((Fibo(1) + Fibo(0)) + Fibo(1)) + Fibo(2)
= ((1 + Fibo(0)) + Fibo(1)) + Ffibo(2)
= ((1 + 1) + Fibo(1)) + Fibo(2)
= (2 + Fibo(1)) + Fibo(2)
= (2 + 1) + Fibo(2)
= 3 + Fibo(2)
= 3 + (Fibo(1) + Fibo(0))
= 3 + (1 + Fibo(0))
= 3 + (1 + 1)
= 3 + 2
= **5**

Le nombre d'appels récursifs est donc très élevé (9 pour Fibo(4), 15 pour Fibo(5), 25 pour Fibo(6) et 21891 pour Fibo(20)), d'autant plus qu'il existe une méthode itérative simple qui calcule simultanément Fibo(n) et Fibo(n-1) :

```
Fonction Fibo(n : Entier) : Entier
Variables
        i, fn, fn_1, fn_2 : Entier
Début
        fn_1 ← 1
        fn_2 ← 1
        Pour i de 2 à n Faire
                fn ← fn_1 + fn_2
                fn_2 ← fn_1
                Fn_1 ← fn
        FinPour
        Fibo ← fn
Fin
```

V. Récursivité indirecte ou croisée

On dit qu'on a une récursivité indirecte ou une récursivité croisée lorsqu'une procédure A, sans appel récursif, appelle une procédure B qui appelle A.

Exemple

```
Procédure A(paramètres)
Début
        …
        B(valeurs)
        …
Fin
```

```
Procédure B(paramètres)
Début
        …
        A(valeurs)
        …
Fin
```

VI. Choix entre itération et récursivité

Le choix de la solution itérative peut être imposé par le langage de programmation. En effet, certains langages tels que Cobol et T. Basic ne sont pas récursifs.

D'autre part, seuls les problèmes dont la solution se base sur une relation de récurrence avec une condition de sortie peuvent être résolus de façon récursive.
Outre ces deux contraintes, le choix doit être fait uniquement en fonction des critères d'efficacité (contraintes de temps et d'espace). Si la solution récursive satisfait ces critères, il n'y a pas lieu de chercher systématiquement une solution itérative.

Exercice : tours de Hanoi

Le problème des tours de Hanoi est un grand classique de la récursivité car la solution itérative est relativement complexe. On dispose de 3 tours appelées A, B et C. La tour A contient n disques empilés par ordre de taille décroissante qu'on veut déplacer sur la tour B dans le même ordre en respectant les contraintes suivantes :

- On ne peut déplacer qu'un disque à la fois
- On ne peut empiler un disque que sur un disque plus grand ou sur une tour vide.

Ainsi, le paramétrage de la procédure déplacer sera le suivant :

Procédure déplacer(n : Entier ; A, B, C : Caractère)

Lorsque la tour A ne contient qu'un seul disque, la solution est évidente : il s'agit de réaliser un transfert de la tour A vers B. Ce cas constitue donc la condition de sortie (point d'appui).

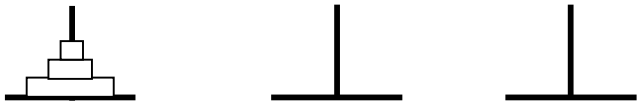
Ainsi, pour déplacer n disques de A vers B en utilisant éventuellement C, il faut :

1- déplacer (n-1) disques de A vers C en utilisant éventuellement B
2- réaliser un transfert du disque de A sur B
3- déplacer (n-1) disques de C vers B en utilisant éventuellement A.

```
Procédure déplacer (n : Entier; A,B,C : Caractère);
Début
    Si (n=1) Alors
        Ecrire(A,' -> ',B)
    Sinon
        déplacer(n-1, A, C, B);
        Ecrire(A,' -> ',B);
        déplacer(n-1, C, B, A);
    FinSi
Fin
```

Voici le résultat de l'exécution de cette procédure avec 3 disques placés initialement sur A (figure 14) :

```
A -> B
A -> C
B -> C
A -> B
C -> A
C -> B
A -> B
```



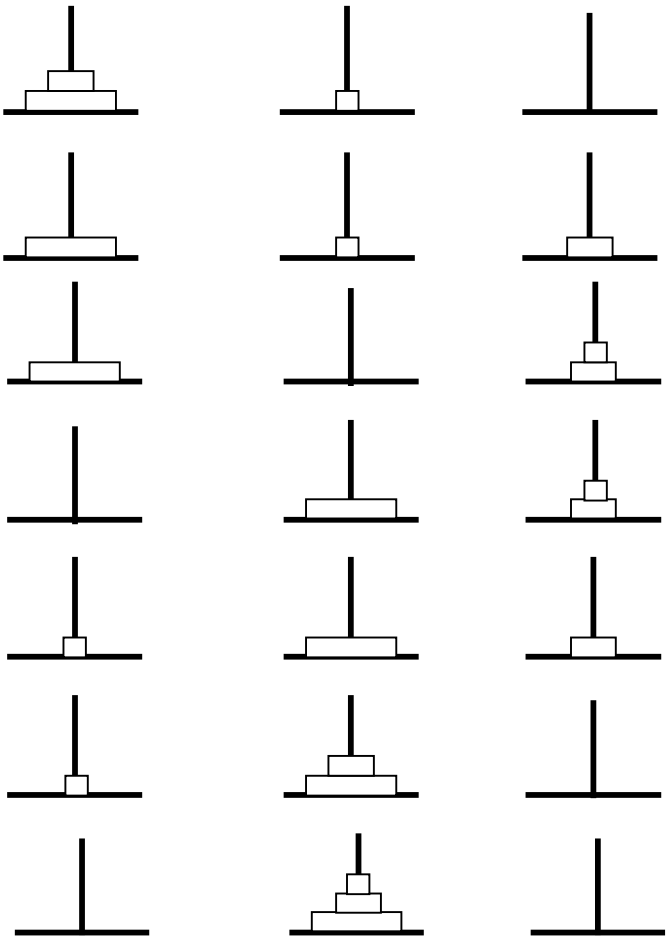


Figure 14. Trace d'exécution de la procédure déplacer(3,A,B,C)

EXERCICES D'APPLICATION

Exercice 10.1 : calcul du PGCD par la méthode d'Euclide

Ecrire une fonction récursive PGCD_Euc qui retourne le PGCD de 2 entiers A et B en utilisant l'algorithme d'Euclide qui s'appuie sur les propriétés suivantes :

PGCD(A,B) = B	Si B est un diviseur de A
PGCD(A,B) = PGCD(B,A mod B)	Sinon

Exemple : PGCD(36,20) = PGCD(20,16) = PGCD(16,4) = 4.

Exercice 10.2 : nombre de combinaisons de p éléments parmi n.

Ecrire une fonction récursive CNP qui calcule le nombre de combinaisons de p éléments pris parmi n éléments différents par la formule suivante :

$$C_n^1 = n \qquad C_n^n = 1$$
$$C_n^p = C_{n-1}^p + C_{n-1}^{p-1}$$

Exercice 10.3 : palindrome

Ecrire une fonction récursive Palind qui vérifie si une chaîne de caractères est un palindrome ou non.

Formellement, une chaîne S de n caractères est un palindrome si et seulement si :

$$\forall\, i, 1 \le i \le n \; div \; 2,\; S[i] = S[n - i + 1]$$

Une condition nécessaire est que les caractères extrêmes soient identiques et que la sous chaîne privée des caractères extrêmes soit également un palindrome.

Exercice 10.4 : fonction d'Ackermann

Ecrire une fonction récursive Ack qui calcule Ack(n,m) selon la formule suivante :

- Ack(0,m) = m + 1
- Ack(n,0) = Ack(n-1,1)
- Ack(n,m) = Ack(n-1,Ack(n,m-1))

Vérifier que : Ack(1,n) = n + 2, Ack(2,n) ≈ 2 * n, Ack(3,n) ≈ 2^n.

Exercice 10.5 : fonction 91 de MacCarthy

1- Ecrire une fonction récursive f qui calcule f(n) selon la formule suivante :

- f(n) = n - 10 Si n > 100
- f(n) = f(f(n+11)) Sinon

2- Calculer f(96).

Exercice 10.6 : fonction de Morris

Soit la fonction récursive suivante :

```
Fonction g(m, n : Entier) : Entier
Début
        Si (m = 0) Alors
                g ← 1
        Sinon
                g ← g(m – 1,g(m,n))
        FinSi
Fin
```

Calculer g(1,0).

SOLUTIONS DES EXERCICES

Exercice 10.1 : calcul du PGCD(a,b) par la méthode d'Euclide

On suppose que a et b sont strictement positifs.

```
Fonction PGCD_Euc(a,b : Entier) : Entier
Début
        Si (a Mod b = 0) alors
                PGCD_Euc ← b
        Sinon
                PGCD_Euc ← PGCD_Euc(b,a mod b)
        FinSi
Fin
```

Exercice 10.2 : nombre de combinaisons de p éléments parmi n

```
Fonction CNP(n,p : Entier) : Entier
Début
        Si (p = 1) alors
                CNP ← n
        Sinon
                Si (p = n) Alors
                        CNP ← 1
                Sinon
                        CNP ← CNP(n-1,p) + CNP(n-1,p-1)
                FinSi
        FinSi
Fin
```

Exercice 10.3 : palindrome (version récursive)

```
Fonction Palind(s : Chaîne) : Booléen
Début
        Si (Longueur(s) < 2) Alors
                Palind ← Vrai
        Sinon
                Palind ← (s[1]=s[longueur(s)]) ET
                (Palind(copie(s,2,longueur(s)-2)))
        FinSi
Fin
```

Exercice 10.4 : fonction d'Ackerman

```
Fonction Ack(a,b : Entier) : Entier
Début
        Si (a = 0) Alors
                Ack ← b + 1
        Sinon
                Si (b = 0) Alors
                        Ack ← Ack(a-1,1)
                Sinon
                        Ack ← Ack(a-1,Ack(a,b-1))
                FinSi
        FinSi
Fin
```

Exercice 10.5 : fonction 91 de MacCarthy

```
Fonction F91(x : Entier) : Entier
Début
        Si (x > 100) Alors
                F91 ← x - 10
        Sinon
                F91 ← F91(F91(x + 11))
        FinSi
Fin
```

F91(96) = F91(F91(107)) = F91(97) = F91(F91(108)) = F91(98) = F91(F91(109)) = F91(99) = F91(F91(110)) = F91(100) = F91(F91(111)) = F91(101) = 91.

Exercice 10.6 : fonction de Morris

g(1,0) = g(0,g(1,0))
= g(0,g(0,g(1,0)))
= g(0,g(0,g(0,g(1,0))))
= …

Impossible de calculer g(1,0) ; la récursivité ne s'arrête plus.

Leçon 11 : Structures de données dynamiques

Objectif

Manipuler correctement des structures de données dynamiques.

I. Introduction

Le but de cette leçon est de gérer un ensemble fini d'éléments dont le nombre varie au cours de l'exécution du programme. Les éléments de cet ensemble peuvent être de différentes sortes : nombres entiers ou réels, chaînes de caractères ou des objets informatiques plus complexes comme les identificateurs de processus ou les expressions arithmétiques.

On ne s'intéressera pas aux éléments de l'ensemble en question mais aux opérations que l'on effectue sur cet ensemble. Plus précisément, les opérations que l'on s'autorise sur un ensemble E sont le suivantes :

- *tester* si l'ensemble E est vide
- *ajouter* l'élément x à l'ensemble E
- *vérifier* si l'élément x appartient à l'ensemble E
- *supprimer* l'élément x de l'ensemble E

Cette gestion des ensembles doit, pour être efficace, répondre à deux critères parfois contradictoires : un minimum d'espace mémoire utilisé et un minimum d'instructions élémentaires pour réaliser une opération.

II. Les variables dynamiques

Jusqu'ici, nous avons utilisé des variables dites « **statiques** ».

Une ***variable statique*** est caractérisée par les propriétés suivantes :

- elle est déclarée en tête du bloc où elle est utilisée
- elle occupe un espace mémoire dont la taille est fixée dès le début pour qu'on y place ses valeurs
- l'accès à la valeur se fait par le nom de la variable.

Au contraire, une ***variable dynamique*** est caractérisée par les propriétés suivantes :

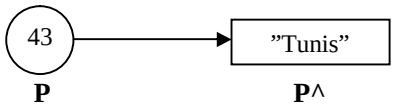
- elle peut être créée et détruite au cours de l'exécution du bloc dans lequel elle est déclarée
- l'espace mémoire rendu libre peut être récupéré
- l'accès à la valeur se fait par un **pointeur**.

II.1 Variable pointeur et variable pointée

Un **pointeur** P est une variable statique dont les valeurs sont des adresses. Une variable dynamique pointée par P sera notée **P^**.

Le type de la variable pointée est appelé **type de base**.

Exemple



La variable pointeur P a pour valeur 43. Elle pointe sur l'espace mémoire P^ d'adresse 43 et dont le contenu est la chaîne "Tunis".

L'algorithe suivant permet de créer et initialiser un pointeur P :

```
Algorithme Pteur
Types
        Pointeur = ^Chaîne
Variables
        P : Pointeur
Début
        Allouer(P)     (* création du pointeur P *)
        P^ ← "Tunis"
Fin.
```

II.2 Opérations sur les pointeurs

Il ne faut pas confondre les opérations sur les pointeurs avec les opérations sur les variables pointées (figure 15).

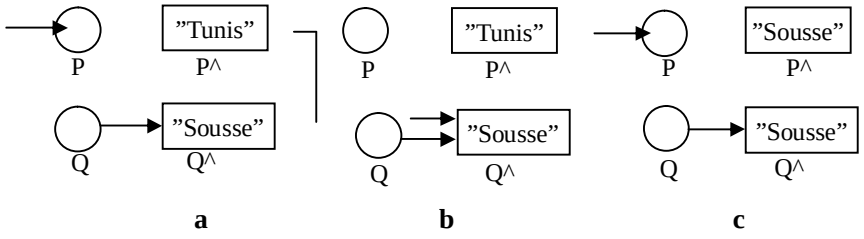


Figure 15. opérations sur les pointeurs

Supposons qu'on soit dans la situation initiale illustrée par la figure 15a :

Si l'instruction :

$$P \leftarrow Q$$

est exécutée, nous passerons à la nouvelle situation illustrée par la figure 15b. Dans ce cas, on a :

$$P^\wedge = Q^\wedge = \text{"Sousse"}$$

et si l'on modifie la valeur de Q^, P^ sera également modifié et restera égal à Q^.

Par contre, si l'instruction :

$$P^\wedge \leftarrow Q^\wedge$$

est exécutée, nous passerons à la nouvelle situation illustrée par la figure 15c. Dans ce cas, on a également :

$$P^\wedge = Q^\wedge = \text{"Sousse"}$$

mais si l'on modifie la valeur de Q^, P^ ne sera pas modifié.

III. Les listes chaînées

La structure de liste est utilisée pour représenter un ensemble d'éléments contenus chacun dans une **cellule**. Celle-ci contient en plus de l'élément, l'adresse de l'élément suivant appelée **pointeur**. La référence vers la première cellule est contenue dans un pointeur nommé **tête de liste**. Une liste chaînée est accessible uniquement par sa tête (figure 16).

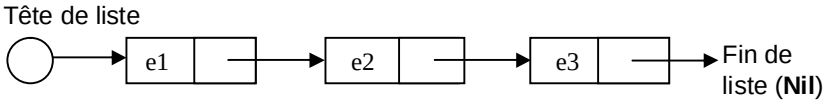


Figure 16. Structure d'une liste chaînée

Nil est une constante prédéfinie qui indique la fin de la liste.

En supposant que les éléments de la liste sont des entiers, celle-ci se déclare de la façon suivante :

```
Types
        Liste = ^Cellule
        Cellule = Struct
                        Elem : entier
                        Suiv : Liste
                FinStruct
Variables
        L : Liste
```

III.1. Création d'une liste chaînée

La procédure suivante permet de créer une liste chaînée de n éléments de type entier.

```
Procédure CréatListe(n : Entier, Var L : Liste)
Variables
        Tête, P : Liste
        i : Entier
Début
        Allouer(Tête)
        Ecrire("Entrer l'élément de tête : ")    Lire(Tête^.Elem)
```

```
        Tête^.Suiv ← Nil
        L ← Tête
        Pour i de 2 à n Faire
                Allouer(P)
                Ecrire("Entrer un entier : ")      Lire(P^.Elem)
                P^.Suiv ← Nil
                L^.Suiv ← P
                L ← P
        FinPour
        L ← Tête
Fin
```

III.2 Parcours d'une liste chaînée

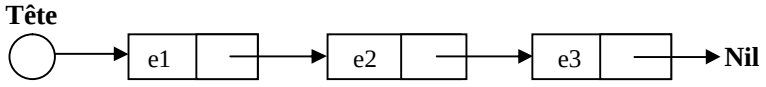
- La procédure ***itérative*** suivante permet de parcourir et afficher les éléments d'une liste chaînée.

```
Procédure AffichListe_itér(L : Liste)
Variables
        P : Liste
Début
        P ← L
        TantQue (P # Nil) Faire
                Ecrire(P^.Elem)
                P ← P^.Suiv
        FinTQ
Fin
```

- Le parcours peut se faire également de façon ***récursive*** :

```
Procédure AffichListe_récur(L : Liste)
Début
        Si (L # Nil) Alors
                Ecrire(L^.Elem)
                AffichListe_récur(L^.Suiv)
        FinSi
Fin
```

III.3 Insertion d'un élément en tête de liste (figure 17)



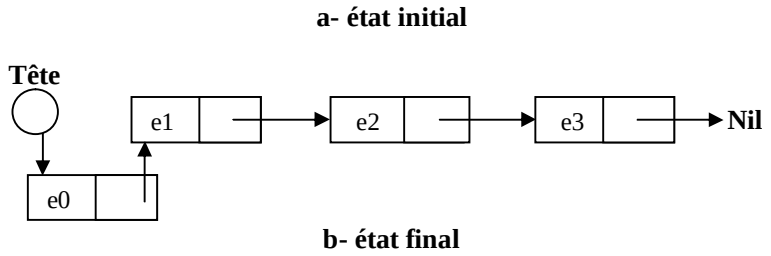


Figure 17. Principe d'insertion d'un élément en tête de liste

On suppose que la liste initiale contient au moins une cellule.

```
Procédure InsertTête(Var L : Liste)
Variables
        P : Liste
Début
        Allouer(P)
        Ecrire("Entrer un entier : ")       Lire(P^.Elem)
        P^.Suiv ← L
        L ← P
Fin
```

III.4 Recherche d'un élément dans une liste chaînée

On veut écrire une fonction **recherche(x : Entier ; L : Liste) : Booléen** qui vérifie si l'élément x figure dans la liste L.

a- version itérative

```
Fonction recherche(x : Entier ; L : Liste) : Booléen
Variables
        P : Liste
        trouve : Booléen
Début
        trouve ← Faux
        P ← L
        TantQue (P # Nil) ET (trouve = Faux) Faire
                Trouve ← (P^.Elem = x)
```

```
                P ← P^.Suiv
        FinTQ
        recherche ← trouve
Fin
```

b- version récursive

```
Fonction recherche(x : Entier ; L : Liste) : Booléen
Début
        Si (L = Nil) Alors
                recherche ← Faux
        Sinon
                Si (L^.Elem = x) Alors
                        recherche ← Vrai
                Sinon
                        recherche ← recherche(x,L^.Suiv)
                FinSi
        FinSi
Fin
```

III.5 Suppression d'un élément d'une liste chaînée

En supposant que la valeur x existe une seule fois dans la liste, supprimer x revient à mettre à jour les liens de façon que le successeur du prédécesseur de x devient le successeur de x (figure 18). Un traitement particulier doit être fait si l'élément à supprimer est le premier élément de la liste.

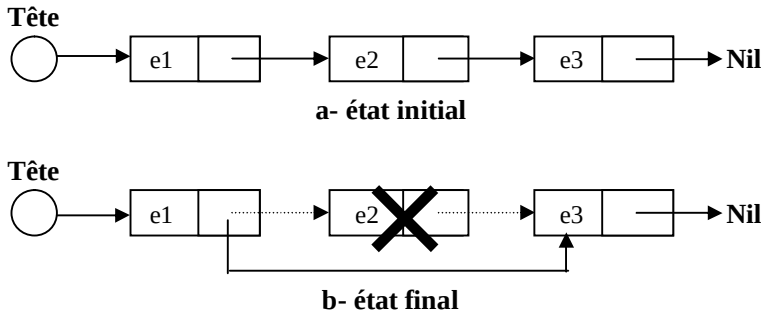


Figure 18. Principe de suppression d'un élément de la liste

a- version itérative

```
Procédure supprimer(x : Entier ; Var L : Liste)
Variables
        P, Q : Liste
Début
        Si (L # Nil) Alors
                P ← L
                Si (L^.Elem = x) Alors
                        L ← L^.Suiv
                        Libérer(P)
                Sinon
                        Q ← L^.Suiv
                        TantQue (Q#Nil)ET(Q^.Elem#x) Faire
                                P ← Q
                                Q ← Q^.Suiv
                        FinTQ
                        Si (Q # Nil) Alors
                                P^.suiv ← Q^.Suiv
                                Libérer(Q)
                        FinSi
                FinSi
        FinSi
Fin
```

L'instruction ***Libérer(P)*** permet de rendre disponible l'espace occupé par P.

b- version récursive

```
Procédure supprimer(x : Entier ; Var L : Liste)
Variables
        P, Q : Liste
Début
        Si (L # Nil) Alors
                Si (L^.Elem = x) Alors
                        P ← L^.Suiv
                        Libérer(L)
                        L ← P
                Sinon
                        supprimer(x, L^.Suiv)
                FinSi
        FinSi
Fin
```

Exercice : longueur d'une liste

Ecrire une fonction récursive qui retourne le nombre d'éléments d'une liste chaînée.

Solution

```
Fonction Longueur(L : Liste) : Entier
Début
        Si (L = Nil) Alors
                Longueur ← 0
        Sinon
                Longueur ← 1 + Longueur(L^.Suiv)
        FinSi
Fin
```

Remarque

Une **liste chaînée circulaire** est une liste particulière dans laquelle le dernier élément pointe sur la tête de la liste (figure 19).

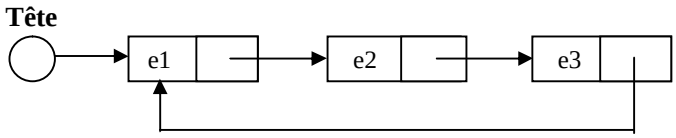


Figure 19. Structure d'une liste chaînée circulaire

IV. Les listes à chaînage double

Ce sont des listes chaînées avec, en plus, un pointeur vers l'élément précédent de chaque cellule. Cette structure permet de parcourir la liste dans les 2 sens et d'accélérer la recherche d'un élément (figure 20) :

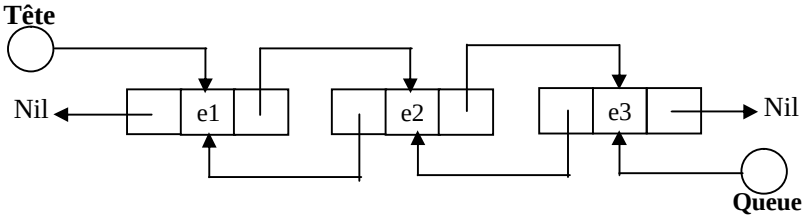


Figure 20. Structure d'une liste à chaînage double

En supposant que les éléments de la liste sont des entiers, celle-ci se déclare de la façon suivante :

```
Types
        Liste = ^Cellule
        Cellule = Struct
                Pred : Liste
                Elem : Entier
                Suiv : Liste
        FinStruct
Variables
        L : Liste
```

IV.1. Création d'une liste à chaînage double

La procédure suivante permet de créer une liste à chaînage double qui contient n éléments de type entier :

```
Procédure CréatListe(n : Entier ; Var L, Queue : Liste)
Variables
        Tête, P : Liste
        i : Entier
Début
        Allouer(Tête)
        Ecrire("Entrer un entier : ")       Lire(Tête^.Elem)
        Tête^.Pred ← Nil
        Tête^.Suiv ← Nil
        L ← Tête
        Pour i de 2 à n Faire
                Allouer(P)
                Ecrire("Entrer un entier : ")       Lire(P^.Elem)
                P^.Pred ← L
                P^.Suiv ← Nil
                L^.Suiv ← P
                L ← P
        FinPour
        Queue ← P
        L ← Tête
Fin
```

IV.2 Parcours inverse d'une liste à chaînage double

La procédure suivante permet de parcourir et afficher les éléments d'une liste à chaînage double en commençant par la queue.

```
Procédure AffichListe(Queue : Liste )
Variables
        P : Liste
Début
        P ← Queue
        TantQue (P # Nil) Faire
                Ecrire(P^.Elem)
```

```
                P ← P^.Pred
        FinTQ
Fin
```

V. Les piles

V.1. Présentation

Une pile est une suite de cellules allouées dynamiquement (liste chaînée) où l'insertion et la suppression d'un élément se font toujours en tête de liste.

L'image intuitive d'une pile peut être donnée par une pile d'assiettes, ou une pile de dossiers à condition de supposer qu'on prend un seul élément à la fois (celui du sommet). On peut résumer les contraintes d'accès par le principe « *dernier arrivé, premier sorti* » qui se traduit en anglais par : **L**ast **I**n **F**irst **O**ut (figure 21).

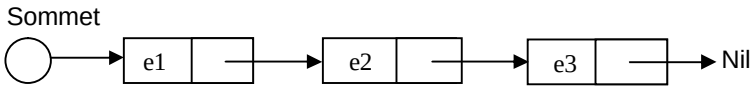


Figure 21. Structure d'une pile

La structuration d'un objet en pile s'impose lorsqu'on mémorise des informations qui devront être traitées dans l'ordre inverse de leur arrivée. C'est le cas, par exemple, de la gestion des adresses de retour dans l'exécution d'un programme récursif.

En supposant que les éléments de la pile sont des entiers, celle-ci se déclare de la façon suivante :

```
Types
        Pile = ^Cellule
        Cellule = Struct
                        Elem : Entier
                        Suiv : Pile
                FinStruct
Var
        P : Pile
```

V.2. Manipulation d'une pile

Du point de vue manipulation, les contraintes d'accès sont matérialisées par les procédures et les fonctions suivantes :

- Procédure Initialiser(Var P : Pile) : crée une pile vide P
- Fonction Pile_Vide (P : Pile) : Booléen : Renvoie la valeur vrai si la pile est vide.
- Procédure Empiler(x : Entier ; Var P : Pile) : ajoute l'élément x au sommet de la pile
- Procédure Dépiler (Var x : Entier ; Var P : Pile) : dépile le sommet de la pile et le met dans la variable x

```
Procédure Initialiser(Var P : Pile)
Début
        P ← Nil
Fin
```

```
Fonction Pile_Vide(P : Pile) : Booléen
Début
        Pile_Vide ← (P = Nil)
Fin
```

```
Procédure Empiler(x : Entier ; Var P : Pile)
Variables
        Q : Liste
Début
        Allouer(Q)
        Q^.Elem ← x
        Q^.Suiv ← P
        P ← Q
Fin
```

```
Procédure Dépiler(Var x : Entier ; Var P : Pile)
Début
        Si NON(Pile_Vide(P)) Alors
                x ← P^.Elem
                Q ← P
                P ← P^.Suiv
                Libérer(Q)
        Sinon
                Ecrire(”impossible, la pile est vide”)
        FinSi
Fin
```

VI. Les files

VI.1. Présentation

Une file est une suite de cellules allouées dynamiquement (liste chaînée) dont les contraintes d'accès sont définies comme suit :

- On ne peut ajouter un élément qu'en dernier rang de la suite
- On ne peut supprimer que le premier élément.

L'image intuitive d'une file peut être donnée par la queue à un guichet lorsqu'il n'y a pas de resquilleurs. On peut résumer les contraintes d'accès par le principe « *premier arrivé, premier sorti* » qui se traduit en anglais par : **F**ast **I**n **F**irst **O**ut (figure 22).

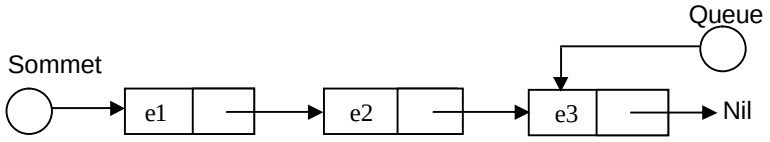


Figure 22. Structure d'une file

La structuration d'un objet en file s'impose en particulier lorsqu'une ressource doit être partagée par plusieurs utilisateurs : il y a formation d'une *file d'attente*. Un exemple est donné par le spooler d'impression qui est un programme qui reçoit, traite, planifie et distribue les documents à imprimer dans un système multiprogrammé.

En supposant que les éléments de la file sont des entiers, celle-ci se déclare de la façon suivante :

```
Types
        Liste = ^Cellule
        Cellule = Struct
                        Elem : Entier
                        Suiv : Liste
                FinStruct
        File = Struct
                        Tête : Liste
                        Queue : Liste
                FinStruct
Variables
        F : File
```

VI.2. Manipulation d'une file

Du point de vue manipulation, les contraintes d'accès sont matérialisées par les procédures et les fonctions suivantes :

- Procédure Initialiser(Var F : File) : crée une file vide F
- Procédure Ajouter(x : Entier ; Var F : File) : ajoute l'élément x à la fin de la file
- Procédure Extraire (Var x : Entier ; Var F : File) : extrait le sommet de la file et le met dans la variable x.

```
Procédure Initialiser(Var F : File)
Début
        F.Tête ← Nil
        F.Queue ← Nil
Fin
```

```
Procédure Ajouter(x : Entier ; Var F : File)
Variables
        P : Liste
Début
```

```
        Allouer(P)
        P^.Elem ← x
        P^.Suiv ← Nil
        Si (F.Queue # Nil) Alors
                F.Queue^.Suiv ← P
        Sinon
                F.Tête ← P
        FinSi
        F.Queue ← P
Fin
```

Dans le cas où la file est vide, comme la queue, la tête de la file doit également pointer vers le nouvel élément.

```
Procédure Extraire(Var x : Entier ; Var F : File)
Variables
        P : Liste
Début
        Si (F.Tête = Nil) Alors
                Ecrire("impossible, la file est vide")
        Sinon
                P ← F.Tête
                x ← F.Tête^.Elem
                F.Tête ← F.Tête^.Suiv
                Libérer(P)
        FinSi
Fin
```

Remarque

Dans les langages qui n'offrent pas le type pointeur, les structures de liste, pile ou file peuvent être implantées sous forme de tableaux.

Ainsi, une liste chaînée d'entiers peut être déclarée de la façon suivante :

```
Constantes
        n = 100
Types
        Indice : 1..n
```

```
        Cellule = Struct
                        Elem : entier
                        Suiv : Indice
                  FinStruct
        Liste = Tableau[Indice] de cellule
Variables
        L : Liste
```

EXERCICES D'APPLICATION

Exercice 11.1 : Inversion d'une liste chaînée

Ecrire une procédure qui permet d'inverser une liste chaînée Ls dans une liste Ls_Inv.

Exercice 11.2 : Tri par bulles d'une liste chaînée

Ecrire une procédure qui permet de trier une liste chaînée Ls dans l'ordre croissant des éléments (utiliser la méthode de tri par bulles).

Exercice 11.3 : Nombres premiers

Ecrire un algorithme qui permet de construire la liste des nombres premiers inférieurs à un entier n donné.

Pour construire cette liste, on commence par y ajouter tous les nombres entiers de 2 à n dans l'ordre croissant. On utilise ensuite la méthode classique du crible d'Eratosthène qui consiste à parcourir successivement les éléments inférieurs à $\sqrt{n}$ et supprimer tous leurs multiples stricts.

Exercice 11.4 : Matrices creuses

Les matrices creuses sont des matrices d'entiers comprenant une majorité de 0 comme dans l'exemple suivant :

$$M = \begin{pmatrix} 0 & 2 & 0 & 0 & 0 \\ 1 & 0 & 3 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 6 & 0 \end{pmatrix}$$

Par souci d'économie, une matrice creuse peut être représentée par une liste chaînée contenant uniquement les éléments non nuls avec leurs indices. Ainsi, la matrice M peut être représentée par la liste suivante :

M = [(2,(1,2)) ; (1,(2,1)) ; (3,(2,3)) ; (6,(4,4)) ; (0,(4,5))]

Les éléments sont rangés par indice croissant (ligne puis colonne). Pour connaître la dimension de la matrice, l'élément d'indice maximal est précisé dans la liste, même si celui-ci est nul.

1- Définir le type Matrice_Creuse
2- Ecrire une fonction qui fait la somme de deux matrices creuses.

SOLUTIONS DES EXERCICES

Exercice 11.1 : inversion d'une liste chaînée

```
Procédure InvertListe(Ls : Liste ; Var Ls_Inv : Liste)
Variables
        P,Q : Liste
Début
        Si (Ls = Nil) Alors
                Ls_Inv ← Nil
        Sinon
                Allouer(Q)
                Q^.Elem ← Ls^.Elem
                Q^.Suiv ← Nil
                Ls ← Ls^.Suiv
                Si (Ls = Nil) Alors
                        Ls_Inv ← Q
                Sinon
                        TantQue (Ls # Nil) Faire
                                Allouer(P)
                                P^.Elem ← Ls^.Elem
                                P^.Suiv ← Q
                                Q ← P
                                Ls ← Ls^.Suiv
                        FinTQ
                        Ls_Inv ← Q
                FinSi
    FinSi
Fin
```

Exercice 11.2 : tri par bulles d'une liste chaînée

```
Procédure TriBul(Var Ls : Liste)
Variables
        P,Q : Liste
        x : Entier
        échange : Booléen
Début
        Répéter
                échange ← Faux
```

```
                P ← Ls
                Q ← Ls^.Suiv
                TantQue (Q # Nil) Faire
                        Si (P^.Elem > Q^.Elem) Alors
                                x ← P^.Elem
                                P^.Elem ← Q^.Elem
                                Q^.Elem ← x
                                échange ← Vrai
                        FinSi
                        P ← Q
                        Q ← Q^.Suiv
                FinTQ
        Jusqu'à (échange = Faux)
Fin
```

Exercice 11.3 : nombres premiers

```
Algorithme Premiers
Types
        Liste = ^Cellue
        Cellule = Struct
                        Elem : Entier
                        Suiv : Liste
                  FinStruct
Variables
        Ls : Liste
        P, Q, R : Liste
        sup,i : Entier
Procédure CreatListe(n : Entier ; Var L : Liste)
Variables
        Tête : Liste
Début
        Allouer(Tête)
        Tête^.Elem ← 2
        Tête^.Suiv ← Nil
        L ← Tête
        Pour i de 3 à n Faire
                Allouer(P)
                P^.Elem ← i
                P^.Suiv ← Nil
```

```
                L^.Suiv ← P
                L ← P
        FinPour
        L ← Tête
Fin

Procédure Prem(n : Entier ; Var L : Liste)
Début
        P ← L
        TantQue (P^.Elem <= Racine(n)) Faire
                Q ← P
                TantQue (Q # Nil) ET (Q^.Suiv # Nil) Faire
                        R ← Q^.Suiv
                        Si (R^.Elem Mod P^.Elem = 0) Alors
                                Q^.Suiv ← R^.Suiv
                                Libérer(R)
                        FinSi
                        Q ← Q^.Suiv
                FinTQ
                P ← P^.Suiv
        FinTQ
Fin

Procédure AffichListe(L : Liste)
Début
        P ← L
        TantQue (P # Nil) Faire
                Ecrire(P^.Elem)
                P ← P^.Suiv
        FinTQ
Fin
Début
        Ecrire(”Entrer la borne sup : ”)  Lire(sup)
        CreatListe(sup,Ls)
        Prem(sup,Ls)
        AffichListe(Ls)
Fin.
```

Leçon 12 : Les arbres

Objectif

Connaître les concepts de base relatifs aux arbres binaires.

I. Introduction

La structure d'arbre est l'une des plus importantes et des plus spécifiques de l'informatique. Elle est utilisée par exemple pour l'organisation des fichiers dans les systèmes d'exploitation, la représentation d'une table des matières, d'un arbre généalogique, etc.

Dans cette leçon, l'accent sera mise sur les **arbres binaires** qui sont un cas particulier des **arbres généraux**.

Une propriété intrinsèque de la structure d'arbre est la ***récursivité***.

II. Définitions

- Un **arbre binaire** B est un ensemble de nœuds qui est soit vide, soit composé d'une racine et de deux arbres binaires disjoints **appelés sous-arbre droit** et **sous-arbre gauche** (figure 23).

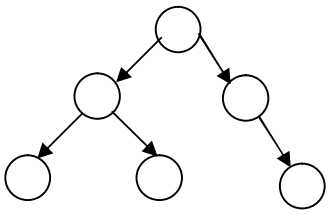


Figure 23. Structure d'un arbre binaire

- Le **père** du sommet x est le sommet p tel qu'il existe un arc entre p et x (seule la racine de l'arbre n'a pas de père).

- Le **frère** de x est un sommet qui a le même père.

- Le **sous-arbre droit** (resp. **gauche**) d'un arbre binaire B est le sous-arbre qui a pour racine le fils droit (resp. gauche) de B.

- Un **nœud interne** est un sommet qui a au moins un fils (gauche ou droit ou les deux).

- Une **feuille** est un sommet qui n'a pas de fils.

- La **hauteur d'un sommet** x est la longueur (en nombre d'arcs) du plus long chemin de x à une feuille.

- La **hauteur d'un arbre** est égale à la hauteur de la racine.

III. Création d'un arbre binaire

Un arbre binaire peut être créé en définissant un type correspondant à un nœud de l'arbre. Soit par exemple :

```
Types
        Arbre = ^Nœud
        Nœud = Struct
                        Valeur : Entier
                        Gauche : Arbre
                        Droite : Arbre
                FinStruct
```

Une telle structure peut être utilisée pour ordonner des éléments, en considérant que le sous-arbre gauche d'un nœud de valeur n contient les éléments inférieurs à n, tandis que le sous-arbre droit contient les éléments supérieurs n (figure 24).

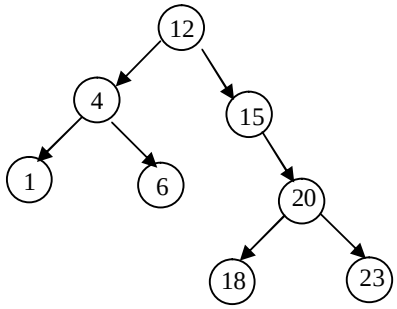


Figure 24. Un arbre binaire ordonné

Une telle structure est pratique pour effectuer des tris rapides. L'algorithme suivant permet de construire un arbre d'entiers à partir d'une suite de nombres entrés par l'utilisateur.

```
Algorithme ArbreBinaire
Types
        Arbre = ^Nœud
        Nœud = Struct
                        Valeur : Entier
                        Gauche : Arbre
                        Droite : Arbre
                FinStruct
Variables
        racine : Arbre
        x : Entier

Procédure Construire(Elt : Entier ; Var B : Arbre)
Début
        Si (B = Nil) Alors
                Allouer(B)
                B^.valeur ← Elt
                B^.gauche ← Nil
                B^.droite ← Nil
        Sinon
                Si (Elt < B^.valeur) Alors
                        Construire(Elt,B^.gauche)
                Sinon
                        Si (Elt > B^.valeur) Alors
```

```
                                Construire(Elt,B^.droite)
                        FinSi
                FinSi
        FinSi
Fin

Début
        racine = Nil
        Ecrire(”Entrer un entier : ”)       Lire(x)
        TantQue (x # 0) Faire
                Construire(x,racine)
                Ecrire(”Entrer un entier :”)        Lire(x)
        FinTQ
Fin.
```

IV. Parcours d’un arbre binaire

On cherche à parcourir un arbre binaire selon une stratégie dite « en profondeur d’abord » ou dans l’ordre **préfixe** illustré sur le diagramme ci-dessous :

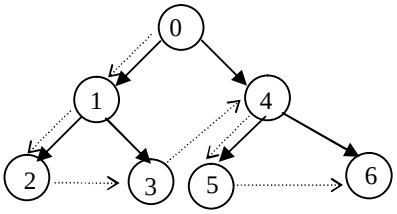


Figure 25. Parcours d’un arbre binaire dans l’ordre préfixe

Les éléments doivent être affichés dans l’ordre suivant : 0 1 2 3 4 5 6.

```
Procédure ParcoursPréfixe(B : Arbre)
Début
        Si (B # Nil) Alors
                Afficher(B^.valeur)
                ParcoursPréfixe(B^.gauche)
                ParcoursPréfixe(B^.droite)
        FinSi
Fin
```

Exercice

1- Ecrire une nouvelle version de la procédure précédente qui fait le parcours de l'arbre dans l'ordre **infixe** ou symétrique. Le résultat doit être dans l'ordre suivant : 6 4 5 0 3 1 2.

2- Ecrire une autre version qui fait le parcours de l'arbre dans l'ordre **postfixe**. Le résultat doit être dans l'ordre suivant : 6 5 4 3 2 1 0)

Solution

```
Procédure ParcoursInfixe(B : Arbre)
Début
        Si (B # Nil) Alors
                ParcoursInfixe(B^.droite)
                Ecrire(B^.valeur)
                ParcoursInfixe(B^.gauche)
        FinSi
Fin

Procédure ParcoursPostfixe(B : Arbre)
Début
        Si (B # Nil) Alors
                ParcoursPostfixe (B^.droite)
                ParcoursPostfixe (B^.gauche)
                Ecrire(B^.valeur)
        FinSi
Fin
```

V. Recherche dans un arbre binaire ordonné

```
Fonction Recherche(x : Entier ; B : Arbre) : Booléen
Début
        Si (B = Nil) Alors
                Recherche ← Faux
        Sinon
                Si (x = B^.valeur) Alors
                        Recherche ← Vrai
                Sinon
                        Si (x < B^.valeur) Alors
                                Recherche(x, B^.gauche)
                        Sinon
                                Recherche(x, B^.droite)
                        FinSi
                FinSi
        FinSi
Fin
```

VI. Arbres binaires particuliers

- On appelle arbre binaire ***complet*** un arbre binaire tel que chaque sommet possède 0 ou 2 fils.

Un arbre binaire complet possède $2P+1$ nœuds (nombre impair) dont P sommets internes et $P+1$ feuilles.

- On appelle arbre binaire ***parfait*** un arbre binaire (complet) tel que chaque sommet est le père de **deux** sous-arbres de **même hauteur**.

Un arbre binaire parfait possède $2^{h+1}-1$ sommets, où h est la hauteur de l'arbre.

- On appelle arbre binaire ***quasi-parfait*** un arbre binaire parfait éventuellement grignoté d'un étage en bas à droite.

Exercice

1- Ecrire une procédure permettant d'ajouter un élément à un arbre binaire (on suppose que cet élément n'existe pas dans l'arbre).

2- Ecrire une procédure permettant de supprimer un élément d'un arbre binaire.

La suppression commence par la recherche de l'élément. Puis :

- si c'est une feuille, on la vire sans problèmes
- si c'est un sommet qui n'a qu'un fils, on le remplace par ce fils
- si c'est un sommet qui a deux fils, on a deux solutions :

  - le remplacer par le sommet de plus grande valeur dans le sous-arbre gauche
  - le remplacer par le sommet de plus petite valeur dans le sous-arbre droit.

Annexe 1 : Proverbes de programmation

Proverbe 1
Définissez le problème complètement.

Proverbe 2
Réfléchissez d'abord vous programmerez plus tard.

Proverbe 3
Utilisez l'analyse descendante.

Proverbe 4
Structurez vos programmes en unités logiques.

Proverbe 5
Soignez la syntaxe tout de suite.

Proverbe 6
Employez les commentaires.

Proverbe 7
Choisissez bien vos identificateurs.

Proverbe 8
Utilisez proprement les variables intermédiaires.

Proverbe 9
Ne touchez pas aux paramètres d'une boucle.

Proverbe 10
Evitez les astuces.

Proverbe 11
Ne supposez jamais que l'ordinateur suppose quelque chose.

Proverbe 12
Testez le programme à la main avant de l'exécuter.

Proverbe 13
Ne vous occupez pas d'une belle présentation des résultats avant que le programme soit correct.

Proverbe 14
Soignez la présentation.

Proverbe 15
N'ayez pas peur de tout recommencer.

Bibliographie

- C. et P. RICHARD
  Initiation à l'algorithmique
  Editions Belin, 1981.
- C. DELANNOY
  Initiation à la programmation »
  Eyrolles, 2000.
- J. COURTIN et I. KOWARSKI
  Initiation à l'algorithmique et aux structures de données
  Dunod informatique, 1987.
- G. CHATY et J. VICARD
  Programmation : cours et exercices
  Ellipses, 1992.
- C. DELANNOY
  Programmer en Turbo Pascal 7
  Eyrolles, 2000.
- B. S. GOTTFRIED
  Programmation Pascal : théorie et applications
  McGraw-Hill, 1992.
- M. SAHNOUN & F. BEN AMOR
  Informatique de base
  C.L.E, 1999.
- Micro Application
  Grand livre Visual C++ 5
  Édition 1998.
- J. MAILLEFERT
  Cours et T.P de langage C
  IUT de Cachan, 1995.
- M. GAFSI, A. ABIDI, M. JARRAYA, M. HASSEN et M.TOUAITI
  Informatique, 7ème année de l'enseignement secondaire
  Orbis impression, 1996.
- P. BRETON, G. DUFOURD et E. HEILMAN
  L'option informatique au lycée
  Editions Hachette.
- M. MOLLARET
  C/C++ et programmation objet
  Armand Colin, 1989.

Table des matières

Algorithmique et structures de données

L'objectif de ce livre est d'expliquer en termes simples et à travers de nombreux exemples et exercices corrigés les éléments nécessaires à l'écriture d'algorithmes corrects et efficaces pour traiter diverses structures de données (tableaux, enregistrements, fichiers, listes chaînées, etc.).

Ouvrage de base pour les étudiants du premier cycle universitaire (ISET, écoles d'ingénieurs, ISG, IHEC, …), ce livre constitue également une référence aux programmeurs débutants et expérimentés qui veulent rafraîchir et compléter leurs connaissances dans le domaine.

Né en 1968 à Djérissa, Baghdadi ZITOUNI est un professeur agrégé en informatique. Il ocuupe le poste de technologue à l'ISET de Ksar-Hellal depuis 1997. Il a également dispensé des cours en informatique et en GPAO à l'ISET de Sousse et à l'école d'ingénieurs de Monastir. Outre l'enseignement de l'informatique, et notamment la programmation, il a participé à plusieurs projets de développement de logiciels qui ont abouti à des produits utilisés au niveau industriel. Il est également l'auteur de plusieurs articles scientifiques publiés dans des journaux spécialisés.