

INF 212 : INFORMATIQUE 3

MBOUH GHOGOMU

2e ANNÉE ENSP 2018-2019

Table des matières

Table des matières	i
Table des figures	iv
1 Rappel	1
1.1 Structure générale d'un algorithme	1
1.2 Les opérateurs	3
1.3 Les structures de contrôle	4
1.4 Les instructions d'entrées-sorties	5
1.5 Les Fonctions	6
1.5.1 Déclaration et définition des fonctions	6
1.5.2 Récursivité	8
1.6 Les structures de données	9
1.6.1 Les énumérations	9
1.6.2 Les tableaux	10
1.7 Exercices du chapitre 1	12
2 Les pointeurs	20
2.1 Généralités	20
2.1.1 Déclaration et manipulation d'un pointeur	21
2.1.2 Arithmétique des pointeurs	22
2.1.3 Initialisation d'un pointeur	23
2.1.4 Double indirection	24
2.1.5 Allocation dynamique de la mémoire	24
2.2 Application des pointeurs.	27

2.2.1	Sur les fonctions	27
2.2.2	Sur les tableaux	28
2.3	Compléments sur les tableaux	31
2.3.1	Passage de tableau en argument	31
2.3.2	Les tableaux de caractères : les chaînes de caractères	32
2.3.3	Une technique usuelle : Le principe du flag	34
2.4	Exercices du chapitre 2	35
3	Analyse des algorithmes	44
3.1	Généralités	44
3.1.1	Temps d'exécution	44
3.1.2	Espace mémoire	46
3.1.3	Différents cas de calcul de complexité	46
3.2	Exemple 1 : Analyse de la fonction factorielle	48
3.2.1	Version itérative	48
3.2.2	Version récursive	49
3.3	Exemple 2 : Analyse du tri par insertion	50
3.4	Ordre de grandeur	52
3.4.1	Définition	52
3.4.2	Le vocabulaire	53
3.5	Exercices du chapitre 3	55
4	Les enregistrements	57
4.1	Généralités	57
4.1.1	Définition	57
4.1.2	Déclaration d'un type et d'une variable Enregistrement	57
4.1.3	Manipulation d'un enregistrement	58
4.1.4	Les tableaux d'enregistrement (ou tables)	61
4.1.5	Tableau de taille variable	62
4.1.6	Tableau circulaire	63
4.1.7	Enregistrement à champs variable	63
4.1.8	Pointeur sur un enregistrement	64
4.2	Les Listes chaînées	65
4.2.1	Définition et Manipulation	65

4.2.2	Les piles	69
4.2.3	Les files	70
4.2.4	Les listes doublement chaînées	72
4.3	Exercices du chapitre 4	73
5	Implémentation en langage C	80
5.1	Généralités	80
5.1.1	C : un langage compilé	80
5.1.2	Les mots-clés	80
5.1.3	Structure générale en programme C	81
5.2	Syntaxe	83
5.2.1	Les Opérations	83
5.2.2	Les Structures de contrôle	86
5.2.3	Les instructions d'entrées-sorties	86
5.2.4	Les énumérations	88
5.3	Les tableaux	89
5.3.1	Les chaînes de caractère en C	91
5.3.2	Les différentes gestions dynamiques	92
5.3.3	Gestion des arguments de la ligne de commande	94
5.4	Les enregistrements	94
5.4.1	Déclaration et utilisation	95
5.4.2	Initialisation et affectation d'une structure	96
5.4.3	Pointeur vers une structure	96
5.4.4	Le type Union	97
5.4.5	Définition des synonymes des types avec typedef	97
5.4.6	Les listes chaînées	98
6	Les arbres	100
6.1	Généralités	100
6.1.1	Définition	100
6.1.2	Représentation d'un arbre	101
6.2	Manipulation	103
6.2.1	Taille	103
6.2.2	Hauteur	104

6.2.3	Liste des éléments	104
6.2.4	Autres fonctions	105

Chapitre 1

Rappel

L'objectif de ce chapitre est de rappeler les notions vues au cours d'informatique de 1ère année. Celles-ci favoriseraient une écriture intuitive des algorithmes en pseudo-code : formalisme indépendant des particularités des langages de programmation et proche du langage humain.

1.1 Structure générale d'un algorithme

Un algorithme est une suite d'instructions, qui prend un ensemble de valeurs comme entrée (Données) et produit un ensemble de valeurs comme sortie (Résultats). Cette suite d'instructions est généralement précédée des commentaires, des déclarations et des fonctions. La Table 1.1 présente la structure d'un algorithme. La description des différentes rubriques :

1. Le nom de l'algorithme **NOM_ALGO**, devrait être assez significatif ; il est obligatoire.
2. Dans le commentaire **Objectif**, il est rappelé le problème à résoudre par l'algorithme ; cette rubrique est obligatoire.
3. Dans le commentaire **Donnée(s)**, les entrées de l'algorithme y sont mentionnées ;
4. Dans le commentaire **Résultat(s)**, le procédé d'obtention des sorties en fonction des entrées y est attendu ;
5. La déclaration des constantes est introduite grâce au mot clé **Constante** ;
Une constante est déclarée sous la forme suivante :

1.	<u>Algorithme</u> NOM_ALGO
2.	/* <u>Objectif</u> :....*/
3.	/* <u>Donnée(s)</u> :....*/
4.	/* <u>Résultat(s)</u> : ... */
5.	/* Partie déclaration */
6.	<u>Constante</u> :
7.	<u>Type</u> :
8.	<u>Variable</u> :
9.	/*les fonctions */
10.
11.	/* partie exécution */
12.	<u>Début</u>
13.	Instruction 1
14.	Instruction 2
15.
16.	Instruction n
17.	<u>Fin.</u>

TABLE 1.1 – Structure générale d’un algorithme.

$NOM_CONSTANTE = valeur$

6. La déclaration des types des variables autres que les types de base (Entier, Réel, Caractère (' '), Booléen (VRAI ou FAUX)) est introduite par le mot clé **Type**.
7. La déclaration des variables est introduite grâce au mot clé **Variable**. Chaque type de variable se trouve sur une nouvelle ligne (voir Table 1.2).

Une variable déclarée au sein d’un algorithme (resp. d’une fonction), est donc créée avec cet algorithme (resp. fonction), et disparaît avec lui (resp. elle). Durant tout le temps de son existence, une telle variable n’est visible que dans la partie exécution de l’algorithme (resp. de la fonction) qui l’a vu naître. En pseudo code, **toutes les variables sont locales**.

type1 : nom_variable1.1, [nom_variable2.1, ... ,nom_variableN.1] type2 : nom_variable1.2, [nom_variable2.2, ... ,nom_variableM.2] typeP : nom_variable1.P, [nom_variable2.P, ... ,nom_variableQ.P]
--

TABLE 1.2 – Déclaration de type.

8. Dans la rubrique **les fonctions**, sont placées toutes les fonctions utiles.
9. La partie exécution, encore appelée **programme principal**, se trouve entre les mots clés **Début** et **Fin**. Cette partie regroupe toutes les instructions exécutables - une seule instruction par ligne. C'est une partie obligatoire.

Exemple :

```

1.  Algorithme PrixTTC
2.  /*Objectif : cet algorithme demande un prix au clavier, prend en compte la TVA
    et affiche le résultat */
3.  /*Donnée(s) : le prix Hors Taxe ht qui est un réel saisi au clavier*/
4.  /*Résultat(s) : afficher à l'écran le prix Tout Taxe Compris :
    prix_TTC = ht*(1+TVA/100)*/
5.  //Partie déclaration
6.      Constante : TVA = 19,54          //la TVA au Cameroun
7.      Variable : réel : prix_TTC, ht
8.  /* les fonctions */
9.  réel : Fonction ajout_TVA (réel : prix_HT)
10.      /*Objectif : Cette fonction ajoute la TVA au prix passé en argument */
11.      /*Donnée : le prix hors taxe prix_HT qui est un réel passé en argument */
12.      /*Résultat : retourne le résultat du calcul : prix_HT*(1+TVA/100) */
13.  Debut
14.      Retourner (prix_HT*(1+TVA/100))
15.  Fin /*fonction*/
16.  /* partie exécution */
17.  Début
18.      Ecrire ("entrer le prix H.T")      // instruction d'entrée
19.      Lire (ht)                          // instruction de sortie
20.      prix_TTC ← ajout_TVA(ht)           // appel de la fonction ajout_TVA()
21.      Ecrire ("Le prix T.T.C est", prix_TTC)
22.  Fin.

```

1.2 Les opérateurs

Les principaux opérateurs sont résumés dans la Table 1.3.

NB : Concernant les opérations logiques :

N	Opérateur	Notation	Exemple	Résultat
1.	Addition	+	$x+y$	La somme de x et y
2.	Soustraction	-	$x-y$	la soustraction y de x
3.	Produit	*	$x*y$	la multiplication de x par y
4.	Division réelle	/	x/y	le quotient de la division réelle de x par y
5.	Division entière	div	$x \text{ div } y$	le quotient de la division entière de x par y
6.	Reste	mod	$x \text{ mod } y$	Le reste de la division euclidienne de x par y
7.	Signe négatif	-	$-x$	la négation arithmétique de x
8.	Affectation	\leftarrow	$x \leftarrow y$	Assigne la valeur de y à x
9.	Inférieur	<	$x < y$	VRAI si x est inférieur à y
10.	Inférieur ou égal	\leq	$x \leq y$	VRAI si x est inférieur ou égale à y
11.	Supérieur	>	$x > y$	VRAI si x est Supérieur à y
12.	Supérieur ou égal	\geq	$x \geq y$	VRAI si x est Supérieur ou égale à y
13.	Égalité	=	$x = y$	VRAI si x est égal à y
14.	Différence	\neq	$x \neq y$	VRAI si x est différent de y
15.	ET logique	ET	$x \text{ ET } y$	VRAI si x et y sont différents de FAUX
16.	OU logique	OU	$x \text{ OU } y$	VRAI si x et/ou y sont différents de FAUX
17.	NON logique	NON	NON x	VRAI si x a la valeur Faux
18.	Appel de fonction	$f(...)$	$f(x)$	Exécute la fonction f(x)

TABLE 1.3 – Les opérateurs en pseudo-code.

Les opérateurs ET logique et OU logique évaluent les opérandes de gauche à droite et le résultat peut être connu dès que l'opérande de gauche est évalué. Ainsi, l'opérande de droite n'est évalué que si celui de gauche est vrai dans le cas de l'opérateur ET logique (respectivement faux dans le cas de l'opérateur OU logique).

Exemple :

Dans l'expression $(i < \text{max}) \text{ ET } (f(14) = 1)$, la fonction f n'est appelée que si $i < \text{max}$.

1.3 Les structures de contrôle

Les structures de contrôle sont résumées dans la Table 1.4.

NB :

- Ne pas modifier le compteur "*varEntiere*" dans une boucle Pour ;
- Utiliser la boucle Pour quand le nombre d'itération est connu d'avance ;

Sélection simple partielle	<u>Si</u> Expression <u>Alors</u> Instruction <u>FinSi</u>
Sélection simple totale	<u>Si</u> Expression <u>Alors</u> Instruction1 <u>Sinon</u> Instruction2 <u>Fsi</u>
Sélection multiple	<u>Cas</u> où(variable) constante1 : instruction1 constante2 : instruction2 ... constanteN : instructionN <u>Sinon</u> instruction <u>FinCas</u>
Structures itératives : Boucle Pour	<u>Pour</u> varEntiere ← initial (pas) final <u>faire</u> Instruction <u>FinPour</u>
Structures itératives : Boucle TantQue	<u>Tantque</u> Expression <u>faire</u> Instruction <u>FinTantque</u>
Structures itératives : Boucle Repeter	<u>Repeter</u> Instruction <u>Tantque</u> Expression

TABLE 1.4 – Les structures de contrôle.

- Utiliser la boucle Répéter quand "Instruction" est exécutée au moins une fois.

1.4 Les instructions d'entrées-sorties

- L'instruction de lecture : **Lire(...)** permet d'entrer des données via le clavier; l'exécution de l'algorithme s'arrête à cette instruction en attendant la frappe des valeurs au clavier;
- L'instruction d'écriture : **Ecrire(...)** permet d'afficher les résultats à l'écran;
NB : Avant de lire une variable, il faut prendre l'habitude d'écrire les libellés à

l'écran afin de prévenir l'utilisateur de ce qu'il doit frapper au clavier (sinon, il pourrait y avoir une incompréhension entre l'utilisateur et l'ordinateur). **Ainsi, avant toute instruction de lecture, il faut une instruction d'écriture.**

1.5 Les Fonctions

Un algorithme long peut :

- procéder aux mêmes traitements, ou à des traitements similaires, à plusieurs endroits ;
- devenir illisible ;
- être difficile à maintenir.

Il faut donc écrire les algorithmes courts en scindant les longs algorithmes en fonctions afin de les appeler dans la partie exécution.

1.5.1 Déclaration et définition des fonctions

La structure d'une fonction est présentée à la Table 1.5. Dans cette structure de

1.	[type_resultat :] <u>FONCTION</u> nom_fonction([type1 : arg11,arg12,..., arg1M type2 : arg21,arg22,..., arg2P ... typeN : argN1,argN2,..., argNQ])
2.	/* <u>Objectif</u> :...*/
3.	/* <u>Argument(s)</u> :...*/
4.	/* <u>Résultat</u> : ... */
5.	//déclaration de variables locales
6.	<u>Variable</u> : ...
7.	/* partie exécution */
8.	<u>Début</u>
9.	Instruction 1
10.	Instruction 2
11.	...
12.	Instruction n
13.	[Renvoyer resultat] //si type_resultat existe
14.	<u>Fin.</u>

TABLE 1.5 – Structure générale d'une fonction.

fonction, on rappelle que :

- `type_resultat` correspond au type du résultat de la fonction, il est facultatif. Une fonction peut fournir comme résultat un type de base, un pointeur ou un enregistrement. Une fonction ne peut pas fournir comme résultat des tableaux ou des fonctions.
- `nom_fonction` est le nom qui identifie la fonction ;
- $$\left\{ \begin{array}{l} type1 : arg11, arg12, \dots, arg1M \\ type2 : arg21, arg22, \dots, arg2P \\ \dots \\ typeN : argN1, argN2, \dots, argNQ \end{array} \right\}$$
 définit les types et les noms des arguments de la fonction. Ces arguments sont appelés paramètres formels, par opposition aux paramètres effectifs qui sont les arguments avec lesquels la fonction est appelée dans le programme principal ;
- Le commentaire "Donnée(s)" de la structure générale d'un algorithme est remplacé par "Argument(s)". Si une fonction n'a pas d'argument, on peut déclarer la liste des arguments par les parenthèses vides () ;
- Le commentaire "Résultat" est toujours utilisé ici au singulier car une fonction retourne au plus un résultat.
- Au cas où "`type_resultat`" existe, le corps de la fonction doit obligatoirement comporter une instruction **retourner** (ou **renvoyer**). Plusieurs instructions **retourner** peuvent apparaître dans une fonction. Le retour au programme appelant sera alors provoqué par le premier **retourner** rencontré lors de l'exécution.

NB :

- Ne pas définir des fonctions à l'intérieur d'une autre fonction. L'ordre des définitions de fonctions dans le programme n'est pas important, mais chaque fonction doit être déclarée ou définie avant d'être appelée.
- Les variables éventuellement déclarées au sein d'une fonction seront locales à celle-ci et n'auront de sens que dans le corps de la fonction. De même, les identificateurs des arguments de la fonction (`arg1`, ... ,`argN`) n'ont d'importance qu'à l'intérieur de celle-ci.
- L'appel d'une fonction se fait par l'expression :
$$nom_fonction(arg1, \dots, argN)$$

L'ordre et le type des paramètres effectifs de la fonction doivent concorder

avec ceux des paramètres formels.

1.5.2 Récursivité

Une fonction récursive est une fonction qui s'appelle elle-même un certain nombre de fois. Elle utilise une pile lors de son exécution. Le rôle de cette pile est de stocker les variables locales et les paramètres d'une fonction. Dans une fonction récursive, toutes les variables locales sont stockées dans la pile, et empilées autant de fois qu'il y a d'appels récursifs. Donc la pile se remplit progressivement, et si vous ne faites pas attention (fixer un point d'arrêt) vous aurez un "débordement de pile". Ensuite, les variables sont désempilées. Toute fonction récursive comporte une instruction (ou un bloc d'instructions) nommée "point d'arrêt", qui indique que le reste des instructions ne doit plus être exécuté. La table 1.6 présente la structure d'une fonction récursive.

1.	<u>[type_resultat :]</u> <u>FONCTION</u> rec (PARAMETRES)
2.	<u>/* Objectif : ... */</u>
3.	<u>Variable</u> : ...
4.	<u>Debut</u>
5.	<u>Si</u> (CONDITION_D_ARRET) <u>alors</u>
6.	INSTRUCTION_D_ARRET
8.	<u>Sinon.</u>
9.	<u>Debut</u>
10.	...
11.	[Renvoyer] rec(PARAMETRE_CHANGES)
12.	<u>Fin</u>
13.	<u>FinSi</u>
14.	<u>Fin.</u>

TABLE 1.6 – Structure d'une fonction récursive.

Les paramètres de l'appel récursif changent ; en effet, le fait de ne rien changer dans les paramètres ferait que l'ordinateur effectuerait un appel infini à cette procédure, ce qui se traduirait en réalité par un débordement de pile, et l'arrêt de l'exécution de la procédure en cours. Grâce à ces changements, l'ordinateur rencontrera éventuellement un ensemble de paramètres vérifiant la condition d'arrêt, et donc à ce moment la procédure récursive aura atteint le point terminal. Ensuite les paramètres ainsi que les variables locales sont désempilées au fur et à mesure

que vous remontez les niveaux.

Exemple : La factorielle d'un nombre

```
1. Entier : FONCTION facto (entier n)
2. /* Objectif : calcul récursif de la factorielle d'un nombre*/
3. /*arguments : un entier n */
4. /*Résultat :  $n! = n \times (n-1)!$  */
5. Variable :
6.         Entier : r
7. Debut
8.         Si(n = 1) alors
9.             r ← 1           //INSTRUCTION_D_ARRET
10.        Sinon.
11.            r ← n * facto(n - 1)
12.        FinSi
13.        Renvoyer (r)
14. Fin.
```

1.6 Les structures de données

Les données des types de bases (booléen, caractère, entier, réel) peuvent être organisées en des ensembles de données, appelés structures de données. Nous rappelons ici les énumérations et les tableaux.

1.6.1 Les énumérations

Les énumérations permettent de définir un type pour des variables qui ne sont affectées qu'à un nombre réduit de valeurs. Dans la partie déclaration des types, on donne :

1. un identifiant
2. = (le signe **égal**)
3. le mot clé **Enumération**
4. dans les accolades, la liste des valeurs que peut prendre une variable de ce type (les différentes constantes).

Une variable de type **énumération** est définie comme les autres variables.

Exemple : Les opérations possibles sur une variable de type **énumération** sont

```
1. Algorithme PLANNING
2. /* Objectif : afficher le planning de la semaine */
3. Type :
4.     TjourSemaine = Enumération{lundi, mardi, mercredi, jeudi, vendredi, samedi,
                                   dimanche}
5. Variable :
6.     TjourSemaine : js
7. Debut
8.     Ecrire ("Entrer le jour de la semaine")
9.     Lire (js)
10.    Cas où(js)
11.        lundi : ecrire("analyse")
12.        mardi : ecrire("informatique")
13.        mercredi : ecrire("chimie")
14.        jeudi : ecrire("algèbre")
15.        vendredi : ecrire("physique")
16.    Sinon
17.        Ecrire ("weekend : pas de programme")
18.    FinCas
19. Fin.
```

les comparaisons et l'affectation.

1.6.2 Les tableaux

Un tableau est un ensemble de valeurs portant le même nom de variable et repérées par un nombre. C'est une structure de donnée permettant de mémoriser des valeurs de même type.

Le nombre qui, au sein d'un tableau, sert à repérer chaque valeur s'appelle **indice**.

Nous définissons un tableau par la syntaxe suivante :

Type : *Id_Type* = *Tableau*[*nombre d'elements*]*de type_des_elements*

Variable : *Id_Type* : *mon_variable*

Où *nombre d'elements* est un entier positif correspondant au nombre maximal d'élément dans le tableau. On l'appelle aussi la taille du tableau ; il est recommandé de déclarer la taille du tableau comme une constante.

On accède à un élément du tableau en lui appliquant l'opérateur [] au nom de la variable.

Exemple :

```
1.  Algorithme Calcul_Moyenne
2.  /* Objectif : demande 10 notes au clavier et calcul leur moyenne */
3.  /* Données : 10 réels tapés au clavier */
4.  /* résultat : le calcul de la moyenne des réels */
5.  Constante :
6.      MAX = 10
7.  Type :
8.      Tnote =Tableau[MAX] de reel
9.  Variable :
10.     Tnote : Note
11.     Réel : Moy, Som
12.  Debut
13.     Som ← 0
14.     Pour i ← 0 (1) MAX-1 faire
15.         Ecrire ("Entrez la note numéro", i)
16.         Lire (Note[i])
17.         Som ← Som + Note[i]
18.     FinPour
19.     Moy ← Som / MAX
20.  Fin.
```

Remarque :

— l'indice peut être exprimé directement comme une constante entière, mais il peut être aussi une variable, ou une expression calculée.

— La taille d'un tableau ne peut être une variable.

— les cases du tableau sont numérotées de 0 à *nombre d'elements* - 1 ;

NB : De même qu'une variable possède toujours une valeur (qui peut être indéterminée), un tableau peut très bien posséder des éléments dont la valeur est indéterminée. Donc avant d'utiliser un élément du tableau, il est donc nécessaire de l'initialiser.

Exemple : Tableau multidimensionnel


```

1.  Algorithme Matrice_Nulle
2.  /* Objectif : initialiser une matrice à zéro*/
3.  Constante :
4.      L = 5
5.      C = 5
6.  Type :
7.      tMat = Tableau[L][C] d'entiers
8.  Variable :
9.      tMat : Mat
10.     Entier : i, j
11. Debut
12.     Som ← 0
13.     Pour i ← 0 (1) L-1 faire
14.         Pour j ← 0 (1) C-1 faire
15.             Mat[i][j] ← 0
16.         FinPour
17.     FinPour
18. Fin.

```

1.7 Exercices du chapitre 1

Exercice1 : Commentez l'algorithme suivant :

ALGORITHME Exo1

/*Objectif : ... */

Variable

Entier : val, double

Debut

val ← 231

double ← val * 2

Ecrire (val)

Ecrire (double)

Fin

Exercice2 : Qu'affiche l'algorithme suivant ?

ALGORITHME Exo2

/*Objectif : rappel sur l'opérateur NON logique*/

Variable

```

        Booleen : flag
    Debut
        flag ← VRAI
        Si NON(flag) alors
            Ecrire ("le SI est exécuté")
        sinon
            Ecrire ("le SINON est exécuté")
        finsi
    Fin

```

Exercice3 : Ecrire un algorithme qui permet d'entrer le prix unitaire hors taxe d'un article et le nombre de pièces de cet article. Cet algorithme affichera le prix total TTC correspondant (la TVA étant une constante).

Exercice4 : Ecrire un algorithme qui demande un nombre réel à l'utilisateur, puis qui calcule et affiche le carré de ce nombre.

Exercice5 : Ecrire un algorithme qui demande un nombre entier à l'utilisateur, et qui l'informe ensuite si ce nombre est "positif", "nul" ou "négatif".

Exercice6 : Ecrire un algorithme qui demande deux nombres entiers à l'utilisateur et qui, sans toutefois calculer le produit de ces nombres, l'informe ensuite si leur produit sera "négatif", "nul" ou "positif".

Exercice7 : Ecrire un algorithme qui permet à l'utilisateur d'entrer trois caractères de alphabet et l'informe ensuite s'ils ont été entrés ou non dans l'ordre lexicographique.

Exercice8 : Ecrire un algorithme qui demande le nombre de chiffre d'une unité de mesure et informe ensuite sur le préfixe de l'unité de mesure à lui accorder :

>Nombrechiffre ? 2 >deca-	>Nombrechiffre ? 4 >kilo-	>Nombrechiffre ? 5 >Je ne sais pas	>Nombrechiffre ? 7 >mega-	...
---------------------------------	---------------------------------	--	---------------------------------	-----

Exercice9 : Sachant qu'il est déconseillé de modifier l'indice d'itération d'une boucle « Pour » dans une instruction située à l'intérieur de la boucle, quelle est la

valeur affichée par l'extrait d'algorithme suivant :

```
...
nombre_fois ← 0
Pour i ← 1 (1) 5 faire
    nombre_fois ← nombre_fois + 1
    i ← i + 1
finPour
Ecrire (nombre_fois)
...
```

Corriger cet extrait afin d'afficher la valeur 5

Exercice10 : Utiliser la structure itérative adéquate, pour écrire un algorithme qui demande à un utilisateur d'entrer un entier autant de fois que cet entier est différent d'une constante **CTE** donnée. Au cas où cet entier est supérieur à **CTE**, on fera apparaître un message : « Essayez plus petit ! », et inversement, « Essayez plus grand ! » si l'entier est inférieur à **CTE**.

Exercice11 : Ecrire un algorithme qui demande un entier positif n et qui calcule la somme des n premiers entiers (les entiers de 1 à n) en utilisant la structure itérative adéquate.

Exercice12 :

1. Ecrire un algorithme qui demande à l'utilisateur de saisir **MAX** entiers positifs, et qui affiche ensuite quel était le plus grand parmi ces **MAX** entiers ; (ne pas utiliser les tableaux, **MAX** est une constante entière).
2. Réécrire l'algorithme précédent, mais cette fois-ci on ne connaît pas d'avance combien l'utilisateur souhaite saisir de nombres. La saisie des nombres s'arrête lorsque l'utilisateur entre un "-1".

Exercice13 :

1. Ecrire une fonction qui a comme argument un entier positif n , calcule de façon itérative sa factorielle et renvoie ce résultat.
2. Appeler la fonction précédente dans un algorithme pour afficher C_n^k , k et n

étant 2 entiers entrés au clavier.

Exercice14 : On considère la suite de Fibonacci définie par :

$$\begin{cases} F(0) = 0 \\ F(1) = 1 \\ F(n) = F(n-1) + F(n-2). \end{cases}$$

1. Ecrire une fonction qui permet de calculer le terme d'ordre n, $F(n)$ de cette suite.
2. Ecrire un programme qui affiche les n premiers nombre de Fibonacci, n étant entré au clavier.

Exercice15 : Extrait de l'Examen 2012

1. Ecrire une fonction d'entête **Entier : puissance(entier : n, p)** qui calcule n^p de façon suivante :

(a) Itérative : $n^p = \underbrace{n * n * \dots * n}_{p \text{ fois}}$.

(b) Récursive : $n^p = \begin{cases} 1 & \text{si } p = 0 \\ n * n^{p-1} & \text{sinon} \end{cases}$

2. Soit l'algorithme suivant :

ALGORITHME ALGO

/*Objectif : à déterminer */

Variable

Entier : n, b, nbre, a, i

Début

Ecrire (entrer 2 entiers positifs)

Lire (nbre, b)

$n \leftarrow \text{nbre}$

$i \leftarrow 0$

$a \leftarrow 0$

Tantque (n>0) faire

$a \leftarrow a + (n \bmod b) * \text{puissance}(10, i)$

$n \leftarrow n \text{ div } b$

$i \leftarrow i + 1$

finTantque

Ecrire (a)

fin

(a) Quelle est la valeur affichée si :

i. $\text{nbre} = 7$ et $b = 2$?

ii. $\text{nbre} = 8$ et $b = 2$?

(b) En déduire l'objectif de l'algorithme ALGO ?

Exercice16 : L'algorithme d'Euclide permet de calculer le pgcd de 2 nombres entiers a et b grâce à la formule de récurrence suivante : $\text{pgdc}(a,b) = \text{pgdc}(b,r)$ avec r le reste de la division euclidienne de a par b ; le pgdc est le dernier reste non nul.

1. Ecrire de la manière suivante une fonction $\text{pgdc}()$ qui renvoie le pgdc de 2 entiers a et b ($0 < b \leq a$), passés en argument :

(a) récursive ;

(b) itérative.

2. Ecrire un algorithme qui permet d'entrer deux entiers positifs et affiche s'ils sont premiers entre eux, ou non. Cet algorithme appelle la fonction précédente.

Exercice17 : Ecrire un algorithme permettant de lire un entier multiple de 5 (cet entier représente un montant) et simule la remise de la monnaie en coupure du Fcfa (ne pas remettre un grand nombre de pièces d'une coupure précise sachant qu'il est possible de remettre un pièce d'une coupure supérieure).

> *Entrer le montant :*

9590

> 1 de "5000F"

2 de "2000F"

1 de "500F"

1 de "50F"

1 de "25F"

1 de "10F"

1 de "5F"

>

Indication : utiliser une fonction (que vous aurez préalablement écrit) d'entête
Entier : FONCTION NbCoupure (Entier : m, c) qui renvoie le nombre de coupure de **c** Fcfa du montant **m**.

Exercice18 : Extrait de CC 2013

On dispose des coupures en nombre illimité des billets de valeurs (V_1, V_2, \dots, V_N) où $V_1 < V_2 < \dots < V_N$. L'objectif étant de rendre la monnaie d'un montant **m** avec le minimum de billets possible.

Pour y arriver, nous utilisons les déclarations suivantes :

CONSTANTE :

$N = 10$

TYPE :

tTab = TABLEAU [N] d'entier

VARIABLE

tTab : V //tableau des coupures des billets par ordre croissant

tTab : Tab //ce tableau est utilisé à la 4^e question ...

1. En s'inspirant de l'exercice précédent, écrire une fonction d'entête

Entier : FONCTION nombre(tTab V,
Entier : m, n)

qui renvoie le nombre "minimal" de billets du montant m à remettre quand on a les n premières coupures du tableau V .

Exemple : nombre(V, 9590, 10) = 8,

où $V = [5, 10, 25, 50, 100, 500, 1000, 2000, 5000, 10000]$.

2. Dérouler puis donner la valeur de "nombre(V, m, 4)" dans les cas de figure suivants :

(a) $V = [1, 100, 500, 1000, 5000, \dots]$, $m = 2400$;

(b) $V = [1, 2, 5, 10, \dots]$, $m = 9$;

(c) $V = [1, 4, 6, 10, \dots]$, $m = 9$.

3. Peut-on déduire que l'objectif de notre fonction est atteint ?

4. (a) Écrire la fonction d'entête

**Entier : FONCTION minimum(tTab Tab,
entier n)**

qui renvoie l'entier le plus petit des **n** premiers éléments d'un tableau **Tab**.

(b) Soient les fonctions suivantes :

Entier : FONCTION f(tTab V,
Entier : t, i)

Début

Si ($i < 0$ OU $t \leq 0$) alors

Renvoyer 0

Fsi

Si ($V[i] \leq t$) alors

Renvoyer ($f(V, t - V[i], i) + 1$)

Sinon

Renvoyer $f(V, t, i - 1)$

Fsi

Fin

Entier : FONCTION nombre2(tTab : V
Entier : m, n)

Variable :

tTab : Tab

Entier : i, j, nb

Début

Pour $i \leftarrow 0$ (1) $n - 1$ faire

Pour $j \leftarrow 1$ (1) m faire

$nb \leftarrow f(V, j, i)$

finPour

Ecrire (nb)

$Tab[i] \leftarrow nb$

finPour

Renvoyer (minimum(Tab, n))

Fin

Qu'affiche la fonction "nombre2(V, m, n)" dans les cas de figure précédents ?
L'objectif initial est-il atteint ?

Chapitre 2

Les pointeurs

2.1 Généralités

Toute variable manipulée dans un programme est stockée quelque part en mémoire centrale. Cette mémoire est découpée en octet identifié par un numéro séquentiel appelé **adresse**.

Par convention, une adresse est notée en hexadécimal et précédée par 0x.

Déclarer une variable, c'est attribuer un nom à une zone de la mémoire. Cette zone est définie par :

- sa position, c'est-à-dire l'adresse de son premier octet ;
- sa taille, c'est-à-dire le nombre d'octets.

Exemple 1 : le code suivant est représenté à la figure 2.1 :

Variable

Entier : n

Debut

n ← 8

fin

Pour accéder à la **valeur** contenue dans une variable, on utilise son nom (ici **n**).

La position de la variable **n** est **0x3C29** et sa taille est **4 Octets**.

Un pointeur est une variable qui contient l'adresse d'une autre variable.

L'adresse contenue dans un pointeur est celle d'une variable appelée **variable pointée**. On dit que le pointeur pointe sur la variable dont il contient l'adresse.

Un pointeur est associé à un type de variable sur lequel il peut pointer. En effet, même si la valeur d'un pointeur (une adresse) est toujours un entier (représenté en

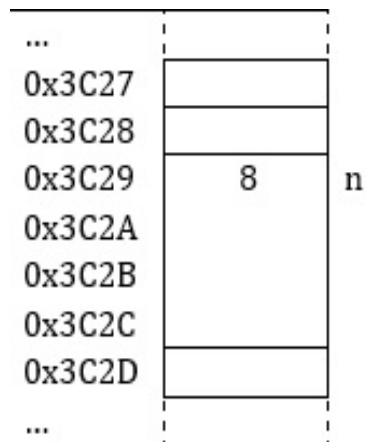


FIGURE 2.1 – Représentation de l'exemple 1.

hexadécimal), le type d'un pointeur dépend du type de la variable pointée. Ainsi, un pointeur vers un entier ne peut pointer que sur les entiers.

Un pointeur est lui-même une variable et à ce titre, il possède une adresse.

2.1.1 Déclaration et manipulation d'un pointeur

On déclare une variable pointeur par l'instruction suivante :

↓ typeVariablePointée nomPointeur

Où typeVariablePointée est le type de l'objet pointé.

Exemple :

Variable :

↓ entier pe //pe est un pointeur vers un entier

↓ caractere pc //pc est un pointeur vers un caractère

Pour manipuler les pointeurs, nous avons besoin des opérateurs suivants :

— **adresse de**, noté @

— **contenu de**, noté ©

L'opérateur "**adresse de**" permet d'accéder à l'adresse d'une variable. Sa syntaxe est :

@nomVariable

Ainsi, @n vaut 0x3C29 (adresse du premier octet de la variable n).

Cette adresse peut être utilisée pour initialiser la valeur d'un pointeur.

L'opérateur "adresse de" peut seulement être appliqué à des variables. Il ne peut être appliqué à des constantes.

L'opérateur "**contenu de**" (appelé encore opérateur **d'indirection**) permet d'accéder directement à la valeur de la variable pointée (on dit qu'on **déréférence** un pointeur). Déréférencer un pointeur consiste à extraire la valeur de la variable sur laquelle il pointe.

La syntaxe est la suivante :

$$\textcircled{c} \textit{nomPointeur}$$

Ainsi, si p est un pointeur vers un entier n, $\textcircled{c}p$ désigne la valeur de n. On peut aussi modifier la valeur d'une variable pointée à travers un pointeur déréférencé.

Exemple 2 :

Variable :

↓ Entier : pe //pe est un pointeur vers un entier
Entier : n ;

Début

n ← 10
p ← @n // p contient l'adresse de n
Ecrire ($\textcircled{c}p$) // affiche 10

fin

La figure 2.2 illustre la mémoire juste avant l'instruction de "fin" de l'exemple 2 :

Nous avons alors :

- n : le contenu de n (soit 10)
- @n : l'adresse de n (soit 0x3C2B)
- p : le contenu de p (soit 0x3C2B)
- $\textcircled{c}p$: le contenu de n (soit 10)
- @p : l'adresse de p (soit 0x1A40)
- **$\textcircled{c}n$ est erroné car n n'est pas un pointeur.**

2.1.2 Arithmétique des pointeurs

La valeur d'un pointeur étant un entier, on peut lui appliquer les opérateurs arithmétiques suivants :

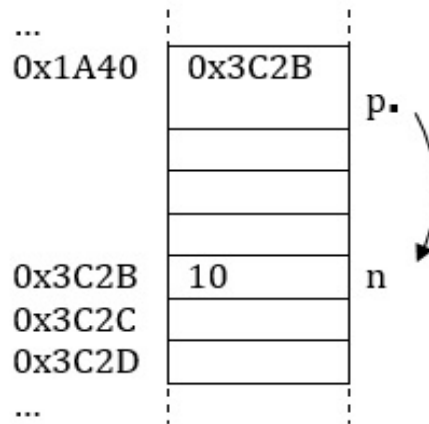


FIGURE 2.2 – Exemple 2.

- l'addition d'un entier i à un pointeur p ($p + i$) : le résultat est un pointeur de même type que le pointeur p .
- la soustraction d'un entier i à un pointeur p ($p - i$) : le résultat est un pointeur de même type que le pointeur p .
- la différence de 2 pointeurs ($p1 - p2$), il faut absolument que les deux pointeurs pointent vers des objets de même type T : Le résultat est un entier dont la valeur est égale à : $(p1 - p2) \text{ div } \text{nbOctet}(T)$.

NB : Soit i un entier et p un pointeur sur un objet de type T (donc déclaré par l'instruction $T : \downarrow p$;). Alors $p+i$ (respectivement $p-i$) désigne un pointeur sur un objet de type T . Sa valeur est égale à celle de p incrémentée (respectivement décrémentée) de $i * \text{nbOctet}(T)$.

2.1.3 Initialisation d'un pointeur

Comme toute variable, un pointeur doit être initialisé. Cette initialisation peut s'effectuer de trois différentes façons :

- Affectation de p à l'adresse d'une autre variable
Si la variable est un pointeur, on peut faire l'affectation directement, sinon on doit utiliser l'opérateur "adresse de".
- Affectation de p à la valeur NIL
On peut dire qu'un pointeur ne pointe sur rien en l'initialisant à NIL (Not

Identified Link).

Attention : Ne jamais déréférencer un pointeur nul. Avant de déréférencer un pointeur, il faut toujours se rassurer qu'il n'est pas nul.

Exemple :

...

Variable :

↓Entier : p1, p2, p3

Entier : n //supposons que n se trouve à l'adresse 0x3C2B

Début

n ← 10

p1 ← @n //affectation de l'adresse de n à p1

p2 ← p1 //affectation de p2 à p1 : p2 contient aussi l'adresse de n

p3 ← NIL //affectation NIL à p3 : p3 est un pointeur nul

fin

- Affectation directe de "contenu de" p (la zone mémoire pointé par p)
Pour cela, il faut d'abord réserver à ©p un espace mémoire de taille adéquate (celui du type pointé par p). L'adresse de cet espace mémoire sera la valeur de p. Cette opération consistant à réserver un espace mémoire pour stocker la variable pointée s'appelle **une allocation dynamique**.

2.1.4 Double indirection

Quand un pointeur pointe sur un autre pointeur, il y a double indirection (figure 2.3) :

On peut accéder à n par p1 en utilisant deux fois l'opérateur de déréférencement :

©©p1 = ©p2 = n.

On pourrait imaginer de la même façon des indirections triples, voire quadruples.

2.1.5 Allocation dynamique de la mémoire

Il y a deux façons d'allouer un emplacement mémoire pour une variable :

- soit par une déclaration, telle qu'on le fait dans la déclaration des variables ;
- soit par une **allocation dynamique**.

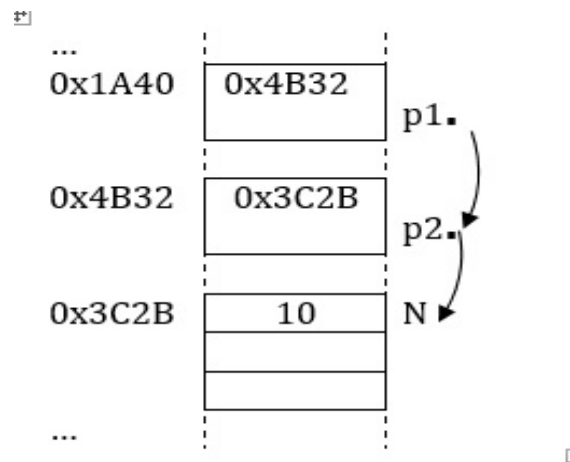


FIGURE 2.3 – Double indirection

La déclaration permet d'allouer un emplacement dans une partie de la mémoire appelée pile. Les variables déclarées dans la pile possèdent un nom. Elles sont créées dès l'entrée dans la **portée** et détruites dès la sortie de leur portée.

L'allocation dynamique permet d'allouer un emplacement dans une autre partie de la mémoire appelé **tas** (ensemble d'octets sans structure particulière, dans lequel on cherche à allouer ou à libérer de l'espace en fonction des besoins). Les variables du tas ne sont pas déclarées, elles sont créées par une instruction spécifique. Elles ne portent pas de nom mais on peut y accéder par leur adresse. Elles peuvent être utilisées partout et jusqu'à leur destruction (il n'y a pas de portée pour les variables du tas).

Fonctions d'allocation dynamique

L'opération consistant à réserver un espace mémoire est réalisée par la fonction "**allouer()**" qui prend en argument la taille de l'espace qu'on veut réserver. Cette taille est donnée par la fonction "**nbOctet()**" qui prend en argument le type de la variable qu'on veut créer. La fonction **allouer()** renvoie l'adresse de la variable qu'on affecte à un pointeur du même type.

Sa syntaxe est la suivante :

nomPointeur ← **allouer(nbOctet(typeVariablePointée))**

Exemple :

Variable :

↓reel p

Début

p ← allouer(nbOctet(réel))

Fin

La fonction "**allouer()**" alloue dans le tas un emplacement ayant la taille du type précisé en argument et elle fournit en retour l'adresse correspondante que nous affectons ici à la variable p. Ensuite, on peut directement accéder à cet emplacement en déréférençant le pointeur : ©p ← 18,5

La fonction "**allouer()**" permet également d'allouer un espacement pour plusieurs variables :

Exemple :

...

Variable :

↓reel p

Debut

p ← allouer(2*NbOctet(réel)) //allocation pour 2 reels

©p ← 14

©(p+1) ← 10 //p+1 est un pointeur de même type que p

...

End

Fonction de libération dynamique

Lorsqu'on n'a plus besoin de l'espace alloué dynamiquement par une fonction d'allocation, il faut libérer cet espace. Ceci se fait à l'aide de la syntaxe suivante :

désallouer(nomPointeur)

Exemple :

...

Début

...

desallouer(p) //cette fonction ne renvoie rien.

Fin

Nous avons libéré l'emplacement alloué précédemment ; cet emplacement ainsi rendu disponible pourra éventuellement être utilisé lors d'une prochaine demande

d'allocation dynamique.

Remarques :

- La fonction de libération dynamique ne supprime pas le pointeur mais la variable pointée ;
- Ne jamais utiliser la fonction de libération dynamique sur un pointeur dont l'espace mémoire n'a pas été alloué.
- Si la mémoire n'est pas libérée à l'aide de la fonction de libération dynamique, alors elle l'est automatiquement à la fin du programme. Mais il faut toujours libérer l'espace mémoire alloué.

2.2 Application des pointeurs.

2.2.1 Sur les fonctions

Contrairement aux fonctions mathématiques, il est possible de changer la valeur de plusieurs variables à l'appel d'une fonction en informatique ; ceci grâce aux pointeurs.

Exemple :

ALGORITHME Permutation

/* Objectif : permuter le contenu de 2 variables initialisées */

Variable :

Entier : x,y

FONCTION echange (↓entier : px, py)

/*Objectif : permuter le contenu de 2 variables passées en paramètre */

variable :

Entier : temp ;

Début

temp ← ©px //temp prend la valeur de la variable pointée par px

©px ← ©py /* la variable pointée par px prend pour valeur celle pointée par py */

©py ← temp // la variable pointée par py prend pour valeur temp

Fin

Debut

x ← 10


```

y ← 20
echange (@x, @y)      //px pointe sur x et py pointe sur y
Fin

```

2.2.2 Sur les tableaux

Une variable de type tableau correspond en fait à un pointeur vers le premier élément du tableau. Ce pointeur est **constant**, ce qui implique en particulier qu'aucune opération globale n'est autorisée sur un tableau. Notamment, un tableau ne peut pas figurer à gauche d'une affectation : on ne peut pas écrire $\text{tab1} \leftarrow \text{tab2}$, tab1 et tab2 étant les variables de type tableau. Il faut effectuer l'affectation pour chacun des éléments du tableau.

Constante

```
MAX = 5
```

Type

```
tTab = Tableau[MAX] d'entier
```

Variable :

```
tTab : tab
```

```
Entier : i
```

```
↓Entier : p
```

Début

```
Pour i ← 0 (1) MAX-1 faire
```

```
    tab[i] ← i*i
```

```
finPour
```

```
...
```

Fin

Ainsi, la déclaration `tTab : tab` définit un tableau de MAX entiers et `tab` est un pointeur constant (non modifiable) dont la valeur est l'adresse du premier élément du tableau. Autrement dit, `tab` a pour valeur `@tab[0]` (ie 0x1A40). On peut donc utiliser un pointeur initialisé à `tab` pour parcourir les éléments du tableau.

Comme on accède à l'élément d'indice i du tableau `tab` par l'expression `tab[i]`, on peut déduire la relation entre `[]` et `Ⓒ` suivante :

$$\text{tab}[i] = \text{Ⓒ}(\text{tab} + i)$$

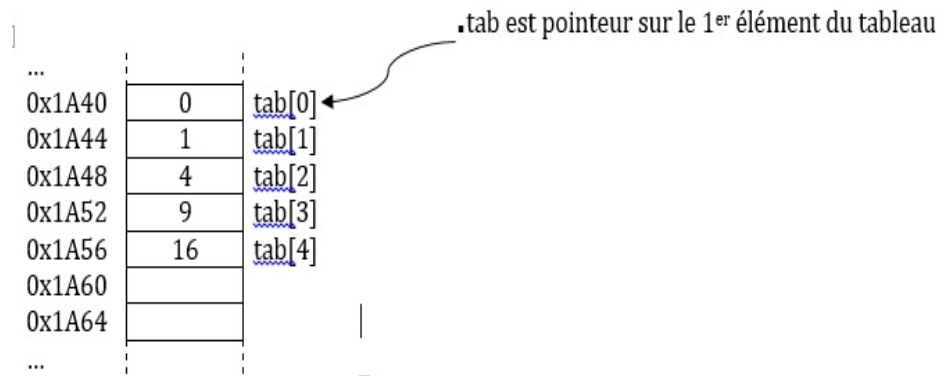


FIGURE 2.4 – tableau et pointeur

Exemple :

Début

```

...
p ← tab
Pour i ← 0 (1) MAX-1 faire
    Ecrire (©p)
    p ← p+1
finPour

```

fin

On a alors les égalités suivantes :

- $\text{tab} + 1 = @(\text{tab}[1])$ (adresse du 2ème élément)
- $\text{©}(\text{tab} + 1) = \text{tab}[1]$ (contenu de la 2ème case)
- $\text{©tab} = \text{tab}[0]$ (contenu du 1er élément)
- $\text{©}(\text{tab} + i) = \text{tab}[i]$ (contenu du (i+1)ème élément)
- $\text{tab} + i = @(\text{tab}[i])$ (adresse du (i+1)ème élément)

Pointeurs et tableaux se manipulent donc exactement de la même manière. Attention cependant puisqu'aucun contrôle n'est effectué pour s'assurer que l'élément du tableau adressé existe effectivement. En outre, la manipulation de tableaux possède certains inconvénients par rapport à la manipulation des pointeurs dû au fait qu'un tableau est un pointeur constant ; on ne peut donc pas créer de tableaux :

- dont la taille est une variable du programme ;
- bidimensionnels dont les lignes n'ont pas toutes le même nombre d'éléments.

Ces opérations deviennent possibles dès que l'on manipule des pointeurs alloués

dynamiquement.

Gestion des tableaux dont la taille est une variable du programme

En pseudo code, et dans la plupart des langages de programmation, la taille d'un tableau doit être connue dans la partie déclaration. Ce qui pose un problème quand on veut que cette taille ne soit connue qu'au moment de l'exécution de l'algorithme. Grâce aux possibilités qu'offre l'allocation dynamique, ce problème est résolu tel que le montre l'exemple suivant :

Exemple :

...

Variable :

↓Reel : p

Entier : n

Début

Ecrire ("entrer le nombre d'éléments ? ")

Lire (n)

$p \leftarrow \text{allouer}(n * \text{nbOctet}(\text{réel}))$ //allocation pour un tableau de n réels

...

desallouer (p)

Fin

Un élément de rang i de ce tableau pourra alors se noter indifféremment p[i] ou ©(p+i).

Gestion des tableaux bidimensionnels dont les lignes n'ont pas toutes le même nombre d'éléments

Avec l'allocation dynamique, il est possible de créer les tableaux à 2 dimensions dont les lignes n'ont pas toutes la même longueur. Un effet un tableau à 2 dimensions peut non seulement être vu comme un pointeur sur une zone mémoire de taille le produit des 2 dimensions ($\text{Mat} = @\text{Mat}[0][0]$), mais aussi comme un tableau de pointeur. Chaque élément du tableau est lui-même un pointeur sur un tableau.

On déclare un pointeur qui pointe sur une variable de type ↓typeVariablePointé de la même manière qu'un pointeur, c'est-à-dire par la déclaration suivante :

⇓⇓typeVariablePointé p

Exemple :

ALGORITHME MatriceInf

/*Objectif : allouer exactement l'espace pour la partie inférieure d'une matrice
carrée*/

Variable :

Entier : n, i

⇓⇓Reel : p //p est du type pointeur vers un pointeur vers un reel

Début

Ecrire ("Entrer le nombre de colonnes : ")

Lire (n)

p ← allouer(n*nbOctet(↓réel)) //allocation pour un tableau de pointeurs

Pour i ← 0 (1) n-1 faire

p[i] ← allouer((i+1)*nbOctet(reel)) /* allocation pour chacune des
lignes en plaçant leur adresse
dans la case correspondante */

finPour

...

Pour i ← 0 (1) n-1 faire

desallouer(p[i])

FinPour

desallouer(p)

Fin

2.3 Compléments sur les tableaux

2.3.1 Passage de tableau en argument

Puisqu'une variable de type tableau correspond à l'adresse du premier élément du tableau, c'est cette adresse qui est passée en argument. On peut donc au besoin modifier le contenu de cette variable.

ALGORITHME Incrémentation

/*Objectif : incrémente et affiche les valeurs d'un tableau*/

Constante :

MAX = 5

Type :

tTab = Tableau[MAX] d'entier

Variable :

tTab : tab

Entier : i

FONCTION incr_tab (↓Entier : t,
Entier : n)

/*Objectif : incrémente les éléments d'un tableau*/

Variable :

Entier : i

Début

Pour i ← 0 (1) n-1 faire

t[i] ← t[i]+1

FinPour

Fin

Début

Pour i ← 0 (1) MAX-1 faire

tab[i] ← i

FinPour

incr_tab (tab, MAX)

Pour i ← 0 (1) MAX-1 faire

Ecrire (tab[i])

FinPour

Fin

2.3.2 Les tableaux de caractères : les chaines de caractères

Une chaine de caractères est un tableau dont les éléments sont de type caractères.

Une chaine de caractères est donc déclarée comme tout autre tableau.

Constante :

MAX = 3

Type :

tChaine = Tableau[MAX + 1] de caractères

Variable :

tChaine : ch

La représentation interne d'une chaîne de caractères est terminée par le symbole nul '\0'. Vous n'avez pas besoin de l'ajouter explicitement. Ainsi, pour un texte de N caractères, il faut prévoir $N + 1$ octets.

Un tableau de caractères peut être initialisé comme suit :

ch[0] ← 'a', ch[1] ← 'b', ch[2] ← 'c' ;

Cela sous-entend que ch[3] = '\0'.

Cependant, ceci devient très vite lourd quand on veut manipuler les mots quelconques. D'où, on peut aussi initialiser un tableau de caractère par une chaîne littérale, à l'exemple de : ch ← "abc" ;

Les opérations possibles sur les chaînes de caractères en pseudo-code sont :

— l'opération d'affectation : ← ;

— les opérations de comparaison pour l'ordre alphabétique : <, >, =, ...

Exemple : "ps" > "code".

— l'opération de concaténation : notée &

Exemple : "pseudo"&"code" = "pseudocode"

Nous avons également plusieurs fonctions prédéfinies sur les chaînes de caractères :

— le nombre de caractère effectif d'une chaîne : **longueur()**, prend en argument une chaîne de caractère et renvoie un entier ;

Exemple : longueur("pseudo code") = 11

longueur("") = 0

— l'extraction d'une sous-chaîne : **sousChaîne()**, prend en arguments une chaîne de caractère et deux entiers n et l pris dans cet ordre. Cette fonction renvoie un extrait de la chaîne, commençant au caractère d'indice $n - 1$ et faisant l caractères de long.

Exemple : sousChaîne("pseudo code", 4, 5) = "udo c"

2.3.3 Une technique usuelle : Le principe du flag

Le flag, en anglais, est un petit drapeau, qui va rester baissé aussi longtemps que l'événement attendu ne se produit pas. Et, aussitôt que cet événement a lieu, le petit drapeau se lève (la variable booléenne change de valeur). Ainsi, la valeur finale de la variable booléenne permet au programmeur de savoir si l'événement a eu lieu ou non.

Exemple :

booléen : FONCTION trouver(Entier n)

/*Objectif : rechercher si l'entier n se trouve dans un tableau Tab de 10 entiers */

/*Argument : la valeur n à rechercher, supposons que Tab est une variable globale/

Variable : Booléen : flag

Entier : i

Debut

flag ← FAUX

i ← 0

TantQue(NON(flag) ET i<10) faire

Si (Tab[i] = n) alors

 flag ← VRAI

Fsi

 i ← i+1

FinTantQue

Renvoyer(flag)

Fin

2.4 Exercices du chapitre 2

Exercice 1 : Déclarer le pointeur p dans les algorithmes suivants ?

<u>Algorithme Question1</u> /*Objectif : déclarer p*/ <u>Variable :</u> reel : x _____ : p <u>Debut</u> x ← 19,5 p ← @x <u>Fin</u>	<u>Algorithme Question2</u> /*Objectif : déclarer p*/ <u>Variable :</u> ↓entier : q _____ : p <u>Debut</u> p ← @q <u>Fin</u>	<u>Algorithme Question3</u> /*Objectif : déclarer p*/ <u>Variable :</u> _____ : p ↓caractère : q <u>Debut</u> ... p ← q+3 <u>Fin</u>
--	---	--

Exercice 2 : soit la déclaration de variable (à gauche) supposée être représenté en mémoire (à droite). Qu'affiche le code suivant ?

<u>Variable :</u>	...		
entier : n	0x1A40		
↓entier : p	0x1A44		n
↓entier : q	0x1A48		
↓entier : r	0x1A52		p
	0x1A56		q
	0x1A60		r
	...		

Debut

```

n ← 5
p ← @n
q ← p
r ← @p
Ecrire("1. " ©p)
Ecrire("2. " @p)
Ecrire("3. "p)
©p ← 8
Ecrire("4. "n)
Ecrire("5. " ©q)
Ecrire ("6. " @©r)
Ecrire ("7. " ©©r)

```

Fin

Exercice 3 : Soient p1 et p2, deux pointeurs pointant vers des réels initialisés respectivement à 14,5 et -5,75. Ecrire l'algorithme ; cet algorithme affiche à la fin les valeurs, les contenus et les adresses des deux pointeurs.

Exercice 4 : soit à gauche un algorithme supposé être exécuté en mémoire de la manière représentée à droite.

<u>Constante</u>		
MAX = 5		
<u>Type</u>		
tTab = Tableau[MAX] d'entier		
<u>Variable</u>		
tTab : tab		
entier : i		
<u>Debut</u>		
Pour i ← 0 (1) MAX-1 faire		
tab[i] ← i*i		
finPour		
<u>Fin</u>		

...		
0x1A40	0	tab[0]
0x1A44	1	tab[1]
0x1A48	4	tab[2]
0x1A52	9	tab[3]
??	16	tab[4]
0x1A60		
0x1A64		
...		

Donner (si possible) le type et la valeur des variables suivantes :

1. var1 = tab + 1
2. var2 = ©(tab + 2)
3. var3 = ©tab + 3
4. var4 = @(tab[4])

Exercice 5 : Donner en illustrant les différents affichages de l'algorithme suivant :

Algorithme Exo5

/*objectif : appeler les fonctions*/

Variable :

entier : a

FONCTION f1(entier : a)

/*objectif : incrementer*/

Debut

a ← a+1

Fin

FONCTION f2(↓entier : p)

```

/*objectif : incrementer*/
Debut
    ©p ← ©p+1
Fin
FONCTION f3(entier : b)
/*objectif : incrementer*/
    b ← b+1
Fin
Debut
    Debut
        a ← 2
        f1(a)
        ecrire("après f1 :", a)
    Fin
    Debut
        a ← 2
        f2(@a)
        ecrire("après f2 :", a)
    Fin
    Debut
        b ← 2
        f3(b)
        ecrire("après f3 :", b)
    Fin

```

Fin

Exercice 6 : Donner en illustrant les différents affichages de l’algorithme suivant :

Algorithme Exo6

/*objectif : appeler les fonctions*/

Variable

Entier : n1, n2, n3

FONCTION f1(entier a,
 ↓entier b)

/*objectif : affecter */

```

Debut
    a ← ©b
Fin

FONCTION f2(↓entier b,
              entier c)
/*objectif : affecter */
Debut
    ©b ← c
Fin

FONCTION f3(↓entier a,
              entier b)
/*objectif : affecter */
Debut
    ©©a ← b
Fin

FONCTION f4(↓entier a,
              entier b)
/*objectif : appeler f3 */
Debut
    f3(@a,b)
Fin

Debut
    n1 ← 2
    n2 ← 4
    n3 ← 6
    f1(n1, @n2)
    Ecrire("après f1 :" n1, n2, n3)
    f2(@n3, ©(@n1))
    Ecrire("après f2 :" n1, n2, n3)
    f4(@n3,n2)
    Ecrire("après f4 :" n1, n2, n3)
Fin

```

Exercice 7 : Écrire un algorithme qui permet de saisir MAX entiers (MAX est

une constante entière); l'algorithme les range d'abord dans un tableau de taille MAX. L'algorithme demande ensuite à l'utilisateur d'entrer un entier n ($0 < n \leq \text{MAX}$) et affiche le $n^{\text{ième}}$ élément du tableau ainsi que son adresse.

Exercice 8 : Reprendre l'exercice précédent, mais avec le nombre d'entiers saisi (ie la taille du tableau) qui est un entier demandé à l'utilisateur.

Exercice 9 : soient les déclarations et la fonction suivantes (où CONDITION est à spécifier) :

Constante

MAX = 100

type

tTab = TABLEAU[MAX]d'entier

Booleen FONCTION Exo9 (tTab tab,

entier n)

/*objectif : vérifier que les éléments du tableau tab sont consécutifs */

Variable

Booléen : flag

Entier : i

Debut

flag \leftarrow VRAI

Pour i \leftarrow 1 (1) n-1 faire

Si CONDITION alors

flag \leftarrow FAUX

Finsi

Finpour

Renvoyer flag

Fin

1. Spécifier la condition CONDITION afin que l'algorithme atteigne son objectif?
2. La fonction Exo9 a le défaut d'examiner la totalité du tableau, même lorsque dès le départ, on a deux éléments non consécutifs. Proposer une correction qui respecte le "principe du flag".

Exercice 10

1. Écrire une fonction qui calcule la combinaison de p dans n en utilisant la

formule suivante : $C_n^p = C_{n-1}^{p-1} + C_{n-1}^p$

2. Écrire un algorithme qui utilise cette fonction afin d'afficher le triangle de pascal sur n lignes, n étant un entier saisi au clavier.

NB : Le triangle de pascal n'utilise que la partie inférieure d'une matrice carrée, ne pas gaspiller l'espace mémoire.

Exercice 11 Écrire un algorithme qui

- permet d'initialiser un tableau à deux dimensions de 3 lignes et 5 colonnes ;
- appelle une fonction préalablement définie (la fonction prend ce tableau en argument et renvoie la plus grande valeur du tableau) ;
- affiche la valeur renvoyée par la fonction.

Exercice 12 l'objectif de l'algorithme de cet exercice est d'afficher le produit de deux matrices carrées. Pour y arriver, nous proposons le découpage en trois fonctions suivantes à définir

- initialiser(??) qui initialise les deux matrices carrées ;
- calculer(?) qui calcule leur produit et renvoi un résultat ;
- afficher(??) qui affiche le résultat ;

afin que le programme principal les appelle seulement.

Exercice 13 Écrire un algorithme qui permet d'initialiser un tableau de moins de 26 caractères (chaîne de caractères de longueur 25 - prévoir un caractère pour '\0') ; l'algorithme affiche le rang et l'adresse du caractère de fin de chaîne '\0'.

Exercice 14 Écrire une fonction qui renvoie le nombre de voyelles contenu dans une chaîne de caractères passée en argument. Utiliser cette fonction dans un programme principal.

Exercice 15 Ecrire une fonction qui purge une chaîne d'un caractère, la chaîne et le caractère étant passés en argument. Si le caractère spécifié ne fait pas partie de la chaîne, celle-ci devra être retournée intacte.

Exemple : `Purge("Bonjour","o")` renverra "Bnjur"

`Purge("J'ai horreur des espaces"," ")` renverra "J'aihorreurdesespaces"

`Purge("Ca purge comme ca purge","y")` renverra "Ca purge comme ca purge"

Exercice 16 Même question que précédemment, mais cette fois, on doit pouvoir fournir un nombre quelconque de caractères à supprimer en argument.

Exercice 17 Ecrire la fonction `Trouve(chaîne1,chaîne2)` qui renvoie un nombre correspondant à la position de chaîne2 dans chaîne1. Si chaîne2 n'est pas com-

prise dans chaîne1, la fonction renvoie zéro.

Exemple : Trouve("Un pur bonheur", "pur") vaut 4

Trouve("Un pur bonheur", "techno") vaut 0

Exercice 18 : Recherche des algorithmes de tri

1. Rechercher sur Internet et donner le principe des méthodes de tri suivantes :
 - Tri bulle
 - Tri par sélection
 - Tri par insertion
 - Tri fusion
 - Tri rapide
2. Ecrire les fonctions de tris précédents afin de les appeler dans un algorithme de la façon suivante : *tri_X (tab, MAX)*, tab étant un tableau de MAX entiers initialisé avant cet appel.

Exercice 19 : Recherche dichotomique

Rechercher sur Internet le principe de la recherche dichotomique et écrire deux fonctions (l'une itérative et l'autre récursive) qui recherche un mot saisi au clavier dans un dictionnaire. Le dictionnaire est supposé être codé dans un tableau préalablement rempli, trié et envoyé en argument.

Exercice 20 : Extrait de l'examen de rattrapage 2013-2014

Soit le tableau [8,5,2,3,7] sur lequel un algorithme précis de tri a été utilisé pour le trier par ordre croissant. On vous donne toutes les modifications du tableau jusqu'à l'obtention du résultat final : $\rightarrow [2,5,8,3,7] \rightarrow [2,3,8,5,7] \rightarrow [2,3,5,8,7] \rightarrow [2,3,5,7,8]$.

1. Quel est l'algorithme de tri qui a été utilisé ?
2. En déduire toutes les différentes modifications du tableau [3,4,2,2,1] lors de son tri par ordre croissant avec le même algorithme.

Exercice 21 : Soit la déclaration de la fonction fct suivant :

<u>Constante :</u>
MAX = 256
<u>Type :</u>
Tchaine = Tableau[MAX+1] de caractère
<u>Variable :</u>
Tchaine : A,B,C
<u>Fonction</u> fct(Tchaine A, B, C)
//Objectif : concaténer
<u>Debut</u>
A ← A & "VA"
B ← B & "TC"
C ← "ON" & A
<u>Fin</u>

Supposons l'initialisation suivante :

A ← "RA";

B ← "CIL";

C ← "B";

Qu'affiche les extraits d'algorithmes suivants :

Remarque : l'affichage n'est pas nécessairement significatif :

1. fct(A,B,C); Ecrire(A&B&C).
2. fct(C,A,B); Ecrire (sousChaine(A&B&C, 4,3)).
3. fct(B,C,sousChaine(A&B&C, 1,3)); Ecrire(A&B&C).

Exercice 22 : Extrait du CC 2017

Soit la déclaration suivante représentant un tableau de bits.

<u>Type</u>
tbit = Enumeration{0,1}
<u>Variable</u>
↓ tbit : ch1, ch2
entier : n, m, s, nb

1. Écrire la partie exécution de l'algorithme principal qui :
 - lit 2 entiers au clavier n et m et utilise la fonction permutate() si $n > m$;
 - appelle 2 fois la fonction read() pour initialiser ch1 (resp. ch2) avec n (resp. m) bits;
 - appelle la fonction match() pour afficher le nombre de fois que ch1 apparaît dans ch2.

2. Écrire les fonctions suivantes :

- (a) **permute()** qui permute les valeurs de n et m passés en argument ;
- (b) **read()** qui permet d'initialiser un nombre binaire (ch) de n bits, n passé en l'argument. Cette fonction utilise la fonction prédéfinie *Random()* qui renvoie aléatoirement un entier appartenant à l'intervalle $[0, +\infty[$.
Exemple : l'instruction $n \leftarrow Random()$; affecte à la variable n un entier compris dans l'intervalle $[0, +\infty[$; il va falloir ramener n dans $\{0, 1\}$
- (c) **is_equal()** qui renvoie VRAI si 2 nombres binaires de longueurs égales, passés en argument sont égaux ; *is_equal()* renvoie FAUX sinon.
- (d) **match()** qui consiste ici à trouver le nombre d'occurrence **nb** des positions s tel que $ch1[i] = ch2[i + s]$ (avec i parcourant $ch1$). Cette fonction utilise une seule boucle itérative et la fonction **is_equal()** précédente. $ch1$ et $ch2$ sont parmi les arguments de la fonction.
Exemple : Si $ch1=0001$ et $ch2=000010001010001$ alors *match()* renvoie "3"
- (e) **first_match()** qui modifie **match()** de manière à ce qu'elle renvoie la première position s du motif $ch1$ dans $ch2$; elle renvoie -1 si $ch1$ n'apparaît pas dans $ch2$.
Exemple : Sur l'exemple précédent, *first_match()* renvoie "2"

Chapitre 3

Analyse des algorithmes

3.1 Généralités

Analyser un algorithme c'est prévoir les ressources nécessaires à cet algorithme. Ces ressources sont le plus souvent le temps d'exécution et la mémoire utilisée. En général, en analysant plusieurs algorithmes ayant le même objectif, on arrive aisément à identifier le plus efficace ou l'algorithme qui utilise le moins d'espace mémoire. Dans la littérature, on parle aussi d'évaluer la **complexité en temps** et la **complexité en espace**.

3.1.1 Temps d'exécution

Pour pouvoir évaluer le temps d'exécution d'un algorithme, on supposera que les instructions sont exécutées les unes après les autres (aucune exécution n'est en simultanée) et que l'exécution de chaque ligne de pseudo code (qui fait un effort de ne contenir qu'une seule instruction élémentaire) demande un temps constant. Deux lignes différentes peuvent prendre des temps différents, mais chaque exécution de la i -ème ligne prend un temps t_i , t_i étant une constante.

Le temps d'exécution d'un algorithme dépend des données en entrée : le tri d'un millier de nombres prend plus de temps que le tri de trois nombres. En général, le temps d'exécution d'un algorithme croît avec la taille de l'entrée ; on a donc pris l'habitude d'exprimer le temps d'exécution d'un algorithme en fonction de la taille de son entrée. Avant de définir le temps d'exécution d'un algorithme, il nous faut définir la taille d'une entrée manipulée par cet algorithme.

La taille d'une entrée

La notion de taille d'une entrée, dépend du problème étudié. Pour de nombreux problèmes, tels le tri et la recherche, la notion la plus naturelle est le nombre d'éléments constituant l'entrée, par exemple la longueur n du tableau à trier. Pour beaucoup d'autres problèmes, comme la multiplication et l'addition de deux entiers binaires, la meilleure mesure de la taille d'une entrée est le nombre de bits nécessaires à sa représentation.

Cependant, pour les raisons de simplicité, nous supposerons que la taille des données des types de base (booléen, caractère, entier, réel) est constante ; exception faite pour les grands entiers. Pour chaque problème, il faut indiquer la mesure utilisée pour exprimer la taille d'une entrée.

Le temps d'exécution pour une entrée particulière

Le temps d'exécution d'un algorithme pour une entrée particulière est le nombre d'instructions élémentaires qui doivent être exécutées. Ces instructions qui ont un temps d'exécution constant peuvent être exécutées plusieurs fois dans un algorithme. Si une instruction i demande un temps t_i pour être exécutée et est exécutée n fois, il compte pour $t_i * n$ dans le temps d'exécution total. Le temps d'exécution de l'algorithme est la somme des temps d'exécution de chaque instruction.

Les commentaires ne sont pas des instructions exécutables, ils consomment par conséquent un temps nul.

Remarque : Deux différentes instructions prennent des temps différents pour être exécuté ; ainsi dans la pratique, il convient de choisir une instruction élémentaire particulière afin de calculer le nombre de fois qu'elle est exécutée dans l'algorithme. Le choix est bon si le nombre d'exécution total de toutes les instructions est proportionnel au nombre d'exécution de cette instruction particulière.

Les instructions élémentaires suivantes peuvent être choisies :

- Les affectations ;
- les instructions de contrôle ;
- Les instructions de lecture et d'écriture ;
- Les instructions de comparaison ;
- Les instructions arithmétique (addition, soustraction, multiplication, division, modulo, partie entière) ;

— Les appels des fonctions.

Exemple de choix :

- pour un problème de recherche d'un élément dans un tableau, on choisit généralement comme instruction élémentaire la comparaison ;
- pour un problème de multiplication des matrices réelles, on calcul le nombre de multiplications réellement effectué.

3.1.2 Espace mémoire

L'espace mémoire est le nombre d'octets utilisé lors de l'exécution de l'algorithme. Il peut se résumer au nombre d'octets nécessaire pour représenter les nouvelles variables utilisées.

Pour les algorithmes récursifs, il faut considérer en outre l'espace requis pour gérer l'ensemble des appels récursifs. Cette gestion est organisée au travers d'une pile dont la taille est égale au nombre d'appels récursifs à un facteur constant près.

En conclusion, la complexité en temps et la complexité en espace permettent de comparer les algorithmes. Considérer dans leur ensemble, il peut cependant ne pas exister de meilleur algorithme : on peut avoir deux algorithmes A et B de même objectif, l'algorithme A s'exécute plus rapidement que l'algorithme B mais l'algorithme B utilise moins d'espace mémoire que l'algorithme A.

3.1.3 Différents cas de calcul de complexité

Même pour des entrées ayant la même taille, la complexité d'un algorithme peut dépendre de l'entrée *particulière* ayant cette taille. Par exemple, un algorithme de tri par exemple peut demander des temps différents pour trier deux tableaux de même taille, selon qu'ils sont déjà plus ou moins triés partiellement. Nous parlerons du **cas le plus favorable**, celui où le tableau en entrée est déjà trié, et du **cas le plus défavorable**, celui où le tableau en entrée est trié en sens inverse.

Complexité dans le cas le plus défavorable (le pire des cas)

La complexité (temporelle ou spatiale) dans le pire des cas est la complexité maximale pour une quelconque entrée de taille précise. Par défaut, quand nous évo-

quons la complexité d'un algorithme, nous considérerons la complexité dans le pire des cas. En effet, c'est une borne supérieure du temps d'exécution (resp. l'espace nécessaire) associé à une entrée quelconque ; connaître cette valeur nous permettra donc d'avoir une certitude que l'algorithme ne mettra jamais plus de temps (resp. ne nécessitera jamais plus d'espace) que cette limite. Ainsi, point besoin de se soucier sur les ressources puisqu'on ne va jamais rencontrer un cas encore pire. De plus, pour certains algorithmes, le cas le plus défavorable survient assez souvent. Par exemple, quand on recherche un élément qui n'existe pas dans un tableau.

Complexité dans le cas le plus favorable (le meilleur des cas)

La définition diffère de celle dans le pire des cas en ne considérant que le nombre minimal d'instructions élémentaires sur l'ensemble des entrées de taille n .

Complexité dans le cas moyen

La définition diffère de celle dans le pire des cas en considérant non le nombre maximal d'instructions élémentaires mais la moyenne du nombre d'instructions élémentaires sur l'ensemble des entrées de taille n .

3.2 Exemple 1 : Analyse de la fonction factorielle

3.2.1 Version itérative

Numéro	Algorithme	Temps	Fois	Espace
	Entier : <u>FONCTION</u> factoIter(Entier : n)			4o
	/* <u>Objectif</u> : calcul itératif de $n!$ */			
	/* <u>Arguments</u> : un entier n */			
	/* <u>Résultat</u> : $n! = 1 \times 2 \times 3 \times \dots \times n$ */			
	<u>Variable</u>			
	Entier : i, res;			2*4o
	<u>Début</u>			
1	res \leftarrow 1	t_1	1	
2	<u>Pour</u> i \leftarrow 1 (1) n <u>Faire</u>	t_2	$n+1$	
3	res \leftarrow res*i;	t_3	n	
	<u>finPour</u>			
4	renvoyer (res)	t_4	1	
	<u>Fin</u>			

Complexité temporelle : Plus la valeur de n est grande, plus le temps d'exécution est important; la complexité (taille) de l'entrée ici est le nombre d'octets (nombre de bits) que nécessite la représentation de n , c'est-à-dire $\log_2 n$. Cependant nous allons calculer la complexité sur n , nous y reviendrons sur $\log_2 n$ plus tard.

Remarque : Quand la boucle Pour se termine normalement (c'est-à-dire, suite au test effectué dans l'entête de la boucle), le test est exécuté une fois de plus que le corps de la boucle.

Pour calculer le temps d'exécution $T(n)$ de l'algorithme factoIter(n), on additionne les produits des colonnes temps et fois, ce qui donne :

$$T(n) = t_1 + t_2(n + 1) + t_3(n) + t_4$$

$$\Rightarrow T(n) = (t_2 + t_3)n + (t_1 + t_2 + t_4)$$

Ce temps d'exécution peut être exprimé sous la forme $an + b$, a et b étant les constantes dépendantes du temps des instructions élémentaires. C'est donc une **fonction linéaire de n** .

Le nombre de multiplication est une fonction linéaire de n ($= t_2 * n$); la multiplication est donc un bon choix comme instruction élémentaire pour calculer la complexité en temps.

Complexité spatiale : Quelques soit le nombre n , on déclare trois variables entières (n, i et res). Donc, la complexité en espace de cet algorithme est une constante.

3.2.2 Version récursive

Numéro	Algorithme	Temps	Fois	Espace
	Entier : FONCTION factoRec(Entier : n) /* <u>Objectif</u> : calcul récursif de $n!$ */ /* <u>Arguments</u> : un entier n */ /* <u>Résultat</u> : $n! = n \times (n-1)!$ */			4o
	<u>Variable</u> Entier : res ;			4o
	<u>Début</u>			
1	<u>Si</u> $n = 0$ <u>Alors</u>	t_1	1	
2	res \leftarrow 1	t_2	1	
	<u>Sinon</u>			
3	res \leftarrow $n * \text{factoRec}(n-1)$	t_3	1	
	<u>finSi</u>			
4	renvoyer (res)	t_4	1	
	<u>Fin</u>			

Lorsqu'un algorithme contient un appel récursif à lui même, sa complexité peut souvent être décrit par une équation de récurrence, qui décrit la complexité globale pour un problème de taille n à partir de la complexité pour les entrées de taille moindre.

Complexité temporelle : Soit $T(n)$ le temps d'exécution d'un problème de taille n .

Si la taille est égale à 0, la solution prend un temps constant ($t_1 + t_2 + t_4$), puisque toutes ces instructions sont des instructions élémentaires ; notons cette constante c_1 .

Sinon ($n > 0$), le temps est égale à $(t_1 + t_3 + t_4)$ avec $t_3 = t_{3'} + T(n - 1)$ (puisque l'on affecte le résultat de la multiplication de factoRec($n-1$) par n). Notons par la constante $c_2 = t_1 + t_{3'} + t_4$; on obtient alors la récurrence suivante :

$$T(n) = \begin{cases} c_1 & \text{si } n = 0 \\ c_2 + T(n - 1) & \text{sinon} \end{cases}$$

Ainsi :

$$T(n) = c_2 + T(n - 1) = c_2 + c_2 + T(n - 2) = \dots = n * c_2 + T(0) = n * c_2 + c_1$$

C'est donc une **fonction linéaire de n** .

Complexité spatiale : Nous avons n appels récursifs et chaque appel crée 2 variables entières (8 octets); la fonction factoRec(n) nécessite une pile d'espace $8 * n$ Octets. La complexité en espace est donc une **fonction linéaire de n** .

3.3 Exemple 2 : Analyse du tri par insertion

Soit l'algorithme du tri par insertion suivant :

CONSTANTE

MAX = 10

TYPE

Tvecteur = TABLEAU[MAX] de réel

FONCTION TriInsertion (Tvecteur : Tab
entier : n)

// Objectif : trier dans l'ordre croissant les n premiers éléments du tableau Tab

/* Résultat : On fait n itérations, à l'itération i :

- les premiers $(i - 1)$ éléments de la liste sont triés.
- on insère d'une manière séquentielle le i -ème élément parmi les premiers $(i - 1)$ éléments en décalant les éléments vers la droite.*/*

VARIABLE

entier : i, j

réel : x

	<u>Debut</u>
1	<u>Pour</u> $i \leftarrow 1$ (1) $n-1$ <u>faire</u>
2	$x \leftarrow \text{Tab}[i]$ //insérer $\text{Tab}[i]$ dans la liste triée $\text{Tab}[0], \text{Tab}[1], \dots, \text{Tab}[i-1]$
3	$j \leftarrow i-1$
4	<u>Tantque</u> $((j > 0) \text{ ET } (\text{Tab}[j] > x))$ <u>Faire</u>
5	$\text{Tab}[j+1] \leftarrow \text{Tab}[j]$
6	$j \leftarrow j-1$
	<u>FinTantque</u>
7	$\text{Tab}[j+1] \leftarrow x$
	<u>FinPour</u>
	<u>Fin</u>

La taille de l'entrée ici est le nombre d'éléments du tableau, c'est-à-dire n .

L'instruction élémentaire est le nombre de comparaisons.

Calculer la complexité en temps $T(n)$, revient à donner le nombre de comparaison en fonction de n .

Pour tout $i = 1, 2, \dots, n-1$, notons par t_i le nombre de fois que le test de la boucle Tantque, en ligne 4 est exécuté pour cette valeur de i . On a :

$$T(n) = a * n + b * \sum_{i=1}^{n-1} t_i$$

a et b étant le coût des lignes n° 1 et n° 4.

— Complexité dans le cas le plus favorable :

Le cas le plus favorable est celui où le tableau est déjà trié dans l'ordre croissant. Pour tout $i = 1, 2, \dots, n-1$, on trouve alors que $\text{Tab}[j] \leq x$ en ligne 4 quand j a sa valeur initiale de $i-1$. Donc $t_i = 1$ pour $i = 1, 2, \dots, n-1$ et le temps d'exécution associé au meilleur des cas est alors :

$$\begin{aligned} \Rightarrow T(n) &= a * n + b * \sum_{i=1}^{n-1} 1 \\ \Rightarrow T(n) &= a * n + b * (n-1) \\ \Rightarrow T(n) &= (a + b) * n - b \end{aligned}$$

ce temps est une **fonction linéaire de n**

— **Complexité dans le cas le plus défavorable :**

Le cas le plus défavorable est celui où le tableau est trié dans l'ordre décroissant. On doit comparer chaque élément $Tab[i]$ avec chaque élément du sous-tableau trié $Tab[0], Tab[1], \dots, Tab[i-1]$. Donc $t_i = i$ pour $i = 1, 2, \dots, n-1$ et le temps d'exécution associé au pire des cas est alors :

$$\begin{aligned}\Rightarrow T(n) &= a * n + b * \sum_{i=1}^{n-1} i \\ \Rightarrow T(n) &= a * n + b * \left[\frac{n * (n+1)}{2} - n \right] \\ \Rightarrow T(n) &= \frac{b}{2} * n^2 + \left(a - \frac{b}{2} \right) * n\end{aligned}$$

Le temps d'exécution du cas le plus défavorable est une **fonction quadratique de n**.

— **Complexité dans le cas le plus moyen :**

Supposons que l'on choisisse au hasard n nombres, auxquels on applique ensuite le tri par insertion. En moyenne, la moitié des éléments $Tab[0], Tab[1], \dots, Tab[i-1]$ sont inférieurs à $tab[i]$ et la moitié lui sont supérieurs. Donc, en moyenne, on teste la moitié du sous-tableau trié $Tab[0], Tab[1], \dots, Tab[i-1]$; auquel cas $t_i = i/2$.

Le temps d'exécution associé au cas moyen est une **fonction quadratique de n**.

3.4 Ordre de grandeur

3.4.1 Définition

Nous avons utilisé des hypothèses simplificatrices pour faciliter notre analyse des algorithmes :

1. D'abord, nous avons ignoré le temps des autres instructions en choisissant une instruction élémentaire particulière ;
2. Ensuite nous avons ignoré le temps réel de cette instruction en employant une constante, pour représenter son coût ;

3. Enfin, nous avons observé que même cette constante nous donne plus de détails que nécessaire (fonction linéaire, fonction quadratique, ...).

Nous allons à présent utiliser une simplification supplémentaire. Ce qui nous intéresse vraiment, c'est **le taux de croissance**, ou ordre de grandeur, de la complexité :

- On ne considéra donc que le terme dominant d'une formule (par exemple $a * n^2$), puisque les termes d'ordre inférieur sont moins significatifs pour n grand.
- On ignorera également le coefficient constant du terme dominant, puisque les facteurs constants sont moins importants que l'ordre de grandeur pour ce qui est de la détermination de l'efficacité du calcul ou de la mémoire utilisée pour les entrées volumineuses.

Exemple :

- dans le pire des cas, le tri par insertion a un temps d'exécution de $\Theta(n^2)$ (prononcer « thêta de n-deux »).
- l'appel récursif de la fonction factorielle occupe un espace de $\Theta(n)$

Soient deux fonctions $f, g : \mathbb{N}^+ \mapsto \mathbb{R}^*$

$$\Theta(f(n)) = \{g(n) : \exists c_1, c_2, n_0 > 0, \forall n \geq n_0 / 0 \leq c_1 f(n) \leq g(n) \leq c_2 f(n)\}$$

Autrement dit, à partir d'un certain rang, la fonction g est égale à f à un facteur constant près. On dit que $f(n)$ **est une borne asymptotiquement approchée de** $g(n)$.

On va donc comparer les algorithmes selon l'ordre de grandeur de leur complexité. On considère généralement qu'un algorithme est plus efficace qu'un autre si son temps d'exécution au pire des cas a un ordre de grandeur inférieur.

Compte tenu des facteurs constants et des termes d'ordre inférieur, cette évaluation peut être erronée pour des entrées de faible volume.

En revanche, pour les entrées de taille assez grande, un algorithme en $\Theta(n^2)$, par exemple, s'exécute plus rapidement, au pire des cas, qu'un algorithme en $\Theta(n^3)$.

3.4.2 Le vocabulaire

- $f \in \Theta(1)$, f est dite constante
- $f \in \Theta(\log n)$, f est dite logarithmique
- $f \in \Theta(n)$, f est dite linéaire

- $f \in \Theta(n^2)$, f est dite quadratique
- $f \in \Theta(n^k)$, f est dite polynomiale
- $f \in \Theta(c^n)$, $c > 1$, f est dite exponentielle

Remarque : Quand on précise la complexité d'un algorithme, il faut définir en fonction de quelle quantité celle-ci est exprimée. Nous pouvons employer l'expression « tel algorithme est de complexité linéaire ». A défaut de toute précision, ceci signifie linéaire en fonction de la complexité (taille) de l'entrée.

Considérons la fonction factorielle décrit en début de chapitre. Nous avons dit que la complexité en temps de cet algorithme est une fonction linéaire de n . Cependant elle doit être considérée comme exponentielle (puisque la taille de l'entrée est $k = \log_2 n$). La fonction complexité est donc $T(k) = 2^k$. C'est une fonction exponentielle.

3.5 Exercices du chapitre 3

Exercice 1

On considère tous les algorithmes de tris écrits à l'exercice n° 18 du Chapitre 2. Calculer la complexité temporelle au pire et au meilleur des cas de chacun de ses algorithmes. Quelles conséquences peut-on en tirer ?

Exercice 2

On considère, pour effectuer la recherche d'un élément dans un tableau, la recherche séquentielle et la recherche dichotomique. Quelle est la complexité temporelle pour chaque algorithme ?

Exercice 3

1. Calculer le nombre d'appels à la fonction Fibonacci écrite sous sa forme récursive lors de l'exécution de $\text{Fibonacci}(n)$;
2. Proposer une fonction itérative calculant la suite de Fibonacci et donner en sa complexité.

Exercice 4

On représente un polynôme $P(x) = \sum_{i=0}^n a_i x_i$ de degré n par un tableau tab de taille $n+1$ tel que $\text{tab}[i] = a_i$ pour $(0 \leq i \leq n)$ Après avoir Écrit les fonctions qui permettant de calculer les éléments suivants, calculer leur complexité temporelle :

- la somme de deux polynômes $P(x)$ et $Q(x)$ de même degré n .
- le produit de deux polynômes $P(x)$ et $Q(x)$ de même degré n .
- la valeur d'un polynôme $P(x)$ de degré n pour une valeur x_0 donnée en utilisant :
 - la fonction $\text{puissance}(x, p)$ qui calcule la valeur x^p qui vous aurez préalablement écrite ;
 - la règle de Hörner

Exercice 5

Soit Tab un tableau de n entiers distincts. Si $i < j$ et $\text{tab}[i] > \text{Tab}[j]$, on dit que le couple (i, j) est une inversion de Tab .

1. Donner les cinq inversions du tableau $[2, 3, 8, 6, 1]$
2. Quel est le tableau dont les éléments appartiennent à l'ensemble $\{1, 2, \dots, n\}$ qui a le plus d'inversions ? Combien en possède-t-il ? Quelle est la relation entre le temps d'exécution du tri par insertion et le nombre d'inversion du

tableau en entrée ? Justifier la réponse.

3. Donner un algorithme qui détermine en un temps $\Theta(n \log_2 n)$, dans le cas le plus défavorable, le nombre d'inversion présentes dans une permutation quelconque de n éléments (*indication* : Modifier le tri par fusion)

Exercice 6

1. Donner un algorithme qui permet de rendre la monnaie d'un montant donné, en utilisant le nombre minimum de pièces, pour n'importe quel ensemble de ($k > 1$) valeurs différentes de pièce.

Par exemple, si le montant donné est 9 et que les ($k=3$) valeurs des pièces est $[1, 4, 6]$, le résultat sera $9 = 4+4+1$.

Indication : Vous inspirer de l'exercice n° 18 du Chapitre 1,

2. Quelle est sa complexité en temps ?
3. Quel est le nombre d'octets au pire des cas pour cet algorithme sachant qu'on manipule les données de type Entier.

Chapitre 4

Les enregistrements

4.1 Généralités

4.1.1 Définition

Un enregistrement (ou variable enregistrement) est un ensemble de variables de type éventuellement différent ; ces variables sont appelées **champs de l'enregistrement**.

Avant de déclarer une variable enregistrement, il faut avoir au préalable défini son type, c'est à dire le type et le nom des champs qui le composent.

4.1.2 Déclaration d'un type et d'une variable Enregistrement

La déclaration d'un type *Enregistrement* se fait, dans la rubrique *Type* de la structure générale des algorithmes de la manière suivante :

<pre>Type Identificateur_type = Enregistrement type_champ1 nom_champ1 type_champ2 nom_champ2 ... type_champN nom_champN finEnregistrement</pre>

Une variable (de type) Enregistrement est déclarée comme celle d'un type de base :

<u>Variable</u>
Identificateur_type nom_variable

Exemple :

Constante

MAX = 10

Type

tChaine = Tableau[MAX + 1] de caractères

tPersonne = Enregistrement

tChaine nom

tChaine prenom

entier age

finEnregistrement

Variable

tPersonne : pers1, pers2, pers3

4.1.3 Manipulation d'un enregistrement

La manipulation d'un enregistrement se fait au travers de ses champs. C'est ainsi que pour afficher un enregistrement, il faut afficher tous ses champs un par un.

Accès aux champs d'un enregistrement

Les champs d'un enregistrement sont accessibles au travers de leur nom, grâce à l'opérateur ".". Ainsi,

nom_variable.nom_champ

représente la valeur mémorisée dans le champ de l'enregistrement. Pour accéder par exemple à l'âge de la variable pers2, nous utilisons l'expression **pers2.age**

Les champs d'un enregistrement, tout comme les éléments d'un tableau, sont des variables à qui nous pouvons faire subir les opérations relatives aux variables des types associés (affectation, entrée, sortie, ...).

Exemple :

ALGORITHME Ecart_d_Age

/* Objectif : saisir les données concernant 2 personnes et afficher leur différence d'âge */

/*Données : Pour chacune des 2 personnes, il faut entrer une chaîne de caractère et un entier représentant respectivement leur nom et leur âge */

/*Résultat : affichage de la valeur absolue de l'âge de la première personne soustrait de l'âge de la seconde.*/

Constante

MAX = 10

Type

tChaine = Tableau[MAX + 1] de caractères

tPersonne = Enregistrement

tChaine : nom

tChaine : prenom

entier : age

finEnregistrement

Variable

tPersonne : pers1, pers2

Début

Ecrire ("Entrez le nom puis l'âge de la première personne")

Lire (pers1.nom, pers1.age)

Ecrire ("Entrez le nom puis l'âge de la seconde personne")

Lire (pers2.nom, pers2.age)

Ecrire ("la différence d'âge entre", pers1.nom, " et ", pers2.nom, " et de :")

Si (pers1.age ≤ pers2.age) alors

Ecrire (pers2.age – pers1.age, "ans")

Sinon

Ecrire (pers1.age – pers2.age, "ans")

finSi

Fin

Affectation globale entre enregistrement

Il est possible d'affecter globalement à une variable de type enregistrement la valeur d'une autre variable de même type. On peut donc par exemple écrire :

pers2 ← *pers1* ; ce qui remplace toutes les affectations suivantes :

$$\begin{cases} pers2.nom \leftarrow pers1.nom \\ pers2.prenom \leftarrow pers1.prenom \\ pers2.age \leftarrow pers1.age \end{cases}$$

Imbrication des enregistrements

Supposons que dans le type *tPersonne*, de l'exemple précédent, nous voulions la date de naissance de la personne en lieu et place de son âge. Une date est composée de trois variables (jour, mois, année) **indissociables** et de **types différents**. Une date correspond donc à un objet du monde réel qu'il est judicieux de représenter par un type Enregistrement à trois champs.

Après avoir au préalable déclaré le type *tDate*, nous pouvons l'utiliser dans la déclaration du type *tPersonne* pour le type de la date de naissance. En effet, un type enregistrement peut-être utilisé comme type pour des champs d'un autre type enregistrement.

Constante

MAX = 10

Type

tChaine = Tableau[MAX + 1] de caractères

tDate = Enregistrement

entier : jour

tChaine : mois

entier : annee

finEnregistrement

tPersonne = Enregistrement

tChaine : nom

tChaine : prenom

tDate : dateNaissance

finEnregistrement

Pour accéder à l'année de naissance d'une personne, il faut utiliser deux fois l'opérateur "." : *pers1.dateNaissance.annee*

Une telle variable est lue de droite à gauche : l'année de la date de naissance de la personne *pers1*.

L'instruction "Avec ... faire"

Dans les exemples précédents, le nom d'un champ était toujours précédé du nom de l'enregistrement auquel il appartient (exemple : pers1.dateNaissance.annee). Ceci peut être pénible dans l'écriture et la lisibilité des algorithmes. Il est alors possible de trouver un nom de champ tout seul, sans le précéder du nom de l'enregistrement. En pseudo code, il suffit d'utiliser l'instruction **Avec ... faire** :

Avec nom_variable faire

...
nom_champ ← ...

...

FinAvec

Exemple :

Avec Pers1.dateNaissance faire

jour ← 10
mois ← "Octobre"
annee ← 2018

FinAvec

4.1.4 Les tableaux d'enregistrement (ou tables)

Il arrive souvent que nous voulions traiter non pas un seul enregistrement mais plusieurs. Par exemple, si nous voulons pouvoir traiter un groupe de personnes, nous n'allons pas créer autant de variables du type tPersonne qu'il y a de personnes. Nous allons créer un tableau regroupant toutes les personnes du groupe : il s'agit alors d'un tableau d'enregistrements.

...

Constante

NP = 20 //nombre de personnes du groupe
MAX = 10

Type

tChaine = Tableau[MAX + 1] de caractères
tPersonne = enregistrement
tChaine : nom

```

entier : age
finEnregistrement
tGroupe = Tableau[NP] de Tpersonne

```

Variable

```

tGroupe : groupe

```

...

Chaque élément du tableau est un enregistrement, contenant plusieurs variables de types différents. Nous accédons à un enregistrement par son indice dans le tableau. Ainsi :

- groupe[2] représente la troisième personne du groupe
- groupe[2].nom représente le nom de la troisième personne du groupe
- groupe.nom[2] n'est pas valide.

4.1.5 Tableau de taille variable

L'allocation dynamique a permis à l'utilisateur de fixer la taille d'un tableau ; initialement, seul le programmeur pouvait le faire. Cependant, une fois fixée, cette taille n'est pas susceptible de changer même si le nombre d'éléments du tableau augmente. Ce qui amène généralement, à fixer une taille aussi grande pour rester dans le tableau. Mais, en agissant ainsi, nous perdons en temps d'exécution et en espace de stockage. Les enregistrements nous permettent de résoudre le problème de temps d'exécution, l'espace mémoire sera résolu en fin de chapitre. Ainsi nous avons la déclaration suivante :

Constante

```

TAILLE_MAX = constante_entière

```

Type

```

tTab = Tableau[TAILLE_MAX] d'un Type_Donné
tEnreg = Enregistrement
entier : taille
tTab : tab
finEnregistrement

```

Ainsi, dans la manipulation d'une variable de type *tEnreg*, au lieu d'aller jusqu'à TAILLE_MAX, on s'arrêtera à *taille* (\leq TAILLE_MAX) dont on peut de surcroit

charger sa valeur.

4.1.6 Tableau circulaire

Au lieu de placer les valeurs dans le tableau à partir de l'emplacement 0, on les place à partir de l'emplacement *tete* (qui est quelconque). Toutes les autres valeurs du tableau sont ensuite placées à la suite de l'emplacement *tete*; l'emplacement 0 suit l'emplacement $TAILLE - 1$ dans un ordre circulaire. Le champ *tete* contient l'emplacement du premier élément du tableau tandis que le champ *queue* contient l'emplacement qui suit le dernier élément du tableau. Nous déclarons un tableau circulaire ainsi :

Constante

$TAILLE_MAX = constante_entière$

Type

tTab = Tableau[$TAILLE_MAX$] d'un *Type_Donné*

tEnreg = Enregistrement

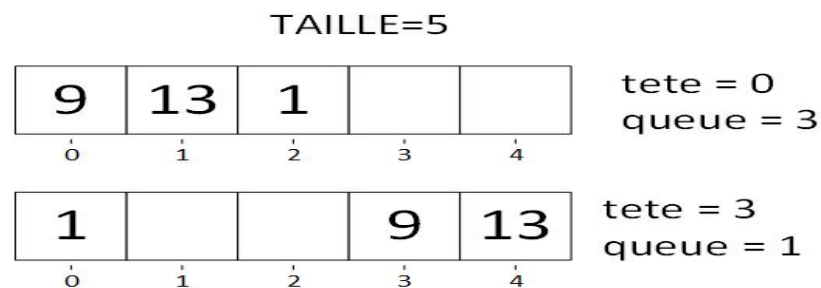
tTab : tab

Entier : tete

Entier : queue

finEnregistrement

Deux représentations possibles du même tableau :



4.1.7 Enregistrement à champs variable

Il est possible de créer les enregistrements dont les champs varient d'une variable à une autre; il suffit d'utiliser un type énumération et le mot clé "cas où" dans la

déclaration des types.

La syntaxe :

Type

tEnum = Enumeration { constante1, constante2, ... ,constanteN }

tEnreg = Enregistrement

type_champ1 : nom_champ1

type_champ2 : nom_champ2

tEnum : nom_champ3

cas où nom_champ3

constante1 : type_champ4 : nom_champ4

constante2 : type_champ5 : nom_champ5

type_champ6 : nom_champ6

...

constanteN : type_champM : nom_champM

finCas

finEnregistrement

4.1.8 Pointeur sur un enregistrement

Un pointeur est très souvent d'utiliser pour mémoriser l'adresse d'un enregistrement.

Exemple :

Constante

MAX = 10

Type

tChaine = Tableau[MAX + 1] de caractères

tPersonne = Enregistrement

tChaine : nom

entier : telephone

FinEnregistrement

Variable

tPersonne : pers

↓tPersonne : p

Debut

```
pers.nom ← "awa"  
pers.telephone ← 242222222  
p ← @pers  
Ecrire (p->nom)           //équivalent à Ecrire(pers.nom)  
Ecrire (p->telephone)      //affiche 242222222
```

Fin

Remarque :

Pour accéder au champ nom de l'enregistrement pointé, on devrait écrire normalement ((@p).nom, mais on utilise une écriture simplifiée "->" (p->nom).

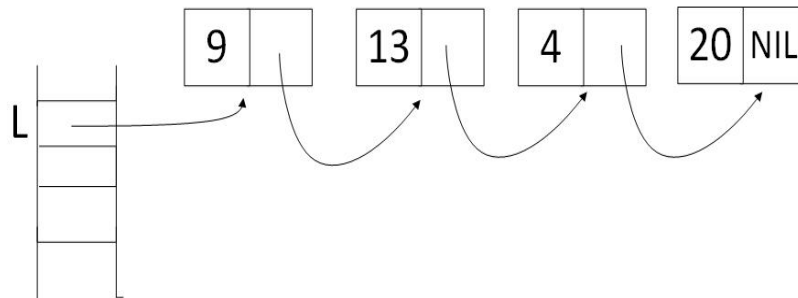
4.2 Les Listes chaînées

4.2.1 Définition et Manipulation

Une liste chaînée est une structure de données dans laquelle les éléments sont reliés linéairement grâce à un pointeur. Il s'agit d'un cas particulier d'enregistrement dont l'un des champs est un pointeur qui pointe sur une variable enregistrement de même type (le pointeur est du type enregistrement de l'élément).

<u>Type</u> tEnreg = Enregistrement Entier : data ↓tEnreg : next //pointeur sur une variable de type tEnreg finEnregistrement	
<u>Variable</u> ↓tEnreg : L	
data	next
	.

Dans une liste chaînée, les éléments sont reliés grâce à un pointeur. Un élément contient les informations de l'élément (ici l'entier data) et un pointeur sur l'élément suivant (ici le pointeur next). Le premier élément, appelé **tête**, n'a pas de précédent. Pour mémoriser son adresse, on déclare un pointeur à part (ici L). Grâce à ce pointeur, on peut accéder au premier élément, qui permet d'accéder au deuxième, qui permet d'accéder au troisième, et ainsi de suite. A partir du poin-



teur sur le premier élément, on peut parcourir toute la liste. Le dernier élément a la valeur NIL dans son champ next, il est appelé **queue** de la liste. La liste est alors définie comme un pointeur sur le premier élément de la chaîne. Les listes chaînées fournissent donc une représentation simple et souple pour un ensemble de données.

Les éléments d'une liste chaînée sont créés dynamiquement au fur et à mesure des besoins. Ainsi, on peut gérer un tableau dont la taille et le contenu évoluent au fil de l'exécution. Il n'y a donc pas de perte d'espace mémoire.

Algorithme GestionListeChaînée

/*objectif : gérer un tableau d'entier grâce à une liste chaînée */

Type

toto = Enregistrement

Entier data

↓toto next

finEnregistrement

Variable

↓toto : L //pointeur sur la tête de la liste

FONCTION afficherListe (↓toto L)

/*Objectif : afficher la liste d'entier L */

Variable

↓toto tmp

Debut

tmp ← L

Tantque (tmp ≠ NIL) Faire

Ecrire (tmp->data)

```

        tmp ← tmp->next
    finTantque
    Ecrire (NIL)
fin //FONCTION

↓toto FONCTION empiler (entier n,
                        ↓toto L)
/*Objectif : ajouter l'entier n en tête de la liste L*/
Variable
    ↓toto pel          //pointeur sur l'élément en cours de création
Début
    pel ← allouer(NbOctet(toto))      /* Crée l'élément et affecte son
                                       adresse au pointeur courant */
    pel->data ← n          // Initialise les informations de cet élément
    pel->next ← L
    Renvoyer pel
fin //FONCTION

↓toto FONCTION enfiler (entier n,
                        ↓toto L)
/*Objectif : ajouter l'entier n en queue de la liste L*/
Variable
    ↓toto pel, tmp
Début
    tmp ← L
    pel ← allouer(NbOctet(toto))      /* Crée l'élément et affecte son
                                       adresse au pointeur courant*/
    pel->data ← n          // Initialise les informations de cet élément
    pel->next ← NIL
    Si (L = NIL) Alors
        Renvoyer pel
    Sinon
        Tantque (tmp->next ≠ NIL) Faire

```



```

        tmp ← tmp->next
    FinTantQue      //déplacement jusqu'au dernier élément de la liste
    tmp -> next ← pel
    Renvoyer L
FinSi
Fin //FONCTION

↓toto FONCTION supprimer (entier n,
                        ↓toto L)
/*Objectif : recherche et supprime la 1ère occurrence de n dans L*/
Variable
    ↓toto pel      //pointeur sur l'élément à supprimer
    ↓toto prec     //pointeur sur l'élément qui précède celui à supprimer
Début
    pel ← L
    prec ← NIL
    //Recherche de l'élément et son précédent
    TantQue ((pel ≠ NIL) et (pel->data ≠ n)) Faire
        prec ← pel
        pel ← pel->next
    FinTantQue
    Si (pel ≠ NIL) Alors      //l'élément à supprimer a été retrouvé
                                //il faut mettre à jour les cellules voisines
        Si (prec ≠ NIL) Alors // ce n'est pas l'élément en tête de liste
            prec->next ← pel->next
        FinSi
        Si (pel = L) Alors      // c'est l'élément en tête de liste
            L ← pel->next
        FinSi
        Desallouer(pel)      // libération de la cellule
    FinSi
    Renvoyer L
Fin // FONCTION

```

```

Début      //le programme principal
              L ← empiler(9,empiler(13,empiler(4, empiler(20,NIL))))
              // L ← enfiler(20, enfiler(4,enfiler(13, enfiler(9,NIL)))) //idem
              afficher(L)
              L ← supprime(4, L)
              afficher(L)

```

Fin

Une liste chaînée peut prendre différentes formes. Elle peut être une **pile** ou une **file**, **simplement chaînée** ou **doublement chaînée**, **triée** ou **non-triée**, **circulaire** ou **non-circulaire**.

- Une liste est simplement chaînée si elle a un seul pointeur sur chaque élément. Elle est doublement chaînée si elle en a deux.
- Si une liste est triée, l'ordre linéaire de la liste correspond à l'ordre linéaire des informations (*champ info*) des éléments de la liste ; l'élément minimum est la tête de la liste et l'élément maximum est la queue, si l'ordre est croissant. Si une liste est non-triée, les éléments peuvent apparaître dans n'importe quel ordre.
- Dans une liste circulaire, le dernier élément (la queue) pointe sur le premier élément (la tête). La liste peut donc être vue comme un anneau d'éléments.

4.2.2 Les piles

Une pile est une structure de données, où l'ajout et le retrait d'un élément se fait toujours au sommet (en tête de liste). Une pile suit la règle **LIFO (Last In, First Out)** : le dernier à entrer est le premier à en sortir.

Terminologie particulière :

- l'extrémité où s'effectuent l'ajout et la suppression s'appelle le **sommet** ;
- ajouter un élément à une pile se dit **empiler** ;
- supprimer un élément de la pile et récupérer son information s'appellent **dé-piler**.

Ces noms font allusion aux piles évoquées dans la gestion des appels des fonctions dans un programme. A chaque fois qu'on appelle une fonction (sous-programme),

on impose au programme d'arrêter l'exécution du code en cours et de se brancher sur le sous-programme. Le programme doit enregistrer l'endroit où il se trouve dans le programme appelant et la valeur des variables à ce moment avant de se brancher vers le sous-programme appelé. Il enregistre ces informations dans une pile. Au retour d'appel du sous-programme, la pile est dépilée, ainsi les informations de l'état avant appel sont récupérées. C'est donc grâce à une pile que le programme gère l'appel en cascade de plusieurs sous-programmes. Cette pile est gérée automatiquement par le système d'exploitation, le programmeur n'a pas à s'en préoccuper.

On ne peut manipuler une pile qu'à travers son sommet. Le sommet d'une pile représentée sous forme de liste chaînée est la tête de cette liste. Le seul pointeur sur la liste est donc le pointeur de tête. La création d'une pile se fait "à l'envers" : le dernier élément créé se retrouve en tête (au sommet).

Remarque : le sommet est une variable déclarée.

4.2.3 Les files

Une file est une structure de donnée où l'insertion d'un élément se fait à une extrémité appelée queue et la suppression d'un élément se fait à une autre extrémité appelée tête. Une file suit la règle **FIFO (First In, First Out)** : le premier à entrer est le premier à en sortir.

Terminologie particulière :

- On appelle **enfiler** le fait d'ajouter un élément dans une file ;
- On appelle **défiler** la suppression d'un élément dans une file ;
- Lorsqu'un élément est enfilé, il prend la place de la **queue** de la file, et l'élément défilé est toujours le premier en **tête** de file.

Les files servent à traiter des données dans l'ordre dans lequel elles arrivent, comme dans une file d'attente à un guichet où le premier arrivé est le premier servi.

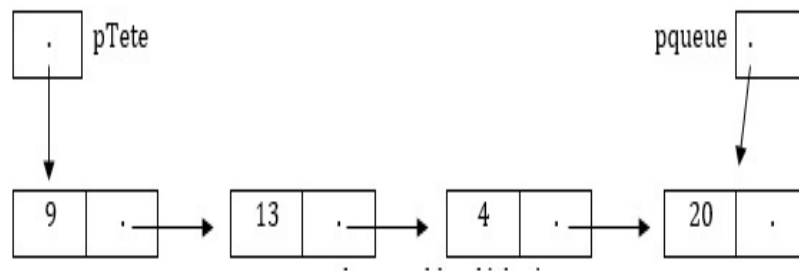
Une file doit être accessible à la fois par la tête pour supprimer un élément et par la queue pour ajouter un nouvel élément. La tête et la queue d'une file correspondent à la tête et la queue de la liste chaînée qui l'implémente.

Dans une file, le premier élément créé doit être en tête et le dernier élément créé doit être en queue. Or, si l'on n'utilise qu'un seul pointeur ptete pour créer la

liste, il va falloir parcourir toute la file pour ajouter un élément (cf enfiler() dans l'exemple ci-dessus).

Une autre solution consiste à utiliser deux pointeurs sur la liste : un pointeur sur la tête et un pointeur sur la queue (cf enfiler2() ci-dessous). **La création se fera par ajout successif en queue** (alors qu'une pile est créée par insertion successive en tête).

Remarque : ptete et pqueue sont des variables déclarées.



Exemple :

...

Variable

↓toto : L //pointeur sur la tête de file

↓toto : Q //pointeur sur la queue de file

↓toto FONCTION enfiler2 (entier n,
↓toto l, q)

/*Objectif : ajouter l'entier n en queue de la liste L et renvoie la nouvelle queue de liste.

/*Arguments : On suppose que la liste en argument n'est pas vide*/

Variable

↓toto pel //pointeur sur l'élément en cours de création

Début

pel ← allouer(NbOctet(toto)) /* Crée l'élément et affecte son
adresse au pointeur courant */

pel->data ← n // Initialiser les infos de cet élément

pel->next ← NIL

Si q ≠ NIL Alors

q->next ← pel

fsi

Renvoyer pel

Fin

4.2.4 Les listes doublement chaînées

Une liste doublement chaînée est une liste où chaque élément pointe sur son suivant et sur son précédent. Le parcours d'une telle liste peut se faire dans les deux sens. Ce type de liste facilite en outre l'ajout d'un élément au milieu de la liste de sorte que celle-ci reste triée.

Le premier élément possède un pointeur qui peut pointer sur NIL, sur la tête ou sur le dernier élément (liste circulaire). Le dernier élément possède un pointeur qui peut pointer sur NIL, sur la queue ou sur le premier élément (liste circulaire).

Type

tListe = enregistrement

entier info

↓tListe psuiv

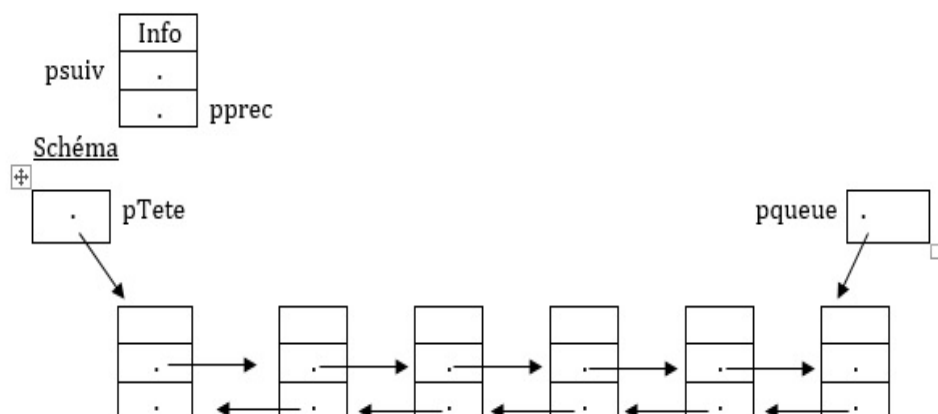
↓tListe pprec

finEnregistrement

Variable

↓tListe ptete, pqueue

Illustration :



4.3 Exercices du chapitre 4

Exercice1

Un nombre complexe z est entièrement défini par ses parties réelle a et imaginaire b . ($z = a + ib$)

1. Déclarez le type `tComplexe`
2. Ecrivez les fonctions donnant les attributs d'un nombre complexe (partie réelle, partie imaginaire, module, argument) ainsi que les fonctions donnant l'addition et la multiplication sur les complexes.
3. Ecrivez les fonctions nommées **lireComplexe()** et **afficherComplexe()** chargées des instructions d'entrée/sortie des variables de type `tComplexe`.
4. Utiliser les questions précédentes pour écrire un algorithme qui lit d'abord deux nombres complexes $c1$ et $c2$ et qui affiche ensuite leur somme et leur produit.

Rappel :

- $(a + ib) + (c + id) = (a + c) + i(b + d)$
- $(a + ib) * (c + id) = (ac - bd) + i(ad + bc)$

Exercice2

Ecrire un algorithme permettant de créer un tableau `Tab_Emp` qui contiendra les informations sur au plus 50 employés d'une entreprise (nom, salaire, `etat_civil` (si marié, le nom du conjoint)), le remplir puis afficher le nombre d'employés dont le salaire est compris entre 200000 et 300000 Fcfa.

Exercice3 (Extrait de l'examen de rattrapage 2010-2011)

Soient les dimensions des figures suivantes :

- Carré : un côté
- Rectangle : une longueur et une largeur
- Cercle : un rayon

Nous voulons enregistrer 1000 figures et calculer pour chacune d'elle le périmètre et la surface.

1. Créer les types suivants :
 - (a) Type Enumeration `tFigure`
 - (b) Type Enregistrement `tDimension` constitué de la surface, du périmètre, de la forme de la figure et des dimensions ci-dessus

- (c) Type Tableau tTabFig de 1000 enregistrements.
2. Écrire une fonction d'entête
- FONCTION calcul(↓tDimension fig,
tFigure form)**
- qui calcule le périmètre et la surface à partir des dimensions fig de la figure qui a la forme form.
3. Écrire un programme principal qui permet :
- (a) de saisir les dimensions des 1000 figures,
 - (b) d'appeler la fonction **calcul()** pour calculer le périmètre et la surface de ces figures,
 - (c) d'afficher les informations en mémoire.
4. Quel est l'espace mémoire nécessaire pour exécuter ce programme ?

Exercice4 (Extrait de l'examen du rattrapage 2014-2015)

Soit la définition d'une liste chaînée triée par ordre croissant suivante :

liste = Enregistrement

entier info

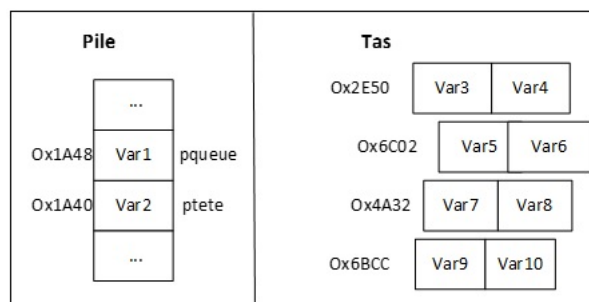
↓liste psuiv

FinEnregistrement

Une variable de cet enregistrement est représentée par :

info	psuiv
------	-------

1. On considère une configuration de la liste chaînée triée suivante :



Sachant que initialement, $Var1 = Var2 = NIL$ et $Var3, Var4, \dots, Var10$ ne sont pas initialisés et en supposant que l'allocation des espaces mémoires se fait par ordre croissant de leur adresse, donner la valeur des variables qui

ont changé après chacune des opérations suivantes (NB : ne pas donner la valeur des variables qui sont restées inchangées d'une opération à une autre) :

- (a) Ajouter l'entier 6 dans la liste
 - (b) Ajouter l'entier 15 dans la liste
 - (c) Ajouter l'entier 10 dans la liste
 - (d) Supprimer l'entier 6 de la liste
 - (e) Ajouter l'entier 4 dans la liste
2. Écrire une fonction d'entête

FONCTION afficher(↓liste ptete,
↓liste pqueue)

qui affiche tous les éléments d'une liste chaînée.

3. On suppose ici que la liste chaînée est non-triée, l'objectif étant de la trier :

- (a) Écrire une fonction d'entête

FONCTION echanger(↓liste pelt1,
↓liste pelt2)

qui permute la position de 2 éléments dans une liste chaînée.

- (b) En déduire la fonction du tri à bulles, la structure de données étant une liste chaînée.

Exercice5 (Extrait de l'examen du rattrapage 2012-2013)

Pour un logiciel de gestion des ventes d'une alimentation, supposons la déclaration de type suivante :

Constante

TAILLE = 20

Type

tTab = TABLEAU [TAILLE + 1] de caractere

tData = Enregistrement /*Le gérant n'entre que ces informations à
chaque vente.*/

tTab nom

entier quantite

entier prix_unitaire


```

finEnregistrement
tResult = Enregistrement    /*Le logiciel attribue un numéro et calcul le
                                prix total de la vente.*/
                                entier numero    /*Ce numéro s'incrémente automatiquement d'un
                                                    entier ∈ [1,9] du numéro de la dernière vente */

                                tData data
                                entier prix_total    // prix_total = data.quantite * data.prix_unitaire
finEnregistrement

tVente = Enregistrement    /*Les lignes complètes des ventes sont
                                répertoriées dans une liste chaînée*/

                                tResult result
                                ↓tVente next
finEnregistrement

```

1. Écrire la fonction d'entête

tResult FONCTION calcul1(tData X,
↓tVente L)

qui permet d'avoir un résultat Y à partir de la nouvelle donnée X et de la liste des ventes L.

Remarque : La fonction Random() renvoie aléatoirement un entier appartenant à l'intervalle $[0, +\infty[$.

2. Écrire la fonction d'entête **FONCTION calcul2 (↓tVente L)** qui calcule et affiche la somme des ventes de la liste L.
3. Écrire la fonction d'entête

↓tVente FONCTION ajoutVente (tData X,
↓tVente L)

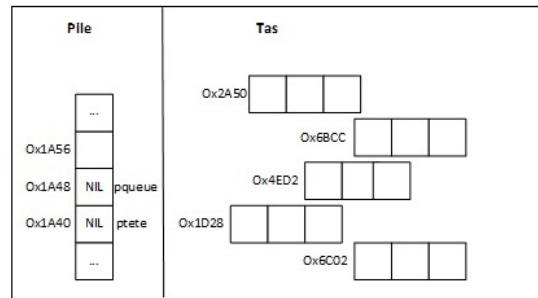
qui appelle la fonction calcul1() pour ajouter la nouvelle donnée X à la fin de la liste des ventes L.

4. Écrire la fonction d'entête

↓tVente) FONCTION supprVente (tData X,
↓tVente L)

qui supprime la dernière occurrence de la donnée X de la liste des ventes L.

Exercice6 (Extrait de l'examen 2014-2015) : Représentations en mémoire d'une liste doublement chaînée triée par ordre croissant
Soit la configuration initiale suivante :



En supposant que l'allocation des espaces mémoires se fait par ordre croissant de leur adresse, dessiner les configurations après chacune des opérations indiquées :

1. Ajouter l'entier 10 dans la liste
2. Ajouter l'entier 15 dans la liste
3. Ajouter l'entier 6 dans la liste
4. Supprimer l'entier 15 de la liste
5. Ajouter l'entier 4 dans la liste
6. Ajouter l'entier 2 dans la liste
7. Supprimer l'entier 6 de la liste
8. Ajouter l'entier 12 dans la liste
9. Ajouter l'entier 8 dans la liste

Exemple : après l'opération **5.** (Ajouter l'entier 4 dans la liste) l'état de la mémoire est le suivant :

Exercice7 : Liste doublement chaînée triée

Soit la déclaration suivante :

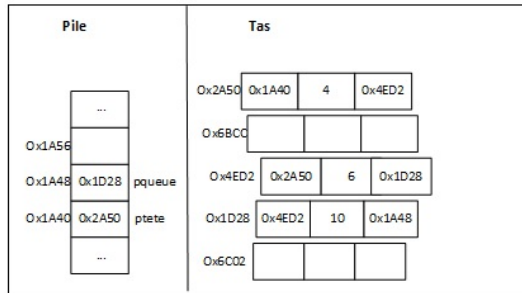
Type

tEnreg = Enregistrement

entier data

↓tEnreg psuiv

//pointeur sur l'element suivant



↓tEnreg pprec //pointeur sur l'element precedent

finEnregistrement

Variable

↓tEnreg ptete //pointeur de tete de liste

↓tEnreg pqueue // pointeur de queue

L'objectif de cet exercice est de créer une liste doublement chaînée d'éléments entiers où les ajouts s'effectuent de façon à ce que l'ordre reste croissant.

Ecrire en pseudo-code les fonctions suivantes permettant d'ajouter et de supprimer un élément de la liste :

1. FONCTION supprime(↓tEnreg : psuppr, ptete, pqueue)
psuppr étant un pointeur sur l'élément à supprimer
2. FONCTION ajout(Entier : n,
↓tEnreg : ptete, pqueue)

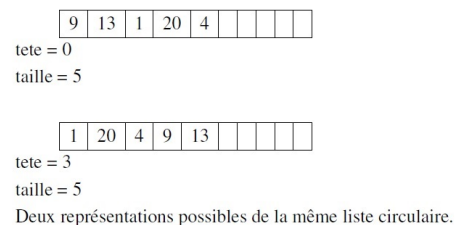
Exercice8 (Extrait de l'examen 2015-2016) : liste circulaire

Une liste circulaire peut être représentée par les structures de données suivantes :

1. Tableau à taille variable :

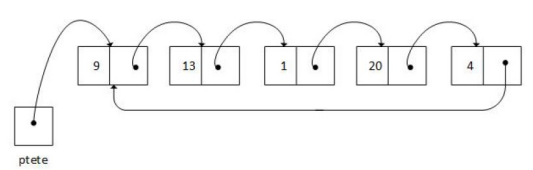
Cette représentation d'une liste circulaire doit vérifier les propriétés suivantes :

- les *taille* éléments de la liste occupent les *taille* premières cases du tableau ;
- pour tout éléments de la liste, si sa valeur est dans la case i du tableau, la valeur de son successeur est dans la case :
$$\begin{cases} i + 1 & \text{si } i < \text{taille} - 1 \\ 0 & \text{si } i = \text{taille} - 1 \end{cases}$$
- la valeur de la tête de la liste se trouve dans la case *tete*.

<u>Constante</u> TAILLE_MAX = 10 <u>Type</u> tTab = Tableau[TAILLE_MAX] d'entier liste = <u>Enregistrement</u> entier tete //position de la tete de liste entier taille //longueur de la liste < TAILLE_MAX tTab Tab <u>finEnregistrement</u>	
---	---

Remarque : la tête de liste ne se trouve forcément pas dans la case 0 du tableau.

2. Liste simplement chaînée :

<u>Type</u> liste = <u>Enregistrement</u> entier valeur ↓liste psuiv <u>FinEnregistrement</u> <u>Variable</u> ↓liste ptete //pointeur de tête de liste	
--	---

3. Liste doublement chaînées :

le champ ↓liste pprec (le lien vers le prédécesseur) est ajouté à la liste simplement chaînée ; le prédécesseur du premier élément est le dernier élément.

Pour chacune de ces 3 structures, écrire les fonctions suivantes et calculer leur complexités (temporelle et spatiale) :

(a) **FONCTION** ajoute_tete(entier n,
↓liste l)

qui ajoute un élément contenant la valeur n en tête d'une liste circulaire l ;

(b) **FONCTION** afficher(↓liste l) qui affiche les éléments d'une liste circulaire en commençant par la tête de la liste et s'arrêtant juste avant de revenir sur la tête de liste.

4. Quelle est la meilleure structure de données pour une liste circulaire ? Justifiez.

Chapitre 5

Implémentation en langage C

5.1 Généralités

5.1.1 C : un langage compilé

Un programme C est décrit par un fichier texte, appelé **fichier source**. Le fichier source est dans un premier temps passé dans un "traducteur" (le compilateur) qui va générer, sauf erreur, un exécutable qui est un **fichier binaire** compréhensible par l'ordinateur.

Les fichiers sources sont écrits dans un éditeur de texte, il en existe plusieurs.

Il existe également un grand nombre de compilateurs C selon les différentes plateformes (Windows, Linux, MAC etc...).

Il existe également un grand nombre d'Environnement de Développement Intégré (IDE) qui regroupe l'éditeur de textes et les outils de compilations dans un seul logiciel et qui facilite la correction du programme.

Dans ce cours, nous essayerons de vous laisser libre champs sur l'IDE avec lequel vous êtes à l'aise; tout comme vous pouvez rester sur le terminal ou l'invite de commande pour lancer les opération de compilation.

5.1.2 Les mots-clés

Un certains nombres de mots sont réservés pour le langage C. voici une liste non-exhaustive : auto; break; case; char; const; continue; default; do; double; else; enum; extern; float; for; goto; if; int; long; register; return; short; signed; sizeof; static; struct; switch; typedef; union; unsigned; void; volatile; while.

5.1.3 Structure générale en programme C

De manière générale, un programme C consiste en la construction des fonctions qui peuvent s'appeler entre elles.

Exemple :

```
/*
*****
**
*   \file PrixTTC.c
*   \author pryde MBOUH GHOGOMU
*   \date 07/11/2018
*   \version 1.0
*   \brief programme qui demande un prix au clavier,
*           prend en compte la TVA et affiche le résultat
**
*****
// Les Directives du préprocesseur
#include<stdio.h>
#define TVA 19.54      // la TVA du Cameroun

// la définition des types

// la déclaration des variables Globales : non conseillé !

// La définition des fonctions

/**
* \fn float ajout_TVA(float prix_HT)
* \brief ajoute la TVA au prix HT et renvoie le prix TTC
* \param float prix_HT
* \return float prix_HT*(1 + (TVA/100))
**/
float ajout_TVA(float prix_HT) {
    return prix_HT*(1 + (TVA/100));
}
```

```

/**
 * \fn int main()
 * \return l'état de l'exécution (0 : exécution correcte)
 */
int main() {
    //déclaration variable locale
    float prix_TTC, HT;

    printf("Entrer le prix H.T. : ");
    scanf("%f",&HT);
    prix_TTC = ajout_TVA(HT);
    printf("prix T.T.C. : %f \n",prix_TTC);
    system("pause");
    return 0;
}

```

Les différentes rubriques d'un programme C sont :

1. les directives du préprocesseur : ce sont les lignes de code source commençant par dièse (#). Les 2 directives les plus utilisées sont :
 - `#include` qui permet d'inclure un fichier. Ici, le fichier **stdio.h** déclare les fonctions standards d'entrées/sorties (STanDard In/Out), qui feront le lien entre le programme et la console (clavier/écran). Dans l'exemple précédent, on utilise les fonctions **printf** et **scanf**.
 - `#define` qui définit une constante. Dans l'exemple précédent, à chaque fois que le compilateur rencontrera le mot TVA, il le remplacera par 19,54.
2. La définition des types : elle consiste à définir les types dérivés à l'exemple du type énumération et du type enregistrement.
3. La déclaration des variables globales : nous n'avons pas abordé ces types de variables au chapitre 1 parce qu'elles ne sont pas recommandées.
4. la définition des fonctions : elle est délimité par les accolades { }. Elle commence par la déclaration du type de variable que nous avons abordé au chapitre 1 (les variables locales). Cette déclaration consiste à donner le type et le

nom des variables. On peut initialiser une variable à sa déclaration. Mais, on ne peut pas utiliser une variable sans l'avoir déclarée.

Après la déclaration des variables, nous avons les (autres) instructions. Une instruction est une expression terminée par un point-virgule.

Le programme principal (**main()**) est une fonction.

Par définition, toute fonction en C fournit un résultat dont le type doit être défini. Le retour du résultat se fait en général à la fin de la fonction par l'instruction **return**. La fonction qui ne fournit pas de résultat est déclarée comme **void**.

Cependant, on peut se contenter de déclarer le prototype d'une fonction (sans le corps) :

```
type_resultat nom_fonction (type1 arg1, ... , typeN argN);
```

Dans ce cas, la définition de la fonction (avec le corps des instructions qui la compose) peut être déclarée plus loin dans le programme. Contrairement à la définition de la fonction, le prototype n'est donc pas suivi du corps de la fonction (contenant les instructions à exécuter). Le prototype est une instruction, il est donc suivi d'un point-virgule !

5.2 Syntaxe

5.2.1 Les Opérations

Des opérations vu au chapitre 1, nous pouvons faire des observations suivantes :

1. Affectation simple est C est le signe = ; à ne pas confondre l'opérateur d'égalité qui lui est représenté par == ;
2. Si les opérandes des opérations arithmétiques sont de types différents, on les convertit vers le type de l'opérande selon l'ordre de priorité suivant :
 - (a) long double
 - (b) double
 - (c) float
 - (d) unsigned long long
 - (e) long long

(f) unsigned long

(g) long

(h) unsigned int

(i) int

Exemple : $2.0/3$ est équivalent à $2.0/3.0$ et le résultat est de type de la variable qui a la valeur 2.0 ;

Cependant, avec la notion de connu sous le nom de "**Cast**", on peut forcer une variable de changer de type. (Exemple : $(\text{float})x$; la valeur de x avec le type float) ;

3. Concernant la division en C, elle est notée par "/" qu'elle soit entière ou réelle. Cependant, le résultat d'une division entre deux variables entières est un entier.

Exemple : $6/4$ donne pour résultat 1 ; alors que $6.0/4.0$ donne pour résultat 1.5

4. Le reste de la division euclidienne de n par p est noté $n\%p$.
5. Des nouveaux symboles qui permettent d'ajouter (resp. de soustraire) un à une variable entière - ie incrémenter (resp. décrémenter).
- **i++** ou **++i** signifie **i=i+1**. L'opérateur préfixe **++i** (resp. suffixe **i++**) incrémente i avant (resp. après) de l'évaluer ;
 - idem pour **i-** - et **-i** .
6. D'autres symboles (ou fonctions) équivalents :

1	≠	!=
2	ET	&&
3	OU	
4	NON	!
5	≤	<=
6	≥	>=
7	↓	*
8	@	&
9	©	*
10	NbOctet()	sizeof()
11	allouer()	malloc()
12	desallouer()	free()
13	Renvoyer	return

Exemple : ↓entier p → int *p;

5.2.2 Les Structures de contrôle

Sélection simple partielle	if (Expression) Instruction
Sélection simple totale	if (Expression) Instruction1 ; else Instruction2 ;
Sélection multiple	switch(variable) { case constante1 : instruction1 break ; case constante2 : instruction2 break ; ... case constanteN : instructionN break ; default : instruction }
Structures itératives : Boucle Pour	for (cpt = initial ; cpt (op) final ; cpt = cpt+pas) Instruction
Structures itératives : Boucle TantQue	while (Expression) Instruction
Structures itératives : Boucle Repeter	do Instruction while (Expression)

Remarque : Les termes comme "Alors" ou "Faire" n'existe pas en C.

5.2.3 Les instructions d'entrées-sorties

Il faut déjà écrire la directive au préprocesseur **#include <stdio.h>** pour utiliser les fonctions suivantes :

1. La fonction `getchar()` : La fonction `getchar()` permet de récupérer un seul caractère à partir du clavier.

Exemple : `var = getchar()` ; où `var` est de type `char`.

2. La fonction `putchar()` : La fonction `putchar()` permet d'afficher un seul caract-

tère sur l'écran de l'ordinateur. putchar() constitue alors la fonction complémentaire de getchar().

Exemple : putchar(var); où var est de type char.

3. La fonction d'écriture à l'écran formatée printf() : La fonction printf() est une fonction d'impression formatée, ce qui signifie que les données sont converties selon le format particulier choisi. Sa syntaxe est la suivante :

printf("chaîne de contrôle", expression1, . . . , expressionN);

La chaîne de contrôle contient le texte à afficher et les spécifications du format correspondant à chaque expression de la liste. Les spécifications du format ont pour but d'annoncer le format des données à visualiser. Elles sont introduites par le caractère %. Le i-ème format de la chaîne de contrôle sera remplacé par la valeur effective de expression i.

Une spécification simplifiée du i-ème format est de la forme suivante : **%type**

Où type précise l'interprétation à donner à expression i. Les valeurs possibles sont détaillées dans la table suivante :

Format	Conversion en	Écriture
%d	int	Décimale signée
%ld	long int	Décimale signée
%u	unsigned int	Décimale non signée
%lu	unsigned long	Décimale non signée
%o	unsigned int	Octale non signée
%lo	unsigned long	Octale non signée
%x	unsigned int	Hexadécimale non signée
%lx	unsigned long	Hexadécimale non signée
%f	double	Décimale virgule fixe
%lf	long double	Décimale virgule fixe
%e	double	Décimale notation exponentielle
%le	long double	Décimale notation exponentielle
%g	double	représentation la plus courte parmi %f et %e
%lg	long double	représentation la plus courte parmi %lf et %le
%c	unsigned char	Caractère
%s	Char*	Chaine de caractère

4. La fonction de saisie scanf() : La fonction scanf() permet de récupérer les don-

nées saisies au clavier, dans le format spécifié. Ces données sont stockées aux adresses spécifiées par les arguments de la fonction `scanf()` (on utilise donc l'opérateur d'adressage `&` pour les variables scalaires). `scanf()` retourne généralement le nombre de valeurs effectivement lues et mémorisées. La syntaxe est la suivante :

```
scanf("chaîne de contrôle", arg1, . . . , argN);
```

La chaîne de contrôle indique le format dans lequel les données lues sont converties. Elle ne contient pas d'autres caractères (notamment pas de `\n`). Comme pour `printf()`, les conversions de format sont spécifiées par un caractère précédé du signe `%`.

Les données à entrer au clavier doivent être séparées par des blancs ou des `<RETURN>` sauf s'il s'agit de caractères.

5.2.4 Les énumérations

Un objet de type énumération est défini par le mot-clef **enum** et un identificateur de modèle, suivi de la liste des valeurs que peut prendre cet objet :

```
//déclaration du type énumération  
enum Id_Type constante1, constante2, . . . , constanteN;  
//déclaration d'une variable du type énumération  
enum Id_Type nom_variable;
```

En langage C, les objets de type **enum** sont représentés comme des `int`. Les valeurs possibles `constante1`, `constante2`, ... , `constanteN` sont codées par des entiers de 0 à N-1. Dans l'exemple suivant, le type `enum bool` associe l'entier 0 à la valeur `FALSE` et l'entier 1 à la valeur `TRUE` :

```
//on rappelle que le type booleen n'existe pas en C
#include <stdio.h>
/**
 * \enum bool{FALSE, TRUE }
 **/
enum bool {FALSE, TRUE };
int main () {
    enum bool b1 = TRUE;
    printf("b = %d \n",b1);
    return 0;
}
```

On peut modifier le codage par défaut des valeurs de la liste lors de la déclaration du type énumération, par exemple : `enum bool {FALSE = 12, TRUE = 23 } ;`

5.3 Les tableaux

La syntaxe

La déclaration du type tableau et la déclaration de la variable de type tableau en pseudo-code se résume à une seule déclaration en langage C :

type_des_éléments nom_Variable[nombre d'éléments];

Exemple :

```
//imprime les éléments du tableau Note
#define MAX 10
int main() {
    int Note[MAX];
    int i;
    ...
    for (i = 0; i < MAX; i++)
        printf("Note[%d] = %d \n",i,Note[i]);
    ...
}
```

NB :

— La déclaration `int Note[MAX];` alloue en mémoire pour l'objet `tab` un espace

de 10*4 octets consécutifs ;

— Ne pas écrire `int Note[Max]` où `Max` est une variable.

Cependant, contrairement au pseudo code, on peut initialiser un tableau en C lors de sa déclaration par une liste de constantes de la façon suivante :

`type_éléments nom_Var[N] = {constante1,constante2,.. . , constanteN } ;`

Exemple :

```
#define MAX 5
int Note[MAX] = { 14, 17, 3, 18, 12 } ;
int main() {
    int i;
    for (i = 0 ; i < MAX ; i++)
        printf("Note[%d] = %d \n",i,Note[i]);
    ...
}
```

On peut donner moins de valeurs d'initialisations que le tableau ne comporte d'éléments. Dans ce cas, les premiers éléments seront initialisés avec les valeurs indiquées, tandis que les autres seront initialisées à zéro.

Si la dimension `n` n'est pas indiquée explicitement lors de l'initialisation, alors le compilateur réserve automatiquement le nombre d'octets nécessaires ; c'est **la réservation automatique**.

Tableau multidimensionnel en C

En C, un tableau à 2 dimensions se déclare comme à l'exemple suivant : `int Mat[10][20]` ; On accède à un élément du tableau par l'expression `Mat[i][j]`.

Pour initialiser un tableau à plusieurs dimensions à la compilation, on utilise une liste dont chaque élément est une liste de constantes :

```

#include <stdio.h>
#define L 3
#define C 2
int tab[L][C] = {{1, 2}, {14, 15}, {100, 200}};
int main() {
    int i, j;
    for (i = 0; i < L; i++) {
        for (j = 0; j < C; j++)
            printf("tab[%d][%d]=%d \n", i, j, tab[i][j]);
    }
    return 0;
}

```

On vient de définir la matrice suivante :

$$Tab = \begin{bmatrix} 1 & 2 \\ 14 & 15 \\ 100 & 200 \end{bmatrix}$$

Si l'on souhaite utiliser la réservation automatique, il faudra spécifier toutes les dimensions sauf la première. L'exemple suivant réserve $2 \times 3 \times 4 = 24$ octets.

```
int Mat[][3] = {{1, 0, 1}, {0, 1, 0}};
```

$$Mat = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

5.3.1 Les chaînes de caractère en C

Nous avons les mêmes considérations en C qu'en pseudo-code concernant les chaînes de caractères. Cependant, les initialisations suivantes sont autorisées :

— initialisation par une liste de constantes caractères

Exemple : `char ch[3] = 'a', 'b', 'c';`

— initialisation par une chaîne littérale

Exemple : `char ch[3] = "abc";`

Cependant, lors de l'initialisation, on peut ne pas indiquer la taille et le compilateur comptera le nombre de caractères de la chaîne littérale pour dimensionner correctement le tableau (sans oublier le caractère de fin de chaîne `'\0'`).

Exemple : `char ch[] = "ch aura 22 caractères";`

Il est également possible de donner une taille égale au nombre de caractères de la

chaîne. Dans le cas, le compilateur comprend qu'il ne faut pas rajouter le caractère de fin de chaîne ;

Exemple : `char ch[12]="pas conseillé";`

5.3.2 Les différentes gestions dynamiques

Concernant la gestion des tableaux dont la taille n'est connue qu'au moment de l'exécution ou la gestion des tableaux à bidimensionnels dont les lignes n'ont pas toutes le même nombre d'éléments, elles sont identiques en pseudo-code et en C. Cependant, quant-au passage du tableau en paramètre, on peut interdire, en C, la modification des éléments du tableau. On utilise pour cela le mot-clé **const** qui spécifie que la variable associée ne peut pas être modifiée.

Exemple :

```
#include <stdio.h>
```

```
#define TAILLE 4
```

```
/**
```

```
 * \fn void print_tab(const int tab[], int nb_elem)
```

```
 * \brief affiche les nb_elem premiers éléments de Tab[]
```

```
 * \param const int tab[]
```

```
 * \param int nb_elem
```

```
 **/
```

```
void print_tab(const int tab[], int nb_elem) {
```

```
    int i;
```

```
    for (i=0; i < nb_elem; i++)
```

```
        printf("tab[%i]=%i \n",i,tab[i]);
```

```
}
```

```
/**
```

```
 * \fn void incr_tab(int tab[], int nb_elem)
```

```
 * \brief incrémente les nb_elem premiers éléments de Tab[]
```

```
 * \param int tab[]
```

```
 * \param int nb_elem
```

```
 **/
```

```

void incr_tab(int tab[], int nb_elem) {
    int i;
    for (i=0; i < nb_elem; i++)
        tab[i]++;
}

/**
 * \fn int main()
 * \return l'état de l'exécution (0 : exécution correcte)
 */
int main() {
    int t[TAILLE] = { 1, 2, 3, 4 };
    incr_tab(t, TAILLE);
    print_tab(t, TAILLE);
    return 0;
}

```

Ainsi, dans la fonction `print_tab`, on ne peut pas modifier le contenu du tableau passé en paramètre.

Cas des tableaux de chaînes de caractères

On peut initialiser un tableau de ce type par :

```
char * jour[] = {"lundi", "mardi", "mercredi", "jeudi", "vendredi", "samedi", "dimanche"};
```

Une boucle d'impression des valeurs du tableau `jour` pourra s'écrire :

```

#define NB_JOUR 7

int i;
for (i=0; i < NB_JOUR; i++)
    printf("%s\n", jour[i]);

```

5.3.3 Gestion des arguments de la ligne de commande

Les tableaux de pointeurs vers des chaînes de caractères sont une structure de données très importante puisqu'elle est utilisée dans la transmission de paramètres lors de l'exécution d'un programme.

Lorsqu'un utilisateur exécute le programme avec les paramètres `param_1`, `param_2`,... ,`param_N`, l'interpréteur collecte tous ces mots sous forme de chaîne de caractères, crée un tableau de pointeurs vers ces chaînes et lance la procédure `main()` en lui passant deux paramètres :

1. un entier contenant la taille du tableau (appelé classiquement `argc`);
2. le tableau des pointeurs vers les chaînes (traditionnellement `argv`).

Pour que le programmeur puisse exploiter ces éléments, la fonction `main()` doit être définie de la façon suivante :

int main(int argc, char * argv[]) {...}

```
/*  
**  
*      \file prog.c  
*      \brief programme qui utilise les paramètres  
**/  
#include <stdio.h>  
int main(int argc, char * argv[]) {  
    int i;  
    printf("Nom du programme : %s \n", argv[0]);  
    for (i=1 ; i < argc ; i++)  
        printf("Paramètre %i : %s \n",i,argv[i]);  
    return 0;  
}
```

5.4 Les enregistrements

Le type d'un enregistrement est appelé type structuré en langage C ; les enregistrements sont parfois appelés structures, en analogie avec ce langage.

5.4.1 Déclaration et utilisation

L'utilisation pratique d'une structure se déroule de la façon suivante :

1. On commence par déclarer la structure elle-même. Le modèle général de cette déclaration est le suivant :

```
struct nom_structure {  
    type_1 membre1 ;  
    type_2 membre2 ;  
    ...  
    type_n membreN ;  
};
```

2. Pour déclarer une variable de type structure correspondant au modèle précédent, on utilise la syntaxe :

struct nom_structure identificateur_var ;

Ou bien, si le type n'a pas encore été déclaré au préalable :

```
struct nom_structure {  
    type_1 membre1 ;  
    type_2 membre2 ;  
    ...  
    type_n membreN ;  
} identificateur_var ;
```

3. On accède aux différents membres d'une structure grâce à l'opérateur membre de structure, noté ".". Le i-ème membre de la variable est désigné par l'expression :

identificateur_var.membrei

Ainsi, le programme suivant définit la structure complexe, composée de deux champs de type double et calcule la norme d'un nombre complexe.

```
#include <math.h>
```

```
/**
```

```
*      \struct Tcomplexe
```

```
*      \brief structure pour les nombre complexe
```

```
**/
```

```
struct Tcomplexe {
```

```

        double reelle;      /* !< Partie entière */
        double imaginaire; /* !< Partie imaginaire */
    };      // ne pas oublier le ";"
    int main() {
        struct Tcomplexe z;
        double norme;
        ...
        norme = sqrt(z.reelle * z.reelle + z.imaginaire * z.imaginaire);
        printf("La norme de (%f + i %f) est %f \n", z.reelle, z.imaginaire, norme);
        return 0;
    }

```

5.4.2 Initialisation et affectation d'une structure

Les règles d'initialisation d'une structure lors de sa déclaration sont les mêmes que pour les tableaux. On écrit par exemple :

struct complexe i = {0. , 1.};

5.4.3 Pointeur vers une structure

En reprenant la structure personne précédente, on déclarera une variable de type pointeur vers cette structure de la manière suivante :

struct personne *p;

On peut affecter à ce pointeur des adresses sur des struct personne.

```

struct personne {...};
int main() {
    struct personne pers;
    struct personne * p;
    p = &pers;    // p pointe maintenant vers pers
}

```

Pour accéder aux membres de la structure pointée :

- il faudrait écrire normalement (*p).nom;
- mais on utilise en pratique l'écriture simplifiée p->nom.

5.4.4 Le type Union

Une union est une structure de donnée qui sert à implémenter la partie variable d'un enregistrement à champs variable.

```
/**
 *      \union data
 *      \brief soit on a un entier, soit un reel
 **/
union data{
    int n;    /* !< soit une valeur de type entier n*/
    float x;  /* !< Soit une valeur de type réel x*/
};

union data u;
printf("entrer un reel");
scanf("%f",&u.x);
printf("le contenu de u est : %d", u.n);
...
```

5.4.5 Définition des synonymes des types avec typedef

Pour alléger l'écriture des programmes, on peut affecter un nouvel identificateur à un type à l'aide du mot-clé typedef :

typedef <type> <synonyme> ;

Exemple :

```
typedef unsigned char UCHAR;
typedef struct {
    double x, y
} POINT;
typedef POINT * P_POINT;    //pointeur vers un POINT
```

A partir de ce moment :

1. l'identificateur UCHAR peut être utilisé comme une abréviation pour le type unsigned char ;

2. l'identificateur POINT peut être utilisé pour spécifier le type de structure associé ;
3. P_POINT permet de caractériser un pointeur vers un POINT ;

Et on pourrait avoir les variables suivantes :

UCHAR c1, c2, tab[100];

POINT point;

P_POINT pPoint;

5.4.6 Les listes chaînées

Tout comme en pseudo code, l'élément de base de la chaîne est une structure appelée cellule qui contient la valeur d'un élément de la liste et un pointeur sur l'élément suivant. Le dernier élément pointe sur le pointeur NULL (défini dans **stddef.h**). La liste est alors définie comme un pointeur sur le premier élément de la chaîne.

Exemple :

```
#include <stddef.h>

/**
 *      \struct toto
 *      \brief Liste chaînée des entiers
 */
struct toto {
    int data;    /* !< un champ data de type int*/
    struct toto *next; /* !< next pointe vers une struct toto*/
};
typedef struct toto *liste;
liste l = NULL;
...
```

La liste chaînée est définie par la structure *struct toto*. Grâce au mot-clef typedef, on peut définir le type liste, synonyme du type pointeur sur une struct toto. On peut alors définir une variable *l* que nous avons initialisé à NULL (liste l = NULL;). Ainsi, pour insérer un élément en tête de liste, on utilise la fonction

suivante :

```
liste insere(int elt, liste Q) {  
    liste L;  
    L = (liste)malloc(sizeof(struct toto));  
    L->data = elt;  
    L->next = Q;  
    return L;  
}
```


Chapitre 6

Les arbres

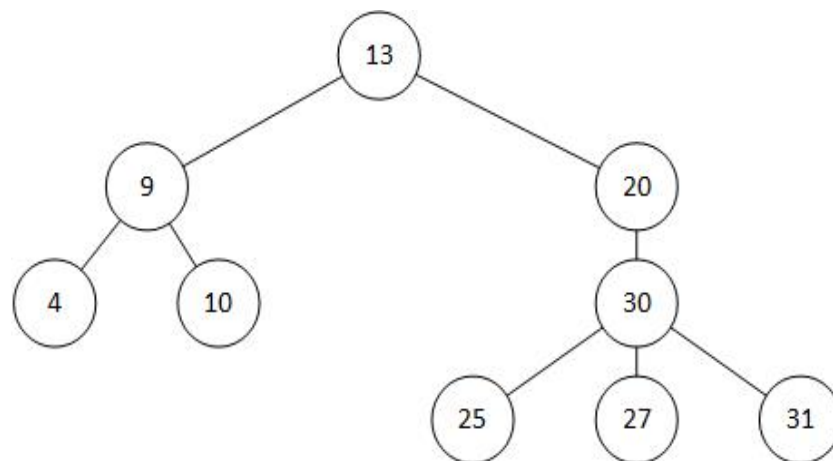
6.1 Généralités

6.1.1 Définition

Les structures de données que nous avons vu jusqu'ici (tableaux, listes, piles, files) sont linéaires, dans le sens où elles stockent les éléments les uns à la suite des autres : on peut les représenter comme des éléments placés sur une ligne.

Un arbre est une structure de données constituée des éléments qui ne sont pas nécessairement sur une ligne ; ces éléments peuvent contenir plusieurs ramifications. ces éléments sont appelés **nœud**.

Les nœuds peuvent avoir des enfants (qui sont d'autres nœuds).



Sur la schéma, le nœud 9 a pour enfant les nœuds 4 et 10, et est lui-même l'enfant

du nœud 13.

Deux types de nœuds ont un statut particulier :

- les feuilles : Les nœuds qui n'ont aucun enfant ;
- la racine : le nœud qui n'a pas de père. Il est unique, sinon on a une autre structure de données qu'on verra au niveau supérieur.

Sur le schéma, la racine est 13 et les feuilles sont 4, 10, 25, 27 et 31.

6.1.2 Représentation d'un arbre

On peut représenter un arbre par une structure de données chaînée dans laquelle chaque nœud contient un champs data (par exemple un entier ou tout autre information) et les autres champs qui sont les pointeurs sur les autres nœuds (ces champs varient selon le type d'arbre) :

- arbre binaire :

On peut utiliser les champs *pere*, *gauche* et *droit* pour stocker les pointeurs vers le père, le fils gauche et le fils droit de chaque nœud d'un arbre ;

Soit un nœud x :

- Si $x.pere = NIL$, alors x est la racine ;
- Si le x n'a pas de fils gauche (resp. droit), alors $x.gauche = NIL$ (resp ; $x.droit = NIL$)

arbre = Enregistrement

Entier val

↓arbre pere

↓arbre droit

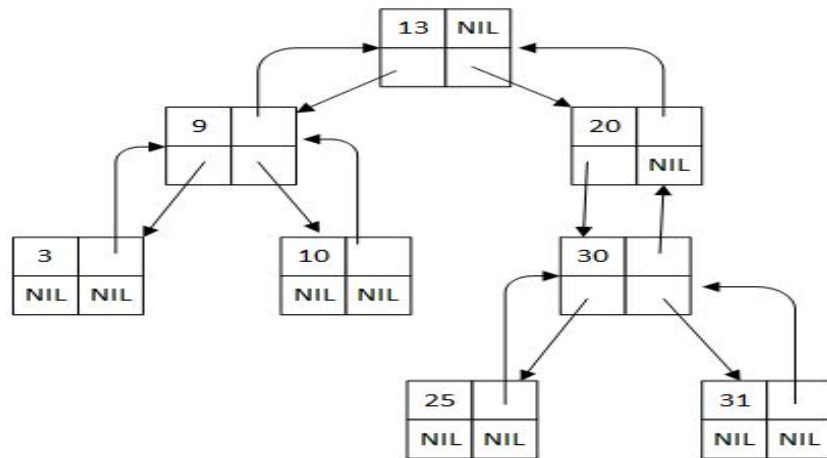
↓arbre gauche

finEnregistrement

Variable

↓arbre : x //pointeur sur la racine de l'arbre

- Arbre avec ramifications non bornées Le schéma de représentation d'un arbre binaire peut être étendu à n'importe quelle classe d'arbres pour laquelle le nombre de fils de chaque nœud vaut au plus une certaine constante k : on remplace les champs gauche et droite par $fil_s_1, fil_s_2, \dots, fil_s_k$. Ce schéma



n'est plus valable quand le nombre de fils n'est pas borné, puisqu'on ne sait pas combien de champs allouer à l'avance. De plus, même si le nombre k de fils est borné par une grande constante, mais que la plupart des nœuds n'ont qu'un petit nombre de fils, on risque de gaspiller beaucoup de mémoire.

Il existe cependant un autre schéma permettant d'utiliser des arbres binaires pour représenter des arbres ayant un nombre arbitraire de fils. Il a l'avantage de n'utiliser qu'un espace en $\Theta(n)$ pour un arbre quelconque à n nœuds. Au lieu de donner à chaque nœud la liste de ses enfants, on lie les enfants entre eux : chaque enfant a un lien vers son frère, et un nœud a donc juste un lien vers le premier de ses fils (le lien *fils_gauche*). Pour parcourir les autres, il suffit ensuite de passer de frère en frère (le lien *frere_droit*).

arbre = Enregistrement

Entier val

↓arbre pere

↓arbre fils

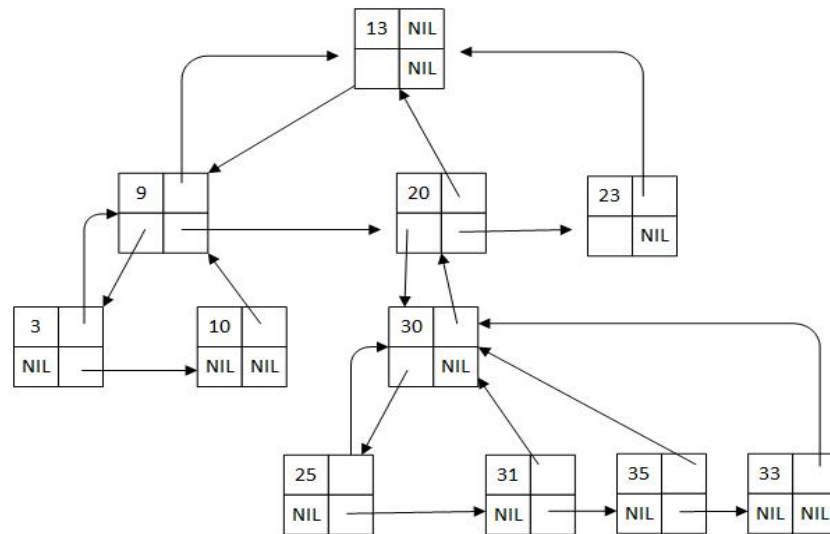
↓arbre frere

finEnregistrement

Variable

↓arbre : x //pointeur sur la racine de l'arbre

Comme précédemment, chaque nœud contient un pointeur sur son père. Tou-



tefois, au lieu d'avoir un pointeur sur chacun de ses fils, chaque nœud x ne possède que deux pointeurs :

- $x.\text{fils}_{\text{gauche}}$ pointe sur le fils le plus à gauche du nœud x
- $x.\text{frere}_{\text{droite}}$ pointe sur le frère de x situé immédiatement à sa droite

Si le nœud x n'a aucun fils, alors $x.\text{fils}_{\text{gauche}} = \text{NIL}$, et si x est le fils le plus à droite de son père, alors $x.\text{frere}_{\text{droite}} = \text{NIL}$.

- Autres représentations d'arbres De nombreux autres schémas sont possibles à l'exemple d'un tableau avec indice. Le meilleur choix dépend du problème à résoudre.

6.2 Manipulation

Pour s'habituer à cette nouvelle structure, on va essayer de d'écrire quelques algorithmes répondant à des questions simples que l'on peut se poser sur des arbres :

6.2.1 Taille

On peut calculer la taille de tout nœud x ; c'est le nombre de nœuds (internes) du sous-arbre enraciné en x (x compris), autrement dit la taille du sous-arbre. Si l'on suppose que $\text{taille}(\text{NIL}) = 0$, alors on a $\text{taille}(x) = 1 + \text{la somme de la taille des fils de } x$.

On procédera par récurrence

FONCTION taille (\downarrow arbre x)

/*Objectif : calcul le nombre de nœud du sous-arbre enraciné en x

x est binaire */

Variable

\downarrow arbre tmp

entier i

Debut

tmp \leftarrow L

Si x = NIL Alors

i \leftarrow 0

Sinon

i \leftarrow 1+taille(x.droit)+taille(x.gauche)

finSi

renvoyer i

fin

6.2.2 Hauteur

La hauteur d'une arbre est la plus grande distance qui sépare un nœud de la racine. On parle parfois de **profondeur** plutôt que de hauteur : cela dépend de si vous imaginez les arbres avec la racine en haut et les feuilles en bas, ou (ce qui est plus courant dans la nature) la racine en bas et les branches vers le haut.

La distance d'un nœud à la racine (la hauteur de ce nœud) est le nombre de nœud sur le chemin entre les deux.

hauteur(arbre) = 1 + la plus grande des hauteurs des fils (ou 0 s'il n'y en a pas)

6.2.3 Liste des éléments

Cette fonction donne les éléments présents dans l'arbre. Il existe plusieurs manières de lister des éléments de l'arbre, et on peut obtenir plusieurs listes différentes, avec les mêmes éléments mais placés dans un ordre différent.

Parcours en profondeur

Parcours en largeur

6.2.4 Autres fonctions

Les fonctions suivantes peuvent servir sur les arbres : RECHERCHER, MINIMUM, MAXIMUM, PRÉDÉCESSEUR, SUCCESSEUR, INSÉRER et SUPPRIMER.