

Université Mohamed Seddik BENYAHIA – JIJEL –
Faculté des Sciences Exactes et Informatique
Département d'Informatique



PROGRAMMATION ET STRUCTURES DE DONNEES 2

Le Cours

Auteur : Dr. KARA Messaoud

Avril 2021

Table des matières

AVANT PROPOS	1
PARTIE 1– PROGRAMMATION MODULAIRE	2
1. Introduction	2
2. Les procédures	4
2.1- Définition d'une procédure	4
2.2- Déclaration d'une procédure	4
2.3- Appel d'une procédure	4
3. Les fonctions	5
3.1- Définition d'une fonction	5
3.2- Déclaration d'une fonction	6
3.3- Appel d'une fonction	6
4. Les variables globales et les variables locales	7
5. Le passage de paramètres	8
5.1- Le passage par valeur (par copie)	8
5.2- Le passage par adresse (par variable ou par référence)	9
6. Les fonctions et procédures récursives	9
6.1- Définition	9
6.2- La structure d'une fonction récursive	10
6.3- Exemple d'une fonction récursive – La factorielle	10
6.4- Comment la récursivité marche ?	11
6.5- Résolution récursive d'un problème	12
6.6- Exemple de procédure récursive – Les Tours de Hanoï	12
7. Synthèse : Procédures, Fonctions, Récursivité	14
PARTIE 2– STRUCTURES DYNAMIQUES	16
1. Introduction	16
2. Les enregistrements (Les structures)	16
Définition	16
Déclaration	16
♦ Accès aux champs d'un enregistrement	17
Cas des structures imbriquées	17
Tableaux d'enregistrements	17
3. Les pointeurs	19
2.1- Définition d'un pointeur	19
2.2- Déclaration d'un type pointeur	19

2.3- Actions sur les pointeurs	19
4. Les Listes Linéaires Chainées (LLC).....	20
3.1- Exemples d'introduction	20
3.2- Définition d'une liste	21
3.3- Déclaration d'une Liste Linéaire Chainée	21
3.4- Accès aux données d'une Liste Linéaire Chainée	22
3.5- Opérations sur les listes	22
3.5.1- Initialisation d'une Liste Linéaire Chainée.....	22
3.5.2- Insertion dans une Liste Linéaire Chainée.....	23
3.5.2.1- Insertion en tête de liste	23
3.5.2.2- Insertion au milieu de la liste	25
3.5.2.2.1- Insertion par position	26
3.5.2.2.2- Insertion dans une liste triée.....	27
3.5.2.3- Insertion à la fin de la liste	28
3.5.3- Consultation	29
3.5.3.1- Affichage des éléments de la liste	29
3.5.3.2- Calcul de la longueur de la liste.....	30
3.5.3.3- Recherche d'une valeur dans la liste.....	30
3.5.4- Modification	32
3.5.4.1- Suppression de la tête de la liste.....	32
3.5.4.2- Destruction de la liste	32
3.5.4.3- Fusion de deux listes triées	33
3.5.4.4- Eclatement d'une liste en deux listes selon le critère de parité	35
3.5.4.5- Inversion d'une liste.....	35
5. Les algorithmes récursifs sur les listes.....	36
4.1- Affichage récursif de la liste.....	36
4.2- Affichage inversé de la liste.....	37
4.3- Recherche d'une valeur dans la liste	37
4.4- Calcul de la longueur d'une liste.....	37
4.5- Calcul du nombre d'occurrences d'une valeur donnée	38
4.6- Insertion par position	38
4.7- Insertion dans une liste triée.....	39
4.8- Suppression de toutes les occurrences d'une valeur donnée	39
4.9- Destruction de la liste	39
6. Listes linéaires chaînées particulières.....	40
5.1- Les Listes bidirectionnelles (doublement chaînées).....	40
5.2- Les Listes circulaires (anneaux)	42

7. Synthèse sur les listes	42
8. PILES	43
7.1- Définition, principe, domaines d'application	43
7.2- Exemple	43
7.3- Modèle	44
7.4- Implémentation	44
7.4.1- Implémentation d'une Pile en utilisant une Liste	45
7.4.2- Implémentation d'une Pile en utilisant un tableau	46
7.5- Synthèse	47
9. FILES	48
8.1- Définition, principe, domaine d'application	48
8.2- Exemple	48
8.3- Modèle	49
8.4- Implémentation	49
8.4.1- Implémentation d'une File en utilisant une Liste	49
8.4.2- Implémentation d'une File en utilisant un tableau	52
8.5- File avec priorité	54
8.6- Synthèse	55
10. Les arbres (introduction)	56
Définition	56
PARTIE 3– COURS SUR LE LANGAGE C	57
PARTIE 4– REFERENCES BIBLIOGRAPHIQUES	57

AVANT PROPOS

Ce cours « Programmation et Structures de Données 2 » est destiné aux étudiants en 1^{ière} année de licence au département MI (Département d'Enseignement Fondamental en Mathématiques et Informatique) de l'université de Jijel.

Les objectifs de ce cours :

- 1- Acquérir les notions de base de la programmation modulaire et sur la récursivité
- 2- Comprendre les structures de données dynamiques et leur utilité.

Le document comprend essentiellement deux parties :

- 1- La première partie est consacrée aux cours sur la programmation modulaire (procédures, fonctions, récursivité),
- 2- La deuxième partie traite les structures de données dynamiques (Listes linéaires chaînées, Piles, Files et une brève introduction sur les arbres).

PARTIE 1– PROGRAMMATION MODULAIRE

1. Introduction

Un programme (algorithme) écrit en un seul bloc devient difficile à comprendre dès qu'il dépasse un nombre de lignes. Pour éviter ce problème, la programmation structurée offre deux outils : les procédures et les fonctions. Elles permettent de décomposer le problème en sous problèmes plus faciles à écrire, comprendre et corriger si nécessaire.

La recherche de la solution du problème sera d'identifier au niveau du problème posé un ou plusieurs sous problèmes à résoudre séparément. Chaque sous problème est à son tour traité comme un nouveau problème.

Ainsi la solution d'un problème est décrite par un algorithme principal qui définit la méthode générale de résolution et des sous algorithmes. Un sous algorithme est un algorithme qui décrit la solution d'un sous problème.

Chaque procédure ou fonction est utilisée pour résoudre un sous problème du problème global posé. Elles sont définies dans la partie déclarative d'un algorithme et ont la même structure d'un algorithme. Elles sont constituées d'un entête, de déclarations de variables et d'un corps.

Les objectifs de l'utilisation des procédures et des fonctions sont :

1. **La simplification** : rendre l'écriture des algorithmes plus simple car chaque sous algorithme est traité séparément.
2. **La réutilisation** : Ecrire une procédure (ou une fonction) une seule fois puis la réutiliser plusieurs fois au lieu de la réécrire plusieurs fois.
3. **Faciliter la maintenance et l'évolution des algorithmes (programmes)** : En cas d'erreurs ou pour améliorer un algorithme (programme), on ne modifie qu'une partie de l'algorithme sans toucher les autres parties.

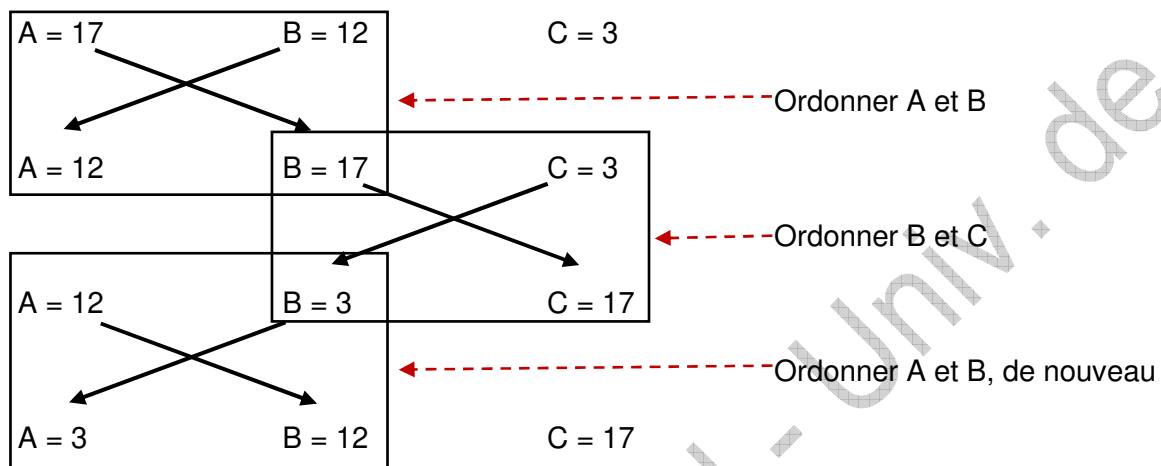
Rappel de la structure générale d'un algorithme

Algorithme	nomAlgorithme
Const	Liste_des_constantes
Type	Liste_des_types
Var	Liste_des_variables
	Liste_des_procédures_et_fonctions
Début	
	Liste des instructions // Corps de l'algorithme (principal)
Fin	

Exemple d'introduction

En s'inspirant de l'algorithme de tri à bulles, écrire un algorithme qui permet d'ordonner dans l'ordre croissant trois entiers A, B et C.

Exemple numérique : Si $A = 17$ et $B = 12$ et $C = 3$ alors A, B et C ne sont pas ordonnés dans l'ordre croissant. On commence par ordonner A et B, puis ordonner B et C et enfin ordonner A et B de nouveau. On a, en fait, trois fois la même opération qui consiste à ordonner deux entiers.



A la fin, on obtient, $A = 3$, $B = 12$ et $C = 17$.

L'algorithme suivant permet de retranscrire les opérations expliquées ci-dessus :

Algorithme TriCroissantVersion1

Var A, B, C, Tmp : **Entier**

Début

```

Ecrire("Donner trois entiers A, B et C")
Lire( A, B, C )
// Ordonner A et B.
Si A > B Alors           // Si A et B ne sont pas dans l'ordre croissant.
    Tmp ← A
    A ← B
    B ← Tmp                // Permuter les valeurs de A et B.
FSi
Si B > C Alors           // Ordonner B et C.
    Tmp ← B
    B ← C
    C ← Tmp
FSi
Si A > B Alors           // Ordonner A et B de nouveau.
    Tmp ← A

```

```

    A ← B
    B ← Tmp
  FSi
  Ecrire("Les trois entiers dans l'ordre : ", A, B, C)
Fin

```

2. Les procédures

2.1- Définition d'une procédure

Une procédure décrit une suite d'actions bien identifiée, qui s'exerce sur un ou plusieurs paramètres (arguments) à travers lesquels la procédure reçoit les données et transmet les résultats des traitements qu'elle réalise.

2.2- Déclaration d'une procédure

Une procédure est déclarée selon la syntaxe suivante :

Procédure nomProcédure([Liste des paramètres formels])	// L'entête
[Var Liste de variables locales]	// Déclarations des variables locales, si nécessaire.
Début	
Liste des instructions	// Corps de la procédure
Fin	

- [Liste des paramètres formels] : c'est une suite de déclarations de variables de la forme :
 nomParamètre1 : Type1 ; nomParamètre2 : Type2 ; ... ; nomParamètreN : TypeN

Remarque : Si un paramètre est modifié par la procédure, il est précédé par le mot clé **VAR**.

Exemple : Procédure qui permet d'ordonner dans l'ordre croissant deux entiers A et B.

Procédure Ordonner(**Var** A : Entier ; **Var** B : Entier)

Var Tmp : Entier

Début

```

    Si A > B Alors // Si les valeurs de A et B ne sont pas dans l'ordre croissant
    |   Tmp ← A
    |   A ← B
    |   B ← Tmp
    |   } // Permuter les valeurs de A et B
  FSi
Fin

```

2.3- Appel d'une procédure

Dans un algorithme, une procédure joue le rôle d'une instruction. Elle est appelée selon la syntaxe suivante : **nomProcédure ([liste des paramètres effectifs])**

L'appel de la procédure provoque son exécution avec les paramètres effectifs.

La liste des paramètres effectifs doit avoir le même nombre, même ordre et les mêmes types que les paramètres formels (Exception : On peut mettre un Entier à la place d'un Réel).

Exemple : Pour appeler la procédure Ordonner sur deux variables X et Y on écrit l'instruction suivante : **Ordonner**(X, Y)

L'algorithme TriCroissantVersion1, donné en introduction, peut être réécrit comme suit :

Algorithme TriCroissantVersion2

Var A, B, C : Entier

Procédure Ordonner(**Var** A : Entier ; **Var** B : Entier)

Var Tmp : Entier

Début

```

    Si A > B Alors           // Si les valeurs de A et B ne sont pas dans l'ordre croissant
    |
    |   Tmp ← A
    |   A ← B
    |   B ← Tmp
    | } // Permuter les valeurs de A et B
    FSi
  
```

Fin

Début

```

    Ecrire("Donner trois entiers A, B et C")
    Lire( A, B, C )
    Ordonner( A, B )      // 1er appel de procédure
    Ordonner( B, C )      // 2e appel de procédure
    Ordonner( A, B )      // 3e appel de procédure
    Ecrire("Les trois entiers dans l'ordre : ", A, B, C)
  
```

Fin

Dans cette nouvelle version de l'algorithme, nous avons écrit la procédure Ordonner une seule fois et nous l'avons utilisé (appelé) trois fois.

3. Les fonctions

3.1- Définition d'une fonction

Une fonction peut être considérée comme un opérateur non primitif (complexe) créé à l'initiative du programmeur. Elle décrit une suite d'actions bien identifiées qui s'exerce sur un ou plusieurs paramètres et réalise le calcul d'une expression et par conséquent elle retourne un résultat unique.

Exemple : La fonction puissance $X^n = X * X * \dots * X$ est un opérateur qui fait N opérations de multiplication. Pareil pour la factorielle : $N! = N * (N - 1) * \dots * 2 * 1$

3.2- Déclaration d'une fonction

Une fonction est déclarée selon la syntaxe suivante :

Fonction nomFonction([Liste des paramètres formels]) : TypeRésultat	// L'entête
[Var Liste de variables locales]	// Déclarations des variables locales, si nécessaire.
Début	} // Corps de la fonction
Liste des instructions	
nomFonction ← résultat	
Fin	

Remarque : La dernière instruction d'une fonction est une affectation. Le nom de la fonction (nomFonction) reçoit le résultat calculé.

Exemple : Ci-dessous une fonction (Facto) qui calcule la factorielle d'un entier positif N.

Fonction Facto(N : Entier) : Entier

Var I, F : Entier

Début

```
F ← 1
Pour I ← 2 à N Faire      // OU Pour I ← 1 à N Faire
    F ← F * I
Fpour
Facto ← F
```

Fin

3.3- Appel d'une fonction

- ✓ Une fonction est utilisée (appelée) dans une expression. Son appel provoque son exécution avec les paramètres effectifs définis dans l'appel et renvoie un résultat.
- ✓ Comme pour une procédure, la liste des paramètres effectifs d'une fonction doit avoir le même nombre, le même ordre et les mêmes types que les paramètres formels.

Exemple :

Pour appeler la fonction facto sur un entier X, on écrit par exemple : $Y \leftarrow \text{Facto}(X)$

Remarque : On peut trouver un appel de fonction :

- ✓ dans une affectation (dans un calcul) : Exemple : $T \leftarrow 1 + \text{Facto}(X) / Y + Z$
- ✓ à la place d'un paramètre non précédé du mot clé Var,
- ✓ dans une opération d'écriture : Exemple : $\text{Ecrire}(\text{Facto}(X))$.

En fait, on peut mettre un appel de fonction, là où on peut utiliser une valeur constante.

Exemple : En utilisant la fonction Facto, on peut écrire un algorithme qui permet de lire un entier positif N et d'afficher sa factorielle comme suit :

Algorithme Factorielle

Var N : Entier

Fonction Facto(N : Entier) : Entier

Var I, F : Entier

Début

F ← 1

Pour I ← 2 à N Faire

F ← F * I

Fpour

Facto ← F

Fin

Début

Répéter

Ecrire("Donner un entier positif ($N \geq 0$) ")

Lire(N)

Jusqu'à $N \geq 0$

Ecrire(N, " != ", Facto(N)) // Appel de fonction

Fin

4. Les variables globales et les variables locales

- ✓ Une variable globale est déclarée dans l'algorithme principal et peut être utilisée dans une ou plusieurs procédures et/ou fonctions ainsi que dans l'algorithme principal.
- ✓ Une variable locale est déclarée dans une procédure ou une fonction et ne peut être utilisée que dans cette procédure (ou fonction).

Exemple : Reprenons l'algorithme TriCroissantVersion2. Ajoutons une variable globale Cpt.

Cette variable Cpt est initialisée par l'algorithme principal et est incrémentée par la procédure Ordonner à chaque fois qu'elle est appelée.

Algorithme TriCroissantVersion3

Var A, B, C, **Cpt** : Entier

Procédure Ordonner(Var A : Entier ; Var B : Entier)

Var Tmp : Entier

Début

Cpt ← Cpt + 1 // Cpt est une variable globale incrémentée à chaque appel.

Si A > B Alors

Tmp ← A

A ← B

B ← Tmp

FSi

Fin

Début // algorithme principal

```
Cpt ← 0 // Nombre d'appels est initialisé à zéro.  
Ecrire("Donner trois entiers A, B et C")  
Lire( A, B, C )  
Ordonner( A, B)      // 1er appel de procédure  
Ecrire(" La procédure Ordonner a été appelée ", Cpt, " fois")  
Ordonner( B, C)      // 2e appel de procédure  
Ecrire(" La procédure Ordonner a été appelée ", Cpt, " fois")  
Ordonner( A, B)      // 3e appel de procédure  
Ecrire(" La procédure Ordonner a été appelée ", Cpt, " fois")  
Ecrire("Les trois entiers dans l'ordre : ", A, B, C)
```

Fin

Remarques : ①-Toutes les variables A, B, C et Cpt sont globales.

②- La variable **Tmp** est locale à la procédure **Ordonner**. Les variables **F** et **I** sont locales à la fonction **Facto**.

③- Les paramètres d'une procédure (ou d'une fonction) sont considérées comme des variables locales. Il n'est pas possible de récupérer leurs valeurs à l'extérieur de la procédure (ou de la fonction).

④- Si une variable globale et une variable locale ont le même nom (même si elles sont de deux types différents), la variable globale devient inaccessible dans la procédure ou la fonction où la variable locale est déclarée.

Pour l'algorithme TriCroissantVersion3, la procédure **Ordonner** utilise deux paramètres **A** et **B** et l'algorithme principal utilise deux variables globales **A** et **B** → les deux variables globales **A**, **B** ne sont plus accessibles dans la procédure **Ordonner**. Par contre les deux variables globales **C** et **Cpt** restent accessibles dans la procédure **Ordonner**.

5. Le passage de paramètres

Il existe deux types (ou modes) de passage de paramètres : Passage par valeur et passage par adresse.

5.1- Le passage par valeur (par copie)

Les paramètres non précédés par le mot clé **VAR**, sont passés à la procédure (ou à la fonction) par valeur. C'est-à-dire, la valeur du paramètre est copiée dans une variable locale sans toucher la variable donnée dans l'appel. C'est cette variable locale qui est utilisée dans la procédure (ou la fonction). Aucune modification de la variable locale dans la procédure (ou la fonction) ne modifiera la valeur de la variable passée en paramètre.

5.2- Le passage par adresse (par variable ou par référence)

Les paramètres précédés par le mot clé **VAR**, sont passés à la procédure par adresse. La procédure travaille directement sur la variable passée en paramètre. Toutes les modifications sur le paramètre seront conservées par la variable passée en paramètre.

Remarques : ①- Tous les paramètres d'une fonction sont passés par valeur.

②- Une fonction retourne un seul résultat à travers le nom de la fonction et non pas à travers les paramètres.

③- Une procédure peut retourner aucun résultat (si la procédure n'utilise que des variables globales ou elle ne fait que de l'affichage), un seul résultat ou plusieurs résultats et cela à travers les paramètres passés par adresse (précédés par le mot clé **VAR**).

6. Les fonctions et procédures récursives

6.1- Définition

Une procédure (ou fonction) est dite récursive si elle s'appelle elle-même. C'est-à-dire dans son corps (liste des instructions) on trouve un appel à la procédure (ou fonction) elle-même.

- L'idée : en supposant qu'on a la solution pour le cas d'ordre (N-1), est ce qu'on pourrait avoir la solution pour le cas d'ordre N ?

- La récursivité est un moyen naturel de résolution de certains problèmes où la version récursive est plus simple à trouver que la version itérative (Exemple : Tours de Hanoï).

- Ce type de programmation permet de réaliser des fonctions définies à partir de relations de récurrence comme :

- La puissance : $X^n = X * X^{n-1}$ et $X^0 = 1$
- La factorielle : $N! = N * (N-1)!$ et $1! = 1$ et $0! = 1$
- Plus Grand Commun Diviseur de deux entiers strictement positifs A et B :
$$\begin{cases} \text{PGCD}(A, B) = \text{PGCD}(B, A \text{ Mod } B) & \text{si } B \neq 0 \\ \text{PGCD}(A, B) = A & \text{si } B = 0 \end{cases}$$
- Suite de Fibonacci :
$$\begin{cases} U_n = U_{n-1} + U_{n-2} & \text{Si } n \geq 2 \\ U_0 = 0 \text{ et } U_1 = 1 & \text{Si } n < 2 \end{cases}$$

6.2- La structure d'une fonction récursive

Une fonction récursive à la structure générale suivante :

Fonction nomFonctRecursive(Parametre1 : Type1 ; ... ; ParametreN :typeN) : TypeRésultat

/* Variables locales s'il y a besoin */

Début

Si condition **Alors** /* Condition d'arrêt */

 nomFonctRecursive ← Résultat /* Cas élémentaire (trivial ou particulier) */

Sinon

 nomFonctRecursive ← ... nomFonctRecursive(p1, ..., pN) ... /* Cas général */

FSi

Fin

Remarques : ①- L'appel d'une fonction (ou procédure) à l'intérieur d'elle-même (dans son corps) est dit appel récursif.

②- Un appel récursif doit obligatoirement être dans une instruction conditionnelle (Si ... Sinon ...). Dans le cas contraire, la récursivité est sans fin. C'est-à-dire, on est en présence d'une boucle infinie.

③- Une procédure récursive a la même structure qu'une fonction récursive. C'est-à-dire, elle doit contenir au moins une structure conditionnelle (Si ... Sinon ...) pour traiter les deux cas : cas particulier et cas général. L'appel récursif de la procédure se fait par une instruction et n'est pas dans une expression.

6.3- Exemple d'une fonction récursive – La factorielle

La fonction suivante calcule la factorielle d'un entier positif N. En se basant sur la relation de récurrence $N! = N * (N-1)!$ et en sachant que : $1! = 1$ et $0! = 1$.

Fonction FactoRec(N : Entier) : Entier

Début

Si (N = 1) OU (N = 0) **Alors**

 FactoRec ← 1

Sinon

 FactoRec ← N * FactoRec(N - 1) /* appel récursif */

FSi

Fin

On peut réécrire l'algorithme qui affiche la factorielle d'un entier positif comme suit :

Algorithme Factorielle

Var N : Entier

Fonction FactoRec(N : Entier) : Entier

Début

```
Si (N = 1) OU (N = 0) Alors
    FactoRec ← 1
Sinon
    FactoRec ← N * FactoRec( N - 1 )    /* appel récursif */
FSi
```

Fin

Début

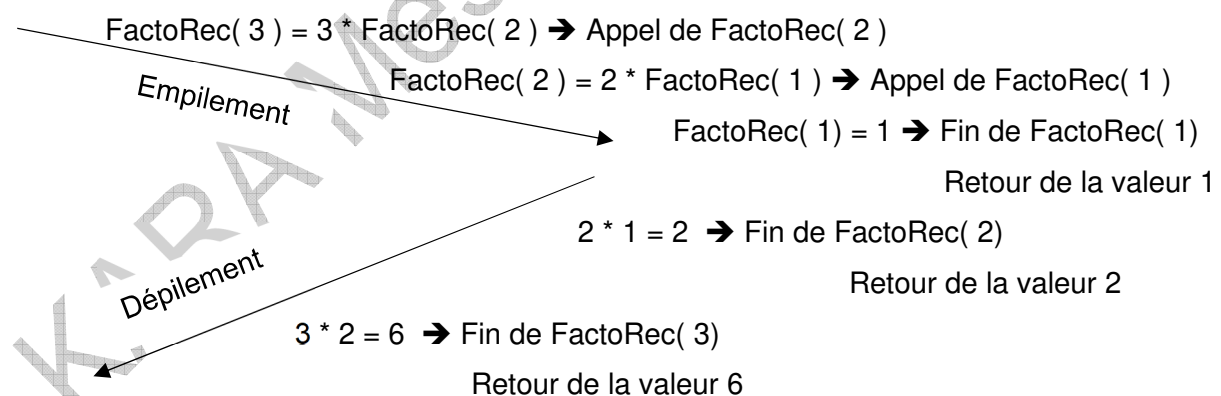
```
Répéter
    Ecrire("Donner un entier positif ( N ≥ 0 ) ")
    Lire( N )
Jusqu'à N ≥ 0
Ecrire( N, " ! = ", FactoRec( N ) )
```

Fin

6.4- Comment la récursivité marche ?

Si l'utilisateur a saisi 3 pour la valeur de N alors pour calculer et afficher le message **3 ! = 6**, l'algorithme fait comme suit :

Appel de FactoRec(3)



Affichage à l'écran du message **3 ! = 6**

Remarques :

Pour pouvoir gérer ces appels, le système dispose d'une **pile d'appels (pile d'exécution)**. Pour chaque appel de fonction (ou de procédure), sont sauvegardées dans cette pile les valeurs des paramètres donnés dans l'appel, les valeurs des variables locales et l'adresse de retour.

A chaque fois qu'une fonction (ou une procédure) se termine, les informations sauvegardées lors de son appel sont supprimées de la pile.

La pile d'appels a une taille limitée, ce qui implique que la récursion ne doit pas être de très grande taille. Sinon, la pile devient pleine et la récursion ne donne pas de résultat !

6.5- Résolution récursive d'un problème

Pour créer une fonction (procédure) récursive, il faut :

1. Décomposer le problème initial en un ou plusieurs sous-problèmes du même type. On résout les sous-problèmes par des appels récursifs.
2. Les sous-problèmes doivent être de taille plus petite que le problème initial. C'est-à-dire que la taille doit diminuer.
3. La décomposition doit en fin de compte arriver à un cas élémentaire qui n'est pas décomposé en sous-problèmes (Condition d'arrêt). On obtient directement un résultat.

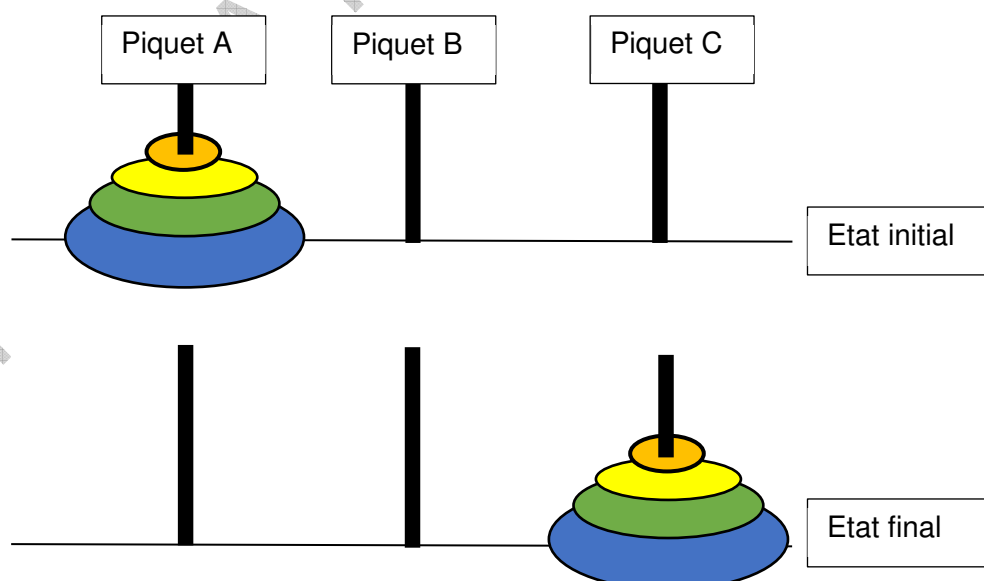
Remarques : L'intérêt d'écrire des fonctions (ou procédures) récursives réside dans une meilleure lisibilité. L'inconvénient est l'accroissement de la taille de la pile d'appels lors de son utilisation et des coûts prohibitifs lors d'une utilisation non optimisée (Exemple de la suite de Fibonacci si programmation directe).

6.6- Exemple de procédure récursive – Les Tours de Hanoï

C'est un jeu. On dispose de trois piquets (**A**, **B** et **C**) et d'un certain nombre de disques empilés sur le piquet **A**. On désire déplacer la totalité des disques sur le piquet **C** en utilisant le piquet **B** comme intermédiaire et en respectant les deux règles suivantes :

- ①- On déplace un disque à la fois.
- ②- Les disques doivent être disposés du plus petit au plus grand sur chaque piquet.

Ecrire une procédure récursive qui permet de modéliser ce problème ainsi que l'algorithme principal permettant de l'utiliser.



Analyse :

Quel est le cas particulier ?

Réponse : Cas où $N = 1$.

Si on a un seul disque, alors il suffit de le déplacer directement du piquet A vers le piquet C.

Quel est le cas général ?

Réponse : Cas où $N > 1$.

Si on a plusieurs disques alors :

1. On déplace les $(N - 1)$ disques du piquet A vers le piquet B en utilisant le piquet C comme intermédiaire.
2. On déplace le dernier disque du piquet A directement vers le piquet C.
3. Enfin, on déplace les $(N - 1)$ disques du piquet B vers le piquet C en utilisant le piquet A comme intermédiaire.

Solution :

Algorithme ToursDeHanoi

Var N : Entier

Procédure déplacer(N : Entier ; T1 : Entier ; T2 : Entier ; T3 : Entier)

/* N : Le nombre de disques à déplacer

T1 : Numéro de la tour Source

T2 : Numéro de la tour Intermédiaire

T3 : Numéro de la tour Destination */

Début

Si $N = 1$ **Alors**

Ecrire ("De la tour ", T1, " à la tour ", T3)

Sinon

déplacer($N - 1$, T1 , T3 , T2)

déplacer(1 , T1 , T2 , T3) // OU Ecrire ("De la tour ", T1, " à la tour ", T3)

déplacer($N - 1$, T2 , T1 , T3)

Fsi

Fin

Début

Répéter

Ecrire("Donner le nombre de disques ($N > 0$) ")

Lire(N)

Jusqu'à $N > 0$

déplacer(N, 1, 2, 3) // Commencer le déplacement des disques

Fin

Remarque :

Les paramètres T1, T2, T3 peuvent être de type Caractère.

Dans ce cas l'entête de la procédure devient :

Procédure déplacer(N : Entier ; T1 : Caractère ; T2 : Caractère ; T3 : Caractère)

L'appel dans le programme principal devient :

déplacer(N, 'A', 'B', 'C')

7. Synthèse : Procédures, Fonctions, Récursivité

- ✓ Les procédures et fonctions (sous-algorithmes) sont créées notamment pour :
 - Faciliter la résolution des problèmes en les découpant en petits problèmes faciles à résoudre,
 - Réutiliser les mêmes opérations au lieu de les réécrire à chaque fois,
 - Faciliter la modification et l'amélioration des algorithmes (des programmes) en modifiant qu'une seule partie de l'algorithme global.
- ✓ Les procédures et fonctions sont définies dans la partie déclarations d'un algorithme juste en dessous des variables.
- ✓ Les procédures sont utilisées pour renommer une suite d'actions.
- ✓ Une procédure est utilisée (appelée) comme une instruction simple.
- ✓ Une fonction est utilisée pour faire un calcul.
- ✓ Une fonction est utilisée comme un opérateur.
- ✓ Les paramètres passés à une procédure sont passés par Valeur (Le paramètre n'est pas modifié) ou par Adresse (Le paramètre peut être modifié).
- ✓ Les paramètres passés à une fonction sont tous passés par Valeur.
- ✓ Les variables globales sont définies dans l'algorithme principal et sont reconnues partout. Dans toutes les procédures et fonctions et dans l'algorithme principal.
- ✓ Une procédure peut retourner zéro, un ou plusieurs résultats à travers le passage des paramètres par Adresse.
- ✓ Une fonction retourne un et un seul résultat à travers le nom de la fonction. En effet la dernière instruction de la fonction doit être une affectation du résultat au nom de la fonction.
- ✓ La récursivité est une technique qui permet de définir une fonction (ou procédure) par l'appel à la fonction (ou la procédure) elle-même. Pour ce faire, il faut définir une relation de récurrence entre les paramètres de la fonction (ou de la procédure) et un cas trivial (particulier) pour que les appels récursifs s'arrêtent.
- ✓ La récursivité permet d'écrire des algorithmes très facilement à des problèmes difficiles à résoudre par des algorithmes itératifs (Exemple : Tours de Hanoï).

- ✓ La version itérative est toujours préférable par rapport à la solution récursive, car cette dernière consomme plus d'espace mémoire. En effet, il faut sauvegarder toutes les données (paramètres et variables locales) de tous les appels récursifs dans une pile nommée « pile des appels » ou « pile d'exécution ».

Dr. KARA Messaoud - Univ. de Jijel

PARTIE 2– STRUCTURES DYNAMIQUES

1. Introduction

Dans cette partie nous nous intéressons aux structures dynamiques ; essentiellement les listes linéaires chaînées. Puis nous aborderons les types abstraits de piles et de files. Nous aborderons d'une manière introductive le type arbre.

2. Les enregistrements (Les structures)

Définition

Un enregistrement (structure) est une structure de données permettant de regrouper dans une seule entité un ensemble de données de types différents associées à un même et seul objet.

Un enregistrement (appelé aussi structure ou article) est constitué de composants appelés champs.

Chaque champ est identifié par un nom qui permet d'y accéder directement et un type.

Le type d'un champ est quelconque : simple ou structuré.

Déclaration

Type Nom_enregistrement = **Enregistrement**

Nom_champ1 : Type1

Nom_champ2 : Type2

...

Nom_champN : TypeN

Fin

- ✓ Nom_champ1, Nom_champ2, ..., Nom_champN : Sont les identifiants des champs de l'enregistrement.
- ✓ Type1, Type2, ..., TypeN : Sont les types associés aux champs.
- ✓ Une fois le type de l'enregistrement est défini, on peut déclarer des variables de ce type :

Var Nom_variable : Nom_enregistrement

Exemple : Les informations concernant un étudiant : nom, prénom, age, sexe, Moyenne du BAC peuvent être représentées à l'aide d'un enregistrement comme suit :

Type Etudiant = **Enregistrement**

Nom : Chaîne

Prénom : Chaîne

Age : Entier

Sexe : Caractère // 'M' : Masculin, 'F' : Féminin
MoyenneBac : Réel

Fin

Var Etudiant1, Etudiant2 : Etudiant // Deux structures (variables) de type Etudiant
e1, e2, e3 : Etudiant // Trois structures (variables) de type Etudiant

- ✓ Un enregistrement peut être représenté par un ensemble de cases. Ces cases peuvent être des tailles différentes, car les types d'un enregistrement ne sont pas forcément les mêmes comme pour un tableau.

	Nom	Prenom	Age	Sexe	MoyenneBAC
Etudiant1	"LAKAB"	"Ism"	19	'M'	12.5

Accès aux champs d'un enregistrement

On accède à une information en précisant le nom de la variable de type enregistrement suivie du nom du champ séparé par un point(.) :

Nom_variable . Nom_champ

Exemple : Pour modifier l'âge de l'étudiant 1 en utilisant l'affectation on doit écrire :

Etudiant1.Age ← 20

- ✓ Le point qui figure dans la syntaxe indique le chemin d'accès : On accède d'abord à la variable Etudiant1 puis on sélectionne le champ Age (ou la partie Age).

Remarque : L'accès aux champs se fait par les noms des champs, donc, l'ordre de déclaration de ces champs n'est pas important.

Cas des structures imbriquées

Un enregistrement peut être imbriqué dans une structure de type tableau ou enregistrement, comme il peut avoir des champs de type structuré quelconque. La notation utilisée pour sélectionner les champs reste la même (L'utilisation du point).

Tableaux d'enregistrements

Il est possible de déclarer un tableau dont les éléments sont de type enregistrement. On définit d'abord le type enregistrement, puis on déclare un tableau dont les éléments sont de ce type d'enregistrement.

Type Nom_enregistrement = **Enregistrement**

Nom_champ1 : Type1
Nom_champ2 : Type2
...
Nom_champN : TypeN

Fin

Var Nom_tableau : **Tableau** [1 .. N] de Nom_enregistrement

Accès aux éléments

On accède d'abord à une case tableaux, en utilisant les crochets [], puis on accède au champ en utilisant le point (.).

Pour sélectionner le deuxième champ du troisième élément du tableau on utilise la syntaxe :

Nom_tableau[3] . Nom_champ2

Exemple : Pour déclarer un tableau d'enregistrements pour manipuler les informations de 100 étudiants, on doit écrire :

Type Etudiant = **Enregistrement**

Nom : Chaîne
Prenom : Chaîne
Age : Entier
Sexe : Caractère // 'M' : Masculin, 'F' : Féminin
MoyenneBac : Réel

Fin

Var Tab : Tableau[1 .. 100] d'Etudiant // Tableau de 100 enregistrements.

Pour modifier les champs de l'étudiant 2, on peut écrire, par exemple :

Tab[2] . Nom ← "Mohamed"
Tab[2] . Prenom ← "Lakabahou"
Tab[2] . Age ← "18"
Tab[2] . Sexe ← 'M'
Tab[2] . MoyenneBac ← 11.7

Manipulation des enregistrements :

L'écriture (La modification)

Lire (Tab[2] . Age)
Tab[3] . MoyenneBAC ← 13

La lecture (La consultation)

X ← Tab[5] . Prenom
Si (Tab[2] . MoyenneBAC < 12) alors // Comparaison
| Ecrire("Admis en MI")
FSi

Remarques

Toute opération sur les enregistrements doit être effectuée par un algorithme : La lecture, l'écriture, la comparaison ne peuvent se faire globalement, il faut lire, écrire ou comparer chaque champ individuellement (un par un).

Une exception : On peut affecter une variable de type enregistrement à une autre variable du même type.

Exemple : **Var** e1, e2 : Etudiant

On peut écrire indifféremment : $e2 \leftarrow e1$ ou champ par champ :

$$\left\{ \begin{array}{l} e2 . \text{Nom} \leftarrow e1 . \text{Nom} \\ e2 . \text{Prenom} \leftarrow e1 . \text{Prenom} \\ e2 . \text{Age} \leftarrow e1 . \text{Age} \\ e2 . \text{Sexe} \leftarrow e1 . \text{Sexe} \\ e2 . \text{MoyenneBac} \leftarrow e1 . \text{MoyenneBac} \end{array} \right.$$

3. Les pointeurs

2.1- Définition d'un pointeur

Un pointeur est une variable qui contient l'adresse mémoire (notée @) d'une autre variable (ou information) stockée en mémoire centrale (RAM).

2.2- Déclaration d'un type pointeur

TYPE nomTypePointeur = ^Type_de_base

Exemple : déclaration d'un type pointeur sur un entier

Type PTREntier = ^Entier

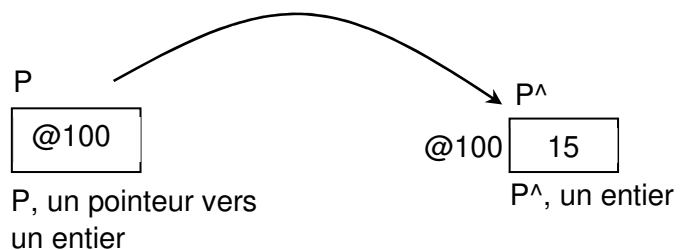
Var P : PTREntier

- P est une variable de type pointeur sur un entier. Elle contient l'adresse mémoire d'un entier.
- P^{\wedge} est une variable entière pointée par le pointeur P.

Exemple : Si on suppose que la variable P (Le pointeur P) est rangée en mémoire centrale (RAM) à l'adresse @020 et que le pointeur contient l'adresse @100 alors $P = @100$ et $P^{\wedge} = 15$.

	RAM
@000	...
P @020	@100
	...
@100	15
	...
@200	

Schématiquement, on représente ces informations comme suit :



2.3- Actions sur les pointeurs

Initialisation

Pour initialiser un pointeur P, on doit lui affecter la valeur **NIL (Not In List)**. NIL est une constante qui représente une adresse particulière qui ne pointe vers aucune donnée. On écrit alors $P \leftarrow \text{NIL}$.

Allocation de l'espace mémoire

Pour réserver de l'espace mémoire pour la variable pointée, il faut utiliser la procédure **ALLouer**. **Allouer(P)** : réserve un espace mémoire pour une donnée pointée par le pointeur P.

Libération de l'espace mémoire

Pour libérer l'espace mémoire (qui a été précédemment réservé avec la procédure Allouer), il faut utiliser la procédure **LIBERER**. **Liberer(P)** : permet de supprimer la donnée pointée par le pointeur P. Cette procédure, ne supprime pas l'adresse contenue dans le pointeur P. On doit lui affecter la valeur NIL pour casser (rompre) la liaison entre le pointeur P et l'ancienne donnée stockée en mémoire.

Remarques

- ①- Un pointeur a un type de données spécifique. Deux pointeurs de types différents ne peuvent pas être affectés l'un à l'autre.
- ②- La déclaration d'une variable pointeur réserve (d'une manière statique) l'espace mémoire nécessaire pour le stockage d'une adresse mémoire, mais ne réserve aucune mémoire pour la variable pointée (la donnée pointée).

4. Les Listes Linéaires Chainées (LLC)

3.1- Exemples d'introduction

Comment faire pour stocker les nombres trouvés si on avait les deux questions suivantes

Q1) Ecrire un algorithme qui permet de trouver et de sauvegarder tous les nombres premiers inférieur à un entier N.

Q2) Ecrire un algorithme qui permet de trouver tous les nombres amis inférieurs à un entier N. Deux entiers A et B sont dits amis si la somme des diviseurs propres de A (A est exclu) est égale à B et la somme des diviseurs propres de B (B est exclu) est égale à A.

Proposition

Utiliser un tableau (structure statique).

- Si un tableau est utilisé, il n'est pas possible de définir sa taille avec précision même si la valeur de N est connue. Il faut donc choisir une taille suffisante dès le début, ce qui peut amener à un gaspillage (perte) considérable de place mémoire.
- Les tableaux ont un deuxième inconvénient : si un élément est supprimé du tableau, il faut déplacer (décaler) tous les éléments qui le suivent et même cela ne modifiera pas la taille du tableau car elle est fixée à la déclaration.

Solution

Donc on a besoin d'utiliser une structure dynamique qui évolue en taille. C'est-à-dire, au début, elle est vide, puis on lui rajoute un premier élément, puis un deuxième, puis un

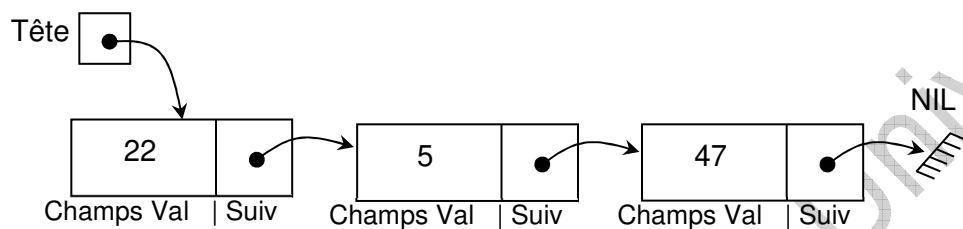
troisième et ainsi de suite. Ces éléments sont reliés entre eux par des pointeurs. Ces éléments constituent des maillons d'une nouvelle structure appelée **UNE LISTE**.

3.2- Définition d'une liste

Une Liste Linéaire Chaînée (LLC) est un ensemble d'éléments (Cellules, Nœuds, Maillons) alloués dynamiquement chaînés (reliés) entre eux. Chaque élément est un enregistrement qui contient au moins deux champs :

- ①- Un ou plusieurs champs qui contiennent l'information (Champ Valeur, Val, Info ou Data).
- ②- Le dernier champ contient un pointeur sur l'élément suivant (Champ Suivant, Suiv).

Schématiquement on peut représenter une liste de trois éléments (nombres entiers) comme suit :



Une liste est définie par un pointeur qui pointe vers le premier élément ; cet élément est appelé tête de la liste.

Le dernier élément de la liste est appelé queue.

3.3- Déclaration d'une Liste Linéaire Chaînée

Type Liste = ^Element

Element = **Enregistrement**

Val : TypeDonnee /* Un type quelconque : entier, réel, booléen, tableau, enregistrement ... */

Suiv : Liste /* Pointeur vers l'élément suivant */

Fin

Var L, P, Q : Liste

Exemple : Pour déclarer une liste d'entiers, on doit écrire :

Type Liste = ^Element

Element = Enregistrement

Val : Entier
Suiv : Liste
Fin

Var tete : Liste // tete est une liste d'entiers.

Remarque : Dans la suite du cours, nous travaillons avec des listes d'entiers. Les mêmes principes étudiés son applicables aux autres types de données.

3.4- Accès aux données d'une Liste Linéaire Chainée

Une liste est un pointeur. Pour accéder à la valeur pointée (élément, nœud, cellule, maillon), on doit utiliser l'opérateur chapeau (^). Chaque élément (nœud, cellule, maillon) est un enregistrement. Pour accéder aux champs de l'enregistrement, on utilise l'opérateur point (.).

Pour accéder au champ de données (Val) on écrit :

L^.Val ← 5 (par exemple)

Pour accéder au champ Pointeur (Suiv) on écrit :

L^.Suiv ← NIL (par exemple)

3.5- Opérations sur les listes

On peut classer les opérations sur les listes en trois catégories : construction (ou modification), consultation (ou exploitation) et destruction (ou suppression).

5.1- Construction :

- 1- Initialisation (création d'une liste vide).
- 2- Insertion (ajout) en tête (au début), au milieu ou en queue (à la fin).
- 3- Insertion par position.
- 4- Insertion dans une liste triée, ...

5.2- Consultation (exploitation) :

- 1- Affichage des éléments de la liste,
- 2- Calcul de la longueur de la liste (nombre d'éléments),
- 3- Recherche d'une valeur si elle existe,
- 4- Nombres d'occurrences, ...

5.3- Destruction (suppression) :

- 1- Suppression d'un élément : En tête, au milieu ou à la fin,
- 2- Destruction de la liste entière.

3.5.1- Initialisation d'une Liste Linéaire Chainée

Si L est une liste, son initialisation consiste à lui donner la valeur NIL (Not In List).

On écrit alors : **L ← NIL**.

Rappel : NIL est une constante qui représente une adresse particulière qui ne pointe vers aucune donnée.

3.5.2- Insertion dans une Liste Linéaire Chainée

Soit **L** le nom de la liste et **P** le pointeur sur le nouvel élément créé.

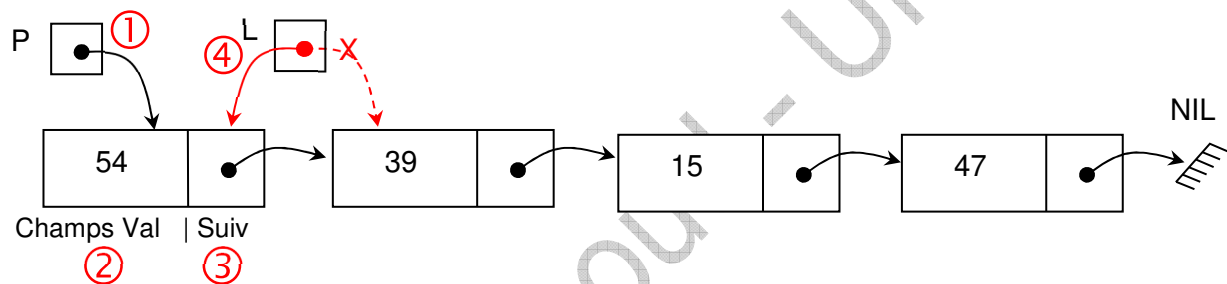
Dans tous les cas d'insertion, cette opération se fait en deux étapes :

Un nouvel élément (cellule) est créé avec l'instruction **ALLOUER(P)**. Puis le champ **Valeur (Val)** est initialisé avec la valeur voulue **P[^].Val ← valeur**.

L'élément créé est inséré dans la liste à l'endroit souhaité. C'est-à-dire, le nouvel élément est relié (chaîné) avec les autres éléments de la liste en adaptant leurs champs **Suivant (Suiv)**.

3.5.2.1- Insertion en tête de liste

Dans ce cas, il faut modifier le pointeur de tête de liste (**L**) et d'établir un lien entre l'élément nouvellement créé et l'élément qui se trouvait en tête (s'il existait). Ces opérations sont expliquées sur le schéma suivant :



On peut écrire une procédure **InsererTete** qui permet d'insérer en tête (au début) d'une liste **L**, une valeur **val** (**val** = 54 dans l'exemple).

Procédure **InsererTete**(Var **L** :Liste, **val** : Entier)

Var **P** : Liste

Début

- ① Allouer(**P**) // Créer un nouvel élément.
- ② **P[^].Val** ← **val** // Initialiser le champ de données (**Val**), insérer la valeur **val**.
- ③ **P[^].Suiv** ← **L** /* Initialiser le champ pointeur (**Suiv**). Créer le lien entre le nouvel élément et l'ancienne tête de la liste */
- ④ **L** ← **P** // Changer la tête (le début) de la liste

Fin

Exercice :

Ecrire un algorithme qui permet de créer une liste à partir des éléments d'un tableau d'entiers. Il faut que l'ordre des éléments dans la liste reste le même que celui des éléments du tableau.

Après la création de la liste, afficher ses éléments ainsi que leur nombre.

Solution :

Algorithme CreationListe

Const N = 10

Type Liste = ^Element

Element = Enregistrement

Val : Entier

Suiv : Liste

Fin

Var Tab : Tableau[1 .. N] d'entier

I : Entier

L, P : Liste // L est la tête de la liste

Procédure InsérerTete(Var L :Liste, val : Entier)

Var P : Liste

Début

① Allouer(P)

② $P^{\wedge}.Val \leftarrow val$

③ $P^{\wedge}.Suiv \leftarrow L$ // On crée le lien entre le nouvel élément et l'ancienne tête de la liste

④ $L \leftarrow P$ // On modifie la tête de liste

Fin

Début

// Lecture des éléments du tableau

Pour I \leftarrow 1 à N Faire

 Lire(Tab[I])

Fpour

L \leftarrow Nil // Initialisation de la liste pour indiquer qu'elle ne contient aucun élément.

/* Création de la liste : Pour que les éléments gardent le même ordre, il faut commencer l'insertion par le dernier élément et continuer jusqu'au premier. Au final, la liste garde le même ordre. Sinon, les éléments de la liste seront dans l'ordre inverse */

I \leftarrow N

TQ I > 0 faire

 InsérerTete(L, Tab[I])

 I \leftarrow I - 1

FTQ

// Affichage et comptage des éléments de la liste

Cpt \leftarrow 0

P \leftarrow L

TQ P \neq Nil Faire // Tant que on n'est pas encore arrivé à la fin de la liste

// Solution 2

Pour I \leftarrow N à 1 PAS = -1 Faire

 InsérerTete(L, Tab[I])

Fpour

```

    Ecrire( P^.Val)      // Affichage de la valeur
    Cpt ← Cpt + 1        // Compter cet élément
    P ← P^.Suiv          // Passer à l'élément suivant
FTQ
    Ecrire("La liste contient ", Cpt, " élément(s)") // Cpt doit être égal à N
Fin

```

Remarque : L'affichage et le comptage seront, plus loin dans le cours, l'objet d'une procédure **AfficherListe** et une fonction **Longueur** respectivement.

3.5.2.2- Insertion au milieu de la liste

Dans une liste simplement chaînée, le sens de parcours est toujours d'un élément (cellule) vers l'élément suivant : on ne peut pas revenir en arrière.

Pour insérer entre deux éléments, on insérera toujours après un élément donné.

Soit **Courant** l'élément après lequel on va insérer le nouvel élément.

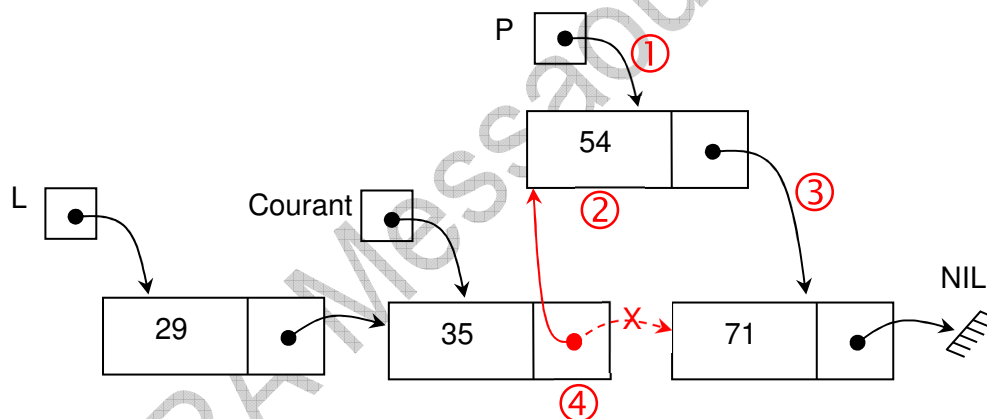
- On mémorise d'abord l'élément suivant de Courant dans le champ Suivant de P.

P^.Suiv ← Courant^.Suiv (Opération ③ sur le schéma ci-dessous)

- On peut maintenant faire pointer le champ Suivant de Courant sur P.

Courant^.Suiv ← P (Opération ④ sur le schéma ci-dessous)

Le schéma ci-dessous illustre l'insertion d'une valeur (54) dans une liste triée.



Procédure InsérerALaSuiteDe(Courant : Liste, val : Entier)

Var P : Liste

Début

- ① Allouer(P)
- ② P^.Val ← val
- ③ P^.Suiv ← Courant^.Suiv
- ④ Courant^.Suiv ← P

Fin

Remarque : Si on a une liste qui contient 4 éléments, on peut insérer un nouvel élément entre le 1^{er} et le 2^e élément ou entre le 2^e et le 3^e élément ou encore entre le 3^e et le

4^eélément. Pour faire le choix de l'endroit où l'insertion doit se faire, il faut avoir un critère. En général, on en a deux :

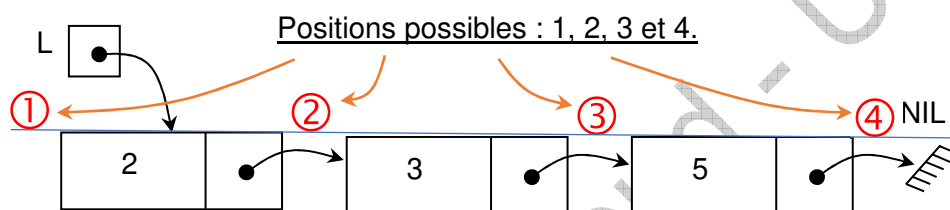
- ✓ Soit la liste est triée et donc chercher où on doit insérer le nouvel élément pour que la liste reste triée.
- ✓ Soit, on insère le nouvel élément par position. Par exemple, on souhaite l'insérer à la 3^e position et donc on va l'insérer entre le 2^e et le 3^e élément. C'est-à-dire, on va l'insérer après le 2^e élément.

3.5.2.2.1- Insertion par position

Le but est d'insérer le nouvel élément à une position donnée, si c'est possible.

Exemple : Si la liste contient trois éléments, on peut (toujours) insérer à la position 1, à la position 2, à la position 3 et à la 4^e position (la dernière). Mais, il est impossible d'insérer à d'autres positions comme la position 5, 6, ...

Sur le schéma suivant, sont montrées les positions possibles si on a une liste de trois éléments.



Procédure InsérerParPosition(Var L : Liste ; Pos : Entier ; val : Entier)

Var Cpt : Entier

A, Courant, P : Liste

Début

Si Pos = 1 Alors

InsérerTete(L, val) // Insertion en position 1 (en tête), opération toujours possible.

Sinon

Cpt ← 0

Courant ← L

TQ (Courant ≠ Nil) ET (Cpt < (Pos – 1)) Faire

Cpt ← Cpt + 1

A ← Courant // Sauvegarder la valeur de Courant

Courant ← Courant^.Suiv // passer à l'élément Suivant

FTQ

Si Pos = Cpt + 1 Alors // Vérifier si la position existe réellement !

① Allouer(P) // Insérer le nouvel élément après l'élément A

② P^.Val ← val

```

    ③ P^.Suiv ← A^.Suiv
    ④ A^.Suiv ← P
  /*Sinon
    Ecrire("Erreur : impossible d'insérer à la position ", Pos)*
  FSi
FSi
Fin

```

3.5.2.2.2- Insertion dans une liste triée

On suppose que la liste peut contenir plusieurs occurrences de la même valeur.

Solution 1

Procédure InsérerListeTriée(Var L : Liste ; val : Entier)

Var P, Q, Courant : Liste

Stop : Booléen

Début

```

  ① Allouer(P)
  ② P^.Val ← val
  Si L = Nil Alors // La liste est vide → Insertion en tête
    ③ P^.Suiv ← Nil // ≡ P^.Suiv ← L
    ④ L ← P
  Sinon
    Si L^.Val > val Alors // Val est supérieure à la 1ière Valeur de la liste
      ③ P^.Suiv ← L // Insertion en tête
      ④ L ← P
    Sinon
      // Insertion au milieu ou à la fin de la liste
      Courant ← L
      Stop ← Faux
      TQ (Courant^.Suiv ≠ Nil) ET (NON Stop) Faire
        Q ← Courant^.Suiv // Q utilisée pour simplifier l'écriture
        Si Q^.Val > val Alors
          Stop ← Vrai
        Sinon
          Courant ← Q
        FSi
      FTQ
      ③ P^.Suiv ← Courant^.Suiv // Que Courant^.Suiv soit Nil ou pas
      ④ Courant^.Suiv ← P
    FSi
FSi
Fin

```

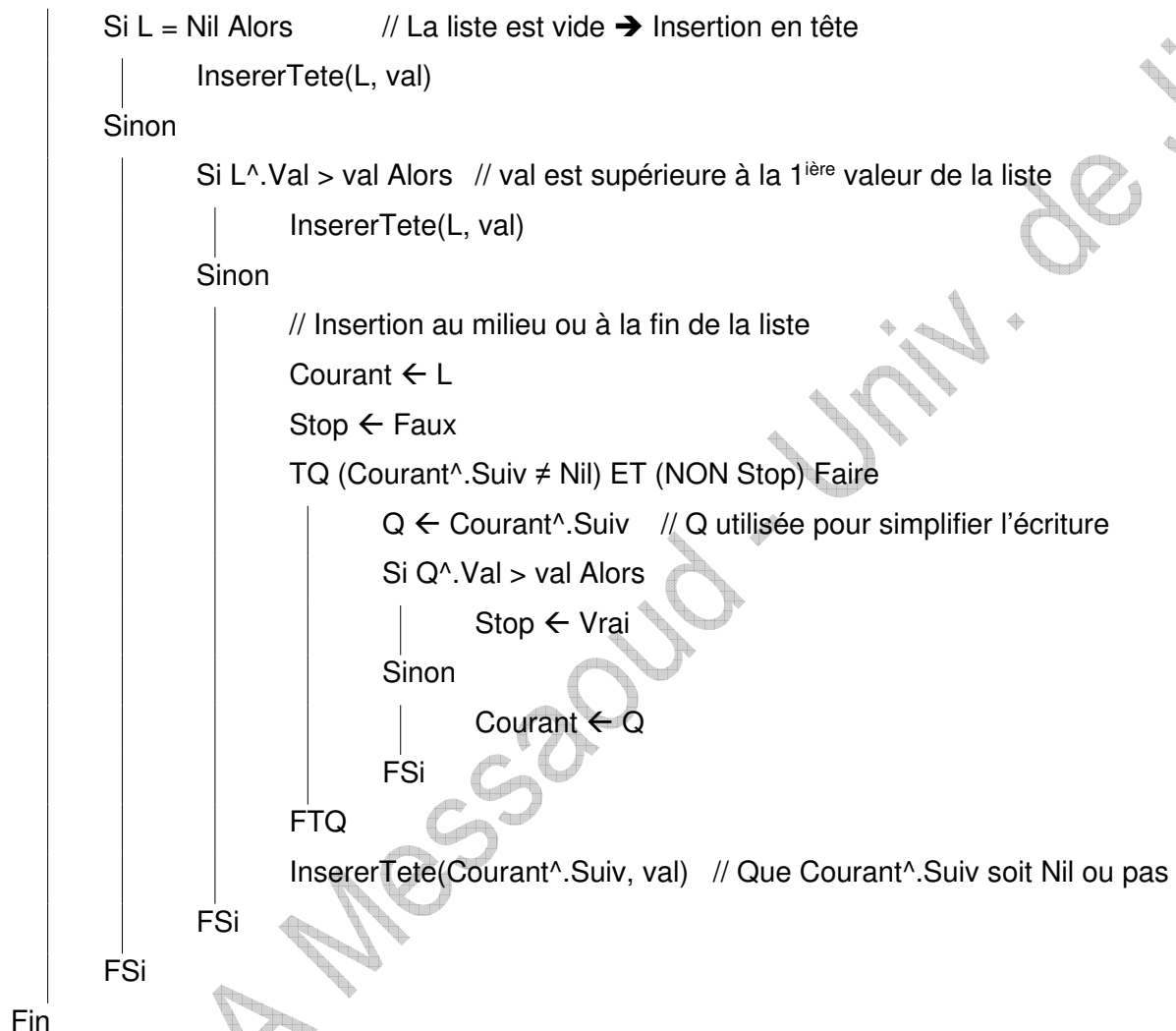
Solution 2

Procédure InsérerListeTriée(Var L : Liste ; val : Entier)

Var Q, Courant : Liste

Stop : Booléen

Début



3.5.2.3- Insertion à la fin de la liste

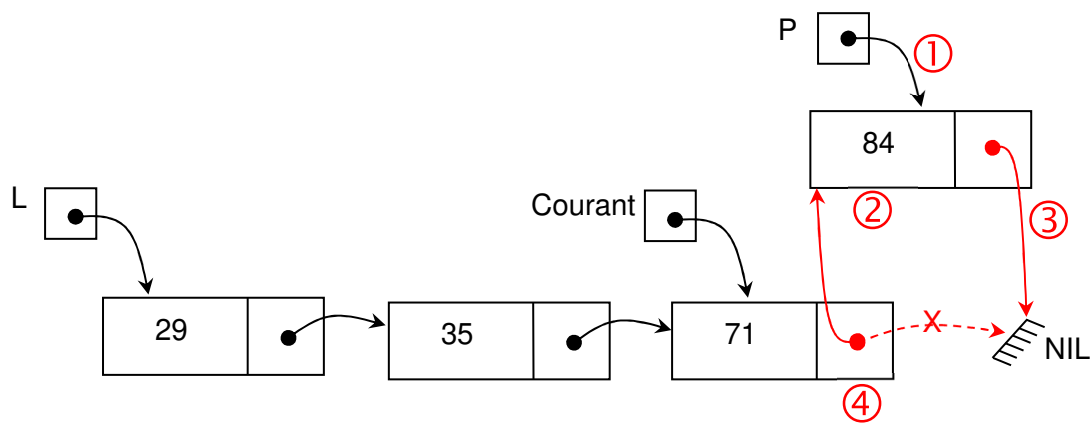
C'est un cas particulier de l'insertion au milieu de la liste : Insérer à la fin de liste, c'est insérer après le dernier élément dont le champ **Suivant** vaut **Nil**. Donc, on doit parcourir la liste jusqu'au dernier élément (Courant), puis insérer le nouvel élément après lui.

Après insertion, le champ Suivant de Courant vaut P et le champ suivant de P vaut Nil.

$P^{\wedge}.Suiv \leftarrow Nil$

$Courant^{\wedge}.Suiv \leftarrow P$

Sur le schéma ci-dessous, sont expliquées les opérations nécessaires pour l'insertion d'un nouvel élément (valeur 84) à la fin de la liste L.



En suivant ces étapes, on peut écrire la procédure **InsererQueue** comme suit :

Procédure InsererQueue(Var L :Liste, val : Entier)

Var P, Courant : Liste

Début

```

① Allouer(P)
② P^.Val ← val
③ P^.Suiv ← NIL
④ Si L = Nil Alors
    |   L ← P // Si la liste est vide, alors InsererQueue ≡ InsererTete.
Sinon
    |   Courant ← L
    |   TQ Courant ^.Suiv ≠ Nil Faire // Parcourir la liste jusqu'au dernier élément.
    |   |   Courant ← Courant ^.Suiv
    |   FTQ
    |   Courant ^.Suiv ← P
FSI

```

Fin

3.5.3- Consultation

Les opérations de consultation (ou d'exploitation) permettent d'utiliser la liste sans lui apporter aucune modification (ni ajout, ni suppression, ni modification d'ordre des éléments, ...). Parmi ces opérations, on peut citer : l'affichage de la liste, le comptage des éléments, la recherche d'une valeur si elle existe, le calcul du nombre d'occurrences d'une valeur, La vérification si la liste est triée ou pas, ...

3.5.3.1- Affichage des éléments de la liste

Pour afficher la liste, on commence par sa tête et on passe par ses éléments un par un et on affiche leurs champs valeur (Val), jusqu'en arrive à la fin de la liste.

Procédure AfficherListe(L :Liste)

Var P : Liste

Début

```

    Si L = Nil Alors
        Ecrire("Liste vide")
    Sinon
        P ← L
        TQ P ≠ Nil Faire
            Ecrire( P^.Val )
            P ← P^.Suiv
        FTQ
    FSi

```

Fin

/* Comme L est passée par valeur, on peut s'en passer de la variable locale P */

```

TQ L ≠ Nil Faire
    Ecrire( L^.Val )
    L ← L^.Suiv
FTQ

```

3.5.3.2- Calcul de la longueur de la liste

La longueur d'une liste est définie comme étant le nombre d'éléments qu'elle contient.

Fonction Longueur(L : Liste) : Entier

Var P : Liste

Cpt : Entier

Début

```

    Cpt ← 0
    P ← L
    TQ P ≠ Nil Faire
        Cpt ← Cpt + 1
        P ← P^.Suiv
    FTQ
    Longueur ← Cpt

```

Fin

/* Comme L est passée par valeur, on peut ne pas utiliser la variable locale P */

```

TQ L ≠ Nil Faire
    Cpt ← Cpt + 1
    L ← L^.Suiv
FTQ

```

3.5.3.3- Recherche d'une valeur dans la liste

Le résultat de recherche peut être soit un Booléen pour dire si la valeur recherchée existe ou n'existe pas, soit un pointeur (de type Liste) sur l'élément qui contient cette valeur. Si la valeur recherchée n'existe pas, le résultat de la fonction sera la constante Nil.

Nous optons ici pour cette dernière option.

Fonction Adresse(L : Liste ; val : Entier) : Liste

Var P : Liste

TRV : Booléen

Début

TRV \leftarrow Faux

P \leftarrow L

TQ (P \neq Nil) ET (NON TRV) Faire

Si P[^].Val = val Alors

TRV \leftarrow Vrai

Sinon

P \leftarrow P[^].Suiv

FSi

FTQ

Adresse \leftarrow P

Fin

Remarque : Pour retourner un booléen, il suffit de retourner la valeur de la variable locale TRV au lieu du pointeur P.

Exercice : Modifier cette fonction pour qu'elle retourne, le nombre d'occurrences de la valeur val.

Solution

Pour compter le nombre d'occurrences, on doit parcourir la liste en entier. Donc, nous n'avons pas besoin de s'arrêter si la valeur val est trouvée. On a aussi besoin d'un compteur (entier) pour compter le nombre d'occurrences.

Fonction NBOccurrences(L : Liste ; val : Entier) : Entier

Var P : Liste

Cpt : Entier

Début

Cpt \leftarrow 0

P \leftarrow L

TQ (P \neq Nil) Faire

Si P[^].Val = val Alors

Cpt \leftarrow Cpt + 1 // Compter l'élément trouvé.

FSi

P \leftarrow P[^].Suiv // Poursuivre en allant à l'élément suivant.

FTQ

NBOccurrences \leftarrow Cpt

Fin

3.5.4- Modification

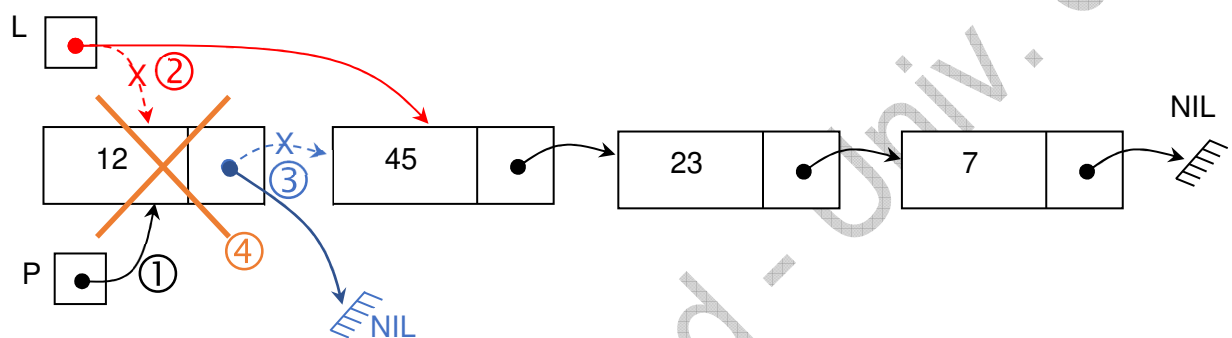
Les opérations de modification permettent de changer l'état de la liste en supprimant des éléments, en modifiant l'ordre des éléments, en modifiant les valeurs enregistrées, en fusionnant deux listes, ... et enfin la destruction complète de la liste.

La suppression, comme l'insertion, peut se faire au début, au milieu ou à la fin.

Parmi ces opérations, nous nous intéressons dans ce cours à la suppression au début et à la destruction de la liste. D'autres opérations seront l'objet de travaux dirigés.

3.5.4.1- Suppression de la tête de la liste

Les opérations nécessaires pour supprimer la tête de la liste sont explicitées sur le schéma ci-dessous.



Procédure SupprimerTete(Var L :Liste)

Var P : Liste

Début

Si L ≠ Nil Alors // On ne peut supprimer un élément que si la liste n'est pas vide !

① $P \leftarrow L$ // Garder l'adresse du 1^{er} élément.

② $L \leftarrow L^{\wedge}.Suiv$ // Modifier la tête de la liste

③ $P^{\wedge}.Suiv \leftarrow Nil$ // Optionnel. Casser le lien entre le 1^{er} et le reste de la liste

④ Libérer(P) // Libérer l'espace mémoire qu'occupait le 1^{er} élément.

FSi

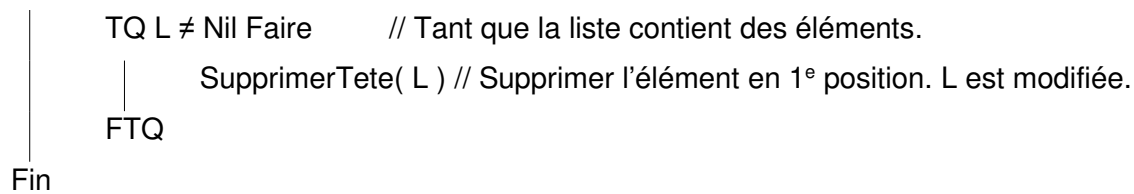
Fin

3.5.4.2- Destruction de la liste

Pour détruire la liste, il est nécessaire de supprimer tous ces éléments. On peut utiliser la procédure SupprimerTete autant de fois que nécessaire, jusqu'à ce que la liste soit vide.

Procédure Supprimerliste(Var L :Liste)

Début



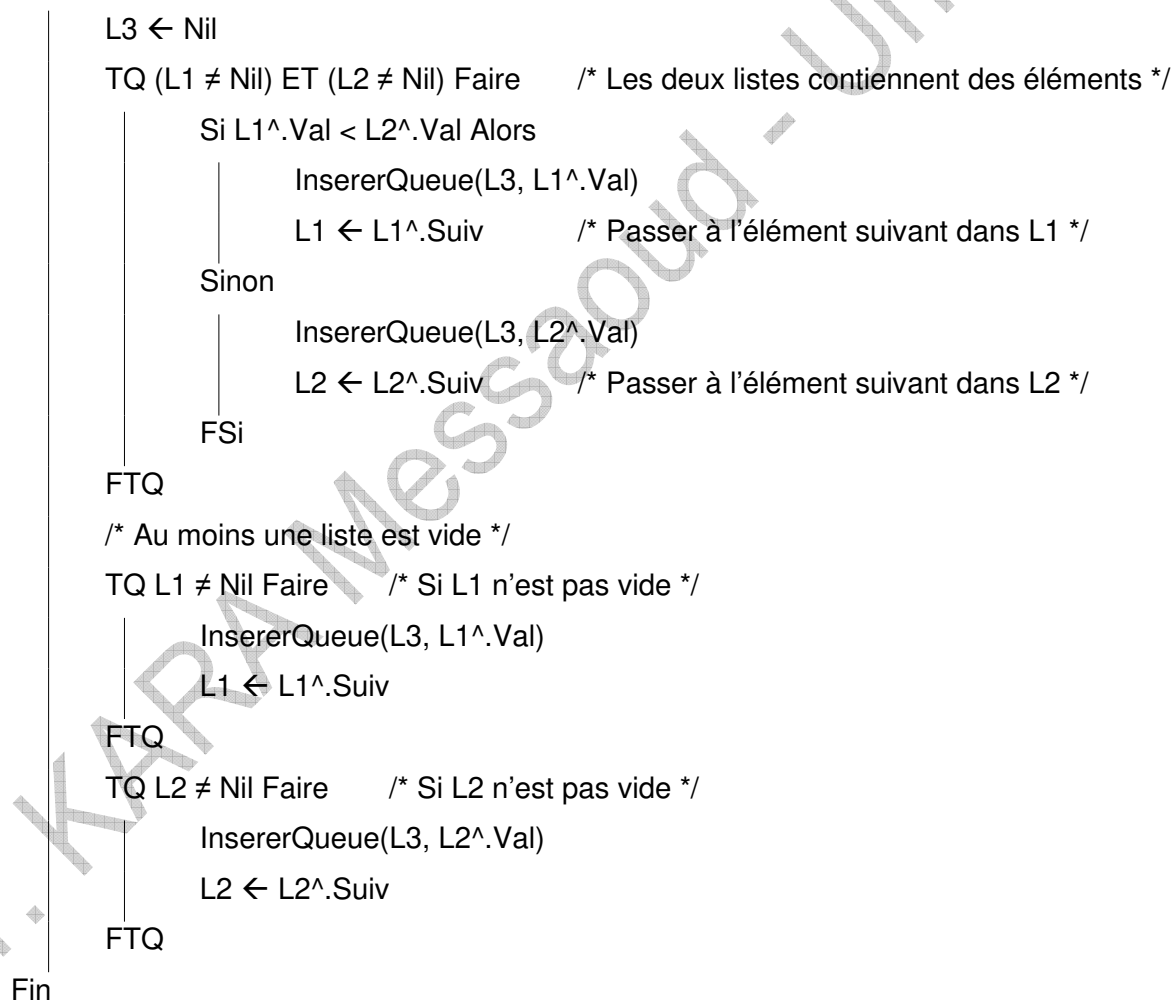
3.5.4.3- Fusion de deux listes triées

Le but est de fusionner deux listes triées pour avoir une troisième liste triée.

Solution 1 (la plus facile) : Les deux listes L1 et L2 ne sont pas détruites. Dans ce cas, il faut créer la liste L3 et insérer les valeurs de L1 et L2 dans le bon ordre dans L3 et à chaque fois à la fin de la liste L3.

Procédure Fusion(L1, L2 : Liste ; Var L3 : Liste)

Début



Solution 2 : Les deux listes L1 et L2 sont détruites et leurs éléments (cellules) sont réutilisés pour construire L3. Les pointeurs sur les éléments suivants sont modifier afin de construire la nouvelle liste L3.

Procédure Fusion(Var L1, L2, L3 : Liste)

Var P, D : Liste

Début

L3 \leftarrow Nil

P \leftarrow Nil

FTQ (L1 \neq Nil) ET (L2 \neq Nil) Faire /* Les deux listes contiennent des éléments */

Si L1[^].Val < L2[^].Val Alors

P \leftarrow L1

L1 \leftarrow L1[^].Suiv /* Passer à l'élément suivant dans L1 */

Sinon

P \leftarrow L2

L2 \leftarrow L2[^].Suiv /* Passer à l'élément suivant dans L2 */

FSi

/* Insertion de l'élément pointé par P à la fin de la liste L3 */

P[^].Suiv \leftarrow Nil /* Retirer l'élément de sa liste d'origine */

Si L3 = Nil Alors

L3 \leftarrow P /* Insertion en tête de la liste L3 */

Sinon

D[^].Suiv \leftarrow P /* Insertion à la fin de la liste L3 */

FSi

D \leftarrow P /* D est le dernier élément de L3 */

FTQ

/* Au moins une liste est vide */

SI L1 \neq Nil Alors /* Si L1 n'est pas vide */

P \leftarrow L1

L1 \leftarrow Nil /* Réinitialiser L1 */

FSi

Si L2 \neq Nil Alors /* Si L2 n'est pas vide */

P \leftarrow L2

L2 \leftarrow Nil /* Réinitialiser L2 */

FSi

/* Insertion de tous les éléments restants de L1 ou de L2 à la fin de la liste L3 */

Si L3 = Nil Alors

L3 \leftarrow P

Sinon

D[^].Suiv \leftarrow P

FSi

Fin

3.5.4.4- Eclatement d'une liste en deux listes selon le critère de parité

Le but est de construire deux listes **L1** et **L2** à partir des éléments d'une liste **L** en se basant sur le critère de parité.

Le principe : Le principe est de parcourir la liste **L** est d'extraire l'élément qui est en tête, puis de l'insérer à la fin de la liste **L1** si la valeur est impaire ou à la fin de la liste **L2** si la valeur est paire. A la fin, la liste **L** originale devient vide.

Procédure Eclater(Var L, L2, L3 : Liste)

Var P, D1, D2 : Liste

Début

```
L1 ← Nil      /* L1 est vide */
L2 ← Nil      /* L1 est vide */
TQ L ≠ Nil Faire
    P ← L
    L ← L^.Suiv
    P^.Suiv ← Nil      /* Un élément est retiré de L à chaque itération */
    Si P^.Val Mod 2 = 1 Alors      /* Insertion à la fin de L1 (nombres impairs) */
        Si L1 = Nil Alors
            L1 ← P      /* Initialiser L1 */
        Sinon
            D1^.Suiv ← P /* Insertion après le dernier élément de L1 */
        FSi
        D1 ← P      /* P devient le dernier élément de L1 */
    Sinon /* Insertion à la fin de L2 (nombres pairs) */
        Si L2 = Nil Alors
            L2 ← P      /* Initialiser L2 */
        Sinon
            D2^.Suiv ← P /* Insertion après le dernier élément de L2 */
        FSi
        D2 ← P      /* P devient le dernier élément de L2 */
    FSi
FTQ
```

Fin

3.5.4.5- Inversion d'une liste

Le principe : On parcourt la liste du début jusqu'à la fin, et on sauvegarde les adresses (pointeurs) des trois (3) éléments qui se suivent : P (Précédent), Q (Courant) et L (Suivant)

et on modifie le champ Suiv de l'élément courant (Q). Il ne pointera plus vers l'élément suivant (L), mais sur l'élément précédent (P).

Procédure inverserListe(Var L : Liste)

Var P, Q : Liste

Début

P ← Nil

TQ L ≠ Nil Faire

Q ← L /* L'élément courant (actuel) */

L ← L^.Suiv /* L'élément suivant */

Q^.Suiv ← P /* Suiv pointe sur l'élément précédent (Nil au début). */

P ← Q

FTQ

L ← P

Fin

5. Les algorithmes récursifs sur les listes

Dans cette section, nous nous intéressons à la version récursive de quelques opérations que nous avons déjà écrit leurs algorithmes en version itérative.

Parmi ces opérations, nous nous intéressons à l'affichage de la liste, la recherche d'une valeur et le nombre d'occurrences d'une valeur.

Dans toutes ces opérations, nous cherchons le (ou les) cas particulier(s) puis le cas général.

- Le cas particulier est souvent le cas où la liste est vide (ou a été entièrement parcourue).
- La cas général, lorsque la liste n'est vide. On traite l'élément courant (qui est en tête de liste), puis on fait un appel récursif pour traiter la liste contenant les éléments suivants.

4.1- Affichage récursif de la liste

Procédure AfficherListeREC(L : Liste)

Début

Si L = Nil alors

Ecrire("Liste Vide")

Sinon

Ecrire(L^.Val) // Op1

Si L^.Suiv ≠ Nil Alors

/* Si ce n'est pas le dernier élément, on fait un appel récursif pour afficher les éléments suivants. Sinon on s'arrête */ // Op2

AfficherListeREC(L^.Suiv)

FSi

FSi

Fin

4.2- Affichage inversé de la liste

Le but est d'afficher les éléments de la liste dans l'ordre inverse de leur présence dans la liste.

On sait que dans une liste (simplement chaînée), chaque élément ne donne accès qu'à l'élément qui le suit ; par conséquent écrire une version itérative pour cette opération n'est pas facile.

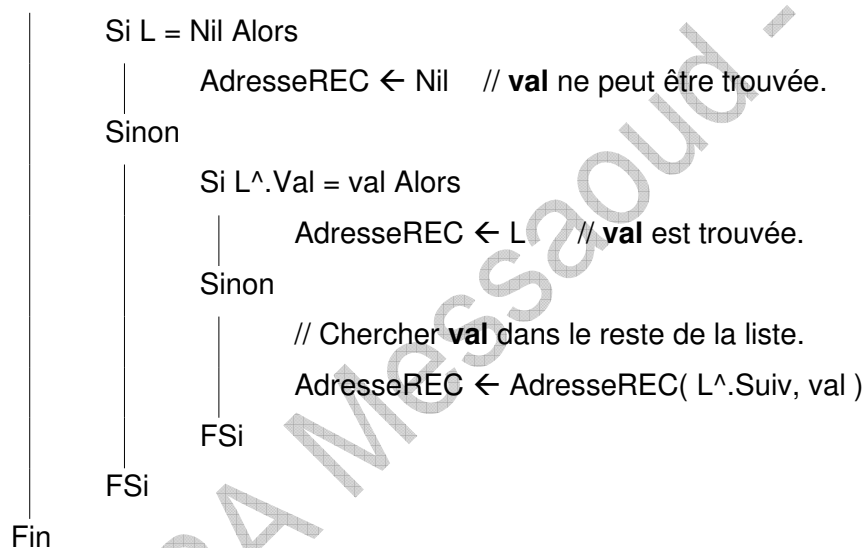
Cependant, pour écrire une version récursive, il suffit d'inverser d'ordre des opérations **Op1** et **Op2** de l'algorithme précédent pour obtenir un affichage inversé des éléments de la liste.

4.3- Recherche d'une valeur dans la liste

- Si la liste est vide, aucune valeur ne peut être trouvée dans la liste.
- Si la liste n'est pas vide, on vérifie si l'élément en tête de liste est égal à la valeur recherchée. Si c'est le cas, on a trouvé la valeur et on s'arrête. Sinon, avec un appel récursif, on vérifie si la valeur recherchée existe dans le reste de la liste.

Fonction AdresseREC(L : Liste ; val : Entier) : Liste

Début



Remarque : Si nous n'avons pas besoin du pointeur sur l'élément qui contient la valeur **val** et nous souhaitons que la réponse soit : Vrai ou Faux, il suffit de changer le type de retour en **Booléen** et à la place de **Nil** mettre **Faux** et à la place de **L** mettre **Vrai**.

4.4- Calcul de la longueur d'une liste

Si la liste est vide, sa longueur est nulle (égal à zéro). Sinon, on compte l'élément courant, puis avec un appel récursif on ajoute la longueur du reste de la liste.

Fonction LongueurREC(L : Liste) : Entier

Début

```

|
|   Si L = Nil Alors
|       |
|       |   LongueurREC ← 0
|       |
|       |   Sinon
|       |       |
|       |       |   LongueurREC ← 1 + LongueurREC(L^.Suiv)
|       |       |
|       |       |   FSi
|       |       |
|       |   FSi
|
|   Fin
```

4.5- Calcul du nombre d'occurrences d'une valeur donnée

Si la liste est vide, le nombre d'occurrence est nul (égal à zéro). Sinon, on vérifie si l'élément courant contient la valeur recherchée, si oui elle est comptée et ajoutée au nombre d'occurrences dans le reste de la liste, sinon, le nombre d'occurrences sera le nombre de fois où la valeur existe dans le reste de la liste seulement.

Fonction NombreOccurREC(L :Liste ; val : Entier) : Entier

Début

```

|
|   Si L = Nil Alors
|       |
|       |   NombreOccurREC ← 0
|       |
|       |   Sinon
|       |       |
|       |       |   Si L^.Val = val Alors
|       |       |       |
|       |       |       |   NombreOccurREC ← 1 + NombreOccurREC( L^.Suiv, val )
|       |       |       |
|       |       |       |   Sinon
|       |       |       |       |
|       |       |       |       |   NombreOccurREC ← NombreOccurREC( L^.Suiv, val )
|       |       |       |       |
|       |       |       |       |   FSi
|       |       |       |       |
|       |       |       |   FSi
|       |       |   FSi
|       |
|   Fin
```

4.6- Insertion par position

Procédure InsérerPos(Var L : Liste ; val : Entier ; pos : Entier)

Début

```

|
|   Si pos = 1 Alors
|       |
|       |   InsérerTete(L, val)
|       |
|       |   Sinon
|       |       |
|       |       |   Si L ≠ Nil Alors
|       |       |       |
|       |       |       |   InsérerPos(L^.Suiv, val, pos - 1)
|       |       |       |
|       |       |       |   /* Sinon
|       |       |       |       |
|       |       |       |       |   Ecrire("Erreur : Position ", pos, " n'existe pas. Insertion impossible") */
|       |       |       |       |
|       |       |       |       |   FSi
|       |       |       |   FSi
|       |       |
|       |   FSi
|
|   Fin
```

4.7- Insertion dans une liste triée

Procédure InsérerListeTriée(Var L : Liste ; val : Entier)

Début

```
Si L = Nil Alors      // La liste est vide → Insertion en tête.
|
|   InsérerTete(L, val)
|
Sinon
|
|   Si L^.Val > val Alors // Val est supérieure à la valeur en tête de liste.
|   |
|   |   InsérerTete(L, val)
|   |
|   |   Sinon
|   |   |
|   |   |   InsérerListeTriée(L^.Suiv, val) // Insérer val dans la liste restante.
|   |   |
|   |   FSi
|   FSi
FSi
```

Fin

4.8- Suppression de toutes les occurrences d'une valeur donnée

Procédure SupprimerVal(Var L : Liste ; val : Entier)

Début

```
Si ( L ≠ Nil ) Alors
|
|   Si L^.Val = val Alors
|   |
|   |   SupprimerTete( L )
|   |   SupprimerVal( L, val ) // La valeur de L a changé.
|   |
|   |   Sinon
|   |   |
|   |   |   SupprimerVal( L^.Suiv, val )
|   |   |
|   |   FSi
|   FSi
FSi
```

Fin

4.9- Destruction de la liste

Procédure SupprimerListe(Var L : Liste)

Début

```
Si ( L ≠ Nil ) Alors
|
|   SupprimerTete( L ) // Supprimer l'élément en 1e position.
|   SupprimerListe( L ) // La valeur de L a changé. Répéter !
|
FSi
```

Fin

6. Listes linéaires chaînées particulières

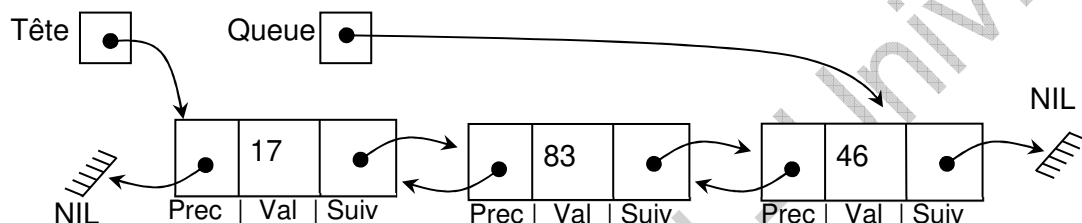
5.1- Les Listes bidirectionnelles (doublement chaînées)

Une liste linéaire chaînée est une liste où on ne peut effectuer qu'un parcours de gauche à droite. On peut qualifier cette liste de monodirectionnelle. Si on cherche à parcourir la liste de droite à gauche, il faut ajouter un deuxième pointeur permettant l'accès à la cellule précédente (champ Prec). On qualifie alors la liste de bidirectionnelle (ou doublement chaînée).

Définition

Une liste bidirectionnelle est une LLC que l'on peut parcourir dans les deux sens : de gauche à droite et de droite à gauche.

Schématiquement, une liste bidirectionnelle de 3 éléments se présente comme suit :



Chaque élément est un enregistrement qui contient trois (3) champs :

- ①- Un champ qui contient l'information (Val),
- ②- Un pointeur qui permet d'accéder à l'élément suivant (Suiv),
- ③- Un pointeur qui permet d'accéder à l'élément précédent (Prec).

Déclaration d'une Liste doublement Chaînée (bidirectionnelle)

Type ListeDC = ^Element

Element = Enregistrement

Suiv : ListeDC	// Un pointeur vers l'élément suivant
Val : TypeQuelconque	// Champ de données
Prec : ListeDC	// Un pointeur vers l'élément précédent

Fin

Var Tete, Queue : ListeDC /* Tête : pointeur vers le 1^{er} élément. Queue : pointeur vers le dernier élément */

Remarque : Pour exploiter efficacement ce type de liste, il est préférable d'utiliser deux pointeurs un pour la tête de la liste et un autre pour la queue. Ainsi, on peut par exemple afficher la liste dans le sens inverse en parcourant la liste à partir de la fin et en utilisant les champs Prec.

Opérations sur les listes doublement chaînées

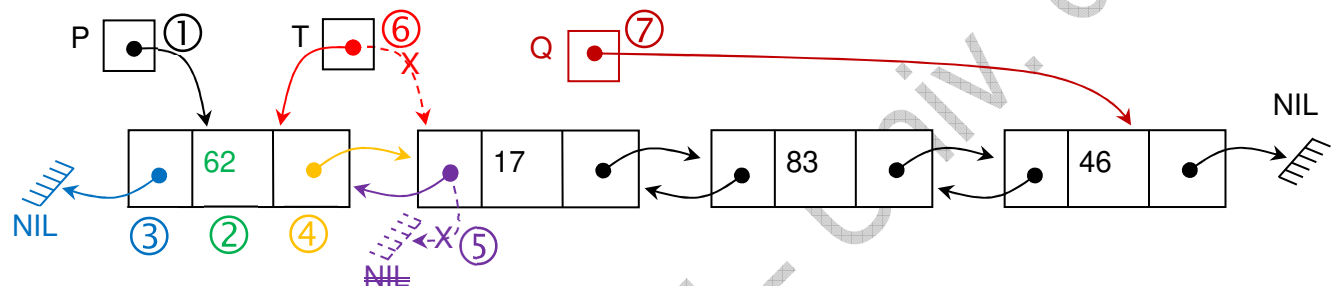
Toutes les opérations possibles sur les listes avec un seul chaînage sont possibles sur les listes doublement chaînées. Exemples : insertion dans une liste vide, insertion en tête,

insertion au milieu, insertion à la fin, suppression au début, suppression au milieu, suppression à la fin, recherche d'un élément, insertion à une position, calcul de la longueur... La seule différence est qu'il faut gérer deux chainages (les pointeurs Suiv et Prec) et gérer deux têtes (début et fin).

Pour illustrer ses opérations, nous choisissons l'exemple de l'insertion d'un nouvel élément en tête d'une liste doublement chaînée.

Insertion en tête d'une liste doublement chaînée

Les actions nécessaires pour insérer un nouvel élément dans une liste doublement chaînée (définie par les deux pointeurs T (Tête) et Q (Queue)) sont montrées sur le schéma ci-dessous :



Les opérations présentées sur le schéma précédent sont retranscrites dans la procédure ci-dessous.

Procédure InsertionTete(Var T : ListeDC ; Var Q : ListeDC ; val : Entier)

Var P : ListeDC

Debut

- ① Allouer(P) // Allouer un nouvel élément
- ② $P^{\wedge}.Val \leftarrow val$ // Remplir le champ Valeur
- ③ $P^{\wedge}.Prec \leftarrow Nil$ // Remplir le champ Précédent
- ④ $P^{\wedge}.Suiv \leftarrow T$ // Remplir le champ Suivant
- Si $T \neq Nil$ Alors
 - ⑤ $T^{\wedge}.Prec \leftarrow P$ // Si ce n'est pas la première insertion dans de la liste.
- FSi
- ⑥ $T \leftarrow P$ // Changer la tête de la liste
- Si $Q = Nil$ Alors
 - ⑦ $Q \leftarrow P$ // Si c'est la première insertion dans la liste.
- FSi

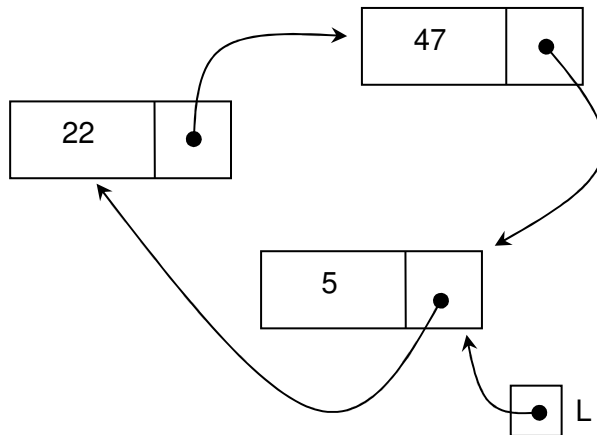
Fin

5.2- Les Listes circulaires (anneaux)

Une liste circulaire ou anneau est une liste linéaire dans laquelle le dernier élément pointe sur le premier. Il n'y a donc ni premier, ni dernier. Il suffit de connaître l'adresse d'un élément pour parcourir tous les éléments de la liste.

Une liste circulaire peut être simplement ou doublement chaînée.

Ci-après une liste circulaire (simplement chaînée) de trois éléments.



Remarque : Pour certaines applications, un tableau peut être utilisé pour représenter des listes (LLCs). Dans ce cas, chaque élément du tableau est un enregistrement qui renferme au moins deux champs : l'information (champ Val) et l'indice vers l'élément suivant (Champ Suiv). L'indice 0 ou -1 peut être utilisé pour remplacer la valeur de Nil. Dans ce cas, le nombre d'élément à créer est limité par la taille du tableau et non plus par la taille de la mémoire. Si le tableau est plein, on ne peut plus créer de nouveaux éléments.

7. Synthèse sur les listes

- ✓ Une liste (linéaire chaînées) est une structure dynamique dont la taille change au cours du déroulement de l'algorithme (l'exécution du programme) : Elle est vide au départ ; au besoin, les éléments sont insérés (ajoutés) ou supprimés (retirés) selon les besoins.
- ✓ Les insertions et les suppressions peuvent se faire au début, au milieu ou à la fin de la liste.
- ✓ Les insertions et les suppressions au milieu de la liste doivent se faire selon un critère : Par position ou le fait de garder la liste triée ...
- ✓ A la fin du déroulement de l'algorithme (l'exécution du programme) la liste doit être détruite.
- ✓ La liste est une structure récursive par excellence où la version récursive d'un traitement est, en général, facile à déduire.

8. PILES

7.1- Définition, principe, domaines d'application

La pile constitue l'un des concepts les plus utilisés dans la science des ordinateurs.

Une pile peut être définie comme une collection d'éléments où les insertions et les suppressions d'éléments se font à la même extrémité de la liste appelée **sommet de pile** (noté SP).

Elle applique le principe LIFO pour Last In First Out (Dernier entré, Premier Servi).

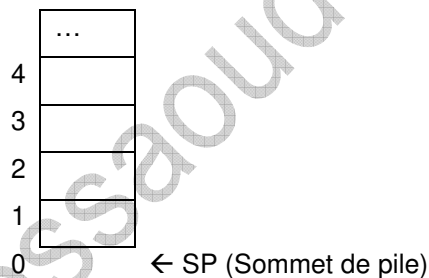
La pile est très utilisée dans le domaine de la compilation : résolution de la portée des variables, récursivité, évaluation d'expression ...

7.2- Exemple

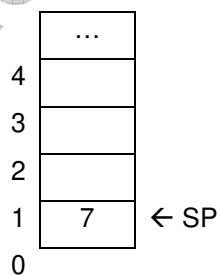
Pour expliciter le fonctionnement d'une pile, prenons l'exemple suivant : On suppose qu'on a une pile et on souhaite insérer les trois éléments (des valeurs entières) 7, 4 et 10 et puis en retirer deux éléments. Chaque insertion ou retrait se fait au sommet de pile.

On peut schématiser ces opérations comme suit :

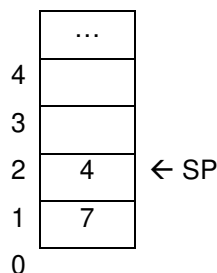
1- Etat initial de la pile : Au départ, la pile est vide, elle ne contient aucun élément, par conséquent, le sommet de pile (SP) ne pointe vers aucun élément.



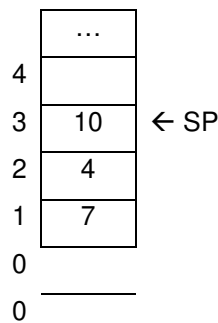
2- Insertion du 1^{er} élément (valeur 7) :



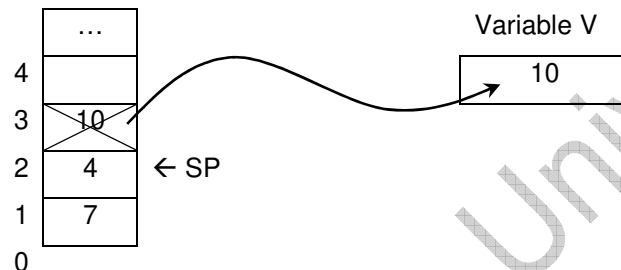
3- Insertion du 2^e élément (valeur 4) :



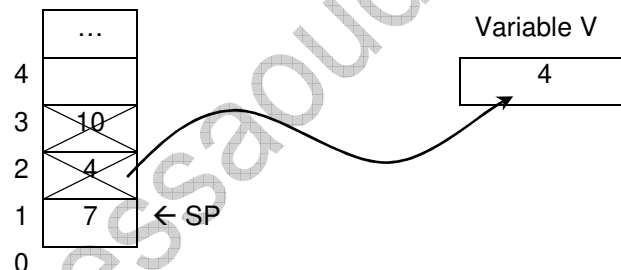
4- Insertion du 3^e élément (valeur 10) :



5- Retrait d'un élément : La valeur de l'élément retiré (le dernier élément inséré) est sauvegardée dans une variable V, le sommet de pile SP, se déplace d'une position.



6- Retrait d'un deuxième élément : La valeur de l'élément retiré est sauvegardée dans une variable V, le sommet de pile SP, se déplace d'une position.



7.3- Modèle

On définit sur les piles une machine abstraite munie de l'ensemble des opérations suivantes :

- ①- CréerPile(P) : Créer une pile vide. Permet d'initialiser la pile P.
- ②- Empiler(P, val) : Ajouter la valeur val en sommet de la pile P.
- ③- Depiler(P, val) : Retirer dans la variable val l'élément en sommet de de la pile P.
- ④- PileVide(P) : Une fonction booléenne permettant de tester si la pile P est vide.
- ⑤- PilePleine(P) : Une fonction booléenne permettant de tester si la pile P est pleine.

7.4- Implémentation

Une Pile peut être implémentée au moyen de :

- Un tableau (Manière statique)
- Une Liste Linéaire chaînée (Manière dynamique)

7.4.1- Implémentation d'une Pile en utilisant une Liste

Dans ce cas, la Pile est exactement une Liste où les insertions et les suppressions se font en tête. On n'utilisera alors que les procédures (sur les listes) **InsererTete** et **SupprimerTete**.

Le type Pile peut être déclaré comme suit :

Type Liste = ^Element

Element = Enregistrement

Val : Entier

Suiv : Liste

Fin

Pile = Liste // Une pile est exactement une liste.

Var P : Pile // Déclaration d'une Pile P.

Le modèle défini sur les Piles peut être implémenté comme suit :

Procédure CreerPile(Var P :Pile)

Début

P ← Nil // Initialisation d'une pile vide.

Fin

Procédure Empiler(Var P : Pile ; Val : Entier)

Début

Si NON PilePleine(P) Alors

InsererTete(P, Val)

/* Sinon

Ecrire("Empiler – Erreur : Pile Pleine") */

FSi

Fin

Procédure Depiler(Var P : Pile ; Var Val : Entier)

Début

Si NON PileVide(P) Alors

Val ← P^.Val

SupprimerTete(P)

/* Sinon

Ecrire("Depiler - Erreur : Pile vide") */

FSi

Fin

Fonction PileVide(P :Pile) : Booléen

Début

PileVide ← P = Nil

Fin

Fonction PilePleine(P :Pile) : Booléen

Début

 PilePleine ← Faux // On suppose que la taille de la mémoire est illimitée.

Fin

7.4.2- Implémentation d'une Pile en utilisant un tableau

En utilisant un tableau pour implémenter une Pile, on a besoin d'un tableau et d'un indice (SP : Sommet de pile) qui indique quel est l'élément qui est en tête de la pile.

Le type Pile peut être déclaré comme suit :

Const N = 50 // Taille Max de la pile.

Type Pile = Enregistrement

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

SP : Entier // Sommet de Pile

Fin

Var P : Pile // Déclaration d'une Pile P.

Le modèle défini sur les piles peut être implémenté comme suit :

Procédure CreerPile(Var P :Pile)

Début

 P.SP ← 0 // Il n'y a aucun élément dans la pile.

Fin

Procédure Empiler(Var P : Pile ; Val : Entier)

Début

 Si NON PilePleine(P) Alors

 P.SP ← P.SP + 1

 P.TAB[P.SP] ← Val

 /* Sinon

 Ecrire("Empiler – Erreur : Pile Pleine") */

 FSi

Fin

Procédure Depiler(Var P : Pile ; Var Val : Entier)

Début

 Si NON PileVide(P) Alors

 Val ← P.TAB[P.SP]

 P.SP ← P.SP – 1

 /* Sinon

 Ecrire("Depiler - Erreur : Pile vide") */

 FSi

Fin

Fonction PileVide(P :Pile) : Booléen

Début

 PileVide \leftarrow P.SP = 0 // Tableau vide.

Fin

Fonction PilePleine(P :Pile) : Booléen

Début

 PilePleine \leftarrow P.SP = N // Tableau plein, il contient N éléments.

Fin

7.5- Synthèse

- ✓ Une pile est une structure abstraite.
- ✓ Dans une pile les insertions et les suppressions d'éléments se font à la même extrémité. Donc ses opérations suivent le principe LIFO (Last In First Out, Dernier entré, Premier Servi).
- ✓ On définit sur les piles une machine abstraite munie de l'ensemble des opérations suivantes : CreerPile (Créer une pile vide), Empiler (Ajouter un élément), Depiler (Retirer un élément), PileVide(Vérifier si la pile est vide), PilePleine (Vérifier si la pile est pleine).
- ✓ Une pile peut être implémenté en utilisant une LLC (Liste Linéaire Chaînée) pour la version dynamique ou avec un tableau pour la version statique.

9. FILES

8.1- Définition, principe, domaine d'application

Une file d'attente peut être définie comme une collection d'éléments dans laquelle tout nouvel élément est inséré à la fin et tout élément ne peut être supprimé que du début. Elle applique le principe « FIFO » pour « First In First Out » qui veut dire « premier entré, premier servi »

La file d'attente est très utilisée dans les systèmes d'exploitation des ordinateurs et surtout dans les problèmes de simulation.

8.2- Exemple

Pour expliciter le fonctionnement d'une file, prenons l'exemple suivant : On suppose qu'on a une file et on souhaite insérer les trois éléments (des valeurs entières) 7, 4 et 10 et puis en retirer deux éléments. Chaque insertion se fait à la fin et chaque retrait se fait par le début.

On peut schématiser ces opérations comme suit :

1- Etat initial de la file : Au départ, la file est vide, elle ne contient aucun élément.

1	2	3	4	...

NombreElements = 0

2- Insertion du 1^{er} élément (valeur 7) :

1	2	3	4	...
7				

NombreElements = 1

3- Insertion du 2^e élément (valeur 4) :

1	2	3	4	...
7	4			

NombreElements = 2

4- Insertion du 3^e élément (valeur 10) :

1	2	3	4	...
7	4	10		

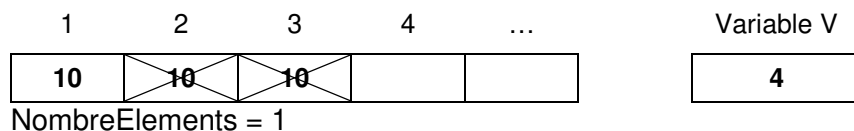
NombreElements = 3

5- Retrait d'un élément : La valeur de l'élément qui se trouve en première position dans la file est sauvegardée dans une variable V. Les autres éléments sont décalés d'un pas. Le nombre d'éléments est décrémenté de un.

1	2	3	4	...	Variable V
4	10	10			7

NombreElements = 2

6- Retrait d'un deuxième élément : même chose que 5-.



8.3- Modèle

On définit sur les files une machine abstraite munie de l'ensemble des opérations suivantes :

- ①- CreerFile(F) : Créer une file vide. Permet d'initialiser la file F.
- ②- Enfiler(F, val) : Ajouter val en queue (à la fin) de la file F.
- ③- Defiler(F, val) : retirer dans la variable val l'élément qui est en tête de la file F.
- ④- FileVide(F) : Une fonction booléenne permettant de tester si la file F est vide.
- ⑤- FilePleine(F) : Une fonction booléenne permettant de tester si la file F est pleine.

8.4- Implémentation

Une file peut être implémentée au moyen de :

- Un tableau (Manière statique)
- Une Liste Linéaire chaînée (Manière dynamique)

8.4.1- Implémentation d'une File en utilisant une Liste

Dans ce cas la file est exactement une liste où les insertions se font à la fin et les suppressions se font en tête. On n'utilisera alors que les procédures (sur les listes)

InsererQueue et **SupprimerTete**.

Le type File peut être déclaré comme suit :

Type Liste = ^Element

Element = Enregistrement

Val : Entier

Suiv : Liste

Fin

File = Liste // Le type File est exactement une Liste.

Var F : File // F est une File (Implémentée par une liste)

Implémentation du modèle (Version 1)

Le modèle défini sur les files peut être implémenté comme suit :

Procédure CreerFile(Var F :File)

Début

F ← Nil // Initialisation d'une file vide.

Fin

Procédure Enfiler(Var F : File ; Val : Entier)

Début

Si NON FilePleine(F) Alors
InsererQueue(F, Val)

```

    /* Sinon
    |
    |   Ecrire("Enfiler - Erreur : File Pleine") */
    FSi

```

Fin

Procédure Defiler(Var F : File ; Var Val : Entier)

Début

```

    Si NON FileVide(F) Alors
    |
    |   Val ← F^.Val
    |   SupprimerTete(F)
    |
    /* Sinon
    |
    |   Ecrire("Defiler - Erreur : File vide") */
    FSi

```

Fin

Fonction FileVide(F :File) : Booléen

Début

```

    FileVide ← F = Nil           // Ou

```

Fin

<pre> Si F = Nil Alors FileVide ← Vrai Sinon FileVide ← Faux FSi </pre>

Fonction FilePleine(F :File) : Booléen

Début

```

    FilePleine ← Faux // On suppose que la taille de la mémoire est illimitée.

```

Fin

Remarque : Pour travailler efficacement avec une file, il est préférable de sauvegarder la tête et la queue. Cela permet d'insérer directement après l'élément qui est en fin de la liste, sans avoir besoin de parcourir toute la liste pour insérer un nouvel élément à la fin de la liste (file). Dans ce cas, la file sera considérée comme un enregistrement composé de deux champs de type pointeur (Liste) :

- ①- Le premier (appelé Tete) pointe sur le premier élément de la liste,
- ②- Le deuxième (appelé Queue) pointe sur le dernier élément de la liste.

Le nouveau type File peut être déclaré comme suit :

Type *Liste* = *^Element*

Element = *Enregistrement*

Val : *Entier*

Suiv : *Liste*

Fin

File = *Enregistrement*

Tete, Queue : *Liste*

Fin

Var *F* : *File* // Déclaration d'une File F.

Implémentation du modèle (Version 2)

Le modèle défini sur les files devient :

Procédure CreerFile(Var F :File)

Début

 F.Tete ← Nil // Initialisation d'une file vide.

 F.Queue ← Nil

Fin

Procédure Enfiler(Var F : File ; Val : Entier)

Début

 Si NON FilePleine(F) Alors

 InsererQueue(F.Queue, Val)

 Si F.Tete = Nil Alors

 F.Tete ← F.Queue // Après insertion du premier élément seulement.

 FSi

 /* Sinon

 Ecrire("Enfiler - Erreur : File Pleine") */

 FSi

Fin

Procédure Defiler(Var F : File ; Var Val : Entier)

Début

 Si NON FileVide(F) Alors

 Val ← F.Tete^.Val

 SupprimerTete(F.Tete)

 Si F.Tete = Nil Alors

 F.Queue ← Nil // Après suppression du dernier élément seulement.

 FSi

 /* Sinon

 Ecrire("Defiler - Erreur : File vide") */

 FSi

Fin

Fonction FileVide(F :File) : Booléen

Début

 FileVide ← F.tete = Nil Ou

Fin

Fonction FilePleine(F :File) : Booléen

Début

 FilePleine ← Faux // On suppose que la taille de la mémoire est illimitée.

Fin

Si F.tete = Nil Alors
FileVide ← Vrai
Sinon
FileVide ← Faux
FSi

8.4.2- Implémentation d'une File en utilisant un tableau

En utilisant un tableau pour implémenter une file, on a deux modes de fonctionnement :

- 1- Utilisation des décalages : A chaque défilement (suppression de l'élément en tête (Tab[1])), tous les autres éléments seront décalés vers la gauche.
- 2- Utilisation circulaire du tableau : A chaque défilement, les éléments du tableau ne sont plus décalés, mais l'indice du début des éléments dans le tableau change. Il ne sera pas toujours 1.

Implémentation du modèle (Version 1 :Utilisation des décalages)

Const N = 50 // Taille Max de la file *

Type File = Enregistrement

```

|
|      TAB : Tableau[1 .. N] d'entier      // Eléments dans la file.
|      NB : Entier      // Nombre d'éléments dans la file.
|
Fin
```

Var F : File // Déclaration d'une File F)

Le modèle défini sur les files peut être implémenté comme suit :

Procédure CreerFile(Var F :File)

Début

```

|      F.NB ← 0 // Aucun élément n'est dans la file = file vide.
|
Fin
```

Procédure Enfiler(Var F : File ; Val : Entier)

Début

```

|      Si NON FilePleine(F) Alors
|          F.NB ← F.NB + 1
|          F.TAB[ F.NB ] ← Val
|      /* Sinon
|          Ecrire("Enfiler - Erreur : File Pleine") */
|      FSi
Fin
```

Procédure Defiler(Var F : File ; Var Val : Entier)

Var I : Entier

Début

```

|      Si NON FileVide(F) Alors
|          Val ← F. TAB[ 1 ] (* Retirer le premier élément *)
|          F.NB ← F.NB – 1
|          (* Décaler tous les autres éléments vers la gauche *)
|          Pour I ← 1 à F.NB Faire
|              F.TAB[ I ] ← F. TAB[ I + 1]
|          FPour
Fin
```



```

    /* Sinon
    |
    |   Ecrire("Defiler - Erreur : File vide") */
    |
    |   FSi
    |
Fin

```

```

Fonction FileVide(F :File) : Booléen
Début
    |
    |   FileVide ← F.NB = 0
    |
    |   Ou
    |
    |   Fin

```

```

    Si F.NB = 0 Alors
    |
    |   FileVide ← Vrai
    |
    |   Sinon
    |
    |   FileVide ← Faux
    |
    |   FSi

```

```

Fonction FilePleine (F :File) : Booléen
Début
    |
    |   FilePleine ← F.NB = N
    |
    |   Ou
    |
    |   (* Si le tableau est plein *)
    |
    |   Fin

```

```

    Si F.NB = N Alors
    |
    |   FilePleine ← Vrai
    |
    |   Sinon
    |
    |   FilePleine ← Faux
    |
    |   FSi

```

Implémentation du modèle (Version 2 :Utilisation circulaire du tableau)

On a besoin de sauvegarder les indices du début et la fin des éléments (comme pour la tête et la queue d'une liste) et laisser une case vide dans le tableau pour différencier le cas où la file est (presque) pleine (contient N-1 élément) et le cas où la file est vide (où les indices début et fin sont égaux).

Le type file devient :

Const N = 50 (* Taille Max de la file *)

Type File = Enregistrement

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

iDebut : Entier (* Indice début des éléments *)

iFin : Entier (* Indice fin des éléments *)

Fin

Var F : File (* Déclaration d'une File F *)

Le modèle défini sur les files devient :

Procédure CreerFile(Var F :File)

Début

F.iDebut ← 1 (* N'importe quelle valeur ∈ [1 ..N] à condition que F.iDebut = F.iFin *)

F.iFin ← F.iDebut

Fin

Procédure Enfiler(Var F : File ; Val : Entier)

Début

Si NON FilePleine(F) Alors

F.TAB[F.iFin] ← Val

F.iFin ← (F.iFin MOD N) + 1 (* Revenir au début du tableau si nécessaire *)

```

    |      /* Sinon
    |      |      Ecrire("Enfiler - Erreur : File Pleine") */
    |      FSi
Fin

```

Procédure Defiler(Var F : File ; Var Val : Entier)

Début

```

    |      Si NON FileVide(F) Alors
    |      |      Val ← F. TAB[ F.iDebut ] (* Retirer l'élément en tête de la file*)
    |      |      F.iDebut ← (F.iDebut MOD N) + 1
    |      |      // Revenir au début du tableau si nécessaire
    |      |      /* Sinon
    |      |      |      Ecrire("Defiler - Erreur : File vide") */
    |      |      FSi
    |
Fin

```

Fonction FileVide(F :File) : Booléen

Début

```

    |      FileVide ← F.iDebut = F.iFin
    |
Fin

```

Fonction FilePleine (F :File) : Booléen

Début

```

    |      FilePleine ← (F.iFin + N – F.iDebut) MOD N = N – 1 (* iFin peut être < iDebut *)
    |
Fin

```

8.5- File avec priorité

Une file d'attente avec priorité est une file d'attente dans laquelle l'opération de défilement récupère l'élément le plus prioritaire. Définir le modèle et l'implémenter.

Le type File peut être déclaré comme suit :

Type Liste = ^Element

Element = Enregistrement

Val : Entier (* L'information utile *)

Prio : Entier (* La priorité de la valeur Val *)

Suiv : Liste

Fin

File = Liste

On peut imaginer deux scénarios :

1- Les insertions se font en queue (normalement) et la fonction de défilement cherche l'élément le plus prioritaire (l'élément avec la valeur Max de la priorité). Si plusieurs éléments

ont la même priorité Max, c'est le premier élément avec la priorité Max qui sera retiré car les éléments sont insérés dans leurs ordres d'arrivée.

2- Les insertions se font par ordre de priorité (c'est exactement une insertion dans une liste triée sur la valeur du champ Prio au lieu du champ Val et les suppressions se font (normalement) en tête de la liste.

Remarque : l'implémentation d'une file avec priorité fera l'objet d'un exercice dans la partie Exercices corrigés.

8.6- Synthèse

- ✓ Une file est une structure abstraite. Elle peut être définie comme une collection d'éléments dans laquelle tout nouvel élément est inséré à la fin et tout élément ne peut être retiré que du début. Elle applique le principe « FIFO » pour « First In First Out » qui veut dire « premier entré, premier servi »
- ✓ On définit sur les Files une machine abstraite munie de l'ensemble des opérations suivantes : CreerFile (Créer une file vide), Emfiler (Ajouter un élément), Defiler (Retirer un élément), FileVide(Vérifier si la file est vide), FilePleine (Vérifier si la file est pleine).
- ✓ Une pile peut être implémenté en utilisant une LLC (Liste Linéaire Chaînée) pour la version dynamique ou avec un tableau pour la version statique.
- ✓ Pour l'implémentation dynamique de la file, une file n'est autre qu'une liste où les insertions se font à la fin et les suppressions se font en tête. Mais pour faciliter les insertions à la fin, il est conseillé d'utiliser deux pointeurs : un pointeur pour la tête de la liste (qui pointe sur le 1er élément) et un deuxième pour la queue de la liste (qui pointe sur le dernier élément).
- ✓ Pour l'implémentation statique de la file, un tableau est utilisé. La solution la plus directe est d'insérer chaque nouvel élément à la fin du tableau (derrière le dernier élément inséré) et en cas de suppression de l'élément en tête (à l'indice 1), tous les autres éléments qui le suivent sont décalés vers la gauche. Comme dans une file d'attente réelle (à un bureau de poste, par exemple), quand la première personne va au guichet, toutes les autres personnes avancent d'un pas.
- ✓ Une file d'attente avec priorité est une file d'attente dans laquelle l'opération de défilement récupère l'élément le plus prioritaire.

10. Les arbres (introduction)

Définition

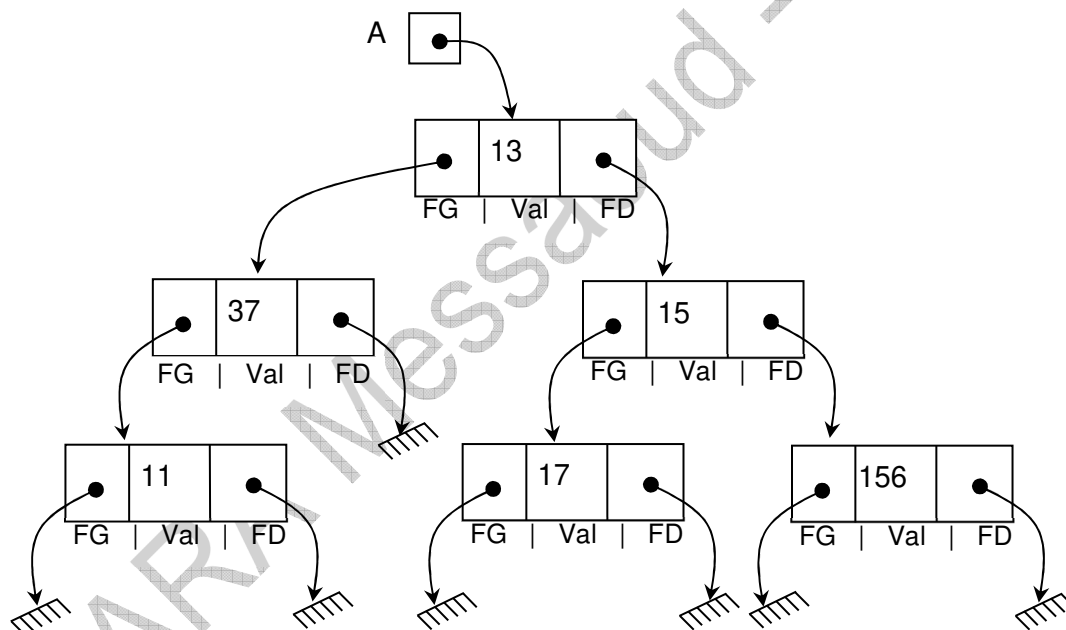
Un arbre est une structure de données hiérarchique, généralement dynamique.

Un arbre peut être considéré comme une liste chaînée non linéaire d'éléments (maillons, nœuds, cellules). Ces derniers forment un graphe orienté où chaque nœud a au plus 1 prédécesseur (appelé parent) et n ($n \geq 0$) successeurs (appelés fils).

En pratique un arbre est représenté de haut en bas. Ce qui permet de ne pas orienter les sens des arcs.

Si le nombre de successeurs de tout nœud est au plus égal à 2, l'arbre est dit binaire. En général, les deux fils sont appelés FD (Fils Droit) et FG (Fils Gauche).

Ci-dessous un exemple d'arbre binaire contenant 6 éléments :



PARTIE 3– COURS SUR LE LANGAGE C

Remarque : Le cours sur le langage C est divisé en plusieurs parties intégrées aux séries de TP. Chaque série, contient la partie nécessaire pour pouvoir la travailler sereinement.

La série N°5 est consacrée aux fonctions, procédures et récursivité.

La série N°6 est réservée aux structures données dynamiques (Listes linéaires chaînées, piles et files).

PARTIE 4– REFERENCES BIBLIOGRAPHIQUES

- [1] M. BELAID, Algorithmique & Programmation en PASCAL, Cours, Exercices, Travaux Pratiques, Corrigés, Bouira-Algérie: Eurl Pages Bleues Internationales, 2008.
- [2] D.-E. ZEGOUR, Structures de Données et de Fichiers Programmation PASCAL et C, Alger-Algérie: Editions Chihab, 1996.
- [3] J. COURTIN, I. KOWARSKI et J. ARSAC, Initiation à l'algorithmique et aux structures de données 1. Programmation structurée et structures de données élémentaires, Paris - France: Dunod, 1994.
- [4] J. COURTIN et I. KOWARSKI, Initiation à l'algorithmique et aux structures de données 2. Récursivité et structures de données avancées, Paris - France: Dunod, 1995.
- [5] C. DELANNOY, Apprendre à programmer en Turbo C, Alger-Algérie: Chihab - Eyrolles, 1994.
- [6] K. KHALFAOUI, «Introduction à la programmation, Support de Cours,» Université de Jijel, Jijel, 2018.
- [7] V. L. Damien Berthet, Algorithmique & programmation en langage C vol.1 : Supports de cours., Istanbul, Turquie : Université Galatasaray, 2014.
- [8] V. L. Damien Berthet, Algorithmique & programmation en langage C vol.2 : Sujets de travaux pratiques., Istanbul, Turquie: Université Galatasaray, 2014.
- [9] V. L. Damien Berthet, Algorithmique & programmation en langage C vol.3 : Corrigés de travaux pratiques., Istanbul, Turquie: Université Galatasaray, 2014.