Table des matières

Chapitre1: Introduction à l'algorithmique

I- Les différentes étapes de résolution d'un problème

Pour résoudre un problème en informatique, il faut passer par 5 étapes :

- Définition et analyse du problème
- Ecriture de l'algorithme
- Programmation
- Compilation du programme
- Exécution et test du programme

1. Définition et analyse du problème

Il s'agit de :

- Définir les données qu'on dispose et les objectifs qu'on souhaite atteindre
- Prévoir des réponses à tous les cas envisageables

Exemple : Si le problème est la résolution d'une équation de second degré $ax^2+bx+c=0$

→ Les données sont a, b et c

→ Les sorties sont x_1 et x_2

→ Les cas : $a=0$ et $b\neq0$, $a=0$ et $b=0$, $a\neq0$ ……

2. Ecriture de l'algorithme

C'est la phase la plus difficile et importante, elle fournit la méthode et la démarche que l'ordinateur va suivre pour résoudre le problème posé.

● *Définition d'un algorithme :*

Un algorithme est une séquence d'étapes de calcul qui utilise des données en entrée pour arriver à des résultats en sortie.

3. *Programmation de l'algorithme*

Il s'agit d'exprimer l'algorithme dans un langage connu par l'ordinateur. Il faut donc choisir un langage de programmation et ensuite traduire l'algorithme sous forme d'un programme exprimé dans ce langage.

4. *Compilation*

Il s'agit de traduire le programme écrit dans un langage de haut niveau en un programme exécutable écrit dans un langage binaire de bas niveau tout en détectant les éventuelles erreurs. Cette tâche est réalisée par le compilateur.

5. *Exécution et test du programme*

Il s'agit de s'assurer que le programme donne un résultat correct dans tous les cas et pour toutes les éventualités.

→ Effectuer plusieurs jeux de tests correspondant aux différents cas et vérifier la validité des résultats.

II- Structure générale d'un algorithme

1. *Schéma général d'un algorithme*

Un algorithme comporte généralement deux parties :

- *Partie déclarative* : elle contient l'entête, la déclaration des constantes et celle des variables.
- *Partie corps de l'algorithme* : elle consiste en une séquence d'actions faisant appel à des opérations de base de l'ordinateur.

Syntaxe :

Algorithme « nom de l'algorithme »

Const

« Liste des constantes avec leurs valeurs »

Var

« Liste des variables suivies par leurs types »

Partie déclarative

Début

« Séquence d'actions »

Fin

} **Partie corps de l'algorithme**

Une action peut être :

→ Action d'affectation ou,

→ Action d'entrée- sortie ou,

→ Action de contrôle conditionnelle simple ou à choix multiple ou,

→ Action de répétition.

2. Définition d'une variable

Une variable est un emplacement mémoire capable de contenir des valeurs de type défini au préalable. Elle peut être définie comme une boite qui admet un nom, une taille, un contenu et une adresse.

→ Le nom de la variable s'appelle identificateur de la variable.

→ La taille dépend du type de la variable (exemple : 2 octets pour un entier, 1 octet pour un caractère, 4 octets pour un réel…)

→ L'adresse désigne le numéro du 1er octet occupé par cette variable en mémoire centrale

Dans un algorithme, les variables sont déclarées comme suit :

Var

Liste des variables suivies par des virgules : type 1

Liste des variables suivies par des virgules : type 2

.

.

Liste des variables suivies par des virgules : type i

→ Dans un algorithme, on peut avoir 0 à plusieurs variables.

Exemple :

Var

X, Y : entier

A : réel

3. Définition d'une constante

La définition d'une constante est la même que celle d'une variable à la différence que la valeur d'une constante reste inchangée tout au long de l'algorithme.

Syntaxe :

Const

Nom const 1 = val 1

Nom const i = val i

Exemple:

Const

Min = 10

Max = 200

4. Les types de base

A toute variable est attribué un type qui définit :

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

Il existe des types simples qui sont prédéfinis tels que les types : entier, réel, caractère ou booléen.

a) Type entier

- Il représente l'ensemble des entiers relatifs tel que : 8, -10, 3……

- Les opérations permises sont : +, -, *, div (division entière) et mod (reste de la division entière)

b) Type réel

- Il représente l'ensemble IR

- Deux formes de représentation : La forme usuelle « a.b » exemple : -4.6, 13.9 ….. ou la forme scientifique a E b exemple : 345 = 3.45 E2 = 0.345 E3
- Les opérations permises sont : +, -, *, /

c) Type caractère

- Il peut être une lettre, un chiffre ou caractère spécial exemple : ‘a’, ‘b’, ‘3’ ….
- Les opérations permises : =, ≠, <, <=, >, >=.

d) Type booléen

- Il représente les deux valeurs ‘Vrai’ et ‘Faux’
- Les opérations : NON, ET, OU

Remarque : Il existe des types composés définis à partir des types de base comme les tableaux, les chaînes de caractère….

Chapitre 2 : Les instructions simples

I- Instruction d'affectation

Cette action permet de ranger une nouvelle valeur dans une variable

Syntaxe

Identificateur var ← <expression>

- Expression peut être :
    - Une variable
    - Une constante
    - Une expression arithmétique
    - Une expression logique

Remarque

- Une constante ne peut jamais figurer à gauche d'une affectation.
- Après une affectation, l'ancien contenu est perdu pour être substitué par le nouveau contenu.
- Une action d'affectation doit se faire entre deux types compatibles.

1. Les expressions arithmétiques

<exp-arith> op_arith <exp-arith>

- Op_arith peut être '+', '-', '/' ou '*'

  Exemple : (Y/2) + x*3

- L'ordre de priorité des opérateurs arithmétiques :
    - \- signe négatif
    - ( ) parenthèses
    - ^ puissance
    - * et / multiplication et division
    - + et – addition et soustraction

2. *Les expressions logiques*

- Les expressions logiques admettent Vrai ou Faux comme résultat.
- Elles peuvent utiliser des opérateurs relationnels (= , ≠, <,<=, >, >=) ou des opérateurs logiques (NON, ET, OU)
- L'ordre de priorité est :

NON	>
ET	>=
OU	<
	<=
	=
	≠

Exemple : (x<6) ET (Y = 20) donne vrai si x<6 et Y = 20 et faux sinon

II- Instruction de lecture ou d'entrée

- Elle permet d'affecter, à une variable, une donnée introduite) partir d'une périphérique d'entrée (clavier).
- Syntaxe :

Lire (nom_var1, nom_var2,……)

- Exemple :

Lire(A) : lit une valeur à partir du périphérique d'entrée et la range dans la case mémoire associée à A.

Lire(X,Y) : lit deux valeurs la première pour X et la deuxième pour Y.

III- Instruction d'écriture ou de sortie

- Elle permet d'afficher des résultats sur un périphérique de sortie (écran). Ce résultat peut être :
  - Une chaîne de caractères délimitée par des '' ''
  - La valeur d'une variable dont le nom est spécifié

- La valeur d'une expression

Syntaxe :

Ecrire (Liste d'expressions séparées par des virgules)

- L'ordinateur évalue tout d'abord l'expression puis affiche le résultat obtenu

Exemple :

Lire (somme)

Lire(Nbre)

Ecrire (''La moyenne est :'', somme / Nbre)

Si l'utilisateur introduit 120 pour somme et 10 pour Nbre alors l'affichage sera : La moyenne est 12.

- ***Exercice 1 :***

  Ecrire un algorithme qui lit deux entiers X et Y et affiche leurs valeurs avant et après permutation

- ***Exercice 2 :***

  Ecrire un algorithme qui lit trois entiers et qui calcule et affiche leur somme, leur produit et leur moyenne.

Chapitre 3: Les structures conditionnelles

Introduction

En programmation, on est souvent confronté à des situations ou on a besoin de choisir entre 2 ou plusieurs traitements selon la réalisation ou non d'une certaine condition d'ou la notion de traitement conditionnel. On distingue deux structures de traitement conditionnel à savoir :

- La structure conditionnelle simple qui consiste à évaluer une condition (expression logique à valeur vrai ou faux) et d'effectuer le traitement relatif à la valeur de vérité trouvée.
- La structure conditionnelle à choix multiple qui consiste à évaluer une expression qui n'est pas nécessairement à valeur booléenne (elle peut avoir plus de deux valeurs) et selon la valeur trouvée, effectue un traitement.

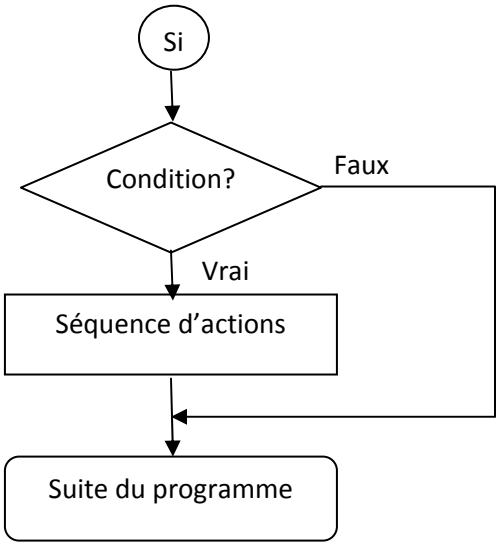
I- Structure conditionnelle simple

1. Forme simple : (Si …. Alors ….. Finsi)

Syntaxe :

Si condition **Alors**
action(s)
Fin si

Dans cette forme, la condition est évaluée. Si elle vaut vrai alors c'est la séquence d'actions qui est exécutée sinon c'est l'action qui suit l'action conditionnelle dans l'algorithme qui est exécutée.
L'exécution de cette instruction se déroule selon l'organigramme suivant :



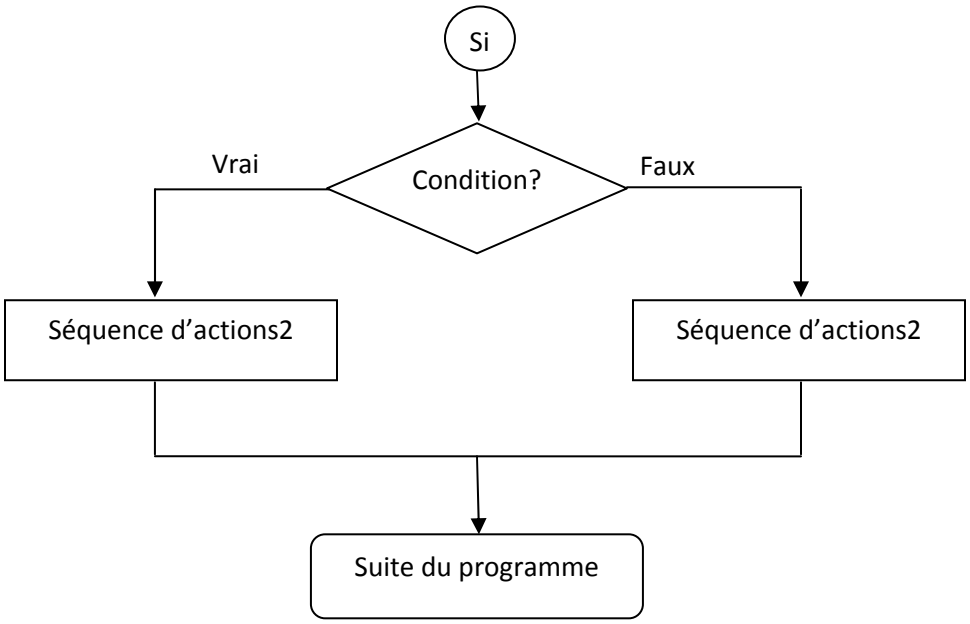
Exemple 1 : Ecrire un algorithme qui permet de saisir un entier et d'afficher impossible d'être diviseur si cet entier est égal à 0.

2. Forme composée : (Si … alors…….sinon)

Syntaxe :

Si condition **Alors**
  Action(s)1
Sinon
  Action(s)2
Fin si

Dans cette forme, la condition est évaluée. Si elle vaut vrai alors c'est la séquence d'actions 1 qui sera exécutée sinon c'est la séquence d'actions 2 qui sera exécutée.
L'exécution de cette instruction se déroule selon l'organigramme suivant :



Exemple 2 :
Ecrire un algorithme qui permet de saisir un entier et d'afficher « pair » si cet entier est pair ou « impair » si cet entier est impair.

3. Forme imbriquée

Syntaxe

Si condition 1 **Alors**
  Action(s)1
Sinon
  Si condition 2 **Alors**
    Action(s)2

Sinon
Si condition N-1 **Alors**
Action(s)N-1
Sinon
Action(s)N
Fin si

Si la condition est vraie, alors la séquence d'actions 1 sera exécutée sinon on évalue la condition 2 si elle est vraie la séquence d'actions 2 sera exécutée. Enfin, si aucune des N-1 conditions est vraie alors on exécute la séquence d'actions N.

Exemple :

Ecrire un algorithme qui permet de saisir deux entiers A et B puis teste si A>B ou A<B ou A=B.

II- Structure conditionnelle à choix multiple

Syntaxe

Selon <sélecteur> faire
<liste de valeurs1> : <traitement 1>
<liste de valeurs2> : <traitement 2>
…..
…..
<liste de valeursN> : <traitement N>
Sinon
<traitement N+1>
Fin selon

- Le sélecteur est un identificateur
- <traitement i> est une séquence d'actions.
- <liste de valeurs i> peut être une constante ou un intervalle de constantes de même type que sélecteur.
- La partie sinon est facultative. Elle est exécutée si aucune des valeurs n'est égale au sélecteur.

Exemple :

Ecrire un algorithme qui permet de lire un numéro de jour de la semaine (compris entre 1 et 7) et d'afficher le nom du jour en toute lettre.

Chapitre 4: Les structures itératives

Introduction

On peut exécuter une action ou un ensemble d'actions non pas infiniment mais un certain nombre de fois : c'est la notion de boucles.

I- La structure « Pour ….. faire…….Finpour »

Syntaxe

Pour vc de vi à vf faire
    Traitement
Finpour

- Vc : compteur de type entier
- Vi et vf : valeur initiale et valeur finale de vc
- Traitement : action ou séquence d'actions à répéter (vf-vi +1) fois.
- La boucle Pour est utilisée lorsque le nombre d'itération est connu à l'avance.
- Vc reçoit une valeur initiale vi pour la première fois, il ne doit pas être modifié par une action de traitement à l'intérieur de la boucle.
- Vc est incrémenté automatiquement par 1 à chaque exécution du corps de la boucle Pour. Cette valeur d'incrémentation est appelée le pas de la boucle.
- L 'exécution de la boucle finit lorsque vc atteind vf.

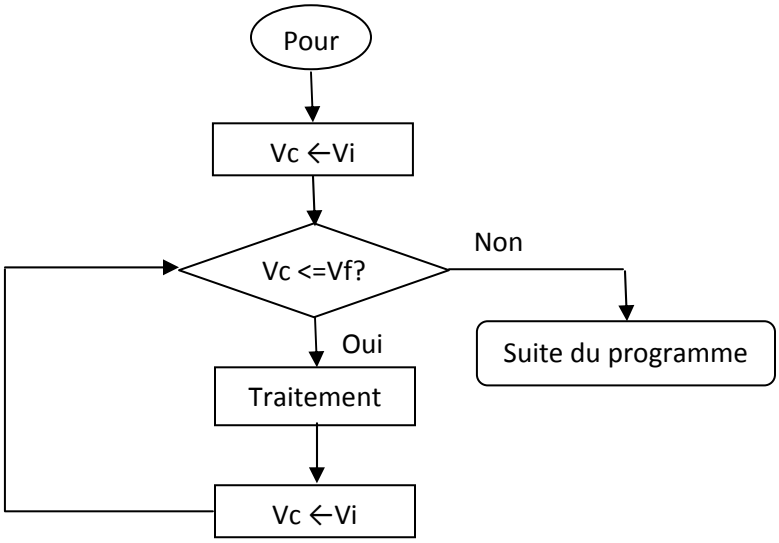


Schéma d'exécution d'une boucle « Pour »

Exemple :
Pour i de 1 à 5 faire
Ecrire (i * 100)
Fin pour
Exécution : i = ~~1~~ ~~2~~ ~~3~~ ~~4~~ ~~5~~ 6
Résultat 100 200 300 400 500

Remarques :
- Une boucle peut être exécutée une ou plusieurs fois.
- Si le pas est différent de 1, il faut ajouter l'option (pas = constante)

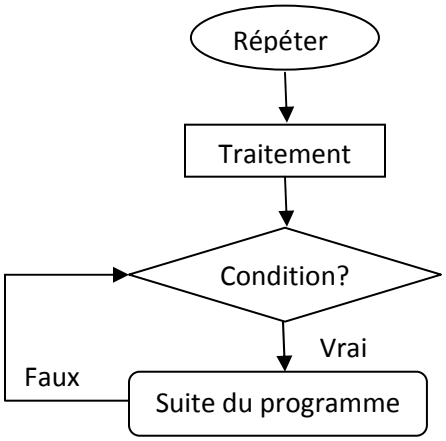
Exemple
Pour i de 5 à 1 (pas = -1) faire
Ecrire (i * 100)
Fin pour
Exécution : i = ~~5~~ ~~4~~ ~~3~~ ~~2~~ ~~1~~ 0
Résultat 500 400 300 200 100

Exemple 1
Ecrire un algorithme qui permet de calculer et d'afficher la somme des nb premiers entiers naturels (nb est saisi à partir de clavier).
Exemple 2
Ecrire un algorithme qui lit un entier n qu'on suppose positif puis affiche tous ses diviseurs.

II- La structure « Répéter …. Jusqu'à »

Syntaxe
Répéter
Traitement
Jusqu'à (condition)

- Condition : condition d'arrêt et de sortie de la boucle
- Traitement : action ou ensemble d'actions à exécuter tant que la condition n'est pas vérifiée, dés qu'elle soit vérifiée, l'exécution du traitement s'arrête.
- Le nombre de répétition n'est pas connu à l'avance.
- Le traitement est exécuté au moins une fois quelque soit le résultat de la condition.
- La condition doit être initialisée avant le début de la boucle et doit être modifiée à l'intérieur de la boucle.



Exemple

i ← 1

Répéter

    Ecrire (i *100)

    i← i +1

jusqu'à (i > 5)

Exécution : i = ~~1~~ ~~2~~ ~~3~~ ~~4~~ ~~5~~ 6

Résultat 100 200 300 400 500

Exemple

Ecrire un algorithme qui permet de calculer la somme des nb premiers entiers en utilisant la boucle répéter jusqu'à

Exemple

Ecrire un algorithme qui permet de calculer la factorielle d'un entier n donné (on suppose que n est un entier positif)

III- La structure « Tantque……. Faire…….. Fintantque »

Syntaxe

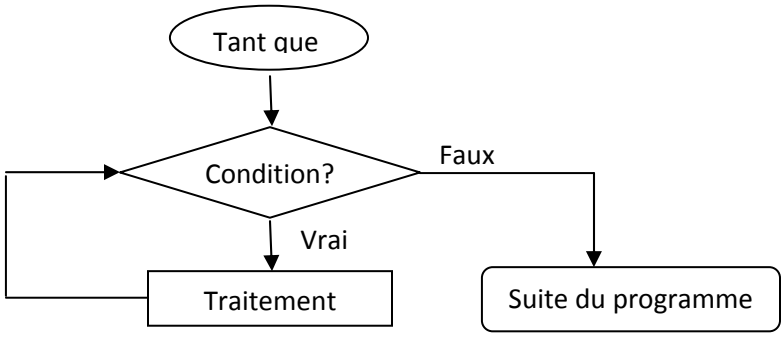
Tantque (condition) faire

        Traitement

Fintantque

- Condition : condition de maintien de la boucle.
- Traitement : Action ou ensemble d'actions à exécuter tant que la condition est vérifiée.
- Le traitement est exécuté tant que la condition est vérifiée sinon on sort de la boucle.
- Si la condition n'est pas vraie dés la première exécution, la boucle ne sera jamais exécutée (0 fois).

- Le nombre de répétition n ‘est pas connu à l’avance.
- La condition doit être initialisée avant la boucle et modifiée à l’intérieur de la boucle.



Exemple

i ← 1
tantque (i≤5)
    Ecrire (i *100)
    i← i +1
Fintantque

Exécution : i = ~~1~~ ~~2~~ ~~3~~ ~~4~~ ~~5~~ 6
Résultat 100 200 300 400 500

Exemple

Reprendre les exercices précédents en utilisant la boucle tant que.

Chapitre 5 : Les sous programmes

Introduction

Le but de l'utilisation de sous programmes :

- décomposition des problèmes en modules (sous problèmes de taille réduite) :
    - Dans un programme plusieurs séquences d'instructions sont appelées plusieurs fois et depuis divers points du programme. Il serait donc plus intéressant d'isoler ces séquences dans un sous programme qui peut être appelé depuis n'importe quel point du programme.
    - L'approche modulaire réduit énormément le nombre d'instructions redondantes (qui se répètent) moyennant l'ajout d'une séquence d'appel pour le module à différents endroits du programme. D'où la réduction de la taille du programme (code).
    - La lisibilité qui facilite notablement la compréhension du programme

- Réutilisation du sous programme

En résumé, le programme sera plus lisible et plus facile à maintenir (à modifier éventuellement par la suite)

Un sous programme est portion de code analogue à un programme. Déclarée dans un programme ou dans un sous programme et dont la partie instruction peut être exécutée plusieurs fois au cours du traitement du programme grâce à des appels. On distingue deux formes de sous programmes : Les procédures et les fonctions.

I- Les procédures

Une procédure est un sous programme qui effectue un traitement (suite d'instructions)

1. Les procédures sans paramètre

Elles sont utilisées pour éviter d'avoir à réécrire plusieurs fois une même suite d'instructions figurant plusieurs fois dans le programme.

Déclaration

Procédure nom_procédure
Déclarations
Début
    Suite d'instructions
Fin

Appel (utilisation)

nom_procédure

Exemple

```
Algorithme Principal
Var x, y : entier
Procédure Affiche                    }
Var                                  }
Début                                } Déclaration de la procédure
        Ecrire (''Bonjour'')         }
Fin                                  }
Début
        Lire(x)
        Lire(y)
        Permute
        Ecrire(x,y)
Fin
```

2. Les procédures avec paramètres

Exemple

```
Algorithme Principal
Var x, y : entier
Procédure Permute(var u,v : entier)
Var z : entier
Début
        z<- u
        u<-v
        v<-z
Fin
Début
        …..
        Permute(x,y)
        ….
        Permute(a,b)
        ….
        Permute(m,n)
Fin
```

- *Paramètres formels*

Une déclaration de procédure peut comporter après le nom de la procédure une liste de paramètres formels dont la syntaxe est la suivante.

Procédure nom_procédure(liste de paramètres formels)

Exemple

```
Procédure Somme(a, b : entier, var c : entier)
Début
c<-a+b
```

Fin

- *Paramètres effectifs*

Au cas où la déclaration d'une procédure comprend des paramètres formels, chaque appel de cette procédure doit comporter des paramètres effectifs compatibles dont la syntaxe est la suivante.

nom_procédure(liste de paramètres effectifs)

Il faut que les deux listes de paramètres formels et effectifs aient le même nombre de paramètres et que les paramètres formels et effectifs correspondants soient compatibles.

Exemple

```
Algorithme Principal
Var t :réel
    X, y : entier
Début
        …
        Permute(x,y,z)          // interdit parce que le nombre de paramètres formels est
                                // différent du nombre de paramètres effectifs
        …
        Permute(x,t)            // interdit parce que les paramètres formels et effectifs ne sont
                        // pas compatibles
Fin
```

3. Passage de paramètres par valeur

les paramètres effectifs sont lus à l'appel de la procédure puis leurs valeurs sont affectées à des variables temporaires locales à la procédure.

Exemple

```
Algorithme passage_valeur
Var x : entier
Procédure Incrémenter(y : entier)
Début
        Ecrire(y)
        y<-y+1
        Ecrire(y)
Fin
Début
        x<-0
        Ecrire(x)
        Incrémenter(x)
        Ecrire(x)
Fin
```

Le x reste intact=0 mais en même temps on veut l'incrémenter. Donc on utilise cette procédure.

4. Passage de paramètres par variable

Les variables d'entrée de la procédure (paramètres effectifs) sont liés aux paramètres formels. Pendant l'exécution du corps de la procédure toute action sur les paramètres formels s'exécutera sur les paramètres effectifs correspondants. Par conséquent, à la sortie de la procédure, les variables peuvent avoir leurs contenus changés.

Exemple

```
Algorithme Passage variable
Var x : entier
Procédure Incrémenter(var y : entier)
Début
        Ecrire(y)
        y<-y+1
        Ecrire(y)
Fin
Début
        x<-0
        Ecrire(x)
        Incrémenter(x)
        Ecrire(x)
Fin
```

Résumé

Le passage de paramètres par valeur est utilisé pour transmettre une valeur à la procédure.
Le passage de paramètres par variable est utilisé pour que la procédure puisse modifier la valeur d'une variable du programme appelant.

II- Les fonctions

Une fonction est un sous programme qui renvoie une valeur d'un seul type. Ce type sera celui de la fonction. La valeur retournée par la fonction dépend en général des paramètres formels (et des variables globales)

- Variables globales : elles sont déclarées à l'extérieur des sous programmes
- Variables locales : elles sont déclarées à l'intérieur du sous programme

Exemple

```
Algorithme Principal
Var x, y : entier          //variables globales
Procédure Proc(z : entier)
Var T :réel                //variable locale
Début
        ….
Fin
Début
        ….
Fin
```

Une même variable peut apparaître localement dans deux sous programmes différents.

Déclaration

Fonction nom_fonction(liste des paramètres formels) : type du résultat retourné
Var Déclarations des variable
Début
    Corps fonction
Fin

Exemple

```
Fonction Max(x, y : entier) : entier
Début
    Si x >=y
    Alors Max<-x
    Sinon Max<-y
    Finsi
Fin
```

Remarque

Le corps de la fonction doit contenir au moins une instruction de retour de la valeur de la fonction comme suit :

nom_fonction <- expression

expression doit être de même type que la fonction

Appel de fonction

Un appel d'une fonction se fait dans une expression

Exemple

```
Algorithme Maximum4réels
Var
    x1, x2, x3, x4 : réel
    y1, y2, y3 : réel
Fonction Max(x, y : réel) :réel
Début
    Si x<=y
    Alors Max<-x
    Sinon Max <-y
    Finsi
Fin
Début
    Lire(x1, x2, x3, x4)
    y1<-Max(x1, x2)
    y2<-Max(x3, x4)
    y 3<-Max(y1,y2)
```

Ecrire(« le maximum est : »,y3)

Fin

Remarque

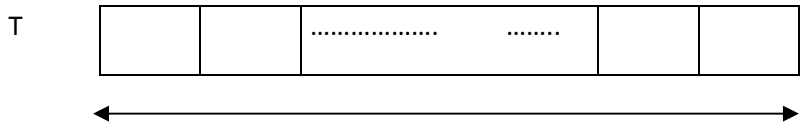
La dernière des valeurs affectées constitue le résultat de l'évaluation de l'appel de la fonction.

Chapitre 6: Les tableaux

I- Les tableaux à une dimension

1- Définition

Un tableau T est une variable structurée formée d'un nombre entier N de variables simples de même type, qui sont appelées les composantes du tableau.
Le nombre de composantes N est alors la dimension du tableau.



N composantes

On dit encore que T est ***un vecteur*** de dimension N.

2- Utilité

- Un tableau est une structure de données constituée d'un nombre fini d'éléments de **même type.**
- Lorsque plusieurs données de même type, généralement destinées au même traitement doivent être accessibles le long d'un programme, on propose d'utiliser la structure d'un tableau.

3- Composantes

Nom : identificateur d'un tableau.

Type-élément : Les éléments d'un tableau sont caractérisés par leur type (entier, réel, caractère,……..).

Indice : Tout type dont les éléments possèdent un successeur (les types scalaires), généralement de type entier.

4- Déclaration

***Nom_tab* : Tableau [premind. .deuxind] *de* type_élément**

Exemples :

T1 : Tableau [1..50] d'entier

T2 : Tableau [1..20] de réel

T3 : Tableau [1..20] de caractère

Remarque :

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

CONST

Nmax = 50

TYPE

Tab : Tableau [1..nmax] d'entier

VAR

T : tab

5- Accès aux composantes d'un tableau

Considérons un tableau T de dimension N

- L'accès au premier élément du tableau se fait par **T[1]**
- L'accès au dernier élément du tableau se fait par **T[N]**

Exemple :

Nom : T	100	200	300	400	500
Indice :	1	2	3	4	5
Contenu	T[1]	T[2]	T[3]	T[4]	T[5]

6- Chargement d'un tableau

Ecrire un algorithme qui permet de remplir un tableau de 5 entiers.

```
ALGORITHME Chargement

VAR

T : Tableau [1..5] d'entier

    i : entier

Début

    Pour i de 1 à 5 Faire

        Ecrire ("T [" , i , "] :")
```

```
            Lire(T[i])
        Fin pour
Fin
```

7- Affichage du contenu d'un tableau

```
ALGORITHME AFFICHER
VAR
T : Tableau [1..5] d'entier
        i : entier
Début
        Pour i de 1 à 5 Faire
        Ecrire (T[i])
        FinPour
Fin
```

8- Méthodes de recherche dans un tableau

8-1- La recherche séquentielle

Problème : Déterminer la première position d'une valeur donnée dans un tableau de N élément. Résoudre ce problème en utilisant la notion de procédures/Fonctions

```
Algorithme RECHERCHE
CONST
        Nmax = 50
TYPE
        Tab : Tableau [1..nmax] d'entier
VAR
        T : tab
        N , val : entier
/*Procédure CHARGEMEMENT*/
Procédure CHARGEMEMENT ( VAR  T : tab ; N :entier)
VAR
        i : entier
DEBUT
        Pour i de 1 à N Faire
                Ecrire ("T [" , i , "] :")
                Lire(T[i])
        Fin pour
FIN
/*Procédure AFFICHE*/
```

```
Procédure AFFICHE ( T : tab ; N :entier)
VAR
        i : entier
DEBUT
        Pour i de 1 à N Faire
        Ecrire ( T[i])
        FinPour
FIN
/*Procédure INDICE*/
Fonction INDICE ( T : tab ; N, val :entier) : entier
VAR
        i, pos : entier
DEBUT
pos← -1
i← 1
Tant que (i ≤ N et pos = -1 ) Faire
        Si (T[i] = val ) alors
                pos ←  i
        Sinon
                i ← i+1
        Finsi
FinTantque
        INDICE ← pos
FIN
/*Programme Principal*/
DEBUT ( P.P)
        Répéter
Ecrire("Donner la taille de T :")
Lire(N)
        Jusqu'à (N>1 et N<=nmax)
Ecrire (" Chargement de T ")
CHARGEMEMENT ( T , N )
Ecrire (" Affichage de T ")
AFFICHE( T , N )
Ecrire (("Donner la valeur à chercher dans T :")
        Lire(val)
Si( INDICE ( T , N, val ) = -1) alors
        Ecrire (val , "n'existe pas dans T ")
sinon
        Ecrire (val , "existe à la position", INDICE ( T , N, val ), "dans T ")
Finsi
FIN
```

8-2- La recherche dichotomique

Problème : Déterminer la première position d'une valeur donnée dans un tableau de N élément ***triés*** dans le sens croissant. Résoudre ce problème en utilisant la notion de procédures/Fonctions.

Principe :

Le principe est de décomposer le tableau T en deux sous tableaux. Trois cas peuvent se produire :

Si val = T[milieu] alors val est trouvé et la recherche est terminée.

Si val < T[milieu] alors on va chercher val dans la partie gauche du tableau T.

Si val > T[milieu] alors on va chercher val dans la partie droite du tableau T.

On poursuit la recherche tant que T[milieu] est différent de val est tant que la dimension de sous tableau reste valide.

```
Fonction Dichotomique ( T : tab ; N, val :entier) : entier
VAR
        i, pos,mil, inf, sup : entier
DEBUT
pos← -1
inf← 1
sup← N
Tant que ( inf ≤ sup et pos = -1 ) Faire
        mil ← (inf + sup) div 2
        Si (T[mil] = val ) alors
                pos = mil
        Sinon
                Si (val<T[mil]  ) alors
                        sup← mil - 1
                sinon
                        inf ← mil + 1
                finsi
        Finsi
FinTantque
        INDICE ← pos
FIN
```

9- Méthodes de tri dans un tableau

9-1- Tri par sélection (par minimum)

<u>Principe</u> :

Le principe de cette méthode est simple. Elle consiste à :

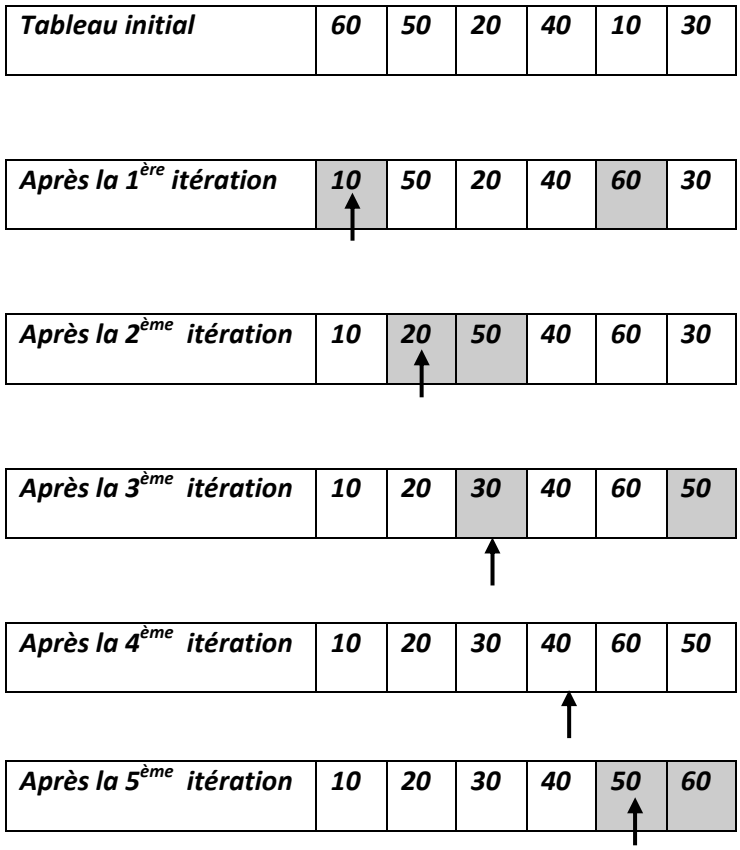
Chercher l'indice du plus petit élément du tableau T[1..n] et permuter l'élément correspondant avec l'élément d'indice 1;

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 TRISELECTION (VAR T : tab ; N : entier)
VAR
        i, j, aux, indmin : entier

DEBUT

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

 Si (i ≠ indmin ) alors
                        aux  ← T[i]
                        T[i] ← T[indmin]
                        T[indmin] ← aux
Finsi

FinPour

FIN
```



9-2- Tri à bulles

Principe **:**

Cet algorithme porte le nom de tri bulle car, petit à petit, les plus grands éléments du tableau remontent, par le jeu des permutations, enfin de tableau. Dans un aquarium il en va de même : les plus grosses bulles remontent plis rapidement à la surface que les petites qui restent collés au fonds.

Il existe plusieurs variantes de cet algorithme :

Une méthode consiste à faire descendre les plus petites valeurs au début du tableau. Dans ce cas, le tableau est parcouru de droite à gauche.

```
                                /*Procédure TRISBULLE */
Procédure TRISBULLE (VAR T : tab ; N : entier)
VAR
        i, j, aux : entier
DEBUT
        Pour i de 1 à n-1 faire
        j ← n
```

```
        Tantque (j ≠ i) faire
                Si (T[j] < T[j - 1] ) alors
                        aux  ← T[j]
                        T[j] ← T[j - 1]
                        T[j - 1] ← aux
                Finsi
                 j  ← j – 1
        FinTantque
         FinPour
FIN
/*Programme Principal*/
DEBUT ( P.P)
        N ← SAISIE_TAILLE ()
        Ecrire (" Chargement de T ")
CHARGEMEMENT ( T , N )
Ecrire (" Affichage de T  avant tri")
AFFICHE ( T , N )
        TRISBULLE (T, N)
Ecrire (" Affichage de T  après tri")
AFFICHE ( T , N )
FIN
```

Tableau initial	***50***	***30***	***20***	***40***	***10***
1ère étape	***50***	***30***	***20***	***10***	***40***
	50	***30***	***10***	***20***	***40***
	50	***10***	***30***	***20***	***40***
	10	***50***	***30***	***20***	***40***
2ème étape	***10***	***50***	***20***	***30***	***40***

10	*20*	*50*	*30*	*40*

3ème étape	*10*	*20*	*30*	*50*	*40*

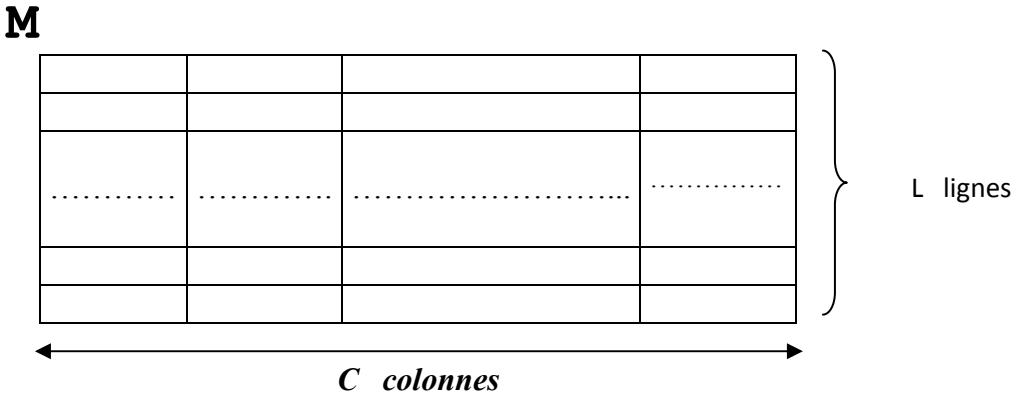
4ème étape	*10*	*20*	*30*	*40*	*50*

II- Les tableaux à deux dimensions

1- Définition

Un tableau à deux dimensions A et à interpréter comme un tableau (unidimensionnel) de dimension **L** dont chaque composante est un tableau (unidimensionnel) de dimension **C**.

On appelle **L** le **nombre de lignes** du tableau et **C** le **nombre de colonnes** du tableau.
Un tableau à deux dimensions contient **L*C** composantes.



2- Déclaration

***Nom_tab* : Tableau [premind. .deuxind , premind. .deuxind] *de* type_élément**

Exemples :
M1 : Tableau [1..30, 1..30] d'entier
M2 : Tableau [1..20, 1..20] de réel
M3 : Tableau [1..20, 1..20] de caractère
Remarque :
Il est également possible de définir une matrice comme dans l'exemple suivant :
CONST
NL = 30

NC = 20

TYPE

MAT : Tableau [1.. NL, 1.. NC] d'entier

VAR

M : MAT

3- Accès aux composantes d'une matrice

Considérons un tableau M de L lignes et C colonnes.

- Les indices du tableau varient de 1 à L, respectivement de 1 à C.
- La composante de la N^{ième} ligne et M^{ième} colonne est notée : A[N,M].

Syntaxe :

<Nom du tableau>[<ligne> ,<colonne>]

Exemple **:** Considérons une matrice de 3 lignes et 4 colonnes

	1	***2***	***3***	***4***
1	**A[1 ,1]**	**A[1,2]**	**A[1,3]**	**A[1 ,4]**
2	**A[2 ,1]**	**A[2 ,2]**	**A[2 ,3]**	**A[2 ,4]**
3	**A[3 ,1]**	**A[3 ,2]**	**A[3 ,3]**	**A[3 ,4]**

4-Chargement d'une matrice

```
Algorithme Chargement

VAR

        M : Tableau [1.. 3, 1..4] d'entier
        i ,j : entier
Début

        Pour i de 1 à 3 Faire
                Pour j de 1 à 4 Faire
                        Ecrire ("M [" , i , ", " , j ,  "] :")
                        Lire (M [i, j])
                Fin pour
        Fin pour
Fin
```

5-Affichage du contenu d'une matrice

```
Algorithme Afficher
VAR
        M : Tableau [1.. 3, 1..4] d'entier
        i,j : entier
Début
        Pour i de 1 à 3 Faire
                Pour j de 1 à 4 Faire
                        Ecrire ( M[i, j])
                Fin pour
Fin pour
Fin
```

Exemple 1

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 + M3

Rappel:

```
/     M1    \   /      M2       \     /          M3           \
| a b c d |   | a' b' c' d' |     | a+a' b+b' c+c' d+d' |
| e f g h | + | e' f' g' h' |  =  | e+e' f+f' g+g' h+h' |
| i j k l |   | i' j' k' l' |     | i+i' j+j' k+k' l+l' |
\           /   \             /     \                     /
```

Procédure SOMME (M1 , M2 : MAT ; VAR M3 : MAT ;n, m : entier)

VAR

        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]
                Fin pour
        Fin pour

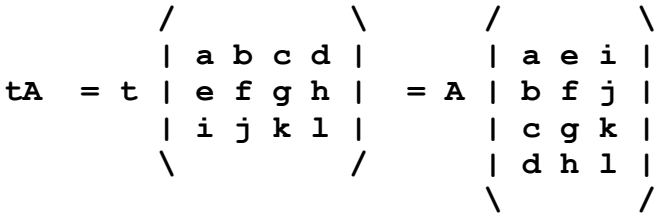
Fin

Exemple 2

Ecrire un algorithme qui effectue la transposition t_A d'une matrice A de dimensions N et M en une matrice de dimensions M et N.

La matrice A sera transposée par permutation des éléments. Résoudre ce problème en utilisant la notion de procédures/Fonctions.

Rappel:



Algorithme CHANGEMENT

CONST

Nmax = 50

TYPE

MAT : Tableau [1.. Nmax, 1.. Nmax] d'entier

VAR

A : MAT

N , M : entier

/*Procédure SAISIE_TAILLE */

Fonction SAISIE_TAILLE() : entier

VAR

nb : entier

DEBUT

Répéter

Ecrire("Donner la taille de T :")

Lire(nb)

Jusqu'à (nb>1 et nb<=nmax)

SAISIE_TAILLE ← nb

FIN

/*Procédure CHARGEMEMENT*/

Procédure CHARGEMEMENT (VAR A : MAT ; N, M : entier)

VAR

i , j: entier

DEBUT

Pour i de 1 à N Faire

Pour j de 1 à M Faire

Ecrire ("A [" , i , ", " , j , "] :")

```
                    Lire (A [i, j])
                Fin pour
        Fin pour
FIN
/*Procédure AFFICHE*/

Procédure AFFICHE (A : MAT ; N, M :entier)

VAR

        i, j : entier

DEBUT
        Pour i de 1 à N  Faire
                Pour j de 1 à M  Faire
                        Ecrire (A [i, j])
                Fin pour
        Fin pour
FIN

/*Procédure transposée*/

Procédure TRANSPOSEE (VAR  A : MAT ; N, M : entier)

VAR

        i, j, Dmax, aux : entier

DEBUT
        Si ( N > M ) alors
        Dmax  ← N
        Sinon
        Dmax  ← M
        Finsi
        Pour i de 1 à Dmax  Faire
                Pour j de 1 à i  Faire
                        aux ← A[i, j]
                        A[i , j] ← A[j , i]
                        A[j , i] ← aux
                Fin pour
        Fin pour
FIN

/*Programme Principal*/
```

DEBUT (P.P)

Ecrire (" Saisie des tailles ")
N ← SAISIE_TAILLE ()
M ← SAISIE_TAILLE()

Ecrire (" Chargement de A ")
CHARGEMEMENT (A, N, M)
Ecrire (" Affichage de A avant transposition ")
AFFICHE (A, N, M)

TRANSPOSEE (A, N, M)

Ecrire (" Affichage de A après transposition ")
AFFICHE (A, M, N)

FIN

Chapitre 7 : La récursivité

I- Notion de récursivité

Une des caractéristiques les plus importantes de programmation est la possibilité pour une **procédure** ou une **fonction** de s'appeler elle-même. On parle de ***récursivité***.

La récursivité est particulièrement utile pour traiter tous les problèmes formalisables de façon récursive, bien qu'il soit possible de programmer des solutions n'utilisant pas la récursivité pour ces problèmes.

Syntaxe

```
Fonction Récursive (par : type_par) : typer_retour
Variable résultat : type_retour
Début
        Si (cdt_arret) alors
                résultat ← val_initiale
        sinon
                résultat ← /* Appel de la fonction Récursive (par2)*/
        finsi
    retour (résultat)
Fin
```

II- Etude d'un exemple

On peut définir le factoriel d'un nombre n non négatif de 2 manières.

Définition non récursive

N ! = N * N-1 * ……. 2 * 1

Définition récursive :

N ! = N * (N – 1) ! et 0 ! = 1

Solution itérative :

```
Fonction FACT ( n : entier ) : entier
VAR
        i, F: entier
DEBUT
        Si (n = 0 ) alors
                FACT  ← 1
        Sinon
                F ← 1
                Pour i de n à 1 ( pas = -1 ) faire
```

```
                F ← F * i
            Finpour
            FACT ← F
        Finsi
FIN
```

Solution récursive :

```
Fonction FACT ( n : entier ) : entier
VAR
        F : entier
DEBUT
        Si (n = 0 ) alors
                F ← 1
        Sinon
                F ←  n * FACT (n-1)
        Finsi
    FACT ← F
FIN
```

/* PROGRAMME PRINCIPAL*/

```
DEBUT ( P.P )
        Répéter
                Ecrire ("Donner un entier n : ")
                Lire ( n )
        Jusqu'à (n ≥ 0 )
        Écrire ("La factorielle de   ", n , " = " , FACT ( n ))
FIN
```

III- Interprétation

Chaque **procédure** ou **fonction récursive** doit comporter une condition d'arrêt (dans notre exemple n=0). Cette condition empêche des appels récursifs sans fin. Habituellement, la condition d'arrêt se présente sous la forme d'une instruction **si…… alors……sinon** qui permet de stopper la récurrence si la condition d'arrêt est satisfaite. Par contre, tant que la condition d'arrêt n'est pas remplie, la procédure (ou la fonction) s'appelle au bon endroit.

On remarque que le processus récursif remplace en quelque sorte la boucle. On remarque aussi qu'on traite le problème à l'envers : on part du nombre, et on remonte à rebours jusqu'à 1 pour pouvoir calculer la factorielle. Cet effet de rebours est caractéristique de la programmation récursive.

Pour la solution récursive : La fonction **FACT** est appelée successivement, une fois dans le programme principal et (N-1) façon depuis elle-même, de façon totalement transparente pour l'utilisateur. Le **seul résultat visible est la réponse finale**.

Lorsque s'exécute un programme récursif, les appels successifs de la fonction (ou procédure) récursive ne sont pas exécutés immédiatement. Ils sont de faits placés dans **une pile** jusqu'à ce que soit atteinte la condition d'arrêt du processus récursif.

Les appels de fonction sont alors exécutés en ordre inverse, au fur et à mesure qu'ils sont retirés de la pile.

Une pile est une structure de données de type **LIFO** (**L**ast **I**n, **F**irst **O**ut) ou **« Dernier Entrée, Premier Sortie »**

IV- Mécanisme de fonctionnement de la récursivité

L'évaluation récursive d'une factorielle donne lieu au traitement, dans l'ordre suivant, des appels de la fonction :

Donner un entier n : 4

FACT (4)
4*FACT (3)
3* FACT (2)
2* FACT (1)
1* FACT (0)
1

Les valeurs effectives sont retournées dans l'ordre inverse :

0 ! = 1
1 ! = 1 * 0 ! = 1 * 1 = 1
2 ! = 2 * 1 ! = 2 * 1 = 2
3 ! = 3 * 2 ! = 3 * 2 = 6
4 ! = 4 * 3 ! = 4 * 6 = 24

Remarques

- La programmation récursive, pour traiter certains problèmes, est **très économique pour le programmeur** ; elle permet de faire les choses correctement, en très peu d'instructions.
- **Tout problème formulé en termes récursifs peut également être formulé en termes itératifs !**

Exercice1 :

Ecrire une fonction **Fib** récursive qui calcule le terme F_n de la suite de Fibbonaci :

$\mathbf{F_0 = 0,}$

$\mathbf{F_1 = 1,}$

$\mathbf{F_n = F_{n-1} + F_{n-2}}$

```
Fonction Fib ( n : entier ) : entier
VAR
        f : entier
```

```
Début
        Selon (n) faire
                0: f ← 0
                1: f ← 1
        Sinon
                f ← Fib (n - 1) + Fib (n - 2)
        Finselon

    Fib ← f
Fin
```

Exercice2 :

Ecrire une fonction puissance **récursive qui donne la puissance entière (positive) d'un nombre réel.**

```
Fonction Puiss( x : réel ; n : entier) : réel
VAR
        P : réel
Début
        Si (n = 0) alors
                P ← 1
        Sinon
                P ← x * Puiss(x, n - 1)
        Finsi
    Puiss ← P
Fin
```

Exercice3 :

Ecrire une procédure ENVERS **récursive qui permet d'afficher les éléments d'un tableau d'entiers à l'envers.**

```
Procédure AFFICHER ( T : tab ; n : entier)
Début
        Si (n ≠ 0) alors
            Écrire (T[n])
            AFFICHER (T , N-1)
        Finsi
Fin
```

Chapitre 8: Les pointeurs

Introduction

La plupart des langages de programmation offrent la possibilité d'accéder aux données dans la mémoire de l'ordinateur à l'aide de ***pointeurs***, c.-à-d. à l'aide de variables auxquelles on peut attribuer les ***adresses*** *d'autres variables*.

I- Adressage de variables

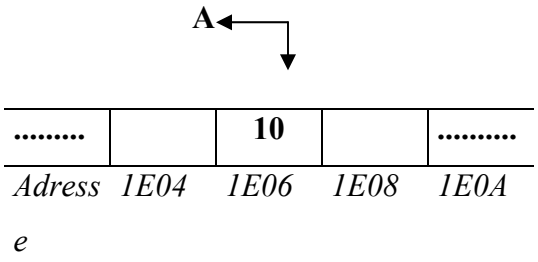
1. Adressage direct

Accès au contenu d'une variable par le **nom** de la variable.

Exemple :

A : entier

A ← 10



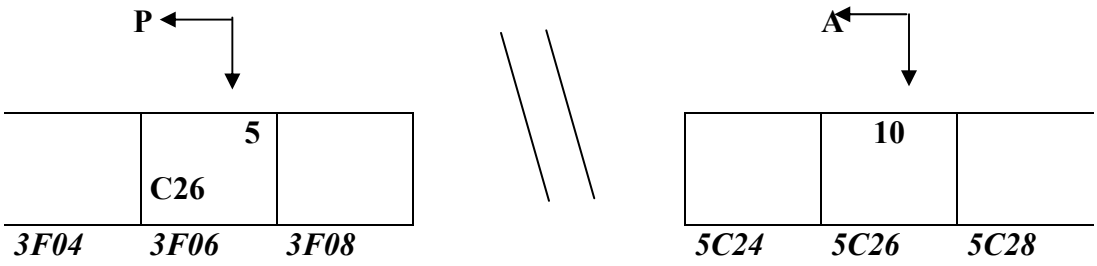
2. Adressage indirect

Si nous ne voulons pas utiliser le nom d'une variable A, nous pouvons **copier l'adresse** de cette variable dans une variable spéciale **P** appelée **Pointeur**. Ensuite, nous pouvons retrouver l'information de la variable **A** en **passant par le pointeur P**.

Accès au contenu d'une variable en passant par un pointeur qui contient l'adresse de la variable.

Exemple :

Soit A une variable contenant la valeur 10 et P un pointeur qui contient l'adresse de A.



II- Les pointeurs

1- Définition

Un pointeur est une variable spéciale qui peut contenir **l'adresse** d'une autre variable.
Si un pointeur **P** contient l'adresse d'une variable A, on dit que ***'P pointe sur A'***.
Un pointeur est limité à un type de données. Il peut contenir :

- L'adresse d'une variable simple de ce type,
- L'adresse d'une composante d'un tableau de ce type.

2- Les opérateurs de base

Lors du travail avec des pointeurs, nous avons besoin :

- D'un opérateur ***'Adresse de'*** : **&** pour obtenir l'adresse d'une variable.
- D'un opérateur ***'contenu de'*** : ***** pour accéder au contenu d'une adresse.
- D'une syntaxe de déclaration pour pouvoir déclarer un pointeur.

a)-L'opérateur 'adresse de' : &

&< nom variable> : ***fournit l'adresse de la variable < nom variable>.***

b)- L'opérateur 'contenu de' : *

< nom pointeur > :** ***désigne le contenu de l'adresse référencée par le pointeur <nom pointeur> .

c)- Déclaration d'un pointeur :

nom pointeur : Pointeur sur type

Déclare un pointeur **<nom pointeur>** qui peut recevoir des adresses de variables.

Remarque

Les pointeurs et les noms de variables ont le même rôle: Ils donnent accès à un emplacement dans la mémoire interne de l'ordinateur. Il faut quand même bien faire la différence:

- Un ***pointeur*** est une variable qui peut 'pointer' sur différentes adresses.
- Le ***nom d'une variable*** reste toujours lié à la même adresse.

Exemple :

PNUM : pointeur sur entier

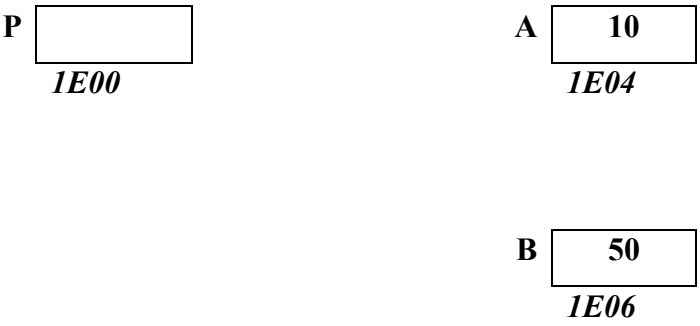
- **PNUM** est un pointeur sur **entier**,
- **PNUM** peut contenir l'adresse d'une variable du type **entier**.
- ***PNUM** est de type **int**,

Exemple1

P : pointeur sur entier

A ← 10

B ← 50

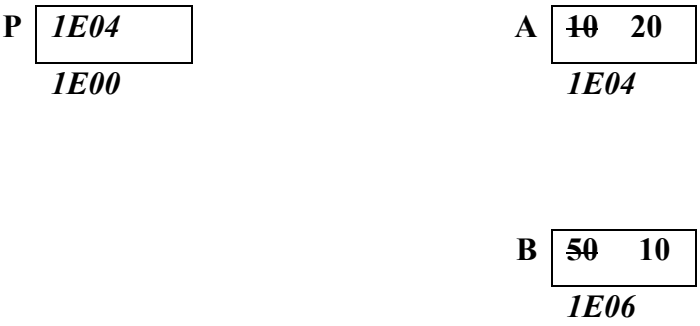


Après les instructions :

P ← &A ↔ P pointe sur A

B ← *P ↔ Le contenu de A est affecté à B

***P ← 20 ↔ Le contenu de A est mis à 20**



Exemple2

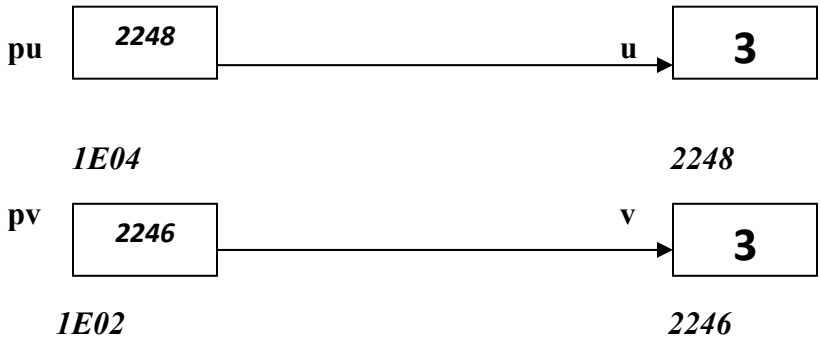
Algorithme XX

VAR

```
        u, v : entier
        pu, pv : pointeur sur entiers
DEBUT
        u ← 3
        Écrire ( "1- u = ", u , "Adresse de u = ", &u)
        pu ← &u
        Écrire ( "2-  *pu = ", *pu , "pu = ", pu)
        pv ← &v
        v ← *pu
        Écrire ( "3-  v = ", v , "Adresse de v = ", &v)
        Écrire ( "4-  *pv = ", *pv , "pv  = ", pv)
FIN
```

Résultat :

1-	u	=	3	Adresse de u	=	2248
2-	*pu	=	3	pu	=	2248
3-	v	=	3	Adresse de v	=	2246
4-	*pv	=	3	pv	=	2246



III- Paramètres d'une fonction

1- Passage des paramètres par valeur

```
Algorithme PAR_VALEUR
VAR
        X, Y : entier
```

```
Procédure PERMUTER (A, B : entier)
VAR
        AIDE : entier
DEBUT
        AIDE ← A
        A ← B
        B ← AIDE
        Écrire ("Dans PERMUTER : A=" , A , "B= "  , B)
FIN
DEBUT ( P.P )
        A ← 30
        B ← 40
        Écrire ("Avant appel de PERMUTER : X=", X, "Y = " , Y)
        PERMUTER ( X, Y )
        Écrire ("Après appel de PERMUTER : X=", X, "Y = " , Y)
FIN
```

Exécution :

Avant appel de PERMUTER : **X = 30**

Y = 40

Dans PERMUTER :

A = 40

B = 30

Après appel de PERMUTER : **X = 30**

Y = 40

X et Y restent échangés.

- Lors de l'appel, les valeurs de X et Y sont copiées dans les paramètres A et B. PERMUTER échange bien le contenu des variables locales A et B, mais les valeurs de X et Y restent les mêmes.
- Pour changer la valeur d'une variable de la fonction appelante, nous allons procéder comme suit :

- ❖ La fonction appelante doit fournir **l'adresse de la variable** et
- ❖ La fonction appelée doit déclarer **le paramètre comme pointeur**.

2- Passage des paramètres par adresse

```
Algorithme PAR_ADRESSE
VAR
        X, Y : entier
Procédure PERMUTER (A, B : pointeur sur entier)
VAR
        AIDE : entier
DEBUT
        AIDE  ← *A
        *A ← *B
        *B ← AIDE
        Écrire ("Dans PERMUTER : A=" , *A , "B= "  , *B)
FIN
DEBUT ( P.P )
        A ← 30
        B ← 40
        Écrire ("Avant appel de PERMUTER : X=", X, "Y = " , Y)
        PERMUTER (&X, &Y)
        Écrire ("Après appel de PERMUTER : X=", X, "Y = " , Y)
FIN
```

Exécution :

Avant appel de PERMUTER : **X = 30**

Y = 40

Dans PERMUTER :

A = 40

B = 30

Après appel de PERMUTER : **X = 40**

Y = 30

Lors de l'appel, les adresse de X et Y sont copiées dans les pointeurs A et B. PERMUTER échange ensuite le contenu des adresses indiquées par les pointeurs A et B.

Remarque

Par défaut lorsque l'on déclare un pointeur, on ne sait pas sur quoi il pointe. Comme toute variable, il faut l'initialiser. On peut dire qu'un pointeur ne pointe sur rien en lui affectant la valeur NULL

i : entier

p1, p2 : pointeur sur entier

p1 ← &i

p2 ← NULL

IV- Allocation dynamique de mémoire

Pour réserver l'espace mémoire pour un tableau ou une variable quelconque, on utilise souvent une opération d'allocation statique qui est effectué au début d'un programme (section variables), cette manière de réservation présente l'inconvénient que l'espace réservé doit être connu à l'avance (constant)

Exemple

1- A, B : entier

C : réel

2- Type

TAB : Tableau [1..50] : d'entier

VAR

T : TAB

A l'inverse l'allocation dynamique permet de réserver un espace mémoire de taille variable en milieu d'exécution d'un programme à ce moment, on ne peut pas utiliser les noms pour accéder à cette zone mémoire, on utilisera par suite les pointeurs.

On dispose deux primitives « ALLOUER » et « LIBERER ».

La fonction « ALLOUER » permet de réserver un ensemble de cases mémoires et envoyer l'adresse de début de cette zone.

PTR ← ALLOUER (N)

N : nombre d'octet à réserver.

Une fois on n'a plus besoin de cette zone mémoire, on peut libérer avec :

LIBERER (PTR)

Exemple

```
i , N : entier
P : pointeur sur entier
Écrire ("Donner le nombre de cases à réserver :")
Lire(N)
P ← ALLOUER (N * taille (entier))
Pour i de 1 à N
*(P + i) ← 0
```

CHAPITRE 9: Les enregistrements

I- Notion d'enregistrements

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

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

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

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

II- Déclaration des variables de type enregistrement

Syntaxe :

```
Type
    Nom_Enreg = Enregistrement
                  Champ1 : type1
                  Champ2 : type2
                  Champ3 : type3
                  .
                  .
                  Champ n : type n
    Fin enregistrement
```

Exemple :

1- Déclarer un enregistrement qui permet d'identifier une adresse

TYPE

Adresse = Enregistrement

Rue : chaîne

Ville : chaîne

cp : entier

Fin enregistrement

VAR

AD: **Adresse**

2- Déclarer un enregistrement qui permet d'identifier un étudiant

TYPE

Etudiant = Enregistrement

NI : entier

Nom : chaîne[30]

Prénom : chaîne[30]

Adresse : chaîne

D_N : chaîne

Moy : réel

Fin enregistrement

VAR

ET : **Etudiant**

Les champs peuvent des variables ordinaires, des tableaux, des pointeurs ou autres structures.

III- Manipulation des variables de type enregistrement

Les enregistrements ne peuvent pas être référencés globalement dans une instruction car ils sont composés d'éléments de type 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 :

- **ET. NI ← 1249200**

  ⇒ affecte la valeur 1249200 au champ NI de l'enregistrement ET

- **Ecrire ("Le nom de l'étudiant : ", ET. Nom)**

  ⇒ affiche le champ Nom de l'enregistrement ET

- **lire (ET. Moy)**

  ⇒ lit un réel qui sera affecté au champ Moy de l'enregistrement ET

Remarque :

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

```
VAR
        ET1 , ET2 : Etudiant
```

ET1 ← ET2 ↔

ET1. NI ← ET2. NI
ET1. Nom ← ET2. Nom
ET1. Prénom ← ET2. Prénom
ET1. Adresse ← ET2. Adresse
ET1. D_N ← ET2. D_N
ET1. Moy ← ET2. Moy

2- On peut effectuer une comparaison entre 2 enregistrements de même type comme dans l'exemple suivants :

- si (**ET1 = ET2) alors ……**
- Tantque (**ET1 ≠ ET2) Faire ……**

IV- Tableaux d'enregistrement

Exemple :

```
TYPE
        Point = Enregistrement
                Nom : Caractère
                x , y : entier
        Fin enregistrement
```

TAB : Tableau [1..50] de Point

VAR

Courbe : TAB

- La structure **point** pourrait, par exemple, servir à représenter un point d'un plan, point qui serait défini par son nom et ses coordonnées.
- La structure **courbe** représente un ensemble de 50 points.
- **Courbe[i].nom** : Représente le nom du point du rang i du tableau courbe.

  N.B : **courbe.nom[i] n'a pas de sens**
- **Courbe[i].x**

  → désigne la valeur du champ x de l'élément de rang i du tableau courbe.
- **Courbe[4]**

  → Représente la structure de type point correspondant au $4^{ème}$ élément du tableau courbe.

V- Structures comportant d'autres Structures

Exemple :

TYPE

Date = Enregistrement
jour : entier
mois : entier
année : entier
Fin enregistrement
Etudiant = Enregistrement
NI : entier
Nom : chaîne[30]
Prénom : chaîne[30]
Adresse : chaîne
D_N : Date
Moy : réel
Fin enregistrement

VAR

ET : **Etudiant**

- **ET. D_N.année**

  → Représente l'année de naissance correspondant à l'enregistrement ET.

- **ET.D_N**

  → Représente la date de naissance correspondant à la structure ET.

VI- Les pointeurs sur structures et accès aux données

Exemple :

```
TYPE
        Etudiant = Enregistrement
                    NI : entier
                    Nom : chaîne[30]
                    Prénom : chaîne[30]
                    Adresse : chaîne
                    D_N : chaîne
                    Moy : réel
        Fin enregistrement
VAR
    ET : Etudiant
    P1 : Pointeur sur Etudiant
```

- P1 est un pointeur vers un enregistrement de type **Etudiant.**
- Pour accéder à un élément de l'enregistrement en utilisant le pointeur, on utilise la notation suivante :

(*Nom_de_la_variable).membre

Où

Nom_de_la_variable → membre

P1 ← &ET

(*P1).NI ou P1 → NI

Ecrire ((*P1).Nom) ou Ecrire (P1→ Nom)

Exercice 1

- Définir le type d'un nombre complexe écrit sous la forme algébrique.
- Ecrire une fonction qui renvoie la somme de deux nombres complexes.

- ❖ Ecrire une fonction qui renvoie le produit de deux nombres complexe
- ❖ Ecrire une fonction qui renvoie le module d'un nombre complexe.
- ❖ Ecrire le programme principal.

```
Algorithme CALCUL
TYPE
            Complexe = Enregistrement
                        Reel : Réel
                        Imag : réel
            Fin enregistrement
VAR
      C1 , C2, SOM, PROD : Complexe
FONCTION SOMME ( C1, C2 : Complexe ) : Complexe
VAR
      S :  Complexe
Début
      S.Reel ← C1.Reel + C2.Reel
      S.Imag ← C1. Imag + C2. Imag
      SOMME ← S
FIN
FONCTION PRODUIT ( C1, C2 : Complexe ) : Complexe
VAR
      P :  Complexe
Début
      P.Reel ← (C1.Reel* C2.Reel) – (C1. Imag * C2. Imag)
      P.Imag ← (C1.Reel* C2.Imag) – (C1. Imag * C2. Reel)
      PRODUIT ← P
FIN
DEBUT
    Ecrire ("Partie réelle du premier nombre : " )
    Lire (C1.Reel)
    Ecrire ("Partie imaginaire du premier nombre : " )
    Lire (C1.Imag)
    Ecrire ("Partie réelle du deuxième nombre : " )
    Lire (C2.Reel)
    Ecrire ("Partie imaginaire du premier nombre : " )
    Lire (C2.Imag)
    SOM ← SOMME ( C1,C2 )
    Ecrire ("Somme = ", SOM.Reel , "+" , SOM.Imag , "i" )
    PROD ← PRODUIT ( C1,C2 )
    Ecrire ("Produit = ", PROD.Reel , "+" , PROD.Imag , "i" )
FIN
```

Exercice 2

- ❖ Définir un enregistrement qui permet d'identifier un étudiant (NI, Nom, Prénom et date de naissance)
- ❖ Ecrire une procédure SAISIE qui permet de saisir les informations de N étudiants.
- ❖ Ecrire une procédure RECHERCHE qui permet de chercher et d'afficher les information concernant un étudiant donné.
- ❖ Ecrire le programme principal.

```
TYPE
        Date = Enregistrement
                    jour : entier
                    mois : entier
                    année : entier
        Fin enregistrement
        Etudiant = Enregistrement
                    NI : entier
                    Nom : chaîne[30]
                    Prénom : chaîne[30]
                    D_N : Date
        Fin enregistrement
        TAB : Tableau [1..50] de Etudiant
VAR
    ET : TAB
    N : entier
PROCEDURE SAISIE (Var T : TAB, N : entier)
VAR
    i :  entier
Début
    Pour i de 1 à N Faire
        Ecrire ("Etudiant N° : " , i )
        Ecrire ("Donner le numéro d'inscription : " )
        Lire (T[i].NI)
        Ecrire ("Donner le nom : " )
        Lire (T[i].Nom)
        Ecrire ("Donner le Prénom : " )
        Lire (T[i].Prénom)
        Ecrire ("Donner la date de naissance : " )
        Lire (T[i].D_N.jour, T[i].D_N.mois, T[i].D_N.année)
  FinPour
Fin
PROCEDURE RECHERCHE (T : TAB, N : entier)
VAR
    i, num , rech:  entier
Début
```

```
    Ecrire ("Donner un numéro  " )    Lire (num)
    i ← 1       rech ← 0
    Tant que (i ≤ N et rech = 0) faire
                Si ( T[i].NI = num) alors
                        rech ← 1
                Sinon
                        i←i+1
                Finsi
     FinTantque
        Si (rech = 1) alors
                Ecrire("Numéro d'inscription :  ",T[i].NI )
                Ecrire ("Nom : ",T[i].Nom)
                Ecrire ("Prénom : ",T[i].Prénom)
                Ecrire ("Date de naissance :  ",T[i].D_N.jour,  "/",  T[i].D_N.mois,  "/"  ,
T[i].D_N.année))
        Sinon
                Ecrire ("Etudiant inexistant")
        Finsi
Fin
DEBUT
        Répéter
                Ecrire("Donner le nombre d'étudiant :")
                Lire(N)
        Jusqu'à (N>1 et N<=20)
        SAISIE (ET, N)
        RECHERCHE (ET, N)
FIN
```

Chapitre 10: Les listes chaînées

Introduction

Une liste chaînée est une suite ***d'un nombre variable*** d'objets de même type appelés éléments de la liste. Elle est enregistrée sur un support adressable et il existe une action simple permettant de passer d'un élément à l'élément suivant s'il existe.

La liste chaînée est une application typique de l'allocation dynamique de mémoire. C'est pourquoi il est judicieux d'avoir des connaissances sur les ***pointeurs***, les ***structures*** et ***l'allocation dynamique*** de mémoire pour aborder aux listes chaînées.

I- Les listes simplement chaînées

1- Définition

Une liste simplement chaînée est une suite d'un nombre variable d'objets de même type et chaque élément, sauf le dernier, pointe vers son successeur.

2- Description

1- chaque élément d'une liste simplement chaînée est de type composé par :

- Une partie information(s)
- Une partie pointeur

Exemple

Elément = enregistrement
num : entier
suivant : pointeur sur élément
Fin enregistrement

106	α

2- Définir un pointeur de tête qui permet d'accéder au premier élément de la liste

Tête : pointeur sur élément

3-

- Disposer des primitives d'allocation et de libération pour créer ou supprimer des éléments.

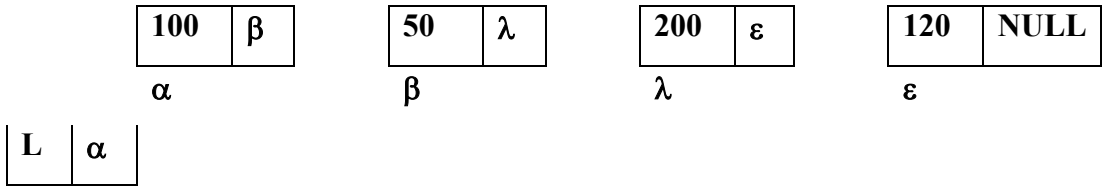
- Création : ALLOUER
- Suppression : LIBERER

❖ Rechercher dans la zone dynamique le premier emplacement capable de contenir un objet de type considéré.

❖ Réserver cet emplacement.

❖ Donner à la variable pointeur la valeur égale à l'adresse mémoire de cet emplacement.

4- De plus, il faut un moyen pour repérer le dernier élément de la liste. Cet élément n'ayant pas de successeur, sa subdivision lien devra avoir la valeur **NULL** indiquant l'adresse du pointage.



<u>Environnement type d'une LSC</u>

```
TYPE
        Elément = enregistrement
                        num : entier
                        suivant : pointeur sur élément
                  Fin enregistrement
        LISTE : pointeur sur élément
VAR
        L : LISTE
```

3- Manipulation des LSC

1- Création d'une liste vide

2- Ajout d'un élément

❖ Ajout en tête

❖ Ajout en queue

3- Affichage

4- Suppression d'un élément

❖ Suppression en tête

❖ Suppression en queue

❖ Suppression d'un élément donné.

5- Taille d'une liste

6- Tri

7- Fusion

8- Eclatement

9- ……

3-1- Création d'une liste vide

Procédure CREATION_LISTE (VAR L : LISTE)

DEBUT

L ← NULL

FIN

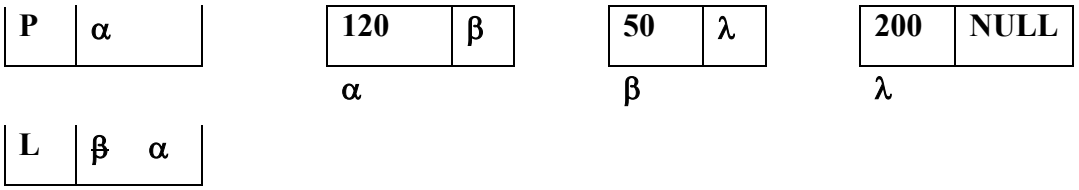
3-2- Ajout d'un élément

a) Ajout en tête

```
Fonction AJOUT_TETE (L : LISTE) : LISTE
VAR
        P : LISTE
        x: entier
DEBUT
        Ecrire ("Donner un entier :  " )
        Lire (x)
        P ←  ALLOUER (taille (Elément)) ou /* ALLOUER (P)*/
        P → num ← x
        P → suivant ← L
        L ← P
        AJOUT_TETE ← L
FIN
```

1- *Liste* vide x = 120

2- *Liste* contient déjà des éléments x = 120



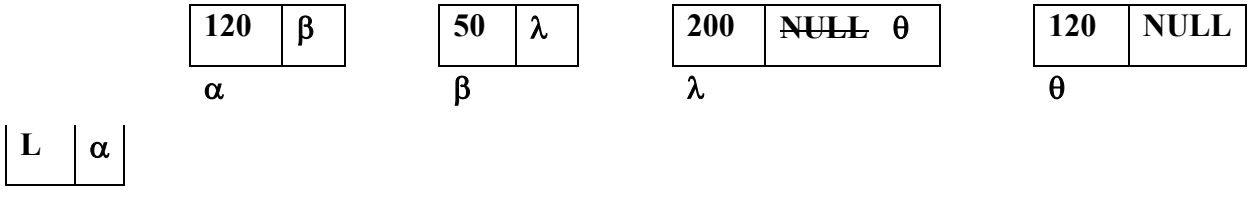
b) Ajout en queue

```
Fonction AJOUT_QUEUE (L : LISTE) : LISTE
VAR
        P, P1 : LISTE
        x : entier
DEBUT
        Ecrire ("Donner un entier :  " )
        Lire (x)
        Si (L = NULL) alors
        P ←  ALLOUER (taille (Elément))
                P → num ← x
                P → suivant ← NULL
                L ← P
        Si non
                P ← L
                Tant que ( P → suivant ≠  NULL) faire
                        P ← P → suivant
                Fin Tant que
                P1 ←  ALLOUER (taille (Elément)) ou /* ALLOUER (P1)*/
                P1 → num ← x
                P → suivant ← P1
                P1 → suivant ← NULL
        Fin Si
        AJOUT_QUEUE ← L
FIN
```

1- *Liste* vide L = NULL

x = 120

2- *Liste* contient déjà des éléments x = 120



P	α β λ
P1	θ

3-3- Affichage

Version itérative

```
Procédure AFFICHE_ITER (L : LISTE)
VAR
        P : LISTE
DEBUT
        P ← L
        Si (P = NULL) alors
                Ecrire ("Liste vide  " )
        Si non
                Tant que ( P ≠  NULL) faire
                        Ecrire (P → num)
                        P ← P → suivant
                Fin Tant que
        Fin Si
FIN
```

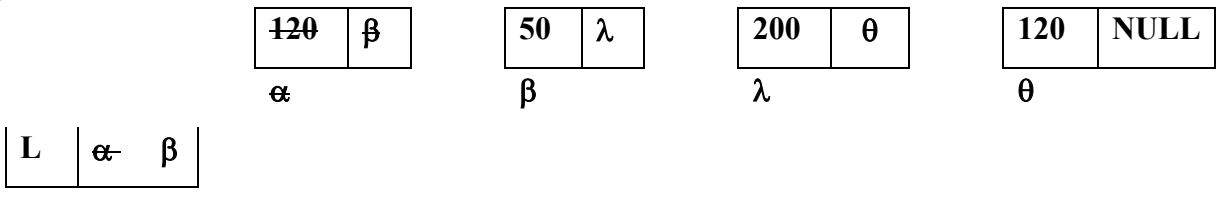
Version récursive

```
Procédure AFFICHE_REC (L : LISTE)
DEBUT
        Si (L ≠   NULL) alors
                Ecrire (P → num)
                AFFICHE_REC (L→ suivant)
        Fin Tant que
        Fin Si
FIN
```

3-4- Suppression d'un élément

a) Suppression en tête

```
Fonction SUPP_TETE (L : LISTE) : LISTE
VAR
        P, Q : LISTE
DEBUT
        P ← L
        Q ← L
        Q ← Q → suivant
        LIBERER  (P)
        SUPP_TETE ← Q
FIN
```

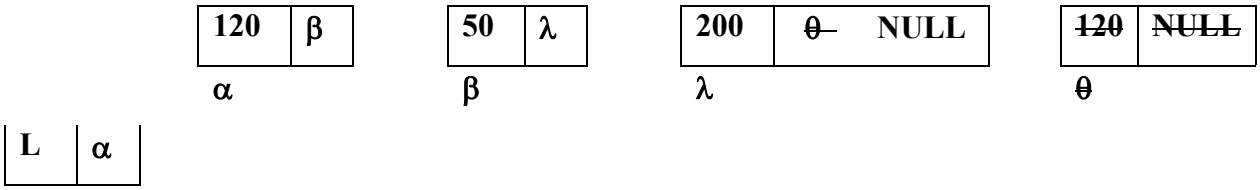


P	α
Q	~~α~~ β

b) Suppression en queue

Fonction SUPP_QUEUE (L : LISTE) : LISTE

```
VAR
        P, Q : LISTE
DEBUT
        Q ← L
        Si (Q → suivant = NULL) alors
                LIBERER  (Q)
                Q ← NULL
        Si non
                P  ← L
                Tant que ( (P → suivant) → suivant ≠  NULL) faire
                        P ← P → suivant
                Fin Tant que
                LIBERER  (P→ suivant)
                P → suivant ← NULL
        Fin Si
        SUPP_QUEUE ← Q
FIN
```



c) Suppression d'un élément donné

Fonction SUPP_ELEMENT (L : LISTE) : LISTE

```
VAR
        P, Q, R : LISTE
        x : entier
        trouve : booléen
DEBUT
        Ecrire ("Donner l'élément à supprimer :  " )
        Lire (x)
        P  ← L                  R  ← L                  trouve  ← faux
        Si (R → num = x) alors
        R ← R → suivant
        LIBERER (P)
```

```
        Si non
                Tant que ( P → suivant ≠ NULL et trouve = faux) faire
                        Si ((P→ suivant) → num = x) alors
                        trouve ← vrai
                        sinon
                        P ← P → suivant
                Fin Si
                Fin Tant que
        Si ( trouve = vrai ) alors
                Q ← P → suivant
                        P → suivant ← Q → suivant
                        LIBERER (Q)
                Si Non
                Ecrire ("élément inexistant " )
        Fin Si
        Fin Si
        SUPP_ELEMENT ← R
FIN
```

3-5- Taille d'une liste

```
Fonction TAILLE_L (L : LISTE) : entier
VAR
        T : entier
DEBUT
        Si ( L = NULL ) alors
                TAILLE_L ← 0
        Si Non
                T← 0
                Tant que ( L ≠ NULL) faire
                T ← T + 1
                L ← L → suivant
        Fin Tant Que
                TAILLE_L ← T
        Fin Si
FIN
```

3-6- Tri d'une liste

```
Fonction TRI_L (L : LISTE) : LISTE
VAR
        P, Q: LISTE
        x : entier
DEBUT
        P ← L
        Tant que ( P ≠ NULL) faire
        Q ← P → suivant
        Tant que ( Q ≠ NULL) faire
                Si (P → num > Q → num ) alors
```

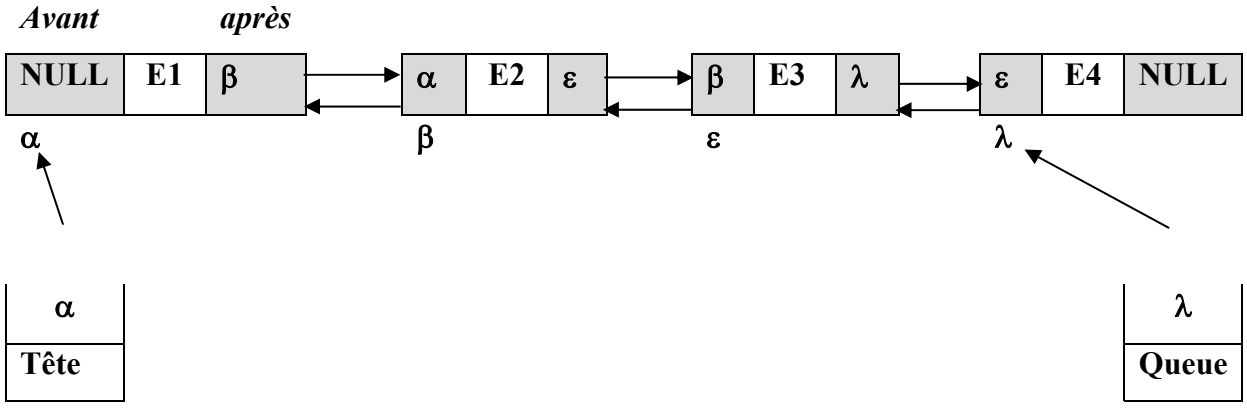
```
                    x ← P → num
                        P → num ← Q → num
                        Q→ num ← x
                    Fin Si
                    Q ← Q → suivant
                Fin Tant Que
                P ← P → suivant
        Fin Tant Que
        TRI_L ← T
FIN
```

II- Les listes doublement chaînées

Dans les listes vues précédemment le parcours ne peut se faire que dans un seul sens : de la tête vers la queue. Pour remédier à cette dissymétrie de l'opération et permettre un parcours aussi bien dans un sens que dans l'autre, on peut construire des listes doublement chaînées (ou listes bilatères)

1- Définition

Une liste bilatère comporte deux pointeurs qu'on appelle « **avant** » et « **après** ». Si un élément à un prédécesseur, il est désigné par le pointeur « avant », s'il a un successeur, il est désigné par le pointeur « après ». Cette structure permet de parcourir la liste dans les 2 sens et d'accélérer la recherche d'un élément.



- **Si Tête = Queue = NULL ⇒ *La liste est vide***
- **Si Tête = Queue ≠ NULL ⇒ *La liste à un seul élément***

2- Environnement type d'une LB

```
TYPE
        Elément = enregistrement
                        avant : pointeur sur élément
                        num : entier
                        aprés : pointeur sur élément
                Fin enregistrement
                LB = enregistrement
                tête = pointeur sur élément
                        queue = pointeur sur élément
                Fin enregistrement
        LISTE = pointeur sur élément
VAR
        L : LB
```

3- Manipulation des LB

3-1- Création d'une liste bilatère vide

```
Procédure CREATION_LB (VAR L : LISTE)
DEBUT
        L.tête          ← NULL
        L.queue         ← NULL
FIN
```

3-2- Création d'une liste bilatère

a) Ajout à gauche

```
Fonction AJOUT_G ( L : LB ; n : entier) : LB
VAR
        P, Q : LISTE
        x, i: entier
DEBUT
        /* Création du 1er élément de la liste*/
        ALLOUER (P)
        Ecrire ("Donner un entier :  " )
        Lire (P → num)
        P → avant ← L.tête
        P → après ← L.queue
        L.tête ← P      L.queue ← P
        /* Insertion du reste des éléments de la LB*/
        Pour i de 2 à n Faire
        Q ← L.tête
                ALLOUER (P)         Lire (P → num)
        P → avant ← NULL
```

```
            Q → avant ← P
            P → après ← Q
            L.tête ← P
        Fin Pour
        AJOUT_G← L
FIN
```

Autre méthode

```
Fonction AJOUT_G ( L : LB ; n : entier) : LB
VAR
        P : LISTE
        x, i: entier
DEBUT
        /* Création du 1er élément de la liste*/
        ALLOUER (P)
        Ecrire ("Donner un entier :  " )
        Lire (P → num)
        P → avant ← L.tête
        P → après ← L.queue
        L.tête ← P      L.queue ← P
        /* Insertion du reste des éléments de la LB*/
        Pour i de 2 à n Faire
                ALLOUER (P)          Lire (P → num)
        P → avant ← NULL
                (L.tête) → avant ← P
                P → après ← L.tête
                L.tête ← P
        Fin Pour
        AJOUT_G← L
FIN
```

b) Ajout à droite

```
Fonction AJOUT_D ( L : LB ; n : entier) : LB
VAR
        P : LISTE
        x, i: entier
DEBUT
        /* Création du 1er élément de la liste*/
        ALLOUER (P)
        Ecrire ("Donner un entier :  " )
        Lire (P → num)
        P → avant ← L.tête
        P → après ← L.queue
        L.tête ← P
        L.queue ← P
        /* Insertion du reste des éléments de la LB*/
        Pour i de 2 à n Faire
```

```
            ALLOUER (P)
            Lire (P → num)
        P → après ← NULL
            (L.queue) → après ← P
            P → avant ← L.queue
            L. queue ← P
        Fin Pour
        AJOUT_D← L
FIN
```

3-3- Affichage

```
Procédure AFFICHE (L : LB)
VAR
        P : LISTE
DEBUT
        Si (L.tête = NULL) alors
                Ecrire ("Liste vide  " )
        Si non
        Si (L.tête = L.queue) alors
                        Ecrire ("La liste à un seul élément : " , L.téte→ num)
        Si non
                Ecrire ("Parcours de gauche vers la droite" )
                P ← L.tête                              /* P ← L.queue : de drote à gauche*/
                        Tant que ( P ≠  NULL) faire
                                Ecrire (P → num)
                                P ← P → après                   /* P ← P → avant */

                        Fin Tant que
                Fin Si
        Fin Si
FIN
```

3-4- Suppression d'un élément

a) Suppression en tête

```
Fonction SUPP_TETE (L : LB) : LB
VAR
        P : LISTE
DEBUT
        Si ( L.tête = NULL) alors
                Ecrire ("Liste vide  " )
        Si non
        Si (L.tête = L.queue) alors
                LIBERER (L.tête)
                        L.tête     ←  NULL
                        L.queue  ←  NULL
                Si Non
```

```
                    P ← L.tête
                    P → après→ avant ← NULL
                    L.tête ← P → après
                    LIBERER  (P)
            Fin Si
    Fin Si
    SUPP_TETE ←L
FIN
```

b) Suppression en queue

```
Fonction SUPP_QUEUE (L : LB) : LB
VAR
    P : LISTE
DEBUT
    Si ( L.tête = NULL) alors
        Ecrire ("Liste vide  " )
    Si non
    Si (L.tête = L.queue) alors
        LIBERER (L.tête)
            L.tête    ←  NULL
            L.queue  ←  NULL
        Si Non
            P ← L.queue
            P → avant→ après ← NULL
            L. queue ← P → avant
            LIBERER  (P)
        Fin Si
    Fin Si
    SUPP_TETE ← L
FIN
```

c) Suppression d'un élément donné

```
Fonction SUPP_ELEMENT (L : LB) : LB
VAR
    P : LISTE
    x : entier
    trouve : booléen
DEBUT
    Ecrire ("Donner l'élément à supprimer :  " )
    Lire (x)
    P  ← L.tête
    trouve  ← faux
    Tant que ( P → après ≠  NULL et trouve = faux) faire
            Si ((P→ num) = x) alors
            trouve  ← vrai
            sinon
            P ← P → après
```

```
            Fin Si
        Fin Tant que
        Si ( trouve = vrai ) alors
        (P → avant) → après   ← P → après
            (P → après) → avant   ← P → avant
            LIBERER (P)
        Si Non
        Ecrire ("élément inexistant  " )
        Fin Si
        SUPP_ELEMENT ← LR
FIN
```

Chapitre 11 : Les Arbres

Introduction

Il est souvent nécessaire d'ordonner les éléments d'un ensemble dans le but d'améliorer la rapidité d'une recherche. Or le maintien d'un ordre entre les éléments d'un tableau ou d'une liste est relativement coûteux. Pour un tableau par exemple l'insertion d'un élément nécessite de déterminer sa place, en parcourant le tableau depuis le début (*k* comparaisons) puis de décaler les (*n*-*k*) éléments successeurs pour ménager une place. Donc une complexité en O(*n*) avec *n* le nombre d'éléments du tableau.

Les arbres de recherche sont des structures de données qui permettent de réduire la complexité en temps mais pas la complexité de la programmation.

Exemples d'organisation :

- Organisation des fichiers dans les systèmes d'exploitation
- Organisation d'un fichier sous forme de table de matière
- Organisation d'un fichier sous forme d'un questionnaire
- Arbre généalogique

L'intérêt de cette organisation est de laisser à l'utilisateur le soin de regrouper les fichiers à sa convenance tout en maintenant une structure hiérarchique.

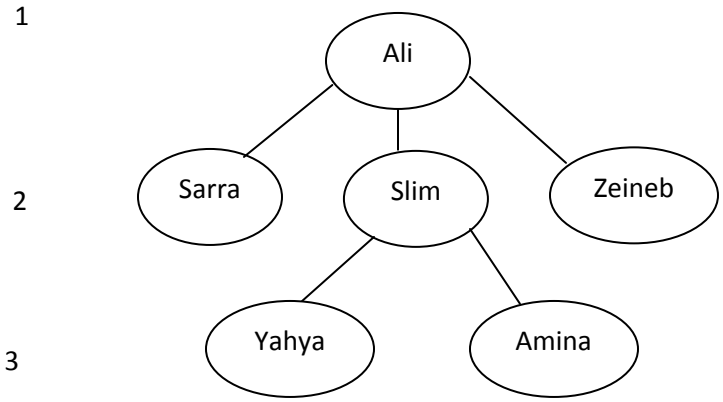
I- Les arbres généraux

1. Définitions

Arbre : un arbre est une structure de données composée d'un ensemble de nœuds. Chaque nœud contient une information et des pointeurs vers d'autres nœuds (d'autres sous-arbres).

Exemple : arbre généalogique

Niveau



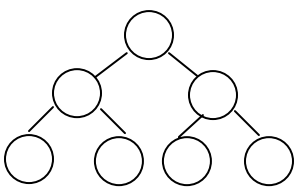
Remarques :

- La structure vide est un arbre
- Chaque nœud possède un certain nombre de ***fils***
- Chaque ***fils*** a un ***père*** unique

Définitions :

- **Racine** : Il existe un unique nœud de niveau 1 autre qui n'est pointé par aucun autre nœud : c'est la *racine* de l'arbre. Le nœud *Ali* est la racine de l'arbre généalogique de l'exemple précédant.
- **Feuilles** : Ce sont des nœuds qui ne possèdent aucun fils (nœuds sans sous arbres).
- **Niveau** : Le niveau de la racine est égal à 1. Le niveau d'un nœud autre que la racine est égal au niveau de son père + 1.
- **Taille** : On appelle taille d'un arbre le nombre total de nœuds de cet arbre. La taille de l'arbre généalogique de l'exemple est égale à 6.
- **Hauteur /Profondeur d'un arbre :** C'est le niveau maximum atteint par la branche la plus longue ; sur l'exemple précédant, la hauteur est égale à 3.
  La hauteur de l'arbre vide est égale à 0.

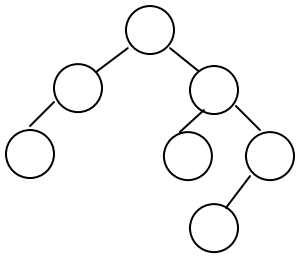
- **Arbre binaire** : C'est un arbre tel que chaque nœud a au plus 2 fils : un ***fils droit*** et un ***fils gauche***.
- **Degré d'un nœud** : On appelle degré d'un nœud le nombre de fils de ce nœud. Le degré d'un nœud d'un arbre binaire est égal à 0,1 ou 2.
- **Degré d'un arbre :** C'est le degré maximal atteint par un nœud. Le degré d'un arbre binaire est égal à 2. Si le degré d'un arbre est égal à **N**, l'arbre est dit ***N-aire***.
- **Arbre binaire complet :** C'est un arbre binaire de taille égale à 2^h -1, tel que *h* est la hauteur de l'arbre. Voici un exemple :



h = 3

Taille = 2^3-1= 7 noeuds

- **Arbre binaire équilibré** : Un arbre binaire est équilibré si pour chaque nœud, les hauteurs des sous arbres gauche et droit diffèrent d'au plus de 1. L'arbre ci-dessous est équilibré.



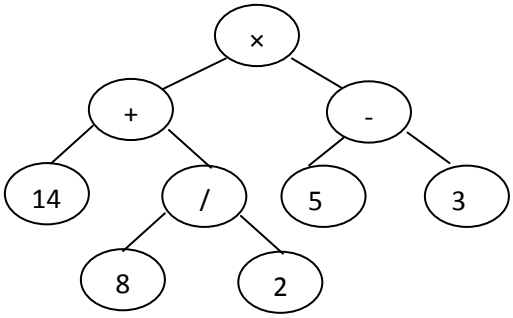
2. Exemples d'application

-Expression arithmétique

On peut utiliser un arbre pour représenter une expression arithmétique ayant des opérateurs binaires :

$(14 + (8/2)) \times (5 - 3)$

L'arbre binaire correspondant est :



Arbre généalogique
Structure d'un chapitre (sections, paragraphes)
Tous les problèmes de classification

II- Les arbres binaires

Un arbre binaire est un arbre tel que chaque nœud a au plus 2 fils: un *fils droit* et un *fils gauche*.

Il s'agit de mémoriser les arbres et leurs relations de dépendances père fils. La mémorisation se fait par **allocation dynamique**.

1. ***Déclaration***

```
Nœud = enregistrement
        Op : caractère
        gauche : pointeur sur noeud
        droite : pointeur sur noeud
       Fin enregistrement
Arbre = pointeur sur nœud
```

2. ***Fonction de création d'un nœud***

```
fonction CreerNoeud(operateur : caractère, SAG: arbre, SAD: arbre) : arbre
var
 nouv : arbre;
debut
nouv ← Allouer(taille(nœud))
nouv→gauche ← SAG
nouv→droite ← SAD
CreerNoeud ← nouv
Fin
```

3. ***Fonction de création d'une feuille***

```
Fonction CreerFeuille(operand: caractère) : arbre
Debut
        CreerFeuille ← Créer(operande, NULL, NULL)
fin
```

- **Exemple : Création de l'arbre binaire représentant l'expression arithmique**

```
Fonction CreerArbre():arbre
Debut
   creerArbre ← CreerNoeud("×", CreerNoeud("+", CreerFeuille(14), CreerNoeud("/",
CreerFeuille(8),   CreerFeuille(2))), CreeNoeud("-",
CreerFeuille(5),CreerFeuille(3))) ;
 fin
```

4. Parcours d'un arbre binaire

Un algorithme de parcours d'un arbre est un procédé permettant d'accéder à chaque nœud de l'arbre. Un certain traitement est effectué pour chaque nœud (test, écriture , comptage, etc.), mais le parcours est indépendant de cette action. On distingue deux catégories de parcours :

- **En profondeur :** On explore branche par branche
- **En Largeur :** On explore niveau par niveau

4.1 Les différentes méthodes de parcours en profondeur d'un arbre

Dans un parcours en profondeur, le principe est le suivant : on explore une branche le plus profondément possible, ensuite on revient en arrière pour essayer un autre chemin.
Il y'a six types de parcours possibles. Soit les notations suivantes :

P : Père
SAG : Sous arbre gauche
SAD : Sous arbre droit

Nous ne considérons dans ce cours que le parcours droite-gauche. Les autres parcours s'en déduisent facilement par symétrie.

	Gauche-Droite	**Droite-Gauche**
Préfixé	P.SAG.SAD	P.SAD.SAG
Infixé	SAG.P.SAD	SAD.P.SAG
Postfixé	SAG.SAD.P	SAD.SAG.P

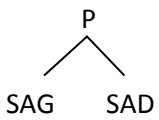


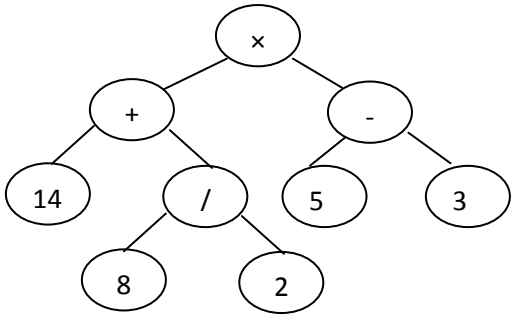
Fig 1. Les six types de parcours d'un arbre binaire

- **Le Parcours Préfixé**

  La racine est traitée en premier

  1 Traiter la racine
  2 Parcours préfixé du SAG
  3 Parcours préfixé du SAD

  - **Exemple : expression arithmétique**



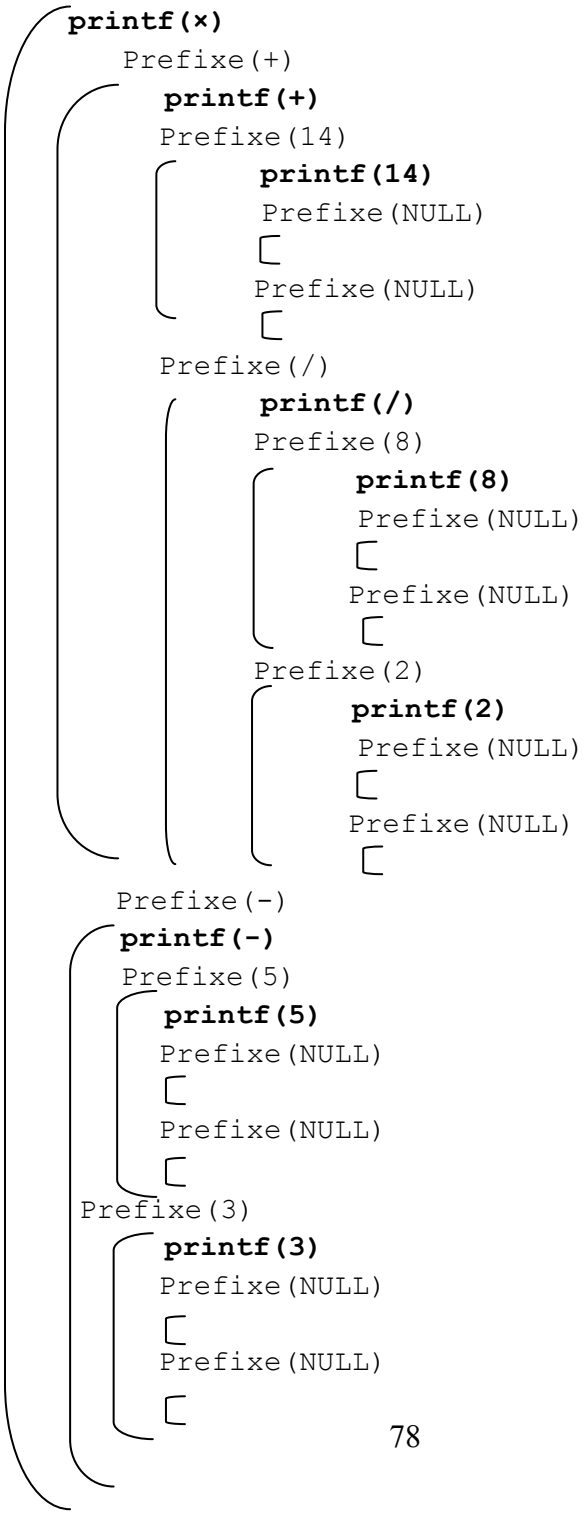
L'affichage correspondant à un parcours préfixé :

×, +, 14, /, 8, 2, -, 5, 3 → L'opérateur ici est traité avant les opérandes.

```
Procédure Prefixe(racine : arbre)
debut
    si( racine # NULL) alors
```

Ecrire(racine→op)
Prefixe(racine→gauche)
Prefixe(racine→droite)
finsi
fin

- **Exécution**



- **Le Parcours Infixé**

  La racine est traitée entre les deux appels récursifs

  1 Parcours Infixé du SAG
  2 Traiter la racine
  3 Parcours Infixé du SAD

  - **Exemple : expression arithmétique**

  L'affichage correspondant à un parcours infixé de l'arbre donné précédemment:

  14, +, 8, /, 2, ×, 5, -, 3

  - **Algorithme**

```
Procédure Infixe(racine : arbre)
debut
    si( racine # NULL) alors
    Infixe(racine→gauche)
     Ecrire(racine→op)
    Infixe(racine→droite)
    finsi
fin
```

- **Le Parcours Postfixé**

  La racine est traitée après les deux appels récursifs

  1 Parcours Infixé du SAG
  2 Parcours Infixé du SAD
  3 Traiter la racine

  - **Exemple : expression arithmétique**

  L'affichage correspondant à un parcours postfixé de l'arbre donné précédemment:

  14, 8, 2, /, +, 5, 3, -, ×

- **Algorithme**

```
Procédure Postfixe(racine : arbre)
debut
     si( racine # NULL) alors
     Postfixe(racine→gauche)
      Postfixe(racine→droite)
      Ecrire(racine→op)
     finsi
fin
```

4.2 Le parcours en largeur d'un arbre

Cette méthode consiste à visiter l'arbre étage par étage. Pour notre exemple d'expression arithmétique le parcours donne l'affichage suivant :

×, +, -, 14, /, 5, 3, 8, 2,

Ce parcours nécessite l'utilisation d'une **file d'attente** qui contient initialement la racine. A chaque étape on extrait l'élément en tête de file et on le remplace par ses successeurs droit et gauche en queue de la file. On effectue ce traitement jusqu'à ce que la file soit vide.

Remarque : Par nature même de la structure arborescente, on utilise plutôt le parcours en profondeur.
Dès fois, le type de résultat nécessite un parcours en largeur, par exemple pour dessiner un arbre.
La file d'attente utilisée est une liste chaînée (linéaire) dont les éléments sont des pointeurs sur un nœud de l'arbre.

```
Type
     Cellule = enregistrement
     {
     Elt : arbre
     Suiv : pointeur sur cellule;
     }
     File : pointeur sur cellule
```

Le parcours en largeur s'écrit alors :

```
     Procédure EnLargeur(racine : arbre, F : file)
     Var
     tmp : arbre
     Debut
     si(racine # NULL)
          Enfiler(F,racine)
     Tant que (FileVide(F))= faux) faire
```

```
        Defiler(F, temp) ;
        Ecrire(tmp→op)
        si(tmp→gauche #NULL)
                Enfiler(F,tmp→gauche)
        Finsi
        si(tmp→droite #NULL) ;
                Enfiler(F,tmp→droite)
        Finsi
        Fin Tantque
Fin
```

5. Primitives d'un arbre binaire

5.1 Test d'un arbre vide

La fonction **ArbreVide** retourne 1 si l'arbre est vide (de taille =0), 0 sinon

```
Fonction ArbreVide(racine : arbre) : entier
{
ArbreVide ← racine==NULL ;
}
```

5.2 Calcul de la taille d'un arbre

La fonction **TailleArbre** parcourt récursivement l'arbre et retourne sa taille ; c'est-à-dire le nombre de ses nœuds.

```
 Fonction TailleArbre(racine : arbre) : entier
Debut
Si ArbreVide(racine) = 1 alors
        TailleArbre ← 0 ;
sinon
        TailleArbre ← 1 + TailleArbre(racine→droite)
             + TailleArbre(racine→gauche)
Finsi
Fin
```

5.3 Recherche d'un nœud de valeur donnée

La fonction **Recherche** recherche récursivement un nœud de valeur donnée en commençant par la racine. La fonction retourne un pointeur sur le nœud si la valeur recherchée existe, sinon elle retourne NULL.

```
Fonction Recherche(racine : arbre, val : caractère) : arbre
Var
ptr_trouv : arbre
Debut
si (racine=NULL)
        ptr_trouv←NULL
sinon
        si (racine→op=val)
                ptr_trouv ← racine
        sinon

                ptr_trouv←Rechercher(racine→gauche,val) ;
                si(Non(ptr_trouv)) alors
                 ptr_trouv←Rechercher(racine→droite,val) ;
          finsi
finsi
                Recherche ←ptr_trouve
Fin
```