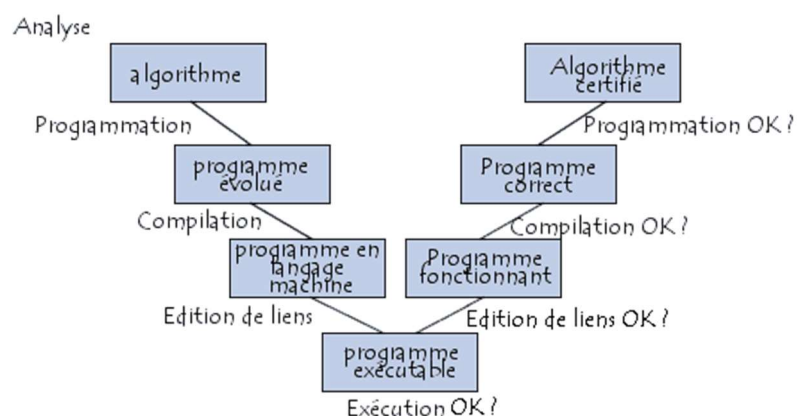


# DWWM



## Algorithmique

Formation

### Développeur Web et Web Mobile

Module : 03

### Développer la partie front-end d'une application web

Séquence : 03

### Développer une interface utilisateur web dynamique

Séance : 01

### Ecrire un algorithmique

Libellé réduit :	Algo
Type de document :	Ressource
Auteur :	DB
Version :	2
Date de mise à jour :	22/07/2022



## Sommaire

<b>1</b>	<b>définition</b>	<b>1</b>
1.1	Généralités	1
1.2	Les notions liées aux algorithmes	1
<b>2</b>	<b>Le Pseudo-Langage</b>	<b>2</b>
2.1	Introduction	2
2.2	Les commentaires	2
2.3	Les variables	2
2.3.1	Définition	2
2.3.2	La déclaration	3
2.3.3	L'affectation	3
2.4	Les instructions de saisie et d'affichage	3
2.4.1	La saisie	3
2.4.2	l'affichage	4
2.5	Expression	4
2.6	Les constantes	4
2.7	La séquence	5
2.8	La sélection	5
2.8.1	La construction SI-ALORS-SINON (dite alternative)	5
2.8.2	La construction SELON (choix multiple)	7
2.9	Itération	8
2.9.1	La construction TANTQUE-FAIRE	8
2.9.2	La construction REPETER-TANTQUE	9
2.9.3	La construction POUR_FAIRE	9
2.10	Exemples	10
2.10.1	Exemple 1 : commenté	10
2.10.2	Exemple 2 : à réaliser selon l'énoncé fourni	11
2.11	Référence à d'autres algorithmes	11
<b>3</b>	<b>Autres représentations</b>	<b>12</b>
3.1	l'organigramme	12
3.1.1	Symboles principaux	12
3.2	Représentation des constructions de base	14
3.2.1	Sélection	14
3.2.2	Itération	14
3.3	Les graphes de nassi-schneidermann	14
3.3.1	Sélection	14
3.3.2	Itération	14
3.4	Les arbres algorithmiques	15
3.4.1	Séquence	15
3.4.2	Sélection	15
3.4.3	Itération	15
<b>4</b>	<b>La modularité</b>	<b>16</b>
4.1	Concepts généraux	16
4.2	Le passage des arguments	20
4.2.1	Le passage par valeur	20

afpa®	Auteur	Nom région	Formation	Date Mise à jour	
	DB	GRN 164	DWWM	22/07/2022	310-Cours_Algorithmie.docx

4.2.2	Le passage par adresse	20
4.2.3	Le passage par référence	21
<b>4.3</b>	<b>La visibilité d'une variable</b>	<b>21</b>
<b>4.4</b>	<b>Passage par argument ou variables globales ???</b>	<b>22</b>
<b>4.5</b>	<b>Récursivité des modules</b>	<b>23</b>
<b>4.6</b>	<b>Avantages de la modularité</b>	<b>23</b>
<b>5</b>	<b>Les structures de données</b>	<b>25</b>
<b>5.1</b>	<b>Introduction</b>	<b>25</b>
<b>5.2</b>	<b>Les types de base</b>	<b>25</b>
5.2.1	Les données scalaires	25
<b>5.3</b>	<b>Les tableaux</b>	<b>26</b>
5.3.1	Les agrégats, enregistrements ou structures	27
<b>5.4</b>	<b>Les structures dynamiques</b>	<b>27</b>
5.4.1	Les piles	27
5.4.2	Les files	28
5.4.3	Les structures chaînées	29
<b>5.5</b>	<b>CONCLUSION :</b>	<b>32</b>
<b>6</b>	<b>Les fichiers</b>	<b>33</b>
<b>6.1</b>	<b>Généralités</b>	<b>33</b>
6.1.1	Déclaration	34
6.1.2	Ouverture	34
6.1.3	Fermeture	35
<b>6.2</b>	<b>Les fichiers séquentiels</b>	<b>35</b>
6.2.1	En lecture	35
6.2.2	En écriture ou en ajout	35
<b>6.3</b>	<b>Les fichiers à accès direct ou relatif</b>	<b>36</b>
6.3.1	En lecture et écriture	36
6.3.2	En déplacement	36
<b>6.4</b>	<b>Les fichiers séquentiels indexés</b>	<b>36</b>
6.4.1	Déplacement	37
6.4.2	Lecture et Ecriture	37
6.4.3	suppression	37

# 1 DEFINITION

Un **algorithme** décrit une succession d'opérations qui, si elles sont correctement exécutées par l'ordinateur, produiront le résultat déterminé ou le **processus désiré**.

L'algorithme est aussi appelé parfois le **procédé de traitements**.

Il faut noter qu'un algorithme **n'est pas une solution unique** : pour réaliser une même activité, **plusieurs algorithmes sont possibles**.

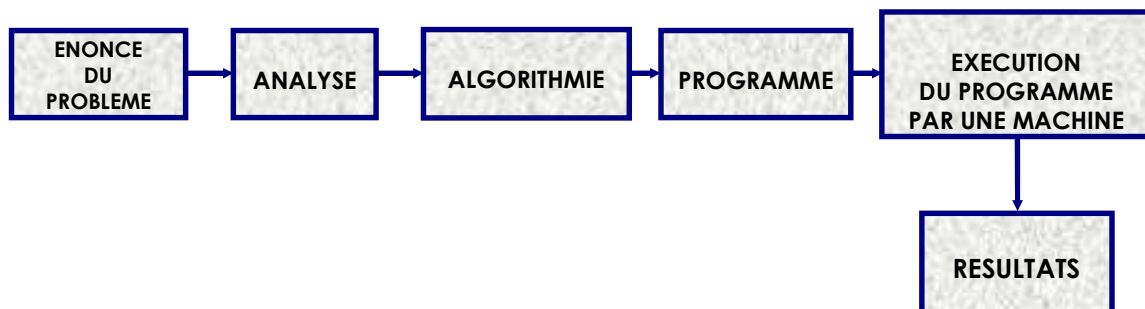
## 1.1 GENERALITES

La notion d'algorithme **n'est pas spécifique à l'informatique**. Certains algorithmes décrivent toutes sortes de tâches quotidiennes. Ainsi les différents exemples analysés dans un chapitre suivant (appel téléphonique, gestion d'un carnet d'adresses, préparation d'une tasse de café) sont des algorithmes.

En règle générale, l'entité qui exécute une tâche est appelée **processeur**. Un processeur peut être une personne, un ordinateur ou tout autre dispositif électronique ou mécanique.

Un processeur obéit aux actions (les exécute) que l'algorithme lui décrit. L'exécution d'un algorithme implique **l'exécution de chacune de ses étapes constitutives**.

Le schéma ci-dessous permet de résumer la situation de l'algorithmie dans le processus de la programmation d'une tâche quelconque.



## 1.2 LES NOTIONS LIEES AUX ALGORITHMES

Différentes **notions** sont étroitement liées aux algorithmes. On trouve entre autres :

- La **puissance** : c'est la capacité d'un algorithme à traiter des problèmes plus ou moins **complexes**.
- La **dimension** : c'est la taille prise par un algorithme (théoriquement incompressible) pour **exécuter le processus**.
- La **précision** : c'est la **fidélité** et l'**exactitude** offertes par l'algorithme dans sa réalisation pour **atteindre les objectifs prévus**.
- La **rapidité** : c'est la capacité d'un algorithme à fournir un résultat dans un minimum de temps.
- **L'identité** : c'est le **rôle intrinsèque** rempli par l'algorithme (composant indépendant).

## 2 LE PSEUDO-LANGAGE

### 2.1 INTRODUCTION

Le **pseudo-langage** ou **pseudo-code** remplit une fonction importante dans la conception de programme.

Il permet de **présenter l'étude d'un module de programme** avec des **instructions évoluées** utilisant des phrases en **langage quasi-naturel** en évitant ainsi l'énumération de détails.

Le **pseudo-langage** fournit une **base structurée** à l'écriture de programme puisque l'adhésion à des constructions formelles renforce la structure.

Il est une **passerelle** entre **les données de l'analyse** et **les détails du codage**.

Il offre une possibilité supplémentaire de **documentation de l'application développée**.

Un **algorithme** développé en **pseudo-code** est un ensemble d'instructions qui doit être exécuté pour remplir une activité. Il est exprimé en un **langage quasi naturel** et **non dans un langage de programmation**.

### 2.2 LES COMMENTAIRES

Il est possible de **placer des lignes de commentaires dans le pseudo-code**.

Suivant les ouvrages, on trouve principalement 3 types de commentaires :

- ceux débutant par # et se finissant par #
- ceux débutant par { et se finissant par }
- ceux dérivant des langages informatiques /\* ... \*/ pour un bloc de commentaire ou // pour une ligne ou fin de ligne

### 2.3 LES VARIABLES

Un algorithme est en général prévu pour être codé puis exécuté par une machine informatique. Pour stocker des valeurs ou des résultats de calcul, on utilise des **variables** et on fait des opérations algébriques, relationnelles, ... pour les **manipuler**.


#### 2.3.1 Définition

Une **variable informatique** (différent d'une variable mathématique) est un objet qui peut prendre une valeur. Cette valeur peut évoluer au cours de l'exécution de l'algorithme. Une variable porte un nom ou **identificateur**. Le **type** de la variable détermine l'ensemble des valeurs que peut prendre cette variable.

L'identificateur d'une variable doit être auto-informant. L'utilisation de soulignés permet de composer des identificateurs formés de plusieurs mots.

*Exemples :*      *nbre\_articles, delta, result\_hors\_taxe*

Une autre écriture de plus en plus utilisée est la notation en « CamelCase » qui consiste à mettre en majuscule la première lettre de chaque mot (sans compter le 1<sup>er</sup> mot pour les variables).

	Auteur	Nom région	Formation	Date Mise à jour	Page 2 / 41
	DB	GRN 164	DWWM	04/10/2022	310-Cours_Algorithmie.docx

Exemples : *nbreArticles, delta, resultHorsTaxe*

Les variables sont assimilables à des boîtes aux lettres de programmes : elles possèdent un nom, contiennent une valeur, et représentent une adresse.

Les variables sont typées en fonction du genre de contenu qu'elles représentent : entier, réel, caractère, chaîne (de caractères), booléen, monétaire...

### 2.3.2 La déclaration

La déclaration des variables se fait avant le traitement.

La syntaxe est la suivante :

**variables** <liste d'identificateurs> : **type**

Exemples : **variables**      val, unNombre : **entier**  
                                  nom, prenom : **chaîne**

Exemple de PSEUDO-CODE :

*# CALCUL DU MONTANT TTC A PARTIR DU MONTANT HT #*  
*variables montantHT, montantTTC, tva : réel*

*lire(montantHT)*  
*tva ← 18.6*

*montantTTC ← montantHT + (montantHT x tva)/100*

La valeur d'une variable évolue en cours de traitement.

### 2.3.3 L'affectation

L'instruction «  $\text{prixTTC} \leftarrow \text{quantite} * (\text{prixHT} + (\text{prixArticleHT} * \text{tva}) / 100)$  » modifie la valeur de la variable prixTTC.


Cette modification de la valeur s'appelle l'**affectation**. L'ancienne valeur prise par prixTTC est **écrasée** par cette affectation. Plusieurs symboles sont utilisables pour exprimer une affectation. On trouve :

$\text{PI} \leftarrow 3.14$   
 $\text{PI} := 3.14$   
 $\text{PI} = 3.14$

Remarque : l'**incrément** d'une variable s'écrit :  $X \leftarrow X + 1$  (on ajoute 1 à la valeur de X et le résultat est réaffecté à X).

## 2.4 LES INSTRUCTIONS DE SAISIE ET D'AFFICHAGE

### 2.4.1 La saisie

	Auteur	Nom région	Formation	Date Mise à jour	Page 3 / 41
	DB	GRN 164	DWWM	04/10/2022	310-Cours_Algorithmie.docx

Il est intéressant de demander à l'utilisateur de rentrer des valeurs au clavier qui seront placées dans des variables.

Les mots clés utilisés sont « saisir, entrer, lire ».

Ils seront utilisés sous forme de « fonction » (utilisation des parenthèses).

Exemple :

```
saisir(rayon)
entrer(nom, prenom)
```

## 2.4.2 l'affichage

Un algorithme étant fait pour exécuter un traitement, il est donc nécessaire d'afficher les résultats.

Le mot utilisé est « afficher, imprimer, écrire »

Exemple :

```
afficher("la circonférence est :", circonff)
afficher("la somme des nombre est : ", nbr1 + nbr2)
```

## 2.5 EXPRESSION

Une expression en **pseudo-code** (ainsi que dans tout autre langage) est une suite de mots (mot-clé, opérateurs, variables...) qui possède une signification dans le langage. Cette signification est alors **évaluée** par le processeur pour en fournir un résultat.

```
Exemple :      delta ← b2 - 4ac
                si ( delta >= 0 )
                  alors
                    .....
                sinon
                    ....
                finSi
```

**RAPPEL IMPORTANT** : Le pseudo-langage doit pouvoir être lu **SANS** connaissance particulière.

On doit donc **VEILLER** à la **CLARTE** des instructions.

On peut utiliser dans les expressions des opérateurs arithmétiques, relationnels...

## 2.6 LES CONSTANTES

Pour des raisons de clarté et de compréhension de l'algorithme, il est possible de déclarer des constantes.

Ces constantes permettent de réserver de l'espace mémoire pour stocker des données dont la valeur est fixée pour tout l'algorithme.


La syntaxe est la suivante :

**constantes** <identificateur> ← <valeur>

Exemples :

```
constantes MAX ← 100
           DOUBLEMAX ← MAX × 2
```

L'identificateur sera écrit en majuscule. Séparer les différents mots constituant l'identificateur par un Under score « \_ ».

	Auteur	Nom région	Formation	Date Mise à jour	Page 4 / 41
	DB	GRN 164	DWWM	04/10/2022	310-Cours_Algorithmie.docx



## 2.7 LA SEQUENCE

C'est une **série d'actions** ou **d'instructions** devant être réalisée dans **un ordre linéaire** :

EXEMPLE : *Prendre l'automobile*  
*Aller au supermarché*  
*Choisir les articles*  
*Acheter les articles sélectionnés*  
*Ranger les achats dans l'automobile*  
*Revenir à la maison*  
*Décharger les articles*

Elles sont exécutées **séquentiellement** du haut vers le bas.

Une séquence s'écrit parfois sur une seule ligne, un séparateur tel qu'un point-virgule délimite les instructions :

EXEMPLE : *lire( x ); Lire(y ), z ← x \* y;*  
*afficher( " Le résultat est :", z )*

Pour identifier un bloc de groupe ou bloc d'instructions remplissant une fonction bien définie, on peut encadrer cette séquence par les mots **DEBUT** et **FIN**.

EXEMPLE : *début*  
*X2 ← carré( X )*  
*Y2 ← carré( Y )*  
*résultat ← racine\_carrée( X2 + Y2 )*  
*afficher( " L'hypoténuse est :", résultat )*  
*fin*

## 2.8 LA SELECTION

**Deux types de construction** permettent de **choisir** (ou **sélectionner**) un groupe d'actions au lieu d'un autre en fonction de conditions précises.

### 2.8.1 La construction SI-ALORS-SINON (dite **alternative**)

La condition testée est une **condition BINAIRE**.

EXEMPLE : *si ( X >= 0 ) alors*  
*afficher( " X est positif ou nul " )*  
*sinon*  
*afficher( " X est négative " )*  
*finSi*

Les instructions derrière le mot-clé « **ALORS** » sont appelées la clause « **VRAI** ».  
 " « **SINON** » "

- Si la réponse à la condition est **vraie** (ou oui), l'action ou le bloc d'actions placé à la suite du mot-clé **ALORS** est réalisé (on parle alors de la clause **ALORS**).

- Si la réponse est **fausse** (ou non) alors c'est l'action ou le bloc d'actions placé à la suite du **SINON** qui est réalisé (clause **SINON**).

La question ou la condition est de nature binaire puisqu'elle n'admet que deux solutions :

**FORMAT GENERAL :**

```

si ( < condition binaire > ) alors
    < séquence vraie >
sinon
    < séquence fausse >
finsi

```

Les constructions suivantes sont possibles :

```

si ( c'est le week-end ) alors
    début
        # Séquence #
        je tonds la pelouse
        je lave la voiture
        je lis le journal
    fin
sinon
    je vais travailler
finSi

```

Remarque : les mots début et fin peuvent être omis

**EXEMPLE :** « Chercher le plus grand de 3 nombres avec la contrainte suivante : Le processeur n'est capable de comparer que deux nombres à la fois »

```

Lire( X, Y, Z )
si ( X > Y ) alors
    si ( X > Z ) alors
        afficher( " X est le plus grand des 3" )
    sinon
        afficher( " Z est le plus grand des 3" )
    finSi
sinon
    si ( Y > Z ) alors
        afficher( " Y est le plus grand des 3" )
    sinon
        afficher( " Z est le plus grand des 3" )
    finSi
finsi

```

**L'indentation** et l'écriture des instructions sur des lignes séparées sont des règles visant à améliorer la **lisibilité** et à déterminer l'étendue (scope) des **blocs** d'instructions.

**NB :** Le terme **SINON** est optionnel :

```

je me prépare à sortir
si ( dehors il fait froid ) alors
    je mets un pull
finSi
je sors

```

La condition **binaire** peut être **complexe**. La condition globale est alors la résultante de plusieurs conditions simples connectées entre elles par des opérateurs logiques **booléens**.

*EXEMPLE :      si ( c'est le week-end ) et ( il fait beau ) alors  
   je vais à la campagne  
                 finSi*

La construction SI-ALORS-SINON est en fait une instruction particulière. Il est alors possible de réaliser des constructions imbriquées (ou composées) :

*si ( ce\_jour est dimanche ) alors  
    début  
        si ( c'est le matin ) alors  
            je prends le petit déjeuner  
        finSi  
    je m'accorde des moments de détente  
fin  
finSi*

### 2.8.2 La construction SELON (choix multiple)

Soit le pseudo-code :

*si ( ce\_jour est lundi ) alors  
    faire le programme  
sinon  
    si ( ce\_jour est mardi ) alors  
        faire le travail du mardi  
    sinon  
        si ( ce\_jour est mercredi ) alors  
            conduire les enfants à la garderie  
        sinon  
            .....  
            .....  
    finSi  
finSi  
finSi*

L'imbrication des alternatives **SI-ALORS-SINON-FINSI** dont les conditions s'appliquent à **différentes valeurs concrètes** d'une **même entité** (ou même variable) peut être réalisée par la construction **SELON** (choix multiple) :

*selon ( ce\_jour )  
    Lundi       :     Editer le programme  
    Mardi       :     Compiler le programme  
    Mercredi   :     Mettre au point le programme  
    Jeudi       :     .....  
    .....       :     .....  
finSelon*

Le sélecteur, ici « ce\_jour », permet la sélection d'une action parmi toutes les actions décrites. Par exemple, si ce\_jour est Mercredi, l'action située après le terme « Mercredi : » est réalisée, puis l'exécution passe aux actions situées après le mot-clé FINCAS. Toutes les possibilités du sélecteur ne sont pas nécessairement énumérées.

Elles peuvent aussi être **regroupées** et **associées à une seule action**; ce regroupement est exclusif.

**EXEMPLE :**     *selon ( ce\_jour )*  
                     *Lundi, Mardi :       je fais mon travail de début de semaine*  
                     *Mercredi, Jeudi :   je fais mon travail de milieu de semaine*  
                     *Vendredi, Samedi : je fais mon travail de fin de semaine*  
                     *finSelon*

Dans les situations où toutes les possibilités du sélecteur ne sont pas décrites, une option **AUTRES** peut être utilisée :

*selon ( ce\_jour )*  
                     *Samedi:       Tondre la pelouse*  
                     *Dimanche : Se relaxer*  
                     *Autres :     Se rendre au travail*  
                     *finSelon*

La forme générale pour le choix multiple est :

*selon ( sélecteur )*  
                     *cas #1 :     instructions pour #1*  
                     *cas #2 :     instructions pour #2*  
                     *cas #3 :     instructions pour #3*  
                     *cas #4 :     instructions pour #4*  
                     *autres :    instructions pour autres*  
                     *finSelon*

## 2.9 ITERATION

Elle fournit une structure permettant de **répéter** un certain nombre de fois des actions.

Pour répéter la réalisation d'actions un nombre donné de fois, **plusieurs constructions sont possibles.**

### 2.9.1 La construction TANTQUE-FAIRE

Le format général est :     *tantque ( condition ) faire*  
   *Séquence*  
   *finTantque*

Exemple 1 :     *Lire la température*  
                     *tantque ( la température >= 50° C ) faire*  
                                     *début*  
                                     *Maintenir l'air conditionné*  
                                     *Lire la température*  
                                     *fin*  
                     *finTantque*

Remarque : là aussi, les mots début et fin peuvent être omis

Avec cette construction, la condition est d'abord évaluée. Ensuite, dans le cas d'une réponse **vraie**, l'action suivant le mot-clé **FAIRE** est exécutée.

Les points suivants à considérer pour cette construction sont :

- le corps de la boucle peut contenir plusieurs instructions.
- la condition doit être binaire et peut être complexe (usage d'opérateurs booléens).
- la condition est évaluée en tout premier -----> il est possible que le corps de boucle ne soit jamais exécuté.
- la création d'une boucle infinie est possible : (la condition est toujours vraie)

Exemple 2 :     *# Lecture de 10 caractères #*  
                   *I ← 0*  
                   ***tantque ( I < 10 ) faire***  
                           *Lecture du caractère saisi au clavier*  
                   ***finTantque***

## 2.9.2 La construction REPETER-TANTQUE

Exemple 1 :     ***lire( temperature )***  
                   ***répéter***  
                           ***début***  
                                   *Maintenir l'air conditionné*  
                                   ***lire( temperature )***  
                           ***fin***  
                   ***tanque ( température > 50°C )***

Même remarque : les mots début et fin peuvent être omis

Exemple 2 :     *I ← 0*  
                   ***répéter***  
                           ***début***  
                                   *Lecture du caractère saisi au clavier*  
                                   *I ← I + 1     # Incrémentation*  
                           ***fin***  
                   ***tantque ( I < 10 )***

Les points suivants à considérer pour cette construction sont :

- Le corps de la boucle peut contenir **plusieurs instructions**.
- La condition doit être **binaire** et peut être **complexe** (usage d'opérateurs booléens).
- La condition est évaluée **en sortie de boucle** ---> **la boucle est exécutée au moins une fois**.
- La **création d'une boucle infinie est possible** : (la condition est toujours vraie).

## 2.9.3 La construction POUR\_FAIRE

Cette boucle permet automatiquement l'initialisation, le test et l'incrément de l'index (ici idx) :

Exemple 1 :     ***pour idx ← 0 à 9 faire***  
                           *sonner l'alarme*  
                   ***finPour***

Exemple 2 :     ***pour** compteur  $\leftarrow$  1 à 10 **faire***  
                           ***début***  
                                   *resultat  $\leftarrow$  compteur \* 5*  
                                   *afficher ( resultat )*  
                           ***fin***  
                   ***finPour***

Le format général est :     ***pour** index  $\leftarrow$  valeur1 à valeur2 **faire***  
                                   *séquence*  
                   ***finPour***

Il faut noter que :

- L'initialisation est faite au **début** de la boucle.
- La valeur située après '**A**' représente une **condition** de fin de boucle.
- **L'incrément** est en général égal à **l'unité**, il peut être adjoint le mot-clé **PAS** pour changer la valeur de l'incrément.

Exemple :     ***pour** J  $\leftarrow$  0 à 50 **pas** 2 **faire***  
                           *....*  
                   ***finPour***

Les valeurs **initiales** et **finales** peuvent être des expressions algébriques incluant des variables.

## 2.10 EXEMPLES

### 2.10.1 Exemple 1 : commenté



Préparation d'une omelette

**Enoncé** : Préparer une omelette de 6 œufs.

- ✚ **Phase A : Casser** 6 œufs dans un récipient.
- ✚ **Phase B : Battre** les blancs et les jaunes avec une fourchette.
- ✚ **Phase C : Mettre** de l'huile à chauffer dans une poêle sur la cuisinière.
- ✚ **Phase D** : Lorsque la poêle est chaude, y **verser** le contenu du récipient.
- ✚ **Phase E : Enlever** la poêle de la cuisinière quand l'omelette est cuite.
- ✚ **Phase F : Eteindre** le feu.

La **phase A** n'est pas une **entité atomique** : elle doit être décomposée selon le schéma suivant possible :

afpa®	Auteur	Nom région	Formation	Date Mise à jour	Page 10 / 41
	DB	GRN 164	DWWM	04/10/2022	310-Cours_Algorithmie.docx

Phase A :

**A1** : Poser les 6 œufs sur le plan de travail  
**répéter**  
**début**  
     **A2** : Prendre un œuf sur le plan de travail  
     **A3** : Le casser et verser son contenu dans le récipient  
     **A4** : Jeter la coquille dans la poubelle  
**fin**  
**tantque** ( *il y a des d'œufs sur le plan de travail* )

REMARQUE IMPORTANTE :

Une action qui ne peut plus être décomposée est appelée « action primitive ».

### 2.10.2 Exemple 2 : à réaliser selon l'énoncé fourni



Les photocopies

## 2.11 REFERENCE A D'AUTRES ALGORITHMES

Un **algorithme** fait **référence** à un autre algorithme, chaque fois qu'il a **besoin** de la fonction réalisée par celui-ci, on parle de **procédures** ou **fonctions** (voir modularité).

L'algorithme **appelant** (ou module **appelant**) dépend des valeurs résultantes de l'exécution de l'**algorithme appelé**.

De même, l'**appelé** peut avoir **besoin de valeurs transmises** par l'appelant.

Les variables sont **globales** quand l'**appelant** et l'**appelé** y accèdent à leur gré.

On parle de **passage de paramètres** quand l'**appelant** et/ou l'**appelé** se passent des valeurs explicitement.

Exemple 1 :

```
# lancement moteur #
début
    variables x , y : entier
           temps : heure
    temps ← 2.0
    x ← obtenir_vitesse_moteur()
    y ← obtenir_sens_moteur()
    commande_moteur( x, y, temps )
fin
```

afpa®	Auteur	Nom région	Formation	Date Mise à jour	Page 11 / 41
	DB	GRN 164	DWWM	04/10/2022	310-Cours_Algorithmie.docx

Exemple 2 :

```
...
# tri de tableau #
début
    lire_valeurs( tableau_entiers )
    tri( tableau_entiers )
    afficher( tableau_entiers )
fin
```

Remarques : ceci sera étudié dans le chapitre sur la modularité

### 3 AUTRES REPRESENTATIONS

On peut représenter un algorithme par des « langages » plus graphiques que le pseudo-langage.

- Inconvénients :
  - On est plus loin du langage de codage.
  - Difficile de représenter les variables et les structures de données.
  - La représentation des modules avec leur passage de paramètres est très difficile.

#### 3.1 L'ORGANIGRAMME

L'organigramme est une **représentation graphique** d'un algorithme. Il permet de visualiser les actions et le cheminement de l'exécution d'un programme.

Il existe de **nombreux symboles graphiques** pour identifier les actions décrites par un algorithme.

Remplir les symboles avec du code est un non-sens : l'organigramme appartient à la phase de conception et non à celle du codage.

##### 3.1.1 Symboles principaux

Les symboles sont en nombre important, mais seuls les principaux seront retenus ici et présentés.

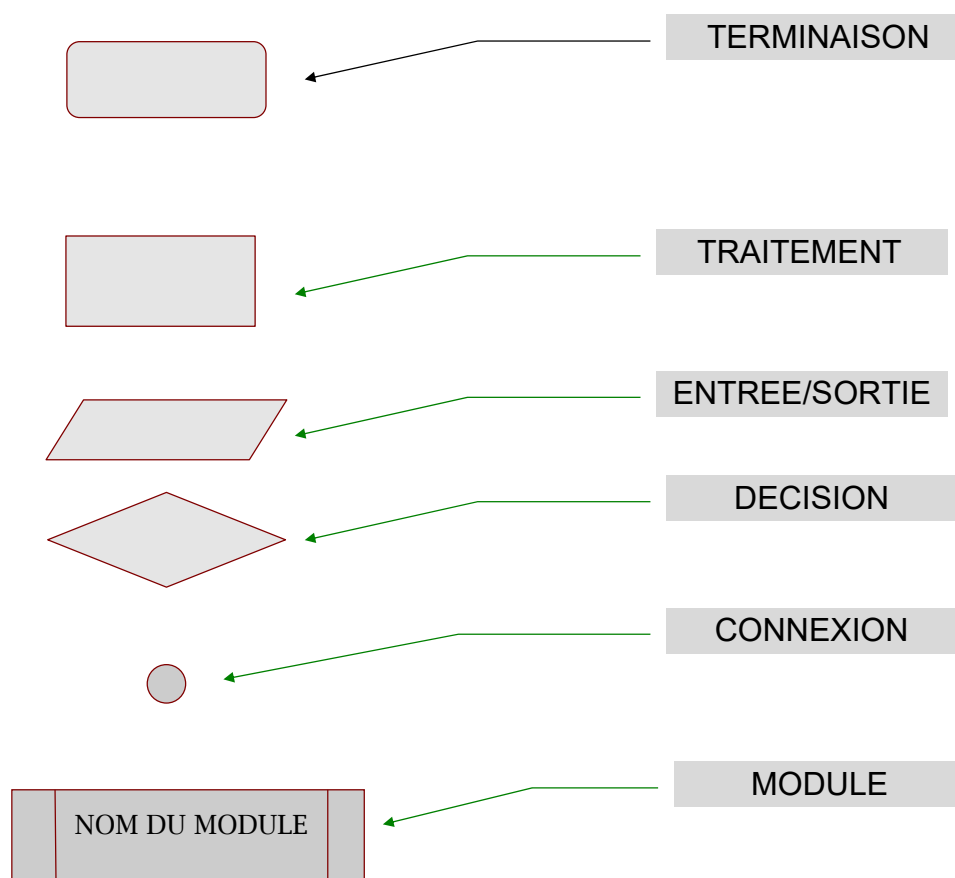
Chaque symbole contient une information relative à l'action exécutée.

Le symbole **TERMINAISON** est utilisé au début ou à la fin d'un organigramme.

En programmation structurée, il n'y a qu'une seule entrée et qu'une seule sortie dans un algorithme. En conséquence, un seul symbole **TERMINAISON** avec les mots **FIN**, **RETOUR**, **STOP**... et un seul symbole **TERMINAISON** avec les mots **DEBUT**, ... existent dans l'organigramme.



## SYMBOLES GRAPHIQUES FONDAMENTAUX



Le rectangle **TRAITEMENT** contient la **description** des actions à exécuter. Un seul arc arrive et un seul arc part d'un rectangle.

Le symbole **E/S** symbolise une **entrée** ou une **sortie** d'information ou de données.

Le symbole **DECISION** correspond à la **condition binaire**. L'arc d'entrée est situé en haut du symbole. Les 2 arcs représentant les 2 résultats possibles le quittent en bas et sur un côté.

Les cercles **CONNEXION** sont utilisés par paires et symbolisent un lien évitant ainsi des enchevêtrements d'arcs.

Le pseudo-langage permet la définition de **modules**. Un symbole particulier existe pour référencer un tel module dans un organigramme.

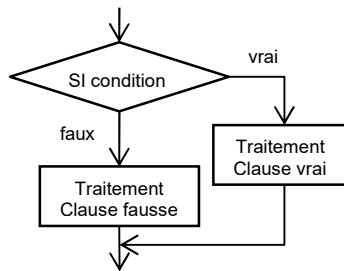
Ce module doit lui-même posséder son **propre organigramme**. C'est à dire posséder une **ENTREE** et une **SORTIE**.

Le parcours des arcs d'un organigramme se fait du **haut vers le bas** et de la **gauche vers la droite**. Les exceptions sont indiquées par des flèches précisant le sens du parcours pour l'arc considéré.

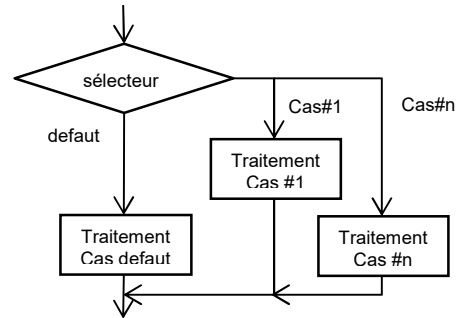
## 3.2 REPRESENTATION DES CONSTRUCTIONS DE BASE

### 3.2.1 Sélection

SI condition ALORS ... SINON ...

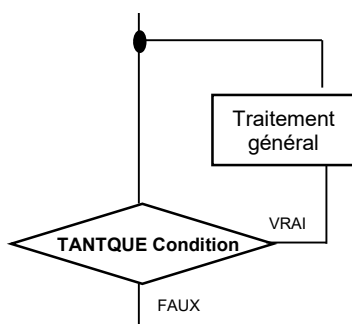


SELON sélecteur

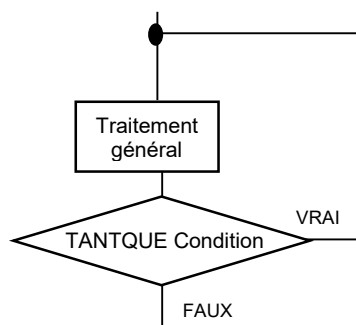


### 3.2.2 Itération

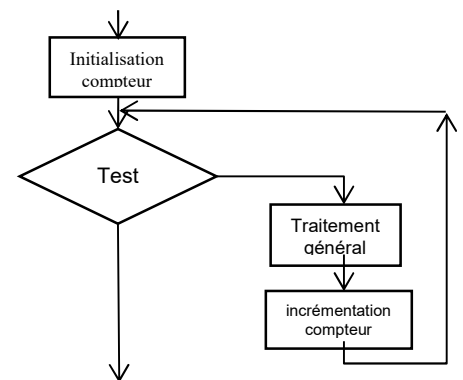
TANTQUE



REPETER\_TANTQUE



POUR\_FAIRE

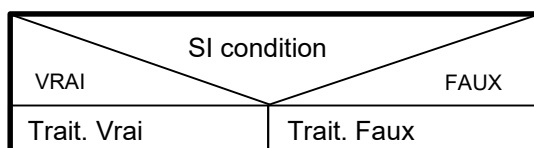


## 3.3 LES GRAPHES DE NASSI-SCHNEIDERMAN

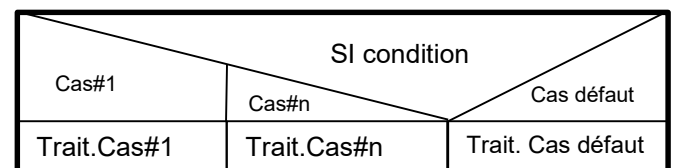
[Isaac Nassi](#) et [Ben Shneiderman](#) ont développé cette représentation en 1972. C'est une représentation graphique basée sur des diagrammes emboîtés (encore connu sous les noms de NSD ou de structogramme).

### 3.3.1 Sélection

SI condition ALORS ... SINON ...

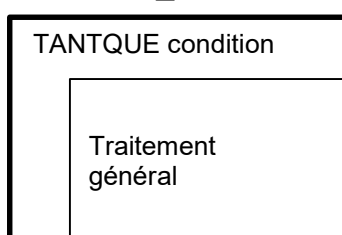


SELON sélecteur

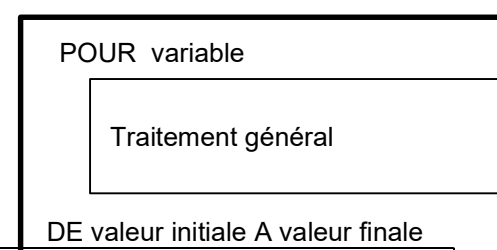
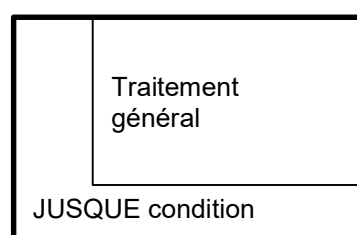


### 3.3.2 Itération

TANTQUE  
POUR\_FAIRE



REPETER\_JUSQUE

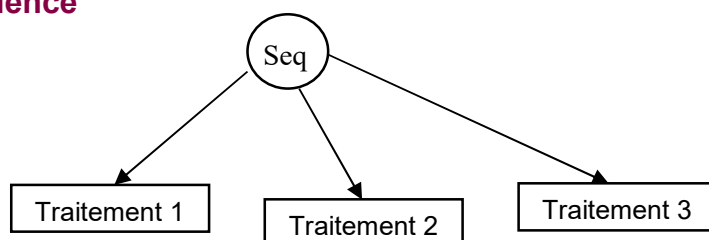


### 3.4 LES ARBRES ALGORITHMIQUES

Les arbres programmatiques sont issus de la théorie des arbres.

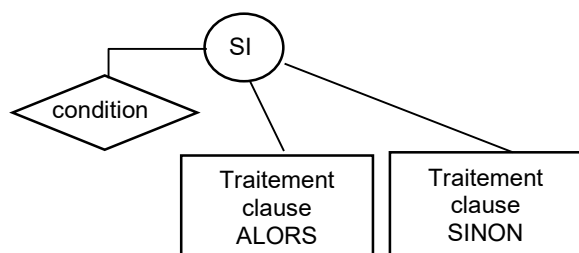
C'est une représentation graphique où la lecture se fait en tournant dans le sens trigonométrique.

#### 3.4.1 Séquence

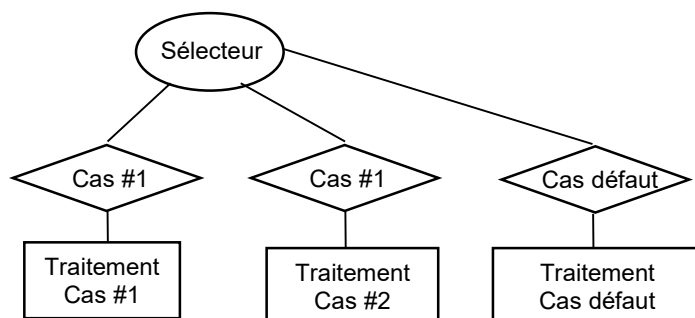


#### 3.4.2 Sélection

SI condition ALORS ... SINON ...

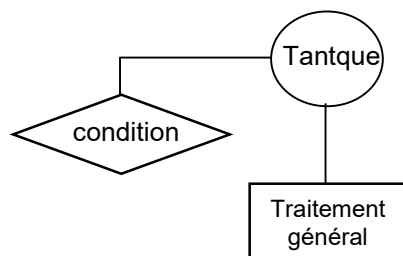


SELON sélecteur

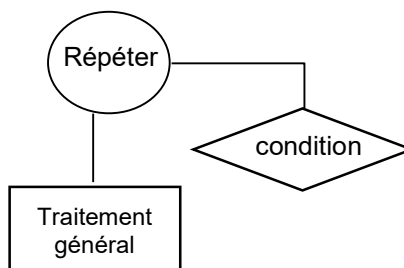


#### 3.4.3 Itération

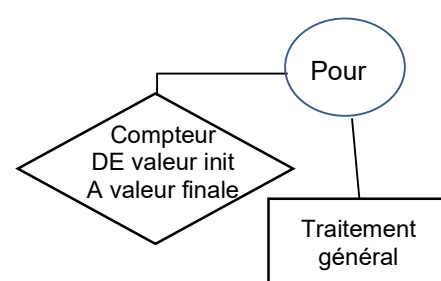
TANTQUE



REPETER\_JUSQUE



POUR\_FAIRE



## 4 LA MODULARITE

### 4.1 CONCEPTS GENERAUX

Dans les chapitres précédents, nous avons vu comment développer des algorithmes grâce à des **procédures d'affinement** des instructions.

A chaque étape de cet affinement, l'algorithme est divisé en **composants plus petits** qui peuvent être définis à leur tour de manière plus détaillée. **L'affinement s'achève** lorsque chaque élément de l'algorithme est exprimé de manière telle que le **processus prévu puisse l'interpréter**.

**Les composants rencontrés** pendant l'affinement sont souvent totalement **indépendants** de l'algorithme principal au sens où ils peuvent être conçus **extérieurement** au contexte dans lequel ils doivent être utilisés.

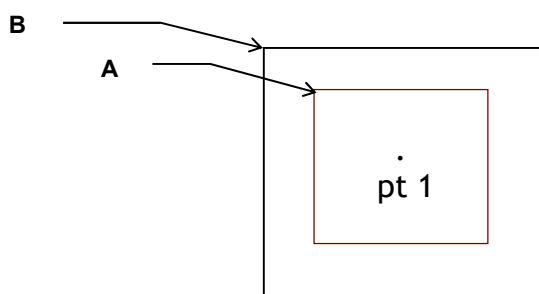
De tels éléments peuvent être conçus par quelqu'un d'autre que le concepteur de l'algorithme principal et peuvent également être utilisés comme éléments d'un autre algorithme.

Le nom donné à chaque procédure doit donc automatiquement exprimer ce qu'elle fait. **On sait qui fait quoi**.

Enfin, la chasse aux « bugs » devient plus facile. Pour trouver une erreur ou modifier un point du programme, il n'y a pas besoin de lire toutes les instructions. On cherche **directement** dans la procédure « **déficiente** ».

Afin d'illustrer ceci, imaginons un problème consistant à dessiner deux carrés concentriques avec une table traçante. Le point **pt1** étant le centre des carrés.

On peut raisonnablement esquisser l'algorithme suivant :



#### Début

- 1- *initialiser la table traçante en pt1*
- 2- *lever la plume*
- 3- *déplacer en A*
- 4- *abaisser la plume*
- 5- *dessiner un carré de 40 unités de côté*
- 6- *lever la plume*
- 7- *déplacer en B*
- 8- *abaisser la plume*
- 9- *dessiner un carré de 80 unités de côté*

*fin*

**Remarque** : La quatrième et la septième instruction de cet algorithme invoquent le dessin d'un carré, qui est une **procédure indépendante** du reste de l'algorithme, ce qui implique que nous pouvons **concevoir** un algorithme qui l'utilise.

Un algorithme qui peut être intégré à un autre algorithme est un **module** (dans certains langages de programmation, c'est aussi appelé une **procédure**, une **routine**, une **fonction**...).

Un tel algorithme - un **module** - est donc utilisé en tant **qu'élément externe**.

afpa®	Auteur	Nom région	Formation	Date Mise à jour	Page 16 / 41
	DB	GRN 164	DWWM	04/10/2022	310-Cours_Algorithmie.docx

Un **module** est un algorithme en soi et peut être conçu **indépendamment** du contexte dans lequel il doit être utilisé. On dit d'un algorithme qui utilise un module « **qu'il l'appelle** ».

Dans l'exemple des carrés précédent, concernant le dessin de chaque carré, il va falloir écrire des instructions permettant le tracé de chacun des côtés avec des consignes très similaires pour les 2 carrés.

*début*

*initialiser la table traçante en pt1*

*lever la plume*

*déplacer en A*

*abaisser la plume*

*pour idx ← 1 à 4 faire*

*tourner à droite*

*tracer sur 40 unités*

*finPour*

*lever la plume*

*déplacer en B*

*abaisser la plume*

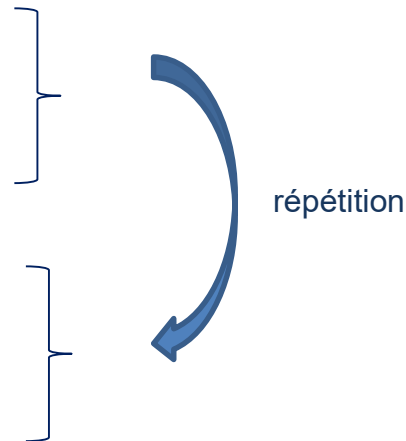
*pour idx ← 1 à 4 faire*

*tourner à droite*

*tracer sur 80 unités*

*finPour*

*fin*

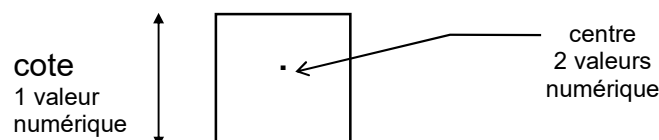


Avec la technique précédente nous avons dû écrire quasiment 2 fois **la même chose**. Si le cahier des charges avait demandé 5 carrés, il aurait fallu définir 5 fois la même chose .....  
!!!!!!

Pourtant qu'est-ce qu'un carré ? C'est simplement une figure géométrique ayant 4 côtés (peu importe leur taille) et 4 angles de 90°.

-----> Il serait donc **intéressant** de réaliser une **procédure** qui satisfasse ces critères et qui puisse prendre en compte la modification de taille. C'est la notion de **procédures avec données** ou **procédures paramétrées**.

On peut donc imaginer la **généralisation** d'une procédure appelée **carre** qui admettrait 2 paramètres nommés **centre** et **cote** et qui permettrait lors de son appel de dessiner un carré dont le centre serait **centre** et la longueur du côté serait **cote**.



Cette procédure ainsi définie constitue un **module à part utilisable** par n'importe quel algorithme qui aurait besoin de ses services.

## C'est la notion même du sous-programme ou de la fonction !

On peut même aller **plus loin** en tenant compte de l'**évolutivité** du module en se disant :

« Ne serait-il pas intéressant pour un proche avenir - en anticipant sur les besoins - de construire un module encore plus général qu'on appellerait Polygone dont le rôle serait de tracer un polygone de centre « centre », de longueur de côté « cote », avec un nombre de côtés « nbr\_cote ». Hein ?? »

La réponse est : Si.

L'appel de ces procédures se ferait ainsi :

```
variables pt_centre : POINT
          longueur : ENTIER
          nombre_cot : ENTIER

pt_centre = ( 5, 4 )
longueur = 10
carre ( pt_centre , longueur ) # Trace un carré de centre pt_centre et de
                              # longueur de côté longueur


pt_centre = ( 22, 54 )
longueur = 15
nombre_cot = 6
polygone ( pt_centre, longueur , nombre_cot )
```

Le pseudo-code de ces modules serait :

```
procedure carre (pt : POINT, long : ENTIER)
DEBUT
  variables idx : ENTIER

  initialiser la table trançante à pt
  abaisser la plume
  POUR idx DE 1 A 4 FAIRE
    tourner à droite
    tracer_longueur ( long )
  FIN_POUR
  lever la plume
FIN

procedure polygone (pt : POINT, long : ENTIER, nbrCote : ENTIER )
DEBUT
  variables idx : ENTIER
  initialiser ....
  POUR idx DE 1 A nbrCote FAIRE
    .....
    tourner d'un angle de 360/nbrCote
    .....
  FIN_POUR
FIN
```

	Auteur	Nom région	Formation	Date Mise à jour	Page 18 / 41
	DB	GRN 164	DWWM	04/10/2022	310-Cours_Algorithmie.docx

**REMARQUE** : Les exemples utilisés ci-dessus sont parfaitement **applicables** avec un langage informatique **évolué** comme le C.

---> Ce qui signifie que lorsque le **PSEUDO-CODE** d'un algorithme est bien **pensé** et validé, le **codage** c'est à dire la **traduction** dans un **langage informatique** ne constitue qu'une formalité quasi-mécanique !!

*Le codage ne représente que l'étape ultime dans le processus de la programmation informatique !!*

Dans les exemples précédents, les paramètres *pt*, *long* et *nbr\_cot* utilisés dans la définition des procédures sont appelés des « **paramètres formels** ».

Lorsque le module *Carre* ou le module *Polygone* est appelé, les paramètres *pt\_centre*, *longueur*, *nombre\_cot* sont appelés « **paramètres réels** » ou « **paramètres actuels** ».

Ce mot « paramètre » peut également s'appeler « **argument** ».

La notation retenue pour les modules ou procédures est la suivante :

```

procedure nomProcedure ( liste des paramètres formels ) : type
    # spécification en commentaire du rôle du module décrit
    DEBUT
        ..... corps du module ...
    FIN

```

**NB** : La notation définissant une fonction en langage **C** est très voisine de celle-ci.

Le commentaire derrière l'accolade ( ou # ) aide le lecteur à comprendre ce que fait le module ; le corps du module décrit comment il le fait. C'est la définition.

On appelle un module de la manière suivante :

```

[ variable = ] nomModule ( paramètres réels )

```

où « **variable** » est la variable de type **type** potentiellement renvoyée par le « module », la « routine » ou encore la « fonction » que l'on appelle.

**NB** : La variable « **variable** » affectée dans le module dit « appelant », à l'issue de l'appel du module dit « appelé », doit naturellement être compatible avec le type de l'expression renvoyée par ce module appelé.

Ce type doit être mentionné dans l'en-tête du module appelé.

C'est ce que le **processeur** interprète comme une directive pour exécuter le **corps du module appelé** dont les **paramètres formels** ont été remplacés par les **paramètres actuels** de l'appel.

Les **paramètres formels** d'un module peuvent être considérés comme la représentation de l'information nécessaire au module lorsqu'il est appelé.

Il doit y avoir naturellement le **même nombre** de **paramètres actuels** et **formels** et chaque paramètre réel doit avoir le même type que son équivalent paramètre formel.

## 4.2 LE PASSAGE DES ARGUMENTS

La **substitution** des **arguments**, ou **paramètres**, **formels** par des **arguments réels** est appelée **passage d'arguments** ou de **paramètres**, en raison du **transfert** d'informations qui a lieu entre le programme appelant et le sous-programme appelé.

Généralement, on distingue **3 types de passage des arguments** :

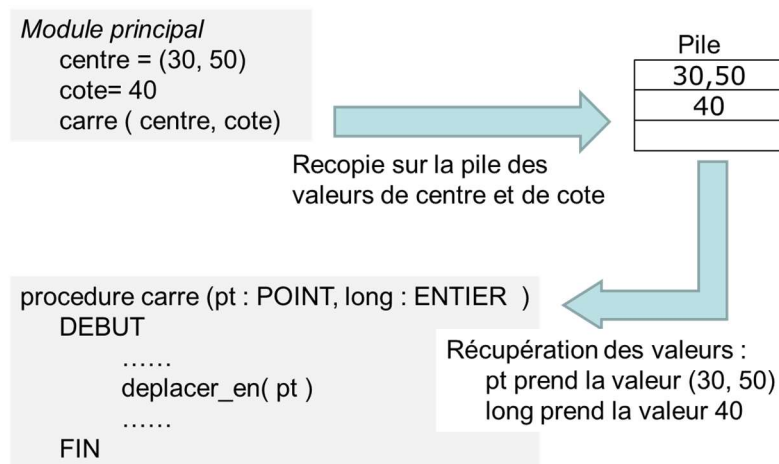
- passage par **valeur**
- par **référence**
- par **adresse**.

Tous ces modes ne coexistent pas dans tous les langages de programmation.

### 4.2.1 Le passage par valeur

Dans ce mode, le sous-programme appelé ne s'intéresse qu'aux **valeurs** des **arguments réels** au moment de l'appel, sans désirer changer la valeur de ces paramètres dans le programme appelant.

La solution informatique adoptée consiste à « **recopier** » les valeurs des arguments du programme appelant dans les arguments correspondant de la procédure appelée, lors de l'appel.



Toutes les opérations portant sur ces arguments locaux à la procédure appelée seront alors **sans effets** sur les arguments réels de l'appelant.

Ce mode de passage présente l'avantage de **protéger** les données du programme principal, au prix de la diminution **de la vitesse d'exécution** et d'une **plus grande occupation de la mémoire**.

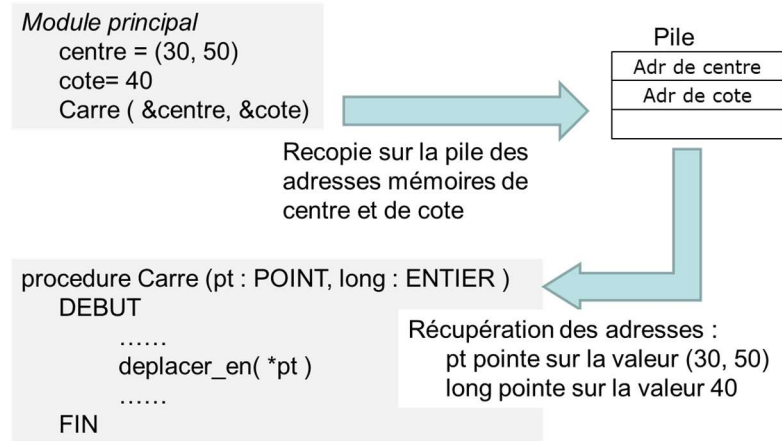
*Ce mode est le passage d'argument par défaut de nombreux langages tel que le C.*

### 4.2.2 Le passage par adresse

Dans ce mode, est transmise à chaque appel non pas la valeur, mais **l'adresse de l'argument réel**.

Ce mode est d'emploi **dangereux**, car **la protection des arguments réels** n'est plus assurée. La modification de la valeur de l'argument dans le sous-programme (appelé ou procédure) entraîne une modification définitive des arguments réels. !!!!

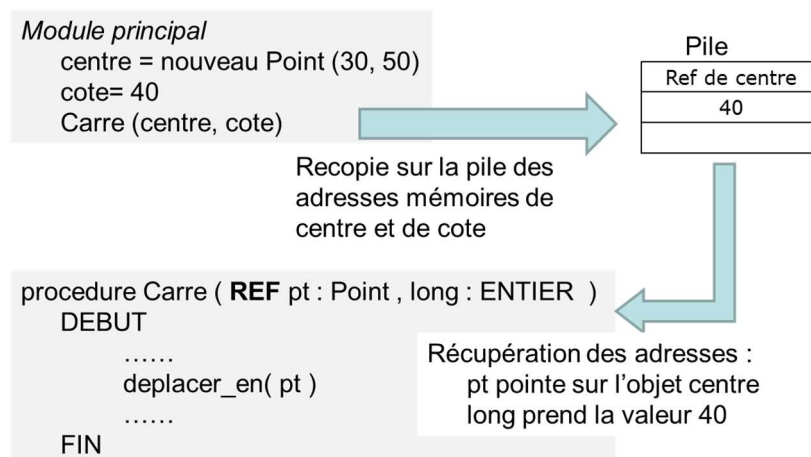




L'avantage de ce mode **est la rapidité d'exécution**. De plus il n'y a pas de 'doublon' pour la même information.

### 4.2.3 Le passage par référence

Dans ce mode, est transmise à chaque appel non pas la valeur, mais **la référence de l'argument réel**. Ce mode de passage est beaucoup utilisé dans les langages de programmation orienté objet.



## 4.3 LA VISIBILITE D'UNE VARIABLE

Les unités de programme peuvent **échanger des informations** en passant des arguments à l'appel d'un sous-programme. Elles peuvent aussi partager les données, et avoir accès à des informations communes sans qu'elles soient transmises comme paramètres. On dit alors qu'elles « **partagent** » les informations.

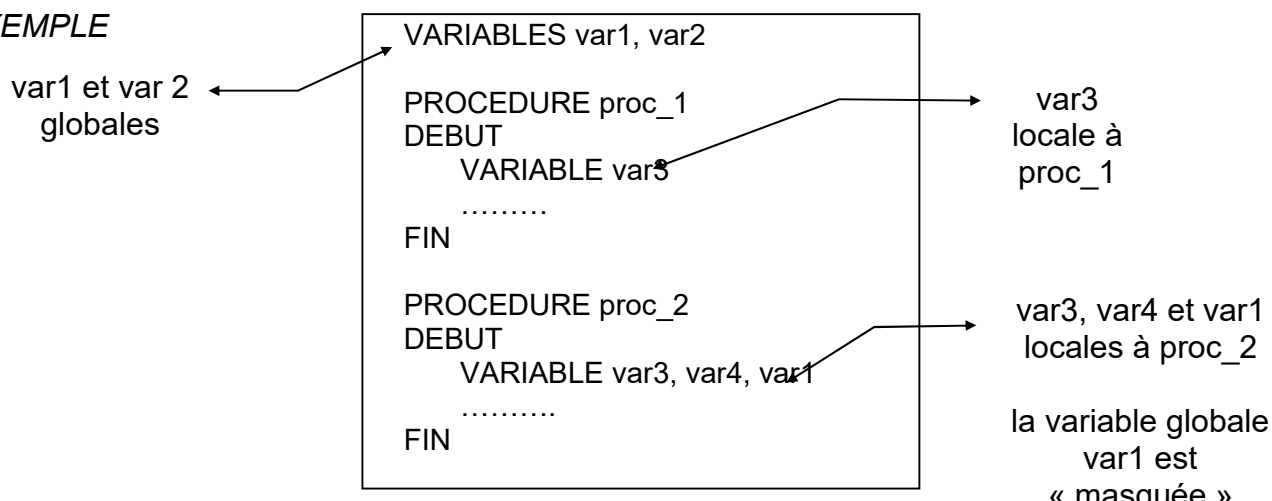
C'est ce que l'on appelle les variables **globales**. Elles ont une **durée de vie** correspondant à l'existence du programme. Leur **portée** est maximum.

On appelle **variable locale** à une unité de programme ou module ou procédure, **une variable qui n'a de sens que pour cette unité**. Leur existence est liée au temps pendant lequel s'exécute le module.

Inversement, **toute variable globale est une variable qui a un sens**, c'est à dire définie, **dans toute procédure du programme**.

On évoque également les termes **visibilité** ou **portée** d'une variable pour mentionner le lieu où elle est déclarée.

### EXEMPLE



Les langages qui possèdent une telle structure, c'est à dire la capacité à tout sous-programme d'accéder à **toute variable locale** lui appartenant et d'accéder à n'importe quelle **variable globale** est appelé **langage à structure de blocs**.

Ainsi, dans les langages structurés, une variable ne peut intervenir **que dans le bloc, ou l'unité de programme, dans laquelle elle a été définie**.

Cela revient à dire que « **la visibilité** » ou « **portée** » d'une variable est **toujours minimale**. Tout se passe comme si la variable ne « portait » pas plus loin que son propre bloc.

Cette règle a pour but d'éviter **les effets secondaires indésirables** et les modifications intempestives qui pourraient avoir lieu pendant l'exécution d'un sous-programme : chaque sous-programme opère sur **ses propres variables**, et la communication entre le programme principal ou appelant et les modules ou sous-programmes est réalisée par l'intermédiaire de **variables d'interfaces prévues à cet effet : les arguments**.

Par conséquent, une bonne règle de programmation consiste à **préférer les variables locales aux variables globales**, afin d'éviter toute interaction nuisible avec le programme principal.

Ces effets indésirables sont appelés **effets de bord**.

## 4.4 PASSAGE PAR ARGUMENT OU VARIABLES GLOBALES ???

Nous avons vu qu'il était possible de transmettre des informations entre entités de **deux manières différentes** :

- Passage d'informations à l'appel d'un sous-programme par **le biais des arguments**.
- Passage d'informations par **variables globales**.

D'une manière générale, la **Programmation Structurée** milite en faveur d'une définition claire des informations transmises et de leur mode de passage.

Or, si le passage d'informations par variables globales permet, dans certains cas, **d'accroître la rapidité d'exécution** dans le cas de sous-programmes très employés, **c'est toujours au détriment de la clarté**, de la lisibilité et donc de la rigueur.

Sans compter les effets de bord qui génèrent tous les problèmes vus précédemment !!!

D'une manière générale, il sera toujours préférable de passer un maximum d'informations dans les arguments des procédures et des fonctions.

Restreindre la **dissémination** des informations, par la définition explicite et l'étroitesse « des interfaces » entre ses différents éléments, n'est plus du goût ou de l'appréciation du programmeur : **cela devient une nécessité vitale pour la conception et la définition de logiciels professionnels.**

#### 4.5 RECURSIVITE DES MODULES

Pour conclure ce chapitre concernant les modules, il est intéressant d'aborder un dernier concept : la **récurtivité**.

Une procédure est dite **récursive** si une de ses propres instructions consiste à s'appeler elle-même. Ce qui impose dans le code du module de prévoir une condition d'arrêt évitant ainsi un appel récursif infini.

*EXEMPLE : calcul de N factoriel*

**Rappel :**  $N! = N * (N - 1) * (N - 2) * (N - 3) * \dots * 2 * 1$   
 $N! = N * (N-1) !$

```

ENTIER facto (x : ENTIER)
  DEBUT
    Variables resultat : ENTIER
    SI ( x = 1 )
      ALORS
        resultat = 1
      SINON
        resultat = x * facto ( x - 1 )
      FINSI
    retourner ( resultat )
  FIN
  
```


#### 4.6 AVANTAGES DE LA MODULARITE

Les avantages que comporte **l'utilisation de modules** peuvent être résumés de la manière suivante :

- Les modules correspondent naturellement à l'affinement progressif de l'algorithme et l'on obtient ainsi une conception descendante.
- Un module est en soi un composant de tout autre algorithme plus important qui l'appelle. Un module et l'algorithme appelant peuvent être conçus **indépendamment** l'un de l'autre, ce qui simplifie la procédure de conception.

- Pour intégrer un module à un algorithme, il suffit de savoir ce que fait le module et non comment il le fait. Ce que fait un module peut être expliqué de manière adéquate par un commentaire de début.
- De même que les modules simplifient la conception des algorithmes, ils simplifient aussi leur compréhension.
- Pour comprendre un algorithme complexe, il suffit de comprendre **l'effet des modules qui le composent**, sans pour autant savoir comment ces résultats sont atteints.
- **La facilité de compréhension** est très importante dans le cas fréquent où un algorithme doit être modifié (particulièrement lorsqu'il doit l'être par d'autres personnes que l'auteur).
- Une fois qu'un module a été conçu, il peut être intégré à tout algorithme qui en a besoin.
- Il est de ce fait possible de construire une bibliothèque de modules, tels que des modules de tri, de résolution d'équations, de manipulations de fichiers...

C'est le premier pas vers la notion de « Composants logiciels » !!

	Auteur	Nom région	Formation	Date Mise à jour	Page 24 / 41
	DB	GRN 164	DWWM	04/10/2022	310-Cours_Algorithme.docx

## 5 LES STRUCTURES DE DONNEES

### 5.1 INTRODUCTION

Nous avons vu dans les chapitres précédents que la **programmation structurée** revient à un **découpage méthodique et hiérarchique** des algorithmes en modules (des programmes en sous-programmes).

**Organiser les actions** est une première étape mais qui demeure **insuffisante** tant que l'on n'a pas abordé ce qui constitue le cœur de la programmation : **les données**.

La programmation structurée passe en effet par l'**organisation des données**, et, qu'il s'agisse **d'actions ou d'informations**, la démarche est similaire : **modularité, analyse logique, implémentation physique ...**

Qu'appelle-t-on une donnée ? Une donnée c'est avant tout une information : 3, 128, « DUPONT », le texte d'un livre, la liste des stagiaires, sont autant de données manipulables par l'informatique.

Les données sont regroupées en **classe** (on dit aussi **type**) dont tous les éléments possèdent les mêmes propriétés : par exemple les nombres peuvent être ajoutés, multipliés, etc ..., un texte peut se voir complété par insertion d'un autre texte, modifié certains de ses mots.

Chaque **classe** comporte donc des **caractéristiques opératoires spécifiques**.

C'est là que réside la notion de structure de données : **définir un type de données revient à décrire l'ensemble des opérations réalisables sur chaque élément appartenant à ce type**.

On parle alors de **type abstrait**, pour bien les différencier de leur **réalisation interne**.

Dans le cadre des structures de données, on distingue deux niveaux de description :

- **le niveau logique** qui s'attache à la description des opérations permises et à leurs propriétés, et ...
- **le niveau physique** qui correspond à la technique et au mode d'implémentation des structures logiques sur ordinateur, compte-tenu des contraintes hard ou soft, celui-ci ne manipulant que des octets.


### 5.2 LES TYPES DE BASE

On appelle **type de base** les types de données élémentaires qui autorisent une **réalisation physique simple et immédiate**.

Les **données scalaires** et **tableaux de données** sont considérés comme étant la base de la programmation.

#### 5.2.1 Les données scalaires

Ce sont les types de données les plus simples qui soient ; leur structure se réduisant à un seul élément : 3, 6, « D » VRAI, en sont des exemples.

	Auteur	Nom région	Formation	Date Mise à jour	Page 25 / 41
	DB	GRN 164	DWWM	04/10/2022	310-Cours_Algorithmie.docx

Ces types de données sont souvent définis **dans le langage lui-même** : **nombres entiers ou réels** (simple ou double précision), **caractères** (de '0' à '9', de 'a' à 'z', de 'A' à 'Z'... plus les **caractères spéciaux de ponctuation**, accentués, échappement...) et les données **booléennes**.

Parfois (comme en C par exemple), il est possible de créer ses propres **données scalaires** par l'emploi de deux mécanismes différents : **l'énumération** qui revient à décrire l'ensemble des valeurs possibles que peut prendre une variable appartenant à ce type, ou **l'intervalle** qui consiste à restreindre l'éventail des valeurs possibles d'un type prédéfini.

Par exemple le type *jour* sera **énuméré** par *lundi, mardi, mercredi, jeudi, vendredi, samedi, dimanche* ; et le type *nombre\_jour* sera **énuméré** par l'intervalle 1....366 ; sous-ensemble des entiers.

Les opérations permises sur toutes les données scalaires sont la **définition** et la **création**, **l'affectation** et la **lecture** d'une valeur dans une variable.

**Certaines opérations sont limitées à un type particulier.** L'addition, la multiplication, la soustraction, la division se retrouvent pour tous les types numériques, avec en outre toutes les comparaisons possibles : égalité, relations d'ordre, ....

Les **données booléennes** autorisent les opérations logiques **ET**, **OU**, **NEGATION** et leur combinaison.

Les caractères permettent la comparaison.

Chaque donnée occupe un certain nombre de cellules mémoire : un **octet** pour les caractères, deux ou plus pour les entiers, quatre ou plus pour les réels, et un seul bit suffit à mémoriser les variables booléennes.

### 5.3 LES TABLEAUX

Les tableaux sont certainement les structures de données les plus connues des programmeurs.


Un tableau peut être considéré soit comme une représentation de l'aspect physique des ordinateurs (suite de cellules mémoires) soit comme **une représentation informatique des matrices mathématiques**.

La programmation structurée, dont le but est de faire de l'informatique une science indépendante des supports ( ordinateurs ou langages de programmation ) fait bien entendu appel à l'aspect matriciel des tableaux.

**Tous les éléments d'un tableau** sont nécessairement **de même type** et peuvent individuellement être sélectionnés par l'utilisation **d'indices**, grâce aux fonctions d'accès aux éléments de la structure.

L'idée de matrices s'étend aisément aux cas de tableaux à plusieurs indices. On parle alors **des dimensions ou des rangs d'un tableau**.

Un tableau à N dimensions peut être considéré comme un vecteur dont les éléments sont des tableaux à N-1 dimensions. Par exemple, un tableau à 3 dimensions (un cube) peut se voir comme un vecteur de matrices (des tableaux à 2 dimensions).

	Auteur	Nom région	Formation	Date Mise à jour	Page 26 / 41
	DB	GRN 164	DWWM	04/10/2022	310-Cours_Algorithmie.docx

### 5.3.1 Les agrégats, enregistrements ou structures

S'il est possible avec les tableaux de regrouper des éléments de même nature, il est souvent intéressant de pouvoir en faire de même avec des données de nature différentes. On parle alors d'**agrégats** ou d'**enregistrements**

Un agrégat possède un identifiant définissant un nouveau type de données. Chaque donnée ou champ qui le constitue à lui aussi son propre identifiant et type.

L'accès à un champ ne s'effectue plus par des indices, mais simplement par la **juxtaposition** du nom de la donnée et de celle du champ séparés par un point « . ».

Ce type de données, appelée **record** en PASCAL et **structure** en C, se rencontre maintenant dans tous les langages de programmation.

*AGREGAT Personne*  
*CHAINE nom*  
*CHAINE prenom*  
*ENTIER age*  
*FIN\_AGREGAT*

*ENREG Personne*  
*CHAINE nom*  
*CHAINE prenom*  
*ENTIER age*  
*FIN\_ENREG*

*STRUCT Personne*  
*CHAINE nom*  
*CHAINE prenom*  
*ENTIER age*  
*FIN\_STRUCT*

La déclaration d'une variable de ce type de donnée : pers1 : Personne

L'accès aux champs se fera par : pers1.nom = "Dupond" ou SI ( pers1.age > 18 ) ...

## 5.4 LES STRUCTURES DYNAMIQUES

Nous avons vu jusqu'à présent des **structures de données statiques** c'est à dire des variables dont l'organisation reste **inchangée** en cours d'exécution du programme. Il en est autrement **des structures dynamiques** que nous allons examiner maintenant.

Ces dernières recouvrent un champ d'application **très vaste** : de la création d'interpréteur ou de compilateur, jusqu'aux programmes d'**Intelligence Artificielle**, l'informatique use et abuse des structures dynamiques : **piles, files, listes chaînées, arbres....**

### 5.4.1 Les piles

Les **piles** sont certainement **les structures dynamiques les plus usitées**.


Une **pile** se définit intuitivement comme un **empilage** d'éléments dont seul le **dernier introduit est visible**. On peut ajouter ou retirer des éléments mais par un seul côté, le dernier entré devient le premier accessible.

D'où le nom de **LIFO** (Last Input First Output).

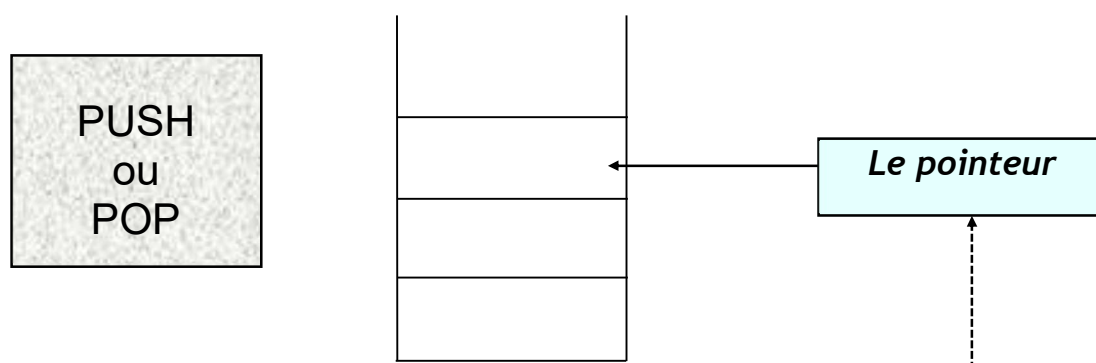
De manière plus rigoureuse, une pile est décrite par un ensemble d'éléments de même type sur lequel sont définies **3 opérations** : 2 fonctions d'accès et un prédicat, c'est à dire une fonction de test.

Les fonctions d'accès servent à placer ou à enlever un élément, le prédicat à déterminer si la pile est vide. On peut les appeler : **Deposer**, **Enlever**, et **Vider**.

Pour représenter une pile, on utilise un vecteur d'éléments et un **pointeur** qui représente le **sommet** de la pile.

	Auteur	Nom région	Formation	Date Mise à jour	Page 27 / 41
	DB	GRN 164	DWWM	04/10/2022	310-Cours_Algorithmie.docx

Ajouter un élément revient à le placer à l'endroit désigné par cet indicateur et à **l'incrémenter** ; pour enlever un élément, il suffit d'accomplir l'opération inverse :



*L'opération PUSH incrémente le pointeur tandis qu'une opération POP décrémente ce pointeur.*

**Les piles sont énormément répandues dans le monde de la programmation.**

Les appels de procédure ou de fonction, sont gérés grâce à des piles.

**L'adresse du programme appelant** est placée dans une pile avec les arguments du sous-programme appelé.

Ce dernier ensuite « **dépile** » ces arguments avant de les utiliser.

A la fin de l'exécution de la procédure, le contrôle est placé à l'adresse située au sommet de la pile.

Lorsqu'une procédure en appelle une autre qui en appelle une autre ... les adresses sont dépilées dans l'ordre inverse afin de redonner le contrôle aux modules appelants.

### 5.4.2 Les files

La **file d'attente**, par certains côtés, ressemble beaucoup à une pile mais, à une pile ouverte.

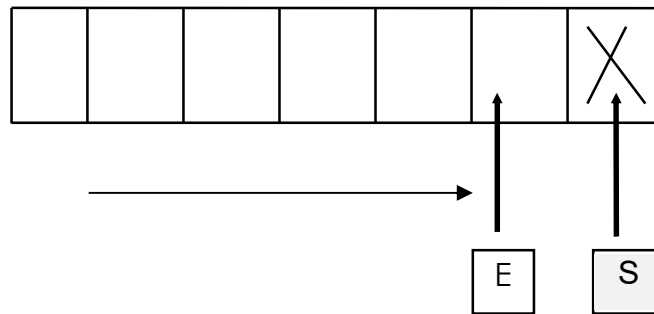
En effet, une file d'attente (que l'on nomme simplement une file) se définit par un ensemble d'éléments sur lequel les 3 opérations : **Deposer**, **Enlever**, **Vide** sont possibles.

Mais à l'inverse d'une pile, une file enlève le premier élément qui lui a été entré. C'est pourquoi on l'appelle **FIFO** ( **F**irst **I**nter **F**irst **O**utput ).

L'implémentation physique d'une file se réalise généralement au moyen d'un tableau et de 2 pointeurs. **L'un représente l'entrée, l'autre la sortie** de la structure. Du fait de l'insertion et de la lecture des éléments par incrémentation des pointeurs, il y a un déplacement continu de ces pointeurs dans la file.



A cet effet, lorsqu'un pointeur arrive au sommet, il est remis à zéro afin de pointer à la base du tableau et être à même de continuer sa tâche.



Lorsque le pointeur d'entrée rattrape celui de la sortie, **la file est pleine**. Si, en revanche, c'est le pointeur de sortie qui rattrape celui d'entrée, alors **la file est vide**.

Ces structures sont surtout utilisées dans les programmes de simulation pour représenter l'attente de personnes ou d'événements, dans les buffers d'entrée/sortie ou, d'une façon générale, en **gestion de processus dans les systèmes d'exploitation multitâches** : l'attente de programmes pour disposer d'une imprimante, par exemple, doit être gérée au moyen d'une file.

### 5.4.3 Les structures chaînées

Nous allons maintenant entrer dans les structures dites « très **dynamiques** ». Aucun système, aucun logiciel puissant ne serait possible à l'heure actuelle sans de telles entités.

Physiquement, nous allons voir que **l'élément essentiel est le pointeur**.

A l'inverse des tableaux dans lesquels les éléments sont sagement alignés les uns à côté des autres, les composantes des structures dynamiques sont **disséminées** dans l'espace mémoire adressable et reliées entre elles grâce aux **pointeurs**.

Pointer consiste à se référer à un élément sans le nommer explicitement.

#### 5.4.3.1 Listes chaînées linéaires

Une **liste linéaire** se décrit logiquement comme **une liste ordonnée de taille variable constituée d'éléments de type défini**, sur laquelle certaines opérations sont rendues possibles.

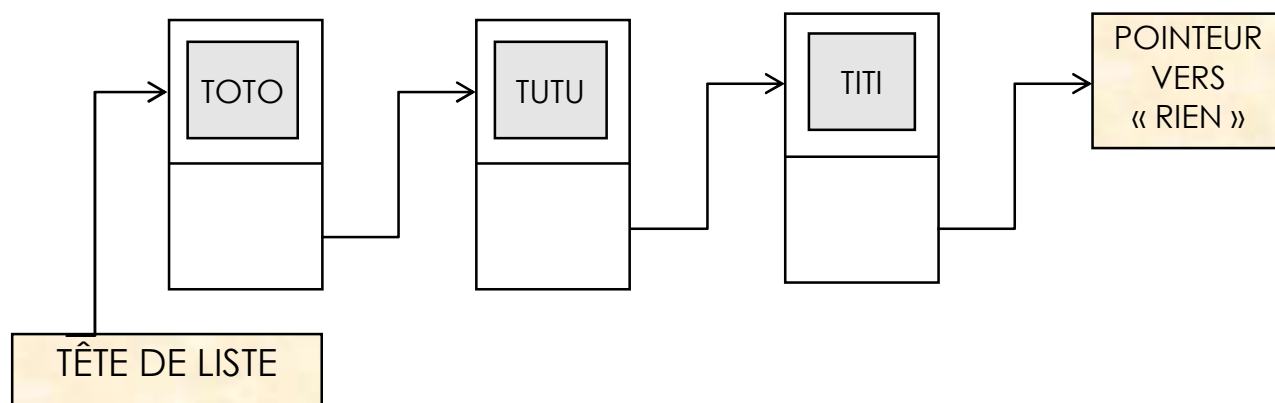
Ces opérations sont les suivantes :

- L'accès à un élément particulier de la liste : celui-ci n'est **pas réalisé** par l'intermédiaire d'un indice mais **par rapport à un autre élément** de la liste grâce aux fonctions : « **Premier** » qui ramène le premier élément de la liste et « **Suivant** » qui permet d'avancer dans la liste et de ramener élément.
- **L'insertion** d'un élément dans la liste.
- La **suppression** d'un élément de la liste.
- Le test déterminant si la **liste est vide**.

On utilisera donc **des listes chaînées linéaires** chaque fois que l'on aura affaire à un ensemble d'éléments **de taille variable**, dans lequel les opérations d'insertion, de suppression et d'accès doivent être **accomplies n'importe où**.

Un texte sur lequel on voudra insérer ou supprimer des lignes de texte est un bon exemple de listes linéaires. Les éléments sont alors les lignes de texte, et la liste le texte lui-même.

Les listes peuvent **se représenter physiquement** sous forme d'un double vecteur : le premier contenant **les éléments**, le second les pointeurs sur **les éléments** :



Une autre possibilité de représentation, surtout utilisable dans les langages de haut niveau tel le C, revient à exprimer l'élément de la liste comme un agrégat de 2 champs :

- « valeur » qui contient l'information.
- « suivant » qui est le pointeur sur l'élément suivant de la liste.

La structure de liste linéaire peut être améliorée de 2 manières différentes :

- par l'emploi d'un double chaînage afin d'obtenir une liste linéaire double.
- en constituant une liste circulaire où le dernier élément pointe sur le premier.

Ces nouvelles structures permettent de pallier certains inconvénients de la structure linéaire simple : **lecture des éléments dans les 2 sens** (listes chaînées linéaires doubles) , **diminution de l'importance accordée au premier élément** de la liste (listes circulaires).

#### 5.4.3.2 Les arborescences

Une **arborescence** (on dit aussi **arbre**) est un ensemble d'éléments organisés de façon **hiérarchique**.

Les arborescences permettent de représenter un très grand nombre de situations et de phénomènes.

**Décomposition** d'un programme en sous-programme, **arbre d'évaluation** d'un jeu de stratégie (échecs, dames, ...), syntaxe d'une expression arithmétique ou d'un langage de programmation, classements divers ....

Les arborescences sont ainsi utilisées dans de nombreux domaines de l'informatique :  
compilation, conception de systèmes d'exploitation, Intelligence Artificielle,  
architectures de base de données ....

On appelle arbre de type T, une structure de données de même type, que l'on dénomme **racine** et **d'une suite d'arbres de même type**, que les l'on dénomme **sous-arbres**.

A l'image d'un arbre généalogique, on appelle « **nœuds fils** » les nœuds issus d'un autre nœud de niveau hiérarchique supérieur, et, « **nœud père** », le nœud père d'un nœud fils.

Afin de pouvoir implanter cette structure, nous allons étudier une arborescence d'un type particulier : **l'arbre binaire**, qui se représente directement en machine, et auquel tout arbre peut se ramener.

Un arbre binaire est un arbre dont chaque nœud ne possède que 2 branches, et pour lequel on fait une différence entre le nœud fils gauche et le nœud fils droit.

L'arbre binaire se définit logiquement par les opérations suivantes :

- **Accès** , qui se différencie en trois fonctions
  - **Racine** , qui lit la racine d'un arbre.
  - **Droite**, qui lit la branche droite d'un arbre.
  - **Gauche** , qui lit la branche gauche d'un arbre.
- **Construction** : création d'un arbre binaire à partir de 2 sous arbres et d'une racine.
- **Test** : fonction vide qui détermine si un sous-arbre est vide ou non.

Il existe 2 représentations physiques possibles d'un arbre binaire :

- Utilisation de tableaux.

Implantée sous forme de tableau, la structure d'un arbre binaire se réduit à 3 vecteurs.  
Le vecteur des valeurs, qui porte la composante **significative** d'un nœud, le vecteur des pointeurs sur les fils gauches et le vecteur des pointeurs sur les fils droits.

- L'autre forme qui emploie la notion d'agrégat, est très utilisée en PASCAL.

Une importante caractéristique des arbres est de pouvoir être **parcourus**, c'est à dire qu'il est possible de se déplacer le long de cette arborescence dans un certain ordre et de traiter les valeurs des nœuds au fur et à mesure de ce parcours.

Les 3 parcours principaux que l'on peut réaliser sur un arbre se dénomment  
« **préordre** », « **postordre** » et « **inordre** » et s'expriment de manière très simple :

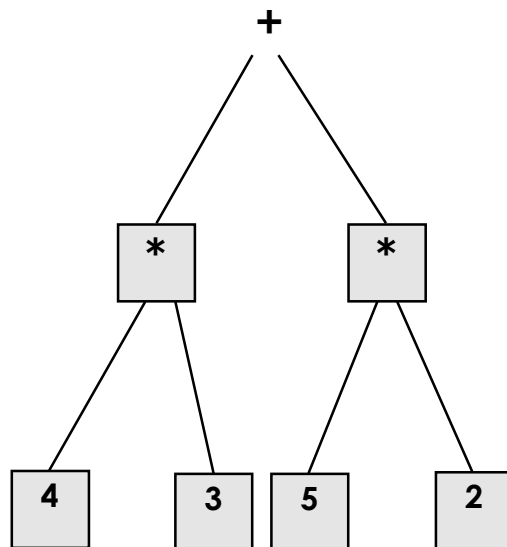
- **préordre** : traiter la racine d'abord, les fils ensuite.
- **postordre** : traiter les fils d'abord, la racine ensuite.
- **inordre** : traiter le fils gauche puis la racine, puis le fils droit.

En parcourant successivement une arborescence d'expression arithmétique en **préordre**, **postordre** et **inordre**, on obtient les notations :

**préordre** : + \* 4 3 \* 5 2 ( préfixée )

**postordre** : 4 3 \* 5 2 \* + ( postfixée )

**inordre** : 4 \* 3 + 5 \* 2 ( infixée )



Les arbres binaires ont de nombreuses applications en tant que tels. L'une d'entre elles, fort utile, **permet de trier des éléments** en créant un arbre binaire de recherche.

L'algorithme de tri revient à créer une arborescence en insérant **systématiquement** les nombres inférieurs à la racine dans le sous-arbre de gauche, et les nombres supérieurs à la racine dans le sous-arbre de droite.

## 5.5 CONCLUSION :

Nous avons ainsi passé en revue quelques-unes des structures de données les plus **classiques** en informatique.

Bien entendu, cette liste n'est pas exhaustive, des structures plus complexes apparaissent sans cesse, surtout dans les domaines de l'intelligence Artificielle et **Techniques Orientés Objets** ... mais elles reviennent toujours à une extension des structures que l'on vient de voir  
.....

## 6 LES FICHIERS

### 6.1 GENERALITES

Quand nous désirons conserver des données sur des références de produits, sur des clients, ou sur des mesures, nous aimerions quelquefois que ces données aient une existence propre, c'est-à-dire qu'elles existent en dehors de tout programme. Elles pourraient alors être utilisées par un autre programme, ou être complétées ou modifiées par le même programme lors d'une exécution ultérieure.

Les fichiers vont nous permettre de conserver des données d'un programme. Ces données pourront être traitées ultérieurement. Il existe plusieurs modes de conservation de données dans un fichier.

C'est un chapitre crucial pour le développement informatique. La finalité d'un programme est en effet d'établir un lien entre l'extérieur et l'espace de stockage des données.

Un premier critère qui différencie les deux catégories de fichiers, est le suivant : les données dans le fichier sont-elles organisées sous forme de lignes successives ?

- Si oui, cela veut dire que le fichier contient le même genre d'informations à chaque ligne (ex : un contact du carnet d'adresses) : on parle d'enregistrements. Ces enregistrements peuvent être codés sous forme ascii, on parlera de fichier « texte ».
  - Dans le cas contraire, les données ne sont qu'une suite d'octets qui n'a de sens que tous ensembles (ex : fichier son, fichier image, un exécutable). Ici on parlera de fichier « binaire ».
- On verra que même si les données sont organisées sous forme d'enregistrement, il est possible de les stocker dans un fichier binaire.

Dans le cas des fichiers textes, les enregistrements peuvent être structurés de deux façons différentes :

- Soit les informations qu'ils contiennent sont séparées par un délimiteur bien précis (ex : le format csv où le séparateur est un « ; »)
- Soit les informations sont de longueur fixe, donc toutes les lignes ont même longueur.

Exemple :

Structure n°1

```
Dejour;Adam;0123456789;adam.dejour@mail.fr
Proviste;Alain;0678912345;alain.proviste@mail.fr
```

Structure n°2

```
Dejour      Adam0123456789      adam.dejour@mail.fr
Proviste    Alain0678912345  alain.proviste@mail.fr
```

L'avantage de la structure n°1 est son faible encombrement, il n'y a aucun espace perdu. Par contre, lors de la lecture, après avoir récupéré une ligne il faut parcourir tous les caractères un par un pour différencier les informations.

La structure n°2 à l'inverse, perd de la place mais la récupération des différentes informations est rapide.

Un autre critère de classification des fichiers est sa méthode d'accès autrement dit comment le système est capable d'aller chercher les données dans le fichier.

On distingue :

- L'accès séquentiel :  
On ne peut accéder qu'à la donnée suivante celle qu'on vient de lire. On lit donc du début du fichier, ligne par ligne, en regardant si dans cette ligne l'information que l'on souhaite se trouve.
- L'accès direct :  
On peut accéder directement à l'information de son choix en précisant le numéro d'enregistrement. Il ne faut pas supprimer un enregistrement sous peine de renuméroter tous les enregistrements.
- L'accès séquentiel indexé :  
Les enregistrements sont repérés par une clé unique. Ceci permet d'accéder à l'enregistrement quel que soit sa place au sein du fichier.

On aura compris qu'un fichier, outre son nom, est caractérisé par :

- son type : texte ou binaire.
- sa méthode d'accès : séquentielle, relative, indexée.

### 6.1.1 Déclaration

Avant de pouvoir utiliser un fichier, il faut au préalable déclarer une variable de type fichier en précisant le type de données qu'il contient :

nomfic : FICHER de type\_élément

où

- nomfic est la variable représentant le fichier,
- type\_élément est le type des éléments du fichier.


### 6.1.2 Ouverture

Il faut ouvrir le fichier avant toute action sur ses données. En général, l'ouverture peut se faire en vue :

- d'une lecture seule : mode LECTURE (READ),
- d'une écriture seule : mode ECRITURE (WRITE) ne peut que rajouter de nouveaux enregistrements, en écrasant le contenu du fichier déjà existant,
- d'un ajout d'enregistrement : mode AJOUT (APPEND) on ne peut que rajouter de nouveaux enregistrements, en fin de fichier,
- ou enfin en vue de modifier des enregistrements : mode MIS\_A\_JOUR (UPDATE) réservé aux fichiers relatif et séquentiel indexé.

L'ajout d'un enregistrement peut donc s'effectuer de deux manières : soit en fin de fichier, soit en écrasant le fichier déjà existant à l'ouverture.

La syntaxe d'ouverture est la suivante :

	Auteur	Nom région	Formation	Date Mise à jour	Page 34 / 41
	DB	GRN 164	DWWM	04/10/2022	310-Cours_Algorithmie.docx

```
ouvrir("nom_du_fichier", nomfic, mode_ouverture)
```

où

- nom\_du\_fichier est le nom du fichier sur le support physique
- nomfic est la variable représentant le fichier lors de la déclaration
- mode\_ouverture est une des constantes vues plus haut.

### 6.1.3 Fermeture

Il faut fermer le fichier lorsque le travail est terminé.

Ceci afin de ne pas perdre des données. En effet un système de « buffer » en mémoire est utilisé pour optimiser la lecture / écriture dans les fichiers.

La syntaxe de fermeture est :

```
fermer(nomfic)
```

où

- nomfic est la variable représentant le fichier lors de la déclaration

## 6.2 LES FICHIERS SEQUENTIELS

On peut ouvrir un fichier séquentiel de trois manières différentes :

### 6.2.1 En lecture

La lecture d'un élément du fichier se fait de la manière suivante :

```
lire( nomfic , varlec )
```

où varlec représente une variable de type type\_élément.

Les données seront lues depuis la première donnée jusqu'à la dernière, dans cet ordre immuable.

La procédure lire() affecte les données du fichier à la variable varlec. Il faut avoir testé au préalable que le fichier ne soit pas à sa fin.

La fonction finfichier(nomfic) qui renvoie un booléen sert à cela :

**Si** finfichier ( nomfic ) **Alors** ...

Cette fonction teste si le dernier élément du fichier a été lu. Elle rend donc vrai quand il n'y a plus rien à lire dans le fichier.


Dans un fichier séquentiel ouvert en lecture, on ne peut pas rajouter de données ou en modifier.

### 6.2.2 En écriture ou en ajout

L'écriture d'un élément du fichier se fait de la manière suivante :

```
écrire ( nomfic , varecr )
```

où varecr représente une valeur de type type\_élément.

	Auteur	Nom région	Formation	Date Mise à jour	Page 35 / 41
	DB	GRN 164	DWWM	04/10/2022	310-Cours_Algorithmie.docx

La procédure écrire() sauvegarde dans le fichier la valeur représentée par valecr.

Remarque : au moment de l'ouverture :

- si le fichier n'existe pas il sera créé et les données seront écrites les unes derrière les autres dans l'ordre d'écriture.
- si le fichier existe au moment de l'ouverture deux cas se présente :
  - le fichier a été ouvert en écriture  
les données qu'il contient sont perdues (le fichier est vidé).
  - le fichier a été ouvert en ajout  
les données seront insérées en fin de fichier

Dans tous les cas, les ouvertures en lecture et écrire, sont exclusives sur une utilisation d'un fichier. Quand un fichier est ouvert en écriture ou ajout, nous pouvons écrire des valeurs dedans, mais si nous voulons les relire, il faut fermer le fichier, puis le rouvrir en lecture pour lire les valeurs qui s'y trouvent.

### 6.3 LES FICHIERS A ACCES DIRECT OU RELATIF

Les fichiers à accès direct permettent de faire des mises à jour des données quelques soient leurs places dans le fichier.

#### 6.3.1 En lecture et écriture

Dans les procédures de lecture et d'écriture, il faudra préciser le numéro d'enregistrement que l'on souhaite atteindre.

La syntaxe de ces procédures sera :

lire(nomfic, num\_enreg, varlec)

ecrire(nomfic, num\_enreg, varecr)

Dans le cas de l'écriture, si on est en fin de fichier, la donnée est ajoutée, sinon la donnée est mise à jour.

#### 6.3.2 En déplacement


Une fonction supplémentaire existe pour se déplacer dans le fichier sans être obligé de lire des données.

La syntaxe de ces procédures sera :

sedeplace(nomfic, num\_enreg)

### 6.4 LES FICHIERS SEQUENTIELS INDEXES

Un des gros problèmes des fichiers à accès direct est la gestion des enregistrements. Si l'on veut supprimer un enregistrement, les numéros des suivant vont être modifiés. Le plus simple dans ce cas, est de laisser les données mais de marquer l'enregistrement comme étant effacé par l'ajout d'un champ supplémentaire.

	Auteur	Nom région	Formation	Date Mise à jour	Page 36 / 41
	DB	GRN 164	DWWM	04/10/2022	310-Cours_Algorithmie.docx



Pour les fichiers séquentiels indexés, un champ unique (numéro de sécurité sociale, nom unique, ...) permet de trouver l'enregistrement correspondant.

Ce champ unique est une clef d'accès à l'enregistrement, c'est à dire que l'on peut retrouver l'enregistrement uniquement en connaissant cette clef quel que soit sa place dans le fichier.

On peut ouvrir un fichier indexé, le fermer, se positionner sur un élément, puis le lire, l'écrire ou le détruire.

La déclaration d'un fichier indexé se fait de la manière suivante :

```
ENREG : type_élément
      clef : TYPE_QUELCONQUE
      // le premier champ est la clef unique d'accès à la donnée
      ....
FIN_ENREG
```

nomfic : FICHER de type\_élément

où nomfic est la variable représentant le fichier, type\_élément est le type des éléments du fichier. Ce fichier n'est à priori pas indexé. C'est l'ouverture du fichier, à sa création qui en fera ou non un fichier indexé.

L'ouverture d'un fichier indexé se fait de la manière suivante :

ouvrir ( 'fichier.dat' , nomfic, indexé )

Ce fichier doit avoir été ouvert à sa création en mode indexé. Un fichier indexé peut être parcouru en mode séquentiel. Pour cela il suffit de l'ouvrir comme un fichier séquentiel, puis de lire séquentiellement ses éléments. Par contre l'écriture ne se fera qu'en mode indexé.

#### 6.4.1 Déplacement

La syntaxe de ces procédures sera :

cherche(nomfic, cle) où cle est la valeur recherchée dans le fichier

#### 6.4.2 Lecture et Ecriture

La syntaxe de ces procédures sera :

lire(nomfic, cle, varlec)

ecrire(nomfic, cle, varecr)

#### 6.4.3 suppression

La syntaxe de ces procédures sera :

supprime(nomfic, cle)