République Algérienne Démocratique et Populaire

Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

Ecole Normale Supérieure Assia Djebar Constantine

Département de l'Informatique et des Mathématiques

2eme année Informatique

Support de cours

Module :

Algorithmique 2

Préparé par : Dr. BETTOU Farida
E-mail : bettou_fn@yahoo.fr

Année 2022

Module : Algorithmique 2

Auteur : Dr. BETTOU Farida

Pré-requis : Algorithmique 1

Description du module

Niveau :2eme Année informatique

Volume horaire hebdomadaire

- Cours : 1h30
- Travaux Dirigés(TD) :1h30
- Travaux Pratique (TP) :1h30

Coefficient : 05

Objectifs pédagogiques

- Introduire les notions de structure complexes et leur implémentation.
- Apprendre à utiliser des structures de données dynamiques
- Apprendre à programmer des pointeurs et manipuler des listes d'éléments avec leurs chaînages.
- Apprendre à déclarer et utiliser des piles (Le principe de fonctionnement d'une pile en stratégie FILO (First In Last Out)) et des files(Le principe de fonctionnement d'une file en stratégie FIFO (First In First Out))
- Ecrire des algorithmes qui manipulent des piles et des files.
- Comprendre la notion de récursivité et Apprendre à écrire des algorithmes récursifs.
- Apprendre à utiliser et manipuler des données sous forme d'arbre binaire.
- Apprendre à exploiter les opérations appliquées à un arbre binaire et arbre binaire de recherche.
- Acquérir la connaissance des données sur les graphes, technique d'Hachage, structure de fichier et méthode d'indexation.

Table des matières

Liste des Figures……. i
Liste des Tableaux……..... ii

Partie A : Structure de données

Chapitre 01 : les Pointeurs

I. Introduction…….. 01
II. Variables statiques et dynamiques…….. 02
II.1 Variables statiques…… 02
II.2 Variables Dynamiques……. 03
III. Adressage de variables…….. 04
III.1 Adressage direct……. 04
III.2 Adressage indirect ……. 04
IV. Déclaration et initialisation de pointeurs…… 05
V. Operateurs & et *…… 06
VI. Operateurs ++ Et –……. 07
VII. Allocations dynamique de la mémoire…….. 08
VII.1. La fonction *sizeof (stdlib.h)*…….. 08
VII.2. La fonction `malloc(N)` ……. 10
VII.3. La fonction `free(N)` …….. 11

Chapitre 02 : Les listes chaînées

I. Introduction…….. 13
II. Définition……. 13
III. Déclaration d'un maillon d'une liste chainée……... 14
IV. Opérations de base sur les listes chaînées…… 15
IV.1 Parcours d'une liste……. 15
IV.2 Insertion d'un élément dans la liste……. 16
IV.2.A. Ajout en début de liste ……. 17
IV.2.B. Ajouter à la fin…….. 18

IV.2.C. Ajout en milieu de liste……. 19
IV.3. Supprimer un élément……. 21
IV.3.A Suppression en début de liste…….. 21
IV.3.B. Suppression en fin de liste…….. 22
IV.3.C. Suppression en milieu de liste…… 22
V. Cas particulaire des listes chaînées……... 23
V.1 Liste doublement chaînée…… 23
V.2 Liste circulaire……. 24
V.3 Liste circulaire doublement chaînée……. 24

Chapitre 03 : Les Piles

I. Principe…… 25
II. Opérations sur les piles…….. 26
III. Modélisation et Implémentation des piles……. 26
III. 1 Implémentation par des tableaux (Statique)……. 27
III.1.a Déclaration …… 27
III.1.b Initialisation d'une pile…….. 28
III.1.c Vérification de pile vide……. 28
III.1.d Vérification de pile pleine…….. 28
III.1.e Ajout d'une nouvelle valeur à une pile……... 28
III.1.f Suppression d'une valeur de la pile…… 29
III. 2 Implémentation par des Listes chainées(Dynamique)…… 29
III.2.a Déclaration …… 29
III.2.b Initialisation d'une pile ……. 29
III.2.c Vérification de pile vide……. 29
III.2.e Ajout d'une nouvelle valeur à une pile …… 30
III.2.f Suppression d'une valeur de la pile…… 30

Chapitre 04 : Les files et les file d'attente

I. Principe…….. 31

II. Modélisation……..... 32

II.1 Modélisation par tableau circulaire……... 32

II.2 Modélisation par liste chaînée…….. 34

III. Opérations sur les files d'attente……..... 35

IV. Implémentation de file d'attente …….. 36

IV.1 Implémentation d'une file d'attente F statique……. 36

IV.1.a Fonction d'initialisation de la file d'attente…… 36

IV.1.b Fonction pour vérifier si la file d'attente est vide……..... 36

IV.1.c Fonction pour vérifier si la file d'attente est pleine……. 36

IV.1.d Fonction pour enfiler une valeur à la file d'attente (Enqueue)……... 37

IV.1.e Fonction pour défiler une valeur de la file d'attente (Dequeue)……. 37

IV.2 Implémentation d'une file d'attente F dynamique…….. 38

IV.2.a Fonction d'initialisation de la file d'attente……. 38

IV.2.b Fonction pour vérifier si la file d'attente est vide…… 38

IV.2.c Fonction d'enfiler une valeur à la file d'attente (en queue de la file)…….. 38

IV.2.d Fonction pour défiler une valeur de la file d'attente (En tête de file)…….. 39

V. Exemple d'utilisation ……. 39

Chapitre 05 : Les arbres

I. Définition…… 40

II. Terminologie…….. 41

III. Exemple d'utilisation des arbres…… 42

III.1 Représentation des expressions arithmétique …… 42

III.2 Représentation des chaines de caractère……. 43

III.3 Représentation des arbres généalogiques……... 43

IV. Typologie des arbres……. 43

V. Arbres binaires…… 44

V.1 Définition…… 44
V.2 Types d'arbres binaires…… 45
V.3 Transformation d'un arbre de degré quelconque en arbre binaire…….. 46
VI. Implémentation des arbres…… 48
VI.1 Représentation statique…….. 48
VI.2 Représentation dynamique……. 48
VII. Traitement sur les arbres…….. 49
VII.1 Parcours en profondeur…… 50
VII.2 Parcours en largeur ou hiérarchique…….. 52
VIII. Arbre binaire de recherche……. 53
VIII.1 Rechercher un élément……. 54
VIII.2 Insertion d'un élément dans un ABR……..... 55
VIII.3 Suppression d'un élément d'un ABR……... 56

Chapitre 06 : Technique de hachage

I. Introduction…… 59
II. Type d'organisation des clés dans la table……. 60
II.1 Accès séquentiel……. 60
II.2 Table triée……... 60
II.3 Hachage (HashCoding) ……... 60
III. Fonction de hachage…….. 61
III.1 Hachage par Extraction…….. 61
III.2. Hachage par Compression……. 62
III.3. Hachage par division……. 63
III.4 Hachage par multiplication…… 64
IV. Méthode de résolution de collision…… 65
IV.1 Par chaînage (méthodes indirectes)…… 66
IV.1. a Hachage avec chaînage séparé …… 66
IV.1.b Hachage coalescent…….. 67

IV.2 Par calcul (méthodes directes)…… 70

IV.2.a Hachage linéaires……. 70

IV.2.b Double Hachage……... 71

Chapitre 07 : Les Graphes

I. Définition ……. 72

I.1 Définition Graphe Orienté……. 73

I.2 Définition Graphe non Orienté ……..... 73

II. Utilisation des graphes…….. 74

III. Représentation ……. 74

III.1 Liste d'adjacence ……..... 75

III.2 Matrice d'adjacences…….. 75

IV. Parcours de graphes (en profondeur et largeur)…… 76

IV.1 Parcours en profondeur…….. 76

IV.2 Parcours en largeur…… 78

Chapitre 08 : Récursivité

I. Introduction…… 79

II. Définition…… 80

III. Type de récursivité……. 81

III.1. Récursivité simple ou linéaire…… 81

III.2. Récursivité croisée ou mutuelle……. 81

IV. Conception d'algorithmes récursifs……..... 82

V. Construction de la récursivité…….. 82

VI. Déroulement d'un algorithme récursif……..... 83

VI.1 Version itérative……. 84

VI.2 Version récursive……..... 84

VII. Conclusion …… 85

Partie B : Structure de Fichiers

Chapitre 01 : Les Fichiers

I. Raisons de l'utilisation des mémoires secondaires …….. 86

II. Définition……. 86

III. Fichiers de données et Fichiers textes……. 87

IV. Déclaration de fichiers……..... 87

V. Fichiers à accès séquentiel et fichier à accès direct……... 88

V.1 Fichiers en organisation séquentielle……... 88

V.2 Fichiers en organisation relative (ou fichiers à accès direct)……. 89

VI. Fichiers physique et fichiers logique……. 89

VII. Fonctions sur les fichiers textes…… 90

VII.1 Ouverture des fichiers (La fonction `fopen`) ……... 90

VII.2 Fermeture de fichiers (la fonction `fclose`)……. 91

VII.3 Lecture de fichiers (les fonctions `fgetc, fgets, fscanf`)…… 92

VII.4 Ecriture dans un fichier……. 94

VII.5 Détection de fin de fichier (`feof`)…… 96

VII.6 Accès aux éléments d'un fichier …… 96

VII.7 Détection d'erreur…… 99

Chapitre 02 : Les méthodes d'indexation

I. Introduction……. 101

II. Index primaire…….. 102

II.1 Index à un niveau …….. 102

II.2 Index multiniveaux…… 104

III. Index secondaire……. 105

IV. Avantage des méthodes d'index ……... 108

Annexe (TD)……...…… 109

Références bibliographiques……

Liste des Figures	
Figure 01 : *élément d'un liste chainée*…….	14
Figure 02 : *Liste simplement chainée*……...	15
Figure 03 : *schéma d'ajout en début de liste*……	17
Figure 04 : schéma d'ajout à la fin……..	18
Figure 05 : *schéma d'ajout en milieu de liste (avant un élément dont l'adresse est dans "A"*)……	19
Figure 06 : *schéma d'ajout en milieu de liste (après un élément dont l'adresse est dans "A"*)……	20
Figure 07 *: Schéma de Suppression un élément en début de liste*……	22
Figure 08 : *Schéma de Suppression un élément en fin de liste*…….	22
Figure 09 : *Schéma de Suppression un élément en milieu de liste*……...	23
Figure 10 : *Liste doublement chaînée*……..	24
Figure 11 : *Liste circulaire*……..	24
Figure 12 : *Liste circulaire doublement chaînée*……	24
Figure 13 : *pile d'assiette* ……	26
Figure 14 : *Modélisation d'une pile*……	27
Figure 15 : *Modélisation d'une file d'attente par tableau circulaire*……..	32
Figure 16 : *Etat d'une file d'attente après enfiler 15, 20, 30 et 40* ……	33
Figure 17 : *Etat d'une file d'attente après défiler 15* ……	33
Figure 18 : *Etat d'une file d'attente après enfiler 50* …….	34
Figure 19 : *Modélisation d'une file d'attente par une liste chainée*……	35
Figure 20 : *Exemple d'arbre*……	40
Figure 21 : *Terminologie d'un arbre*……	41
Figure 22 : *Exemple de représentation des expressions arithmétique* ……	42
Figure 23 : *Exemple de représentation des chaines de caractère*……	43
Figure 24 : *Exemple de représentation des arbres généalogique*……	43
Figure 25 : *Exemple d'arbre binaire*……	44
Figure 26 : A*rbre binaire Plein*……	45
Figure 27 : A*rbre binaire Complet*……..	45

Figure 28 : A*rbre binaire parfait*……	46
Figure 29 : *exemple d'arbre m-aire*…….	46
Figure 30 : *Représentation dynamique d'un arbre*……	48
Figure 31 : *Arbre binaire*……	50
Figure 32 : *Parcours d'un arbre en largeur*……	53
Figure 33 : *Arbre binaire de recherche*……	54
Figure 34 : *exemple d'ajout des éléments à un ABR*……	56
Figure 35 : *Exemple de suppression d'une feuille à un ABR* …….	57
Figure 36 : *Exemple de suppression d'un nœud qui a un seul fils à un ABR*……..	57
Figure 37 : *Exemple de suppression d'un nœud qui a deux fils à un ABR*……...	58
Figure 38 : *schéma de table de hachage*…….	60
Figure 39 : *exemple de Codge*…….	61
Figure 40 : *Exemple d'Hachage avec chaînage séparé*……..	66
Figure 41 : *Exemple d'Hachage* coalescent ……	68
Figure 42 : *Exemple d'Hachage* coalescent *secondaires*…….	69
Figure 43 : *Exemple de graphe orienté*……	73
Figure 44 : *Exemple de graphe non orienté*……	73
Figure 45.a : *Liste d'adjacence (Graphe non orienté)*……	75
Figure 45.b : *Liste d'adjacence (Graphe orienté)*……	75
Figure 46 : *Exemple de graphe*……	76
Figure 47 *: Schéma explicatif de la récursivité simple*……	81
Figure 48 *: Schéma explicatif de la récursivité croisée*……..	82
Figure 49 *: Schéma explicatif des appels récursif*……	83
Figure 50 *: Récursivité-factorielle*……...	85
Figure 51 : *Ouverture d'un fichier avec chemin absolu*……..	91
Figure 52 : *Ouverture d'un fichier avec chemin relatif*…….....	91
Figure 53 : *Lecture d'un fichier avec la fonction fgetc*……	92
Figure 54 : *Lecture d'un fichier avec la fonction fgets*……	93
Figure 55 : *lecture d'un fichier avec la fonction fscanf*…….	94
Figure 56 : *écriture d'un fichier avec la fonction fputc*……..	95
Figure 57 : *Lecture d'un fichier avec la fonction fputs*……	95
Figure 58 : *Ecriture d'un fichier avec la fonction fprintf*……	96
Figure 59 *: Position du curseur dans le fichier*……	97

Figure *60* ***:*** *exemple de détection d'erreur*……...	99
Figure 61 *: Schéma explicatif d'Index à un niveau*……	102
Figure 62 *: Schéma explicatif d'Index à 2 niveaux*……	105
Figure 63 *: Schéma explicatif d'Index secondaire*…….	106

Liste des Tableaux	
Tableau 01 : *Etapes de transformation d'un arbre de degré quelconque en arbre binaire*……	47
Tableau 02 : *Représentation statique d'un arbre*……	48
Tableau 03 : *Exemple de parcours en profondeur*……	52
Tableau 04 : *Méthode principales de hachage*……	61
Tableau 05 : *Exemple d'Hachage par extraction des bits*……	62
Tableau 06 : *Exemple d'Hachage par Compression des bits*……	62
Tableau 07 : *Exemple 2 d'Hachage par Compression des bits*……	63
Tableau 08 : *Exemple d'Hachage par division*……	64
Tableau 09 : *Exemple tableau des valeurs de hachage*……	66
Tableau 10 : *Exemple 2 tableau des valeurs de hachage*……	68
Tableau 11 : *Comparaison entre fichiers binaires et fichiers textes*……	87

Partie A

Chapitre 01 : Les pointeurs

I. Introduction

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

En outre, les pointeurs nous permettent d'écrire des programmes plus compacts et plus efficients et fournissent souvent la seule solution raisonnable à un problème. Ainsi, la majorité des applications écrites en C profitent extensivement des pointeurs.

Le revers de la médaille est très bien formulé par Kernighan & Ritchie dans leur livre 'Programming in C':

" ... Les pointeurs étaient mis dans le même sac que l'instruction ***goto*** *comme une excellente technique de formuler des programmes incompréhensibles. Ceci est certainement vrai si les pointeurs sont employés négligemment, et on peut facilement créer des pointeurs qui pointent 'n'importe où'. Avec une certaine discipline, les pointeurs peuvent aussi être utilisés pour programmer de façon claire et simple. C'est précisément cet aspect que nous voulons faire ressortir dans la suite. ...".*

II. Variables statiques et dynamiques

Revenons aux variables. En premier lieu, on distingue les variables *statiques* et les variables *dynamiques*. Les premières existent aussi longtemps que le programme, et sont initialisées (éventuellement) en même temps que lui. Les secondes sont créées (et initialisées éventuellement) à un certain moment dans le programme, puis détruites ultérieurement ; ce processus peut se produire plusieurs fois pour une même variable.

II.1 Variables Statiques

Les variables statiques possèdent les caractéristiques suivantes :

- Leurs formes et leurs tailles sont prédéterminées (déclarées) à l'avance dans le programme.
- Elles existent en mémoire durant toute l'exécution du bloc (programme ou procédure) dans lequel elles sont définies.
- Les variables statiques sont référencées directement par des identificateurs (noms des variables).

Il existe trois sortes de variables statiques. Les *variables globales* sont déclarées en dehors de toute fonction, souvent au début du fichier. Ces variables sont automatiquement initialisées, même si aucune initialisation n'est explicitée par le programmeur : dans ce cas, elles sont mises à zéro.

Deuxième type de variables statiques, celles qui sont *explicitement déclarées* ainsi avec le mot réservé `static` dans une fonction. Une telle variable est également initialisée à zéro si aucune initialisation explicite n'est indiquée. Troisième type de variables statiques, celles qui sont déclarées dans le bloc principal de la fonction main, Ces variables sont en fait des variables locales de main , mais comme la durée de parcours de main est celle du programme entier, elles existent tout au long du programme. De telles variables ne sont cependant visibles que dans la fonction main. De plus elles ne sont jamais initialisées automatiquement.

Exemple

```
#include <stdio.h>
#include <stdlib.h>
int glob1, glob2 = 10; // statiques globales
void f(int arg1, int arg2)
{
     static int stat = 1;  // statique explicite
     arg2 = (arg1+1)*(stat+1);}
```

```
void g(void)
{
     int loc = 0;   // locale
     for (int i = 1; i <= glob2; i++)   // i est locale
           i = (glob1+1)*(loc+1) ;
}
int main()
{
     int mloc1, mloc2 = 7; // statique de main
     f(mloc2, mloc1);
     if (mloc1 > 0) {
        for (int k = 1; k <= 3; k++) f(0,0);
        mloc1 = 0;
        }
     g();
     return 0;
}
```

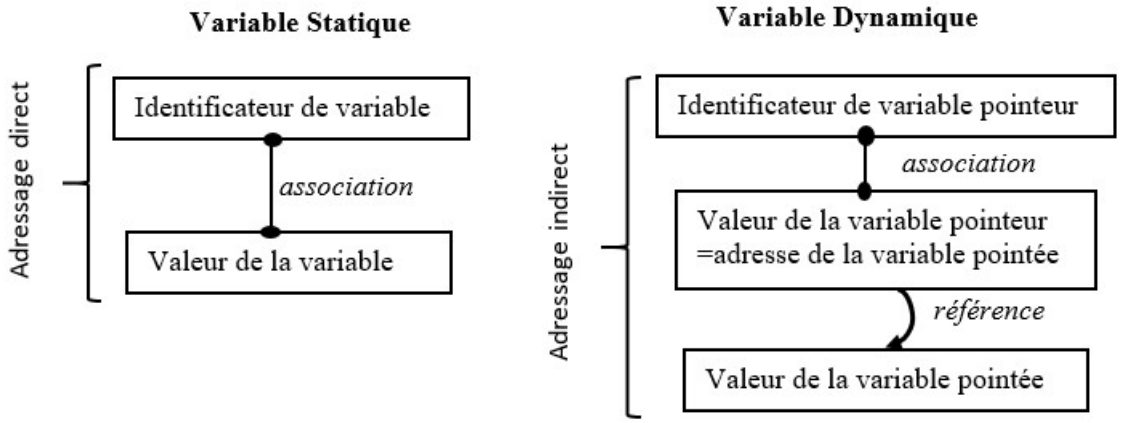
II.2 Variables Dynamiques

Contrairement aux variables statiques, les variables dynamiques :

- Peuvent être créées et utilisées au fur et à mesure des besoins.
- Peuvent être détruites à tout moment afin de récupérer l'espace mémoire devenu inutile.
- Permettent l'insertion et la suppression d'éléments sans toucher au reste des données.

Il existe deux sortes de variables dynamiques. Les variables dynamiques du programmeur sont celles qu'il crée explicitement dans le tas à l'aide de `malloc`, et qu'il détruit quand bon lui semble par la fonction `free`.

Les autres variables dynamiques sont dites *automatiques*, parce qu'elles sont automatiquement créées et détruites par le programme de façon transparente pour le programmeur. Elles sont placées dans la pile. Il en existe différentes espèces.



III. Adressage de variables

Il existe deux modes d'adressage principaux : l'adressage direct, et l'adressage indirect.

III.1 Adressage direct

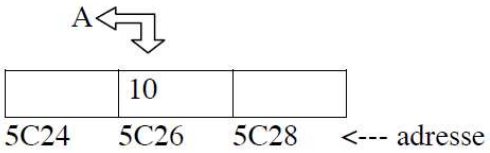
Accès au contenu d'une variable par le nom de cette variable.

Exemple :

int A ; On déclare une variable A,

A=10 ; On lui affecte la valeur 10 (son contenu vaut 10)

Représentation en mémoire :



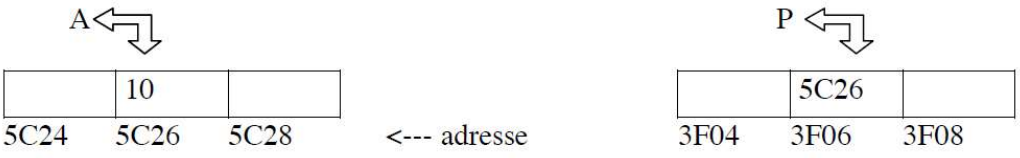
III.2 Adressage indirect

Si on ne veut, ou on ne peut utiliser le nom d'une variable A, on peut copier l'adresse de cette variable dans une variable spéciale P, appelée pointeur. On peut ainsi, par la suite, retrouver l'information de A en passant par P.

Représentation en mémoire :

A : variable contenant la valeur 10

P : pointeur (variable) contenant l'adresse de A (&A ou 5C26)



IV. Déclaration et initialisation de pointeurs

En C, chaque pointeur est limité à un type de données. Il peut contenir l'adresse d'une variable simple de ce type, ou l'adresse d'une composante d'un tableau de ce type. Si un pointeur P contient l'adresse d'une variable A, on dit que P pointe sur A.

Définition :

Un pointeur est une variable spéciale qui peut contenir l'adresse d'une autre variable.

Remarque :

Pointeurs et noms de variables ont le même rôle : ils donnent accès à un emplacement dans la mémoire interne de l'ordinateur, il faut tout de même bien faire la différence :

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

Syntaxe :

```
Type_variable_pointée *Pointeur ;
```

Exemples :

Par exemple, si nous souhaitons créer un pointeur sur int (c'est-à-dire un pointeur pouvant stocker l'adresse d'un objet de type int) et que nous voulons le nommer « ptr », nous devons écrire ceci.

```
int *ptr;
```

L'astérisque peut être entourée d'espaces et placée n'importe où entre le type et l'identificateur. Ainsi, les trois définitions suivantes sont identiques.

```
int *ptr;
int * ptr;
int* ptr;
```

Remarque :

Notez bien qu'un pointeur est toujours typé. Autrement dit, vous aurez toujours un pointeur sur (ou vers) un objet d'un certain type (int, double, char, etc.).

Un pointeur, comme une variable, ne possède pas de valeur par défaut, il est donc important de l'initialiser pour éviter d'éventuels problèmes. Pour ce faire, il est nécessaire de recourir à l'opérateur d'adressage (ou de référencement) : & qui permet d'obtenir l'adresse d'un objet. Ce dernier se place devant l'objet dont l'adresse souhaite être obtenue. Par exemple comme ceci,

```
int a = 10;
int *p;
p = &a;
```

Ou, plus directement, comme cela.

```
int a = 10;
int *p = &a;
```

Faites bien attention à ne pas mélanger différents types de pointeurs ! Un pointeur sur `int` n'est pas le même qu'un pointeur sur `long` ou qu'un pointeur sur `double`. De même, n'affectez l'adresse d'un objet qu'à un pointeur du même type.

```
int a;
double b;
int *p = &b; /* Faux */
int *q = &a; /* Correct */
double *r = p; /* Faux */
```

Lorsqu'un pointeur ne contient aucune adresse valide, il est égal à NULL (Pour utiliser cette valeur, il faut inclure le fichier stdio.h dans le programme).

```
int *p = NULL; /* Un pointeur nul */
```

V. Operateurs & et *

Le langage C met en jeu deux opérateurs utilisés lors de l'usage de pointeurs. Il s'agit des opérateurs & et *.

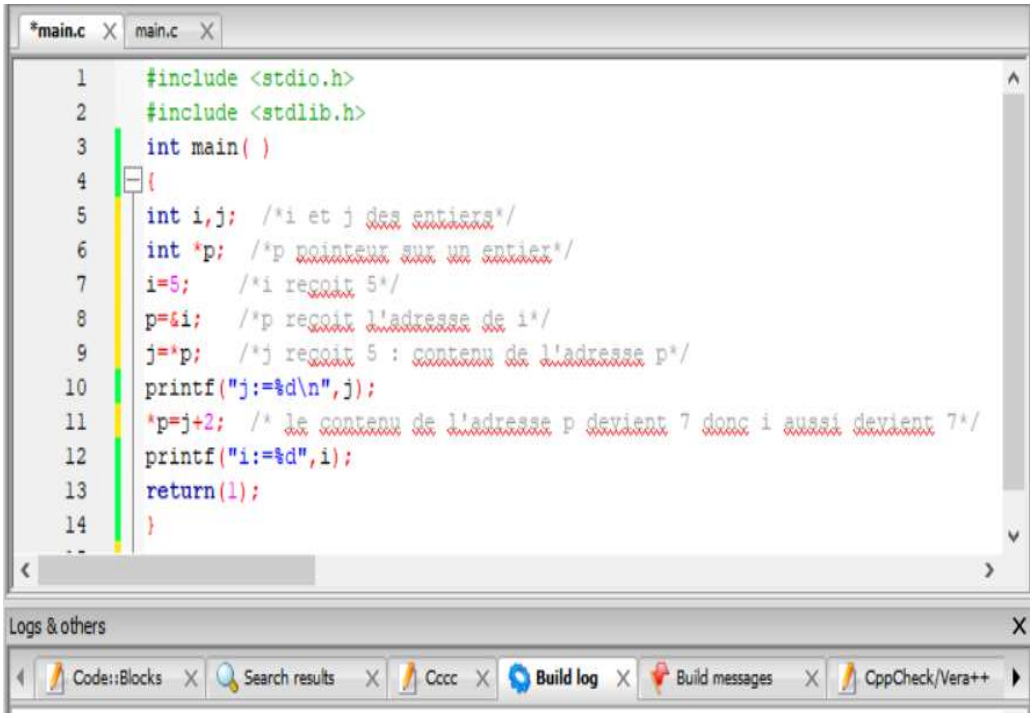
& variable signifie adresse de variable.

*pointeur signifie contenu de l'adresse référencée par pointeur.

* et & ont la même priorité que les opérateurs unaires (!, ++, --), dans une même expression, les opérateurs unaires *, &, !, ++, -- sont évalués de droite à gauche.

Si p pointe sur x (p= &x), alors *p peut être utilisé partout où l'on peut écrire x. Les pointeurs sont aussi des variables, et peuvent être utilisés comme telles. Soient p1 et p2, pointeurs de type entier, p1 = p2 copie le contenu de p2 dans p1, p1 pointe alors sur le même objet que p2.

Exemples :



VI. Operateurs ++ Et --

Un pointeur peut être déplacé d'une adresse à une autre au moyen des opérateurs ++ et --. L'unité d'incrémentation (ou de décrémentation) d'un pointeur est toujours la taille de la variable pointée.

Exemple :

```
#include <stdio.h>
main( )
{
int i; /*i entier, supposons qu'il se trouve à l'adresse 100*/
int *p; /*p pointeur sur un entier*/
p=&i; /*p reçoit 100 : adresse de i*/
p++; /*p s'incrémente de 4 (devient 104)*/
char c; /*c char, supposons qu'il se trouve à l'adresse 200*/
char *q; /*q pointeur sur char*/
q=&c; /*q reçoit 200 : adresse de c*/
q++; /*q s'incrémente de 1 (devient 201)*/
}
```

VII. Allocations dynamique de la mémoire

On doit souvent travailler avec des données dont on ne peut pas prévoir le nombre et la grandeur lors de la programmation. Ce serait un gaspillage de réserver toujours l'espace prévisible. Il existe un moyen de gérer la mémoire lors de l'exécution du programme, c'est *l'allocation dynamique de la mémoire*.

Pour éviter des erreurs fréquemment rencontrées lors de l'utilisation des pointeurs, le programmeur doit, immédiatement après la déclaration d'un pointeur, l'initialiser à l'adresse d'une variable donnée (par exemple, p=&i) ou lui allouer de la mémoire et l'initialiser au choix. Après la déclaration d'un pointeur, la zone mémoire réservée à la variable pointée se trouve dans un espace dynamique (heap). Elle peut être allouée au moyen de la fonction *malloc()* et on utilise aussi la fonctionnalité *sizeof()*.

VII.1. La fonction `sizeof (stdlib.h)`

Selon le type de variable que vous demandez de créer (char, int, double, float…), vous avez besoin de plus ou moins de mémoire. En effet, pour stocker un nombre compris entre -128 et 127 (un char), on n'a besoin que d'un octet en mémoire. C'est tout petit.

En revanche, un int occupe généralement 4 octets en mémoire. Quant au double, il occupe 8 octets. Notre objectif ici est de vérifier quelle taille occupe chacun des types sur un ordinateur. Il y a un moyen très facile pour savoir cela : utiliser l'opérateur *sizeof()*. Contrairement aux apparences, ce n'est pas une fonction mais une fonctionnalité de base du langage C. Vous devez juste indiquer entre parenthèses le type que vous voulez analyser.

Pour connaître la taille d'une int, on devra donc écrire :

```
sizeof(int)
```

À la compilation, cela sera remplacé par un nombre : le nombre d'octets que prend int en mémoire. Chez moi, *sizeof(int)* vaut 4, ce qui signifie que int occupe 4 octets. Testez pour voir, en affichant la valeur à l'aide d'un *printf* par exemple :

```
printf("char : %d octets\n", sizeof(char));
printf("int : %d octets\n", sizeof(int));
printf("long : %d octets\n", sizeof(long));
printf("double : %d octets\n", sizeof(double));
```

Cela affiche :

```
char : 1 octets
int : 4 octets
long : 4 octets
```

```
double : 8 octets
```

Vous remarquerez que long et int occupent la même place en mémoire. Créer un long revient donc ici exactement à créer un int, cela prend 4 octets dans la mémoire.

Exemple 1 :

```
sizeof(int)          → 4 octet
sizeof(double)       → 8 octet
sizeof(4.5)          → 8 octet
int A[10]
sizeof(A)            → 40 octet
char B[5][10]
sizeof(B)            → 50 octet
int D[5][10]
sizeof(B)            → 200 octet
```

Peut-on afficher la taille d'un type personnalisé qu'on a créé (une structure) ?

Oui ! sizeof marche aussi sur les structures !

Exemple :

```c
#include <stdio.h>
#include <stdlib.h>
int main( )
{
typedef struct elementes elementes;
struct elementes
{
    int x;
    int y;
};
printf("elementes : %d octets\n", sizeof(elementes));

    return 0;
}
```

Cela affiche :

```
elementes : 8 octets
```

VII.2. La fonction `malloc(N)`

Entrons maintenant dans le vif du sujet. Je vous rappelle notre objectif : apprendre à demander de la mémoire manuellement. On va avoir besoin d'inclure la bibliothèque <stdlib.h>. Cette bibliothèque contient deux fonctions dont nous allons avoir besoin :

« malloc (« Memory ALLOCation », c'est-à-dire « Allocation de mémoire ») : demande au système d'exploitation la permission d'utiliser de la mémoire ; »

Quand vous faites une allocation manuelle de mémoire, vous devez toujours suivre ces trois étapes :

1. appeler *malloc* pour demander de la mémoire ;
2. vérifier la valeur retournée par *malloc* pour savoir si le système d'exploitation a bien réussi à allouer la mémoire ;
3. une fois qu'on a fini d'utiliser la mémoire, on doit la libérer avec *free*. Si on ne le fait pas, on s'expose à des fuites de mémoire, c'est-à-dire que votre programme risque au final de prendre beaucoup de mémoire alors qu'il n'a en réalité plus besoin de tout cet espace.

Le prototype de la fonction *malloc* est assez simple, vous allez voir :

```
void* malloc(size_t nombreOctetsNecessaires);
```

La fonction prend un paramètre : le nombre d'octets à réserver. Ainsi, il suffira d'écrire *sizeof(int)* dans ce paramètre pour réserver suffisamment d'espace pour stocker un int.!.

```
malloc(sizeof(int));
```

Si je veux créer manuellement une variable de type int en mémoire, je devrais indiquer à *malloc* que j'ai besoin de *sizeof(int)* octets en mémoire. Je récupère le résultat du *malloc* dans un pointeur sur int.

Exemple :

```c
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int* ptr = NULL;

    ptr = malloc(sizeof(int));
    if (ptr == NULL)
    {
        exit(0);
    }
    printf("Quel age avez-vous ? ");
    scanf("%d", ptr);
    printf("Vous avez %d ans\n", *ptr);

    free(ptr);
    return 0;
}
```

Si je veux créer manuellement un tableau dynamique de taille x et de type int, je devrais indiquer à *malloc* que j'ai besoin de *sizeof(x*int)* octets en mémoire.

Exemple :

```c
#include <stdio.h>
#include <stdlib.h>
int main( )
{
int x,i=0;
printf("entrer la taille du tableau dynamique:");
scanf("%d",&x);
int *t=malloc(x*sizeof(int));
for(i=0;i<x;i++){
    printf("entrer le Nombre %d:",i+1);
    scanf("%d",&t[i]);
}
for(i=0;i<x;i++){
    printf("\n%d",t[i]);
}
free(t);
   return 0;
}
```

VII.3. La fonction `free(N)`

Tout comme on utilisait la fonction *fclose* pour fermer un fichier dont on n'avait plus besoin, on va utiliser la fonction *free* pour libérer la mémoire dont on ne se sert plus.

Donc, *free* (« Libérer ») permet d'indiquer au système d'exploitation que l'on n'a plus besoin de la mémoire qu'on avait demandée. La place en mémoire est libérée, un autre programme peut maintenant s'en servir au besoin.

```
void free(void* pointeur);
```

La fonction *free* a juste besoin de l'adresse mémoire à libérer.

Attention : comme ptr est un pointeur, on ne l'utilise pas de la même manière qu'une vraie variable. Pour obtenir la valeur de la variable, il faut placer une étoile devant : *ptr (regardez le printf). Tandis que pour indiquer l'adresse, on a juste besoin d'écrire le nom du pointeur ptr.

Partie A

Chapitre 02 : Les listes chaînées

I. Introduction

Toutefois, les tableaux se révèlent parfois assez limités. Par exemple, si vous créez un tableau de 10 cases et que vous vous rendez compte plus tard dans votre programme que vous avez besoin de plus d'espace, il sera impossible d'agrandir ce tableau. De même, il n'est pas possible d'insérer une case au milieu du tableau.

Les listes chaînées représentent une façon d'organiser les données en mémoire de manière beaucoup plus flexible.

II. Définition

Une liste chaînée est une structure linéaire qui n'a pas de dimension fixée à sa création. Ses éléments de même type sont éparpillés dans la mémoire et reliés entre eux par des pointeurs. Sa dimension peut être modifiée selon la place disponible en mémoire. La liste est accessible uniquement par sa tête de liste c'est-à-dire son premier élément. Chaque élément contient :

- Des informations relatives au fonctionnement de l'application, par exemple des noms et prénoms de personnes avec adresses et numéros de téléphone pour un carnet d'adresse.
- L'adresse de l'élément suivant ou une marque de fin s'il n'y a pas de suivant. C'est ce lien via l'adresse de l'élément suivant contenue dans l'élément précédent qui fait la "chaine" et permet de retrouver chaque élément de la liste.

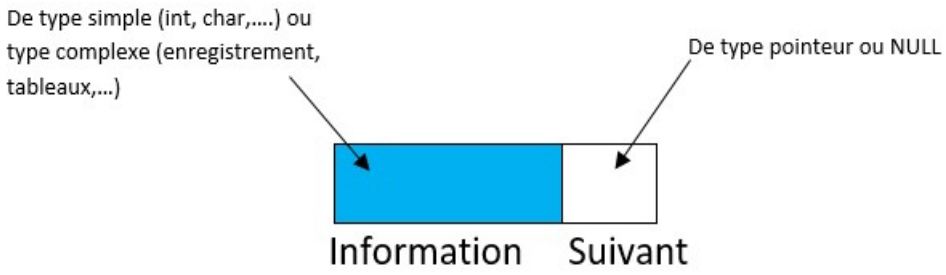


Figure 01 : *élément d'un liste chainée.*

Pour les listes chaînées la séquence est mise en œuvre par le pointeur porté par chaque élément qui indique l'emplacement de l'élément suivant. Le dernier élément de la liste ne pointe sur rien (NULL). On accède à un élément de la liste en parcourant les éléments grâce à leurs pointeurs. Les listes chaînées font appel à la notion de variable dynamique.

Une variable dynamique *e*st déclarée au début de l'exécution d'un programme, elle y est créée, c'est-à-dire qu'on lui alloue un espace à occuper à une adresse de la mémoire, elle peut y être détruite, c'est-à-dire que l'espace mémoire qu'elle occupait est libéré, l'accès à la valeur se fait à l'aide d'un pointeur.

III. Déclaration d'un maillon d'une liste chainée

Pour définir un élément de la liste le type *struct* sera utilisé. L'élément de la liste contiendra un champ ''Info'' et un pointeur suivant. Le pointeur suivant doit être du même type que l'élément, sinon il ne pourra pas pointer vers l'élément. Le pointeur "suiv" permettra l'accès vers le prochain élément.

```
typedef struct Element{
      int Info ;
      struct Element *suiv ;
}Liste ;
```

Nous avons créé ici un élément d'une liste chaînée, correspondant à la Figure 01 que nous avons vue plus tôt. Que contient cette structure ?

- Une Info, ici un nombre de type int : on pourrait remplacer cela par n'importe quelle autre donnée (un enregistrement, un double, un tableau…). Cela correspond à ce que vous voulez stocker, c'est à vous de l'adapter en fonction des besoins de votre programme.
- Un pointeur vers un élément du même type appelé suivant. C'est ce qui permet de lier les éléments les uns aux autres : chaque élément « sait » où se trouve l'élément suivant en mémoire. les cases ne sont pas côte à côte en mémoire. C'est la grosse différence par rapport aux tableaux. Cela offre davantage de souplesse car on peut plus facilement ajouter de nouvelles cases par la suite au besoin.

Grâce au pointeur chaque élément peut contenir l'adresse d'un élément et une liste chainée peut se représenter ainsi :

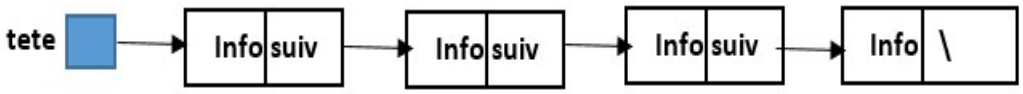


Figure 02 : *Liste simplement chainée.*

L'adresse de la liste proprement dite est donnée par le premier élément, celui à partir du quel tous les autres sont accessibles. Ce premier élément est un pointeur que nous déclarerons dans le main().

```
Liste*tete=NULL;
```

IV. Opérations de base sur les listes chaînées

La création d'une liste, le parcours d'une liste ainsi que l'ajout et la suppression d'un élément sont les principales opérations effectuées sur les listes.

IV.1 Parcours d'une liste

Les cases d'un tableau sont numérotées successivement. Le parcours dans un tableau revient alors à initialiser un indice de parcours à 1 (se placer au niveau de la première case) puis d'incrémenter cet indice à chaque fois pour passer d'une case à une autre.
Les éléments d'une liste ne sont pas numérotés mais reconnus par leurs adresses (l'adresse de chaque élément étant sauvegardée dans la tête de la liste si c'est le premier sinon dans la partie pointeur de l'élément qui le précède).

Le parcours d'une liste revient alors à initialiser un pointeur à la valeur de la tête de liste puis de lui affecter à chaque déplacement l'adresse se trouvant dans la partie pointeur de l'enregistrement jusqu'à ne plus avoir d'élément suivant :

```
void parcour(Liste *L){
if (L==NULL) printf("la liste est vide") ;
else
   while(L!=NULL){
      printf("%d",L->Info);
      L->suiv ;}
}
```

Remarque : La condition d'arrêt change lorsque l'on veut s'arrêter plus tôt que la fin de liste, si l'on veut se placer au niveau du dernier élément pour réaliser un quelconque traitement sur lui la condition serait alors :

```
void parcourfin(Liste *L){
  if (L==NULL) printf("la liste est vide") ;
  else
    {Liste *P=L;
     while(P->suiv!=NULL)
     P->suiv ;}
}
```

IV.2 Insertion d'un élément dans la liste

On peut ajouter (insérer) un élément en début de liste (il deviendra le nouveau premier élément), en fin de liste (il deviendra le dernier élément) ou encore en milieu de liste (entre deux éléments). Voici les étapes d'insertion et de sauvegarde des éléments dans une liste :

- Déclaration d'élément(s) à insérer.
- Allocation de la mémoire pour le nouvel élément.
- Remplir le contenu du champ de données.
- Mettre à jour les pointeurs vers le 1er et le dernier élément si nécessaire.

Pour ajouter un élément dans la liste il y a plusieurs situations :

- A. Insertion dans une liste vide.
- B. Insertion au début de la liste.

- C. Insertion à la fin de la liste.
- D. Insertion au milieu dans la liste.

IV.2.A. Ajout en début de liste

Nous devons créer une fonction capable d'insérer un nouvel élément en début de liste. Pour nous mettre en situation, imaginons un cas semblable à la Figure 03 suivante : la liste est composée de quatre éléments et on souhaite en ajouter un nouveau au début.

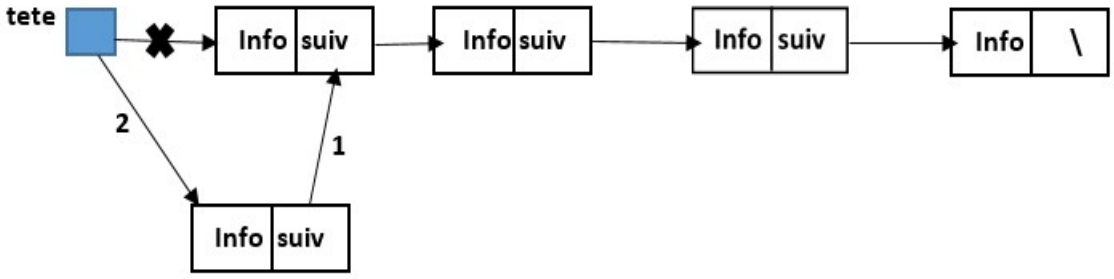


Figure 03 : *schéma d'ajout en début de liste*

Il va falloir adapter le pointeur `tete` de la liste ainsi que le pointeur `suiv` de notre nouvel élément pour « insérer » correctement celui-ci dans la liste. Je vous propose pour cela ce code source que nous analyserons juste après :

```
Liste* AjouterDebut(Liste *L, int nvNombre)
{
    /* Création du nouvel élément */
    Liste *nouveau = malloc(sizeof(Liste));
    if (nouveau == NULL)
    {
        exit(1);
    }
    nouveau->Info = nvNombre;
    /* Insertion de l'élément au début de la liste */
    nouveau->suiv = L;
    L= nouveau;
return L ;
}
```

La fonction `AjouterDebut()` prend en paramètre l'élément de contrôle `L`(qui contient l'adresse du premier élément) et le nombre à stocker dans le nouvel élément que l'on va créer (`nouveau`).

Dans un premier temps, on alloue l'espace nécessaire au stockage du nouvel élément et on y place le nouveau nombre `nvNombre`. Il reste alors une étape délicate : l'insertion du nouvel élément dans la liste chaînée.

Nous avons ici choisi pour simplifier d'insérer l'élément en début de liste. Pour mettre à jour correctement les pointeurs, nous devons procéder dans cet ordre précis :

1. Faire pointer notre nouvel élément vers son futur successeur, qui est l'actuel premier élément de la liste ;
2. Faire pointer le pointeur `L`(la tête de liste) vers notre nouvel élément.

On ne peut pas suivre ces étapes dans l'ordre inverse ! En effet, si vous faites d'abord pointer `L` vers notre nouvel élément, vous perdez l'adresse du premier élément de la liste ! Faites le test, vous comprendrez de suite pourquoi l'inverse est impossible. Cela aura pour effet d'insérer correctement notre nouvel élément dans la liste chaînée (Figure 03) !

IV.2.B. Ajouter à la fin

Pour ajouter à la fin d'abord créer un nouveau maillon, ensuite deux cas se présentent : la liste est vide ou pas. Si la liste est vide le premier prend la valeur du nouveau, Sinon se positionner sur le dernier élément de la liste et ajouter le nouveau à la suite (voir Figure 04) :

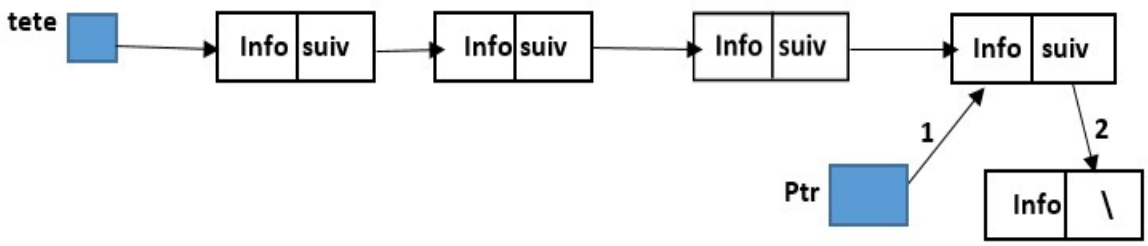


Figure 04 : schéma d'ajout à la fin

Je vous propose pour cela ce code source :

```
Liste *AjouterFin(Liste* L ,int nvNombre)
{    /* Création du nouvel élément */
    Liste *nouveau = malloc(sizeof(Liste));
     nouveau->Info=nvNombre ;
     nouveau->suiv=NULL ;
     if (L == NULL)
      L =nouveau ;
    else
{
    /* Insertion de l'élément à la fin de la liste */
   Liste *Ptr=L ;
while(Ptr->suiv !=NULL)
     Ptr=Ptr->suiv ;
Ptr->suiv=nouveau;}
return L ;
}
```

IV.2.C. Ajout en milieu de liste

On a deux cas possible :

Cas 1 : (avant un élément dont l'adresse est dans "A")

Dans ce cas il faut connaître l'adresse de l'élément A que l'on veut précéder du nouveau élément nouveau (nouveau sera placé donc avant A) :

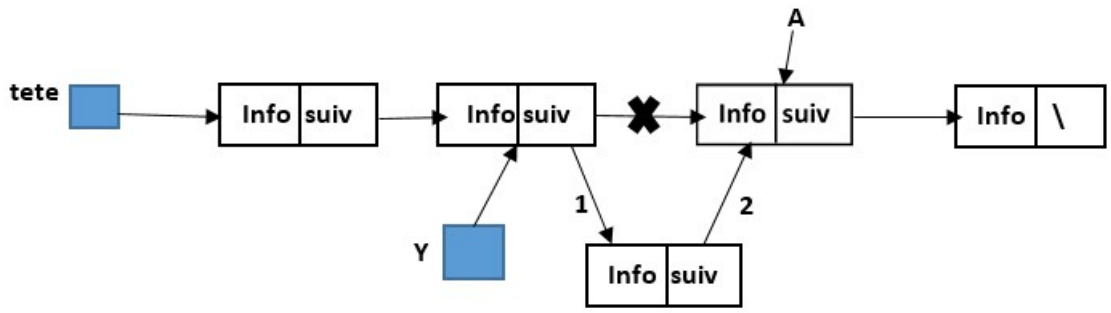


Figure 05 : *schéma d'ajout en milieu de liste (avant un élément dont l'adresse est dans "A")*

Je vous propose pour cela ce code source :

```
Liste* AjoutMilieuAV (Liste *L, Liste *A,int nvNombre)
{Liste * Y;
Liste *nouveau = malloc(sizeof(Liste));
nouveau->Info=nvNombre ;
Y=L ;
while (Y->suiv != A)
    Y = Y->suiv ;
nouveau->suiv =Y->suiv ; /* ou encore nouveau->Suiv = A ; */
Y->suiv = nouveau ;
return L ;
}
```

Remarque :

- Une insertion au milieu se fait entre deux éléments dans une liste avec au moins deux éléments tout autre liste (vide (tete = NULL) ou avec un seul élément (tete->suiv = NULL)) ne peut être utilisée pour une insertion au milieu.

Cas 2 : (après un élément dont l'adresse est dans "A")

Dans ce cas il faut connaître l'adresse de l'élément A que l'on veut faire suivre du nouveau élément X (X sera placé donc après A), on a pas besoin de parcours du liste.

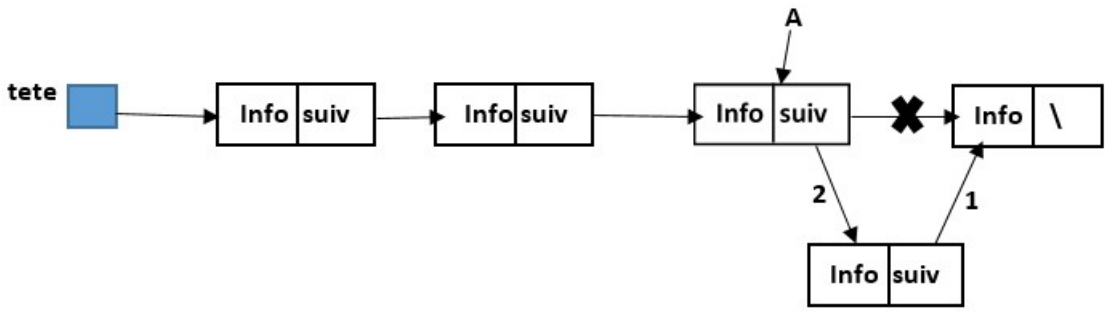


Figure 06 : *schéma d'ajout en milieu de liste (après un élément dont l'adresse est dans "A")*

Je vous propose pour cela ce code source :

```
liste* AjoutMilieuAP (Liste *L, Liste* A,int nvNombre)
{Liste * Y;
Liste *nouveau = malloc(sizeof(Liste));
nouveau->Info=nvNombre ;
nouveau->suiv=A->suiv ;
A->suiv=nouveau ;
return L ;}
```

IV.3. Supprimer un élément

De même que pour l'insertion, nous allons ici nous concentrer sur la suppression du premier élément de la liste, du dernier élément de la liste et un élément au milieu de la liste. La suppression ne pose pas de difficulté supplémentaire. Il faut cependant bien adapter les pointeurs de la liste dans le bon ordre pour ne « perdre » aucune information.

IV.3.A Suppression en début de liste

Avant de suppression un élément de la liste, la vérification de l'existence d'au moins un élément dans la liste (liste non vide) est nécessaire. Dans le cas où le test n'est pas réalisé avant l'appel du sous algorithme, le code devient le suivant :

```
Liste* SupprimeDebut(Liste *L)
{Liste* X ;
if (L==NULL)
printf("erreur ! pas d'élément à supprimer") ;
else
{X = L ;
L = L->suiv;
free(X) ;
return(L);
}}
```

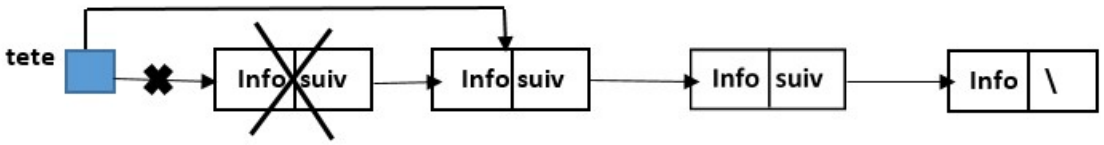


Figure 07 : *Schéma de Suppression un élément en début de liste*

IV.3.B. Suppression en fin de liste

La suppression à la fin suppose de se positionner sur l'avant dernier maillon (`AvPtr`) puis de supprimer le suivant et de mettre suivant à NULL. Il faut que la liste ne soit pas vide et traiter le cas où il n'y a qu'un seul maillon dans la liste.

```
Liste* SupprimeFin (Liste* L)
{Liste* Ptr ;
Ptr=L;
Liste* AvPtr =Ptr ;
while (Ptr->suiv !=NULL)
    {AvPtr=Ptr;
     Ptr=Ptr->suiv ;}
AvPtr->suiv = NULL ;
free(Ptr) ;return(L) ; }
```

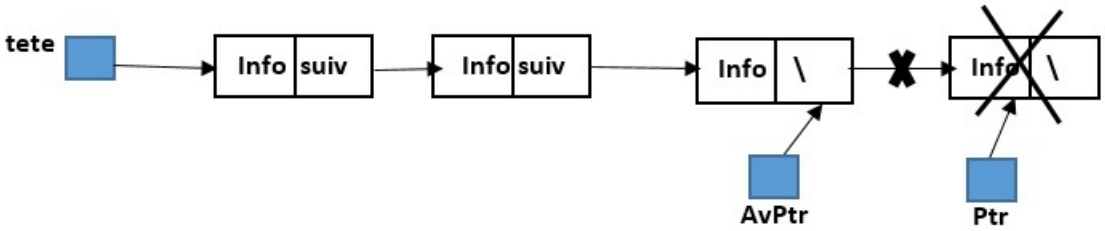


Figure 08 : *Schéma de Suppression un élément en fin de liste*

IV.3.C. *Suppression en milieu de liste*

Dans ce cas, il est nécessaire de connaître l'adresse de l'élément à supprimer (paramètre transmis dans le pointeur A).

Remarque :

Une suppression au milieu se fait dans une liste avec au moins trois éléments et l'adresse de l'élément à supprimer doit être valide et ne doit pas être ni celle du premier ni du dernier élément sinon cela serait respectivement une suppression du début ou en fin (tous ces cas d'erreurs doivent être traités avant d'appeler la procédure de suppression au milieu).

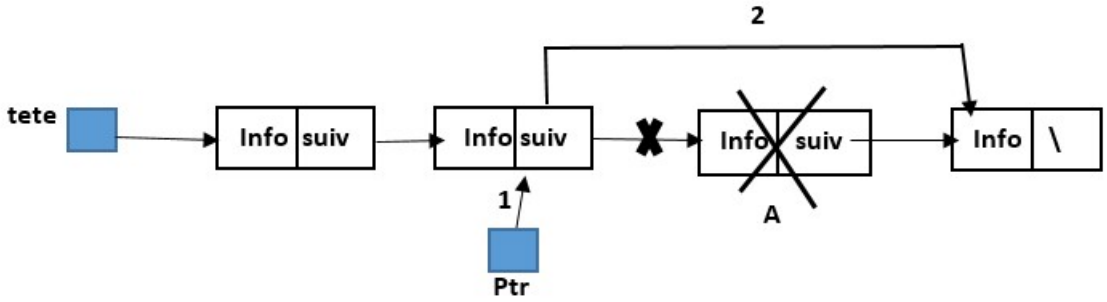


Figure 09 : *Schéma de Suppression un élément en milieu de liste*

```
Liste* SupprimeM (Liste* L,Liste* A)
{Liste* Ptr=L ;
while (Ptr->suiv != A )
Ptr =Ptr-> suiv ;
Ptr->suiv = A->suiv ;
free(A) ;
return(L);}
```

V. Cas particulaire des listes chaînées

Une liste est dite simplement chaînée si chacun de ses nœuds contient une partie donnée (Info) et l'adresse du nœud qui le suiv. Les opérations précédentes sont toutes valides sur ce type. Mais, il existe d'autre type des listes chainées, qu'on va expliquer dans les sections suivantes.

V.1 Liste doublement chaînée

Une liste est dite doublement chaînée si chaque nœud de cette liste contient une partie donnée, l'adresse du nœud qui la suit et l'adresse du nœud qui la précède (donc simplement chaînée avec en plus un chainage arrière). Il n'y a pas d'élément avant le premier nœud, la valeur du pointeur Prec = NULL.

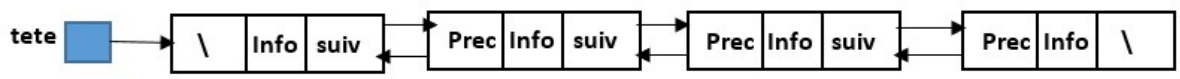


Figure 10 : *Liste doublement chaînée*

La déclaration des listes doublement chaînées nécessite donc un pointeur vers un enregistrement de trois cases :

```
typedef struct NoeudDBL{
    struct noeud *prec;
    int Info;
    struct noeud *suiv;
 }ListDBL;
```

L'intérêt de ce type de listes est le déplacement d'un élément vers son prédécesseur en un seul pas. Il est cependant nécessaire d'établir des changements dans les opérations de base à fin de garantir cette flexibilité.

V.2 Liste circulaire

Une liste est dite circulaire si le dernier nœud de cette liste nous remet au premier nœud de cette liste ; c'est à dire, que la case pointeur du dernier élément ne contient pas NULL mais l'adresse du premier élément (tete). L'absence du NULL dans ce type de liste (sauf le cas d'une liste vide) implique des conditions d'arrêt en rapport avec la tête de liste.

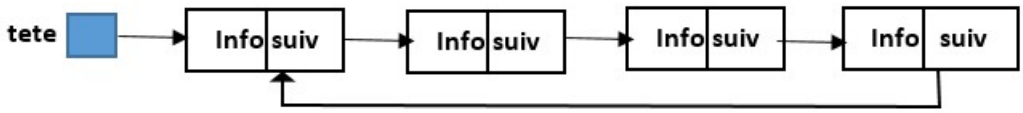


Figure 11 : *Liste circulaire*

V.3 Liste circulaire doublement chaînée

Cette variation regroupe les caractéristiques des listes doublement chaînées (présence du Prec) et des listes circulaires l'absence du NULL (car le Suiv du dernier nœud renvoi vers le premier élément et le Prec du premier nœud renvoi vers le dernier élément.)

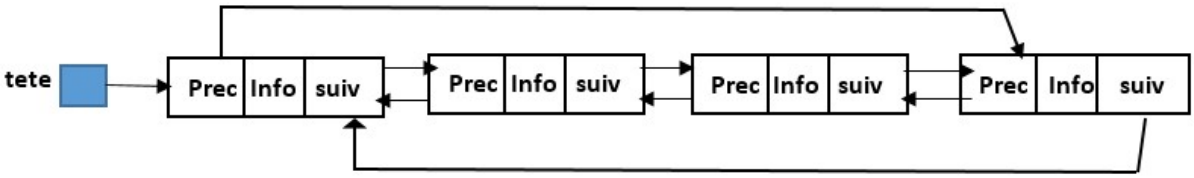


Figure 12 : Liste circulaire doublement chaînée

Partie A

Chapitre 03 : Les Piles

I. Principe

La pile est une structure très utilisée en informatique, surtout dans les langages de programmation (appel de procédures, calcul des expressions arithmétiques,...). Les piles ne sont pas de nouveaux types de données mais plutôt une manière de gérer un ensemble de données. La pile est une liste ordonnée sans fin d'éléments dans laquelle on ne peut introduire ou enlever un élément qu'à une extrémité appelée tête de pile ou sommet de pile. Noter que dans une pile, le dernier élément inséré sera le premier à être supprimé, retiré : on parle de liste LIFO 'Last In First Out' (En anglais : pile = stack). Elles sont très souvent utiles et servent, entre autres, à mémoriser des évènements en attente de traitement.

Il n'y a pas de structures spécifiques prévues dans les langages de programmation pour les piles. Il faut donc les créer de toute pièce sachant que la représentation en mémoire de ces structures de données peut être, selon le besoin, statique (utilisation des tableaux) ou dynamique (utilisation des listes).

Lorsqu'une structure de données est manipulée selon la stratégie Pile, alors les deux opérations d'ajout et de récupération se font uniquement en tête de la pile, appelé aussi le sommet de la pile. Autrement dit, le seul élément accessible est le premier élément.

Exemple une pile de livres, une pile d'assiettes, une pile à un jeu de cartes si l'on se refuse le droit de manipuler plus d'un de leurs éléments à la fois.

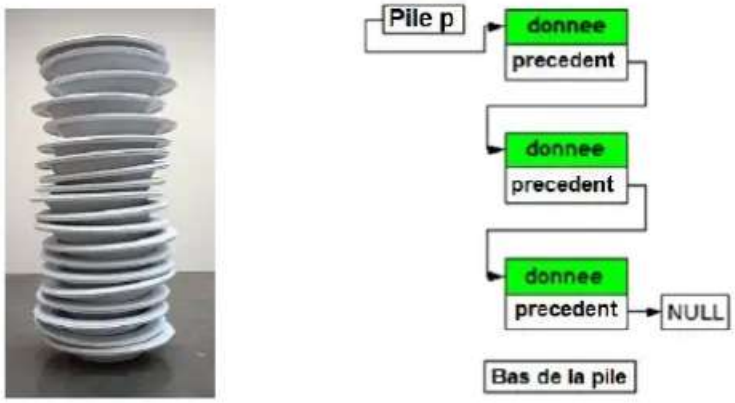


Figure 13 : *pile d'assiette*

II. Opérations sur les piles

En informatique une pile sert essentiellement à stocker des données qui ne peuvent pas être traitées immédiatement, car le programme à une tâche plus urgente ou préalable à accomplir auparavant. En particulier les appels et retours de fonctions sont gérés grâce à une pile appelée pile d'exécution. Les opérations habituelles sur les piles sont :

- *Init-Pile(P)* : Initialisation de la pile (généralement à vide).
- *Pile-Vide(P), Pile-Pleine :* Vérification du contenu de la pile (pile pleine ou vide).
- *Depiler (P) :* Dépiler un objet de P, si elle n'est pas vide, consiste à supprimer de P l'objet placé au sommet (dans la pile d'assiettes seule peut être retirée celle qui se trouve au sommet). L'objet dépilé est retourné comme résultat du traitement(en anglais pop(P)).
- *Empiler (P,x) :* Empiler un objet sur une pile P, si elle n'est pas saturée, consiste à insérer cet objet au sommet de P (dans la pile d'assiettes une nouvelle assiette ne peut être ajoutée qu'au-dessus de celle qui se trouve au sommet)(en anglais push(P,x)).

III. Modélisation et Implémentation des piles

En termes de programmation, une pile est un enregistrement avec :

- Une structure de données pour enregistrées les valeurs (elle peut être statique ou dynamique)
- Une variable sommet qui indique le sommet de la pile.

Les piles peuvent être représentés en deux manières : par des tableaux ou par des listes chainées.

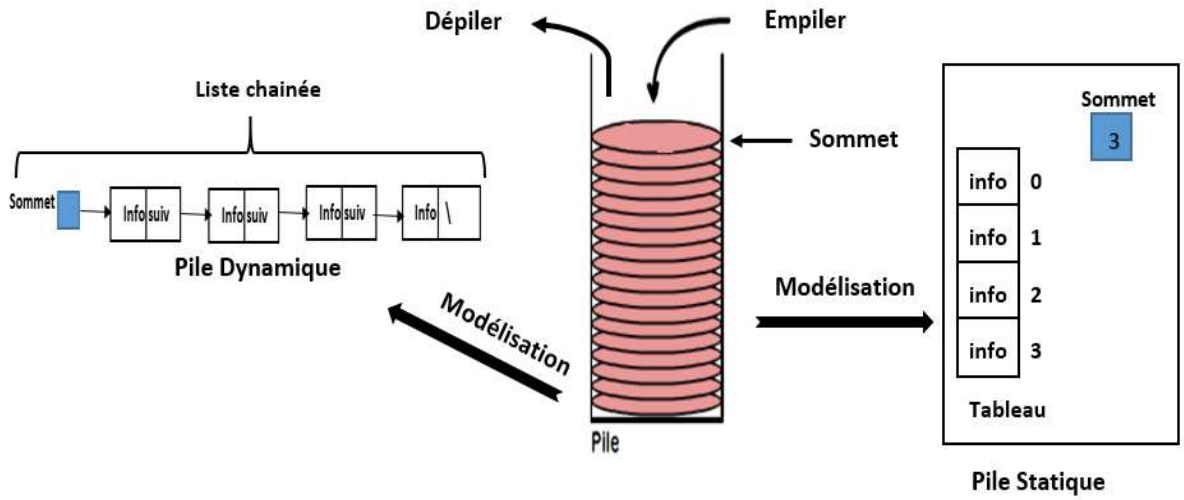


Figure 14 : *Modélisation d'une pile*

III. 1 Implémentation par des tableaux (Statique)

L'implémentation statique des piles utilise les tableaux. Dans ce cas, la capacité de la pile est limitée par la taille du tableau. L'ajout à la pile se fait dans le sens croissant des indices, tandis que le retrait se fait dans le sens inverse.

Dans ce cours nous allons découvrir comment écrire un programme d'implémentation d'une pile en utilisant un tableau. Il implique diverses opérations telles que push, pop, etc.

III.1.a Déclaration

Dans cette stratégie, la pile est un enregistrement qui contient deux champs :

- Les éléments de la pile sont stockés dans un tableau à hauteur maximale prévisible.
- Un champ sommet qui contient l'indice du dernier élément ajout a la pile.

```
#define max 5
typedef struct
{
  int tab[max] ;
  int sommet ;
}TPile;
TPile P ;
```

Remarque :

Les données enregistrées dans la pile peuvent être des entiers, des réels, des caractères, des chaînes de caractères, des booléens, des tableaux.

III.1.b Initialisation d'une pile

Il s'agit d'initialiser le sommet à "-1" pour attester l'absence de valeurs dans la pile.

```
void Init_Pile ()
{P.sommet = -1 ;
}
```

III.1.c Vérification de pile vide

```
void Pile_vide ( )
{
if( P.sommet = -1 )
printf("la pile est vide") ;
else
printf("la pile n'est pas vide") ;}
```

III.1.d Vérification de pile pleine

```
void Pile_pleine ()
{
if (P.sommet = max-1)
  printf("la pile est pleine") ;
else
  printf("la pile n'est pas pleine") ;}
```

III.1.e Ajout d'une nouvelle valeur à une pile

```
void push(int x) {
if (P.sommet == max-1)
  {printf("erreur :pile pliene !") ;
   exit(1);}
else{
   P.sommet=P.sommet+1;
   P.tab[P.sommet]=x;
}}
```

III.1.f Suppression d'une valeur de la pile

```
int pop() {
    int x;
if (P.sommet== -1)
    {
        printf("erreur :pile vide !") ;
return 1 ;
    }
else{x = P.tab[P.sommet];
P.sommet=P.sommet-1;
return x;}

}
```

III. 2 Implémentation par des Listes chainées(Dynamique)

Une pile dynamique est une liste à la quel on attache un pointeur sommet. C'est un enregistrement à une seule case : pointeur qui pointe la dernière valeur traitée dans la liste (sommet).

L'implémentation dynamique utilise les listes linéaires chainées. Dans ce cas, la pile peut être vide, mais ne peut être jamais pleine, sauf bien sûr en cas d'insuffisance de l'espace mémoire. L'empilement et le dépilement dans les piles dynamiques se font à la tête de la Pile.

III.2.a Déclaration

La pile n'est rien qu'un pointeur vers le premier élément de la pile. Les donnes de la pile peuvent être de n'importe quel type : entier, réel, caractère, structure, tableau,…

```
typedef struct Element{
     int Info;
     struct Element *suiv;
   } PILED;
PILED *P ;
```

III.2.b Initialisation d'une pile

```
PILED *sommet=NULL ;
```

III.2.c Vérification de pile vide

Il n'y a pas de pile pleine pour les piles dynamique. La seule possibilité de ne pas pouvoir ajouter un élément c'est d'avoir une mémoire pleine, cas que l'on ne prend pas en considération

ici.

```
void  PILEvide(PILED *P)
{if P==NULL)
  printf("la pile est vide") ;
else
printf("la pile n'est pas vide") ;}
```

III.2.e Ajout d'une nouvelle valeur à une pile

L'ajout d'une valeur à une pile dynamique revient à une insertion en début de liste si l'on considère que le sommet est la tête de la liste.

```
PILED *empiler (int x, PILED *P)
{ PILED *P1 ;
  P=(PILED*)malloc(sizeof(PILED)) ;
  if(P1==NULL)
    {printf("mémoire insuffisant") ;
     exit(1) ;}
  else
   {P1->Info=x ;
    P1->suiv=P ;
    P=P1 ;}
return(P) ;
}
```

III.2.f Suppression d'une valeur de la pile

La suppression d'une valeur dans une pile dynamique revient à effectuer une suppression physique d'un nœud (le premier de liste) et à récupérer dans un paramètre, passé par variable, la donnée enregistrée.

```
int Depiler(PILED *P)
{int X ;PILED *P1 ;
if(P==NULL)
   printf("erreur ! pas d'élément à dépiler") ;
else
  {X = P->Info ;
   P1=P ;
   P =P->suiv;
   free (P1) ;
return(X);}}
```

Partie A

Chapitre 04 : Les files et les file d'attente

I. Principe

Dans certain problème on cherche à stocker des éléments de même type toute en gardant leurs ordre d'arrivé, exemple :

- L'arrivé des patients dans un service d'hôpital.
- La queue des clients devant le guichet d'une banque.
- Les commandes reçues dans un restaurant, la première commande doit être traité en premier avant de passer voir les autres commandes.

Tous ces problèmes ont en commun le respect de l'ordre d'arrivée des éléments, cette stratégie peut être nommé Premier arrivé Premier servi (en anglais First In First Out), une file, permet de réaliser cette stratège ; les premiers éléments ajoutés à la file sont les premiers à être récupérés.

Quand on vous dit file penser directement à une file d'attente où chacun à son tour qu'il doit respecter. Une file est un ensemble de valeurs qui a un début (tète) et une fin (Queue). Enfiler un objet sur une file F consiste à insérer cet objet à la fin de la file F (dans la file d'attente un nouvel arrivant se met à la queue c'est à dire, après la personne arrivée juste avant lui) ; Défiler un objet de F consiste à supprimer de F l'objet placé en début de file (dans la file d'attente seule peut être servie la personne qui se trouve en début de file). L'objet défilé est retourné comme résultat du traitement.

II. Modélisation

Après avoir étudié les piles du point de vue statique et dynamique (voir chapitre précédent), ferons de même avec les files d'attente. Une file d'attente statique est représentée par un tableau circulaire et ses deux indices de parcours tète et queue, un pour indiquer la tête de la file et l'autre pour indiquer la queue de la file.

Quand la file d'attente est dynamique, elle est représentée sous forme d'une liste chaînée avec sa politique de gestion particulière(FIFO).

II.1 Modélisation par tableau circulaire

La première façon de modéliser une file d'attente consiste à utiliser un tableau. L'ajout d'un élément se fera à la suite du dernier élément du tableau. Le retrait d'un élément de la file se fera en enlevant le premier élément du tableau. Il faudra donc deux indices pour ce tableau, le premier qui indique le premier élément de la file et le deuxième qui indique la fin de la file. La figure suivante représente une file d'attente par cette modélisation. Elle reprend la même pile que pour l'exemple de modélisation par tableau.

Voici la structure de données correspondant à une file d'attente représentée par un tableau circulaire.

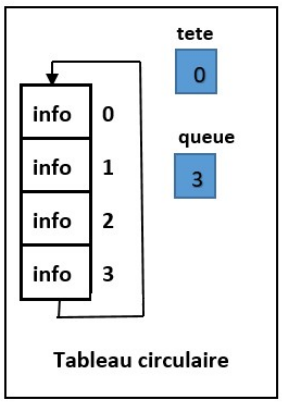


Figure 15 : *Modélisation d'une file d'attente par tableau circulaire*

On peut noter que progressivement, au cours des opérations d'ajout et de retrait, le tableau se déplace sur le bas dans son espace mémoire. A un moment, il va en atteindre le bout de l'espace. Dans ce cas, le tableau continuera au début de l'espace mémoire comme si la première et la dernière case étaient adjacentes, d'où le terme "tableau circulaire". Ce

mécanisme fonctionner a tant que le tableau n'est effectivement pas plein. Pour illustrer, prenons l'exemple suivant.

Exemple :

On applique quatre opérations d'ajout de l'élément de tête. On ajoute également en queue les valeurs suivants et dans cet ordre: 15, 20, 30, 40.

La file d'attente contient alors, et dans cet ordre : 15, 20, 30 et 40 les éléments de la file. .

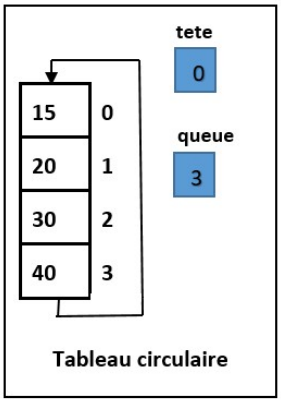


Figure 16 : *Etat d'une file d'attente après enfiler 15, 20, 30 et 40*

Si on défile un élément de la file d'attente, on obtient le résultat suivant.

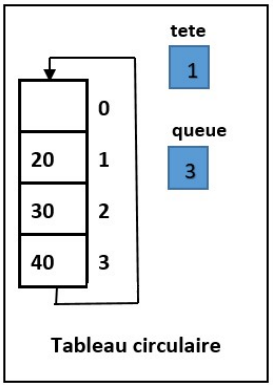


Figure 17 : *Etat d'une file d'attente après défiler 15*

Si enfile un autre élément x=50, la file d'attente est devient comme présenter dans le schéma suivante.

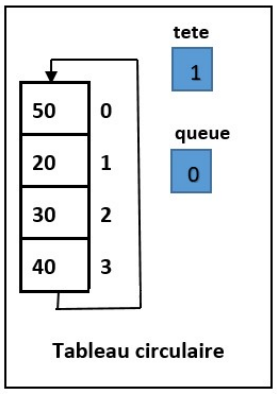


Figure 18 : *Etat d'une file d'attente après enfiler 50*

On s'aperçoit que, ayant atteint la fin du tableau, la file redémarre à l'indice 0. Pour réaliser cela, une structure sera utilisée (ce n'est pas obligatoire, des variables peuvent être utilisées). Voici sa composition :

```
typedef struct {
      int tete;
      int queue;
      int Tab [Maxfile];
        }FileA;
  FileA F1,F2;
```

`Maxfile` est une constante représentant la taille du tableau alloué. *`tete`* est l'indice de tableau qui pointe sur la tête de la file d'attente. *`Queue`* est l'indice de tableau qui pointe sur la case de dernier élément enfiler dans le tableau.

II.2 Modélisation par liste chaînée

La deuxième manière de modéliser une file d'attente consiste à utiliser une liste chaînée en n'utilisant que les opérations *`EnQueue`* et *`Dequeue`*. Dans ce cas, on s'aperçoit que le premier élément entré est toujours le premier élément sorti. La figure suivante représente une file d'attente par cette modélisation.

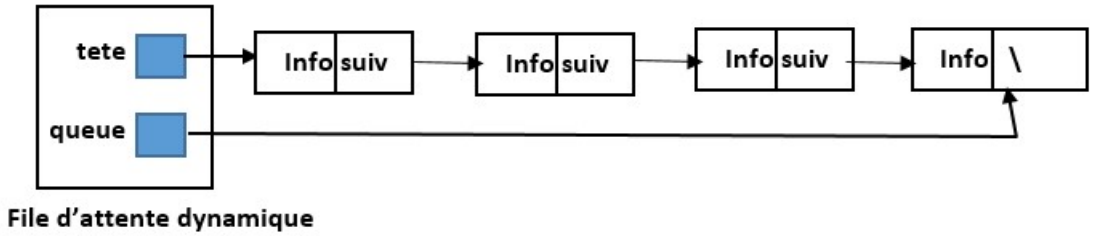


Figure 19 : *Modélisation d'une file d'attente par une liste chainée*

Pour cette modélisation, la structure d'une file d'attente est celle d'une liste chaînée.

```
typedef struct element {
     int info;
     struct element *suiv;
   } pointeur;
typedef struct fileA{
         pointeur* tete ;
         pointeur *queue ;
}FileAD ;
```

III. Opérations sur les files d'attente

Comme pour les piles, la manipulation d'une file d'attente revient à l'appel de fonctions et procédures dites de bases définies une seule fois et utilisées autant de fois qu'il est nécessaire. Ces sous-algorithmes sont :

- *Init_File :* permet d'initialiser une file à vide lors de sa création ;

- *File_vide :* pour vérifier si une file est vide ou non et savoir alors s'il reste des valeurs à traiter ou non ;

- *File_pleine :* pour vérifier s'il est possible de rajouter ou non un nouveau élément (utilisée dans le seul cas des files statiques) ;

- *Enqueue :* permet d'ajouter une nouvelle valeur (envoyé en paramètre par l'appelant) à la file (après le dernier élément de la file qui se trouve au niveau de sa queue et dans le cas d'une file non pleine (file statique)) ;

- *Dequeue :* permet de supprimer une valeur (se trouvant au début de la file) et de la renvoyer en paramètre. Cette opération n'est possible que si la file n'est pas vide.

IV. Implémentation de file d'attente

En termes de programmation, une file d'attente est un enregistrement avec :

- Une structure de données pour enregistrées les valeurs (elle peut être statique ou dynamique).

-Un indice entier "tete" qui pointe le premier élément inséré dans la file et un deuxième indice entier "Queue" qui pointe la dernière valeur ajoutée.

Les file d'attente peuvent être représentés en deux manières : par des tableaux ou par des listes chainées.

IV.1 Implémentation d'une file d'attente F statique

Dans cette section on présente un aperçu sur les principales opérations sur une file. Pour développer ces module, on a choisi de manipuler un tableau circulaire selon la stratégie file.

```
#define Maxfile 10  /* on suppose qu'il y ait 10 éléments*/
typedef struct {
      int tete;
      int queue;
      int Tab[Maxfile];
        }FileA;
  FileA F1 ;
```

On présente dans la suite les opérations de base sur les files d'attente statique :

IV.1.a Fonction d'initialisation de la file d'attente

```
FileA InitialiserFile(FileA F)
   F.tete=0;
   F.queue =-1;
   return(F);
  }
```

IV.1.b Fonction pour vérifier si la file d'attente est vide

```
int FileVide( FileA F)
{  if ((F.tete ==0) && (F.queue ==-1))
     return 1;
return 0;}
```

IV.1.c Fonction pour vérifier si la file d'attente est pleine

```
 int FilePleine(FileA F)
 { if (((F.tete==0)&&(F.queue==Maxfile-1))||
       ((F.tete == F.queue+1) && (F.queue!=-1)))
```

```
   return 1;
return 0;
}
```

IV.1.d Fonction pour enfiler une valeur à la file d'attente (Enqueue)

```
FileA EnqueueFile(int valeur,FileA F)
{
if (FilePleine(F) ==1)
    printf("Erreur: file d'attente pleine!\n");
else
{
   F.queue=(F.queue+1) % Maxfile;
   F.Tab[F.queue] = valeur;
} return(F);}
```

IV.1.e Fonction pour défiler une valeur de la file d'attente (Dequeue)

```
int dequeueFile(FileA F)
{
 int valeur;
 if (FileVide(F) ==1)
     printf("Erreur : File d'attente vide!\n");
else
{
   valeur = F.Tab[F.tete];
   if (F.tete == F.queue)
       /* cela veut dire que la file contient un element*/
{
    F.tete = 0; /* donc elle devient vide*/
    F.queue = -1;
}
else /* La file contient plus d'un élément*/
F.tete=(F.tete+1)%Maxfile;
return (valeur);}}
```

IV.2 Implémentation d'une file d'attente F dynamique

Sans tarder, on va étudier les files d'attente dynamiques sous forme de listes chaînées.

```
typedef struct element {
     int info;
     struct element *suiv;
   } pointeur;
typedef struct fileA{
         pointeur* tete ;
         pointeur *queue ;
}FileAD ;
```

Remarque :

On remarque que c'est pratiquement la même implémentation qu'une pile et c'est leur politique de gestion qui fait la différence.

IV.2.a Fonction d'initialisation de la file d'attente

```
void InitialiserFile(FileAD F)
{
   F.tete=NULL;
   F.queue=NULL;
    }
```

IV.2.b Fonction pour vérifier si la file d'attente est vide

```
int FileVide( FileAD F)
{  if (F.queue ==NULL)
     return 1;
return 0;}
```

IV.2.c Fonction d'enfiler une valeur à la file d'attente (en queue de la file)

```
void EnqueueD(int valeur,FileAD *F)
{ pointeur *ptr=(pointeur*)malloc(sizeof(pointeur));
ptr->info=valeur ;
ptr->suiv=NULL ;
 if (FileVide(F)==1)
    F.tete=ptr;
else
    F.queue->suiv=ptr;
   F.queue=ptr;
```

```
    return(F);
}
```

IV.2.d Fonction pour défiler une valeur de la file d'attente (En tête de file)

```
int DequeueD(FileAD F)
{ pointeur *ptr;
int x;
    if (FileVide(F))
        printf("Impossible la file est vide");
    else
    {
        ptr=F.tete;
        x=F.tete->info;
        F.tete=F.tete->suiv;
        if(F.tete==NULL)
            F.queue=NULL;
        free(ptr);
        return(x);
    }
}
```

V. Exemple d'utilisation

En informatique une file sert essentiellement à stocker des données qui doivent être traitées selon leur ordre d'arrivée. L'exemple le plus connu est celui de l'impression de documents reçus par une imprimante qui imprime le premier document arrivé et termine par le dernier. Ce qui fait que les objets quittent la pile dans l'ordre de leur ordre d'arrivée. Pour cette raison, une file est aussi appelée structure FIFO (First In, First Out) ou (premier arrivé, premier sorti).

Partie A

Chapitre 05 : Les arbres

I.Définition

En algorithmique, un arbre est une structure de donnée fondamentales trés utilisés dans tous les domaines parce qu'ils sont bien adaptés à la représentation naturelle d'informations homogènes organisées et d'une grande commodité et rapidité de manipulation, elle peut être représenté comme un tableau ou un pointeur (statique, dynamique), d'une manière plus simple en définis un arbre comme étant un ensemble des nœuds. Comme schématisé dans la figure suivante :

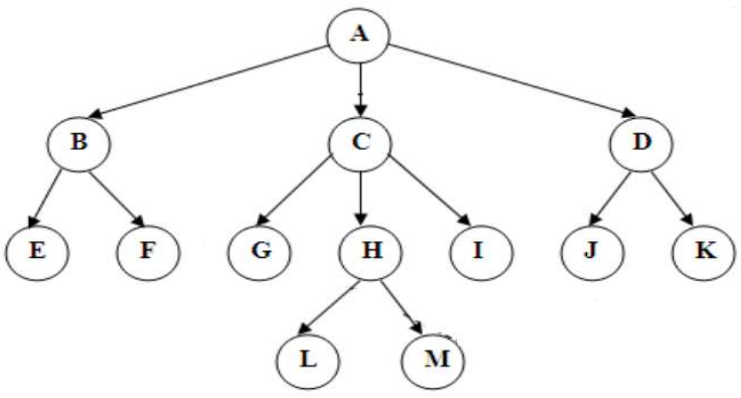


Figure 20 : *Exemple d'arbre*

II.Terminologie

Dans la suite on va définir les notions de base dans les arbres.

- ***Racine :*** est le nœud qui n'a pas de prédécesseur (parent) et possède zéro ou plusieurs fils. La racine constitue la caractéristique d'un arbre.
- ***Feuille :*** est un nœud qui n'a pas de successeur (fils). Une feuille est aussi appelée un nœud externe.
- ***Nœud interne :*** est tout nœud qui admet au moins un successeur (fils).

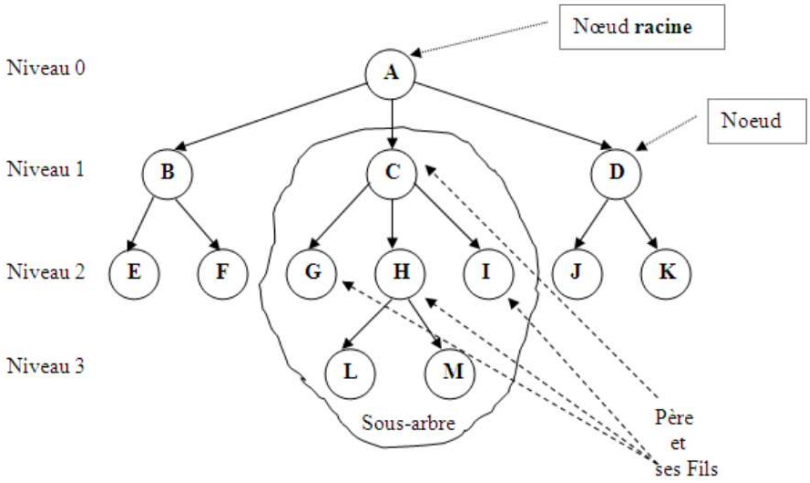


Figure 21 : *Terminologie d'un arbre*

- ***Père :*** est un nœud qui admet au moins un successeur (fils). Dans la Figure 21, D est le père des nœuds J et K.
- ***Fils d'un nœud :*** sont ses successeurs. Dans la Figure 21, G,H et I sont les fils du nœud C.
- ***Frères :*** sont les successeurs ou les fils issus d'un même nœud (parent direct). Dans la Figure 21, G, H et I sont des frères.
- ***Sous arbre :*** est une portion de l'arbre. Dans la Figure 21, le nœud C avec ces trois fils G, H et I, et les deux fils de H constituent un sous arbre.
- ***Taille d'un arbre :*** est le nombre de nœuds qu'il possède. Par exemple la Taille de l'arbre Figure 21 = 13. Un arbre vide est de taille égale à 0.
- ***Degré d'un nœud :*** est le nombre de ses fils. Dans l'exemple de la Figure 21, le degré des nœuds sont :

Degré(A) = 3 Degré(H) = 2 Degré(K) = 0
Degré(C) = 3 Degré(D) = 2 Degré(G) = 0

Degré(B) = 2 Degré(L) = 0 Degré(E) = 0

- ***Degré d'un arbre :*** est le degré maximum de ses nœuds. Dans l'exemple de la Figure 21, degré de l'arbre =3.
- ***Le niveau d'un nœud :*** est la distance qui le sépare de la racine : Le niveau de la racine est égale 0 et le niveau de chaque nœud est égale au niveau de son père plus 1. Dans l'exemple de la Figure 21, on va calculer le niveau du nœud M :

Niveau(A)=0

Niveau(M)=Niveau(H) +1

Niveau(M)= (Niveau(C) +1) +1

Niveau(M)= ((Niveau(A) +1) +1) +1

Niveau(M)= ((0+1) +1) +1=3

- ***La profondeur (hauteur) d'un arbre :*** est le plus grand niveau, c'est à dire la distance entre la racine et la feuille la plus lointaine. Dans l'exemple de la Figure 02, la profondeur de l'arbre est égale à 3.

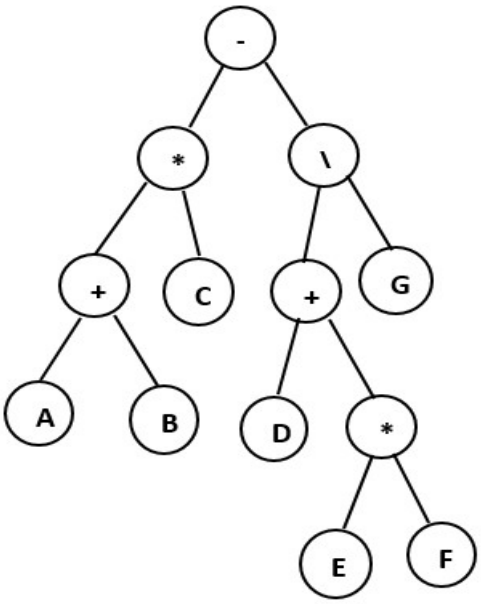
III.Exemple d'utilisation des arbres

Les cas d'usage des arbres peuvent être variés, et de nombreux utilisateur

III.1 Représentation des expressions arithmétique

Soit par exemple l'expression arithmétique suivante :

(A+B)*C-(D+E*F)/G



Taille de l'arbre = 13

Profondeur = 4

Degré d'arbre = 2

Figure 22 : *Exemple de représentation des expressions arithmétique*

III.2 Représentation des chaines de caractère

Soit par exemple les mots suivants :

Information, Informatique, Informer, Inferieur et Infériorité.

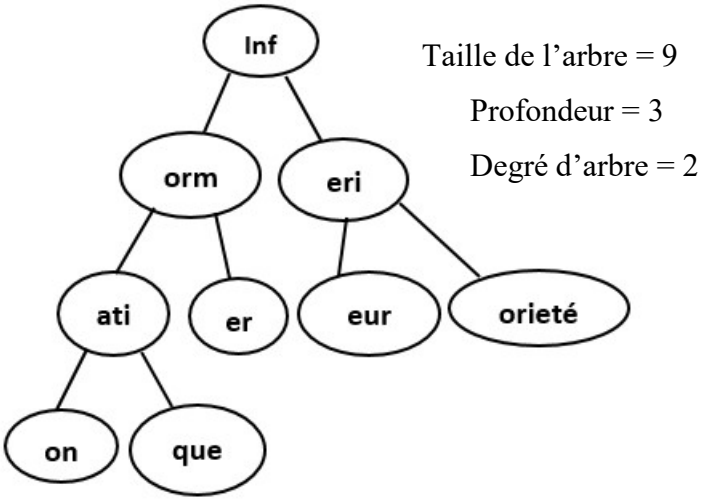


Figure 23 : *Exemple de représentation des chaines de caractère*

III.3 Représentation des arbres généalogiques

Soit par exemple l'arbre suivant :

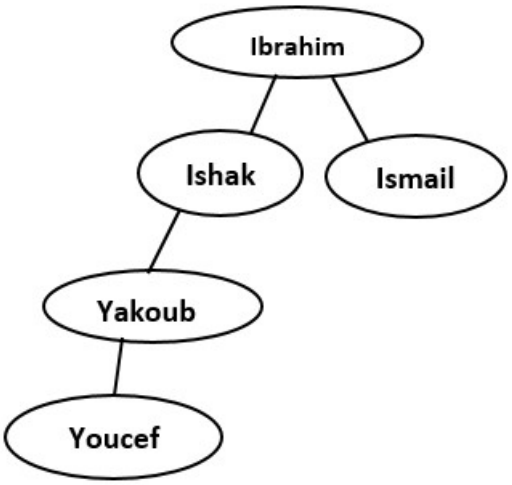


Figure 24 : *Exemple de représentation des arbres généalogique*

IV.Typologie des arbres

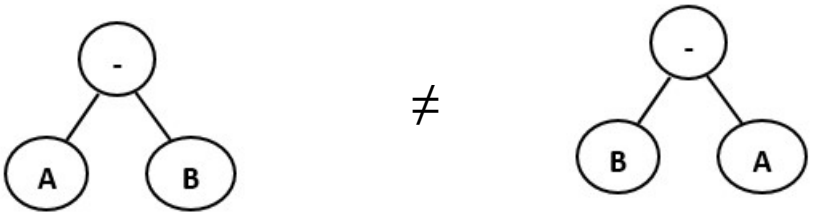
Un arbre se présenter de plusieurs maniéré selon le type de problèmes posées, ils sont classifiés selon :

- **Leur degré m≥2:** On trouve les arbres binaires (m = 2), ternaires (m=3), quaternaires (m = 4) ,…….., m-aire (m>2).
- **La priorité d'ordre :** où chaque nœud possède au moins une clé, les valeurs de sous arbre gauche (droit respectivement.) de la clé sont strictement inférieurs (supérieure ou

égale respectivement.) à la valeur de la clé. On trouve ainsi, les arbres binaires de recherche (où chaque nœud possède une clé) et les arbres m-aire de recherche (où chaque nœud possède (m-1) clé),

Exemple : pour représenter l'opération arithmétique suivante :

A-B ou B-A



- **La propriété d'équilibrage :** où les feuilles se situent au même niveau. On trouve les arbres binaires de recherche équilibré (AVL, rouge et noir, …..) et les arbres m-aire de recherche équilibré (B-arbres, B-arbre*, B-Arbre+, …..).

V.Arbres binaires

V.1 Définition

Un arbre binaire est un arbre où chaque nœud est connecté à deux sous arbres (un sous arbre gauche et un sous arbre droit), C'est un arbre de degré 2, c'est-à-dire que chaque nœud a au plus deux fils. Le premier fils d'un nœud n est appelé Fils gauche (FG) et le deuxième fils est appelé Fils droit (FD).

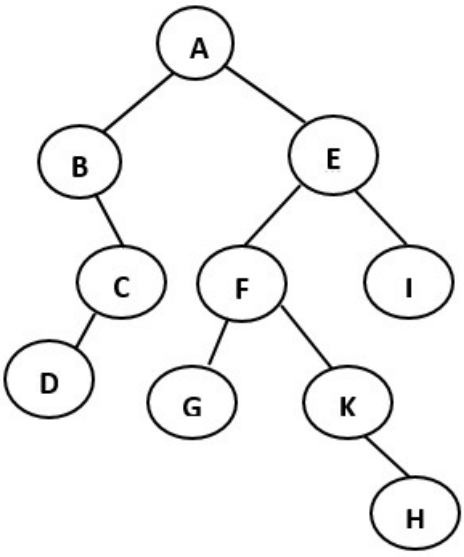


Figure 25 : *Exemple d'arbre binaire*

V.2 Types d'arbres binaires

Il est possible d'avoir des arbres binaires de même taille mais de « forme » très différente :

- Un **arbre binaire plein** (strictement binaire) est un arbre dont tous les nœuds possèdent zéro ou deux fils (si chaque nœud interne a exactement 2 fils différents de NIL.).

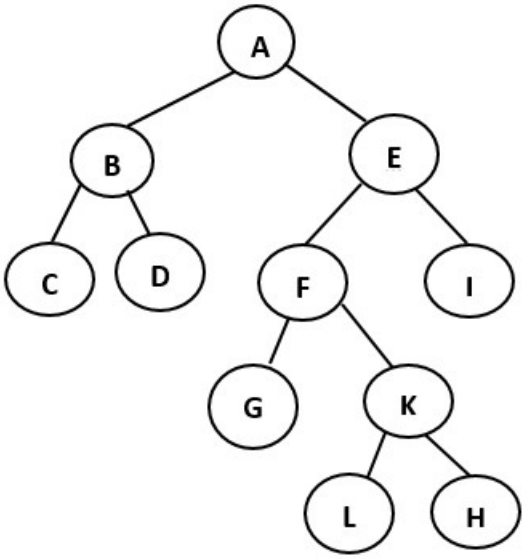


Figure 26 : A*rbre binaire Plein*

- **Arbre binaire complet,** Un arbre binaire est complet si tous les niveaux sont complètement remplis, sauf peut-être le dernier niveau et le dernier niveau a toutes les clés aussi gauches que possible.

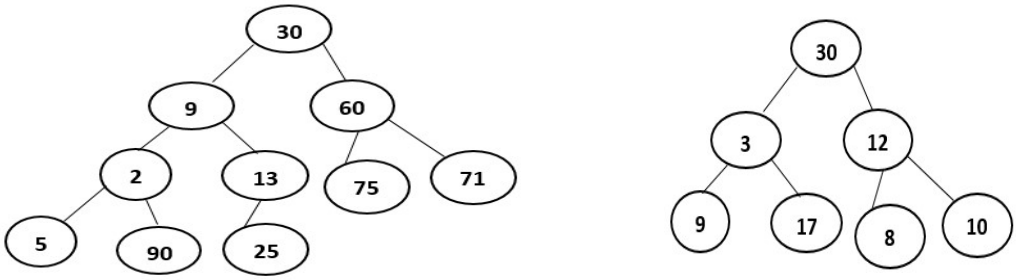


Figure 27 : A*rbre binaire Complet*

- Un **arbre binaire parfait** est un arbre binaire strict dans lequel toutes les feuilles (nœuds n'ayant aucun fils) sont à la même distance de la racine (c'est-à-dire à la même profondeur). Il s'agit d'un arbre dont tous les niveaux sont remplis : où tous les nœuds internes ont deux fils et où tous les nœuds externes ont la même hauteur.

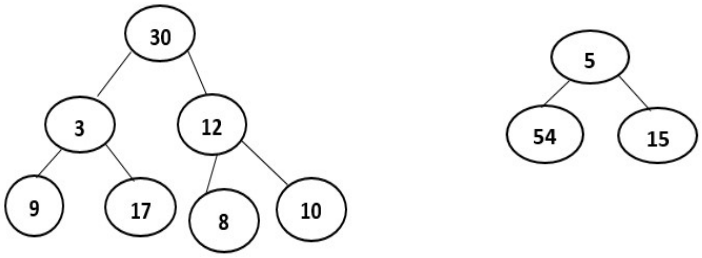


Figure 28 : A*rbre binaire parfait*

V.3 Transformation d'un arbre de degré quelconque en arbre binaire

Dans un arbre de degré quelconque (arbre m-aire), un nœud peut avoir un nombre variable de successeur, c'est pourquoi, il est souvent utile de le transformer en un arbre binaire équivalent. L'arbre binaire se construit par le procédé suivant :

- Etant donné un nœud dans un arbre m-aire, on crée un lien gauche vers son premier fils (fils aîné) et un lien droit vers son frère situé à droite (le plus proche).

Exemple : Soit l'arbre m-aire suivant :

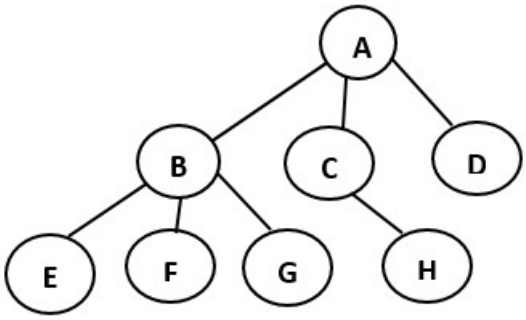


Figure 29 : *exemple d'arbre m-aire*

Il peut être converti en arbre binaire (comme se présenter sur le tableau suivant).

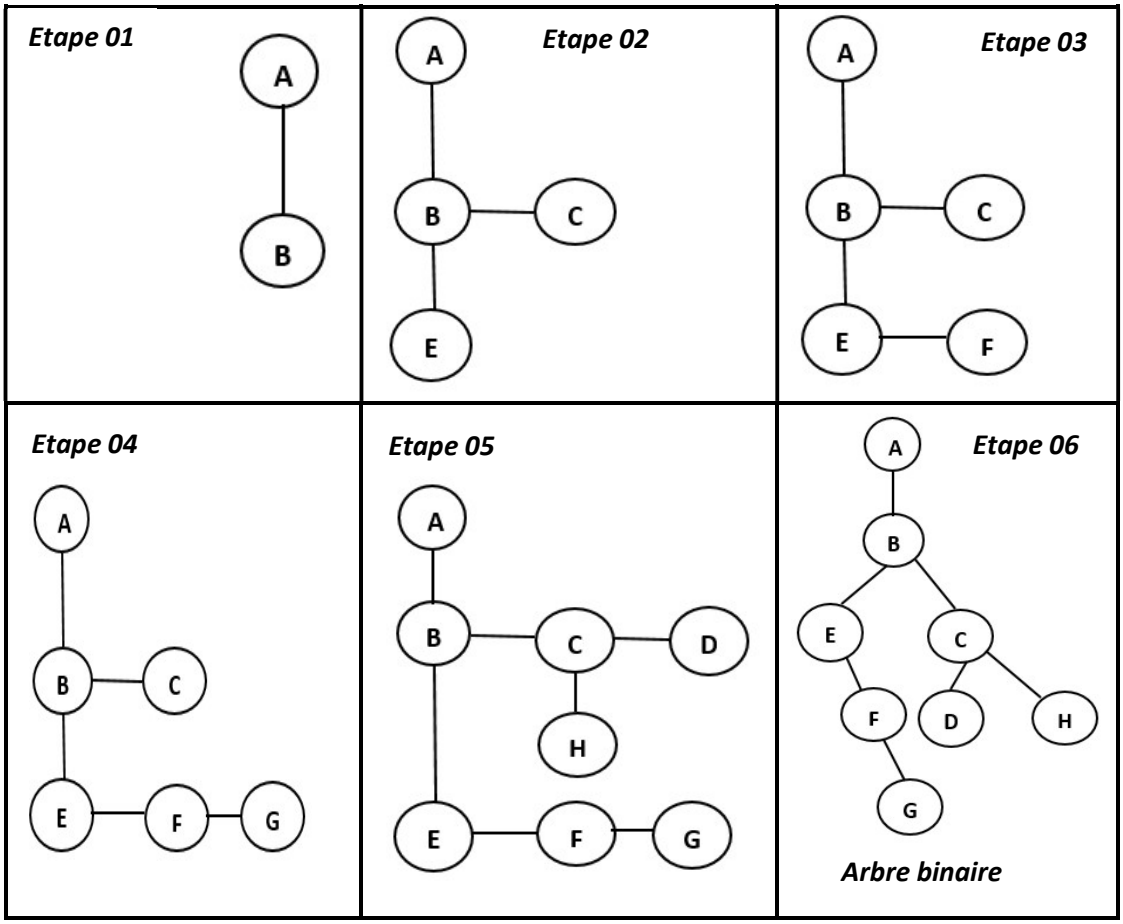


Tableau 01 : *Etapes de transformation d'un arbre de degré quelconque en arbre binaire*

VI.Implémentation des arbres

Les arbres peuvent être représentés par des tableaux, des listes non linéaires ou tous les deux :

VI.1 Représentation statique

L'arbre du premier exemple (Figue 01) peut être représenté par un tableau comme suit :

Num	Information	Fils 1	Fils 2	Fils3
1	A	2	3	4
2	B	5	6	0
3	C	7	8	9
4	D	10	11	0
5	E	0	0	0
6	F	0	0	0
7	G	0	0	0
8	H	12	13	0
9	I	0	0	0
10	J	0	0	0
11	K	0	0	0
12	L	0	0	0
13	M	0	0	0

Tableau 02 : *Représentation statique d'un arbre.*

VI.2 Représentation dynamique

L'arbre du premier exemple (Figue 01) peut être représenté par des pointeurs comme suit :

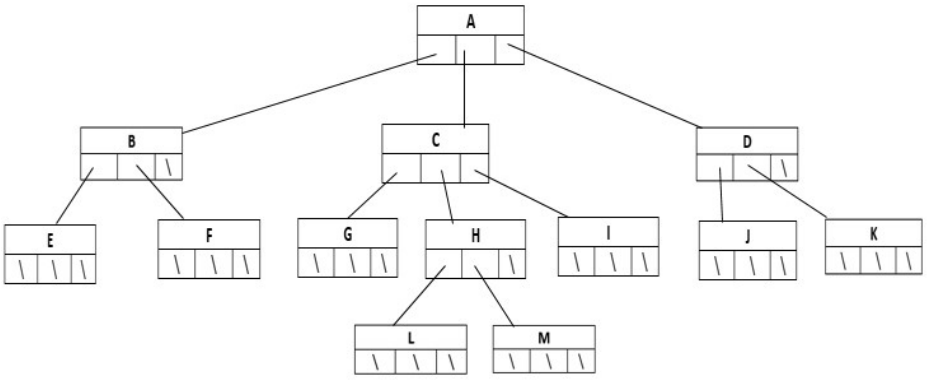


Figure 30 : *Représentation dynamique d'un arbre.*

Nous allons maintenant discuter de l'implémentation des arbres binaire. Tout d'abord, définissons le nœud. Un nœud est une structure (ou un enregistrement) qui contient au minimum trois champs : un champ contenant l'élément du nœud, c'est l'information qui est importante. Cette information peut être un entier, une chaîne de caractère ou tout autre chose que l'on désire stocker. Les deux autres champs sont le fils gauche et le fils droit du nœud. Ces des fils sont en

fait des arbres, on les appelle généralement les sous arbres gauches et les sous arbres droit du nœud. De part cette définition, un arbre ne pourra donc être qu'un pointeur sur un nœud.
Il est important de signaler que la représentation des arbres par un tableau (statique) n'est pas intéressante, si pour cela que nous entamons directement la représentation chainée qui présente par contre une grande souplesse et soit la plus utilisée.

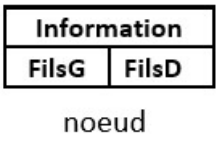
- ***Déclaration d'un arbre non binaire En langage C :***

```
typedef struct noeud * Arbre;
struct noeud
{ int value;
  Arbre fils[4];
};
```

- ***Déclaration d'un arbre binaire En langage C :***

/*Définition d'un nœud*/

```
typedef struct nœud{
  int Val ;
  struct nœud* FilsG ;
  struct nœud* FilsD ;
}Noeud;
```



noeud

/*Définition d'un arbre=un pointeur sur son Nœud racine*/

```
typedef Nœud *Arbre ;
```

VII. Traitement sur les arbres

Diffèrent traitement peuvent être effectue sur les arbres, mais la problématique se situe ou niveau de parcours de l'arbre.
On peut en effet :

- ✓ Parcours un arbre pour effectuer une recherche.
- ✓ Parcours un arbre pour insérer un élément.
- ✓ Parcours un arbre pour supprimer un élément.

Le parcours d'un arbre est l'opération de visiter tous les nœuds de l'arbre, Le parcours d'un arbre peut se faire selon deux stratégies : en profondeur, ou bien en largeur.

- ➢ *En profondeur* : permet d'explorer l'arbre en explorant jusqu'au bout une branche pour passer à la suivante.

- *En largeur (par niveau)* : permet d'explorer l'arbre niveau par niveau. C'est à dire que l'on va parcourir tous les nœuds du niveau 1 puis ceux du niveau deux et ainsi de suite jusqu'à l'exploration de tous les nœuds.

VII.1 Parcours en profondeur

La stratégie consiste à aller en profondeur on visitant les feuilles d'un noeud de l'arbre, puis lorsque toutes les feuilles du noeud ont été visitées, l'algorithme "remonte" au noeud plus haut dont les feuilles n'ont pas encore été visitées.

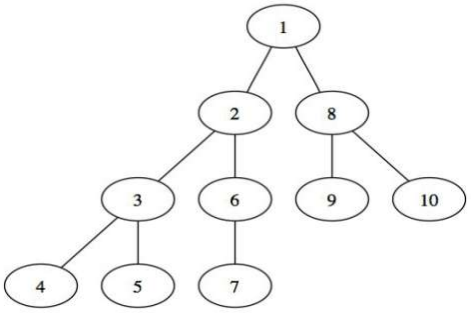


Figure 31 : *Arbre binaire.*

Il existe 3 méthodes de parcours d'un arbre binaire :

- ***Parcours préfixe (Racine, fils Gauche, fils Droit) :*** le noeud racine est traité au premier passage avant le parcours des sous arbre, c'est-à-dire, il consiste à :

-Visiter la racine.

- Parcourir le sous arbre gauche.

- Parcourir le sous arbre droite.

La procédure (récursive) qui affiche les valeurs en parcours préfixé d'un arbre A est :

```
void Prefixe(Arbre A){
        if(A !=NULL){
            printf(A->Val) ;
            Prefixe(A->FilsG) ;
            Prefixe(A->FilsD) ;}}
```

Le résultat de parcours prefixe de l'arbre précèdent est : 1,2, 3, 4, 5, 6, 7, 8, 9, 10.

Le parcours préfixe est utilisé pour créer une copie de l'arbre. Le parcours préfixe est également utilisé pour obtenir l'expression préfixe d'un arbre d'expression.

- Parcours infixe ou symétrique (fils Gauche, Racine, fils Droit) : le noeud racine est traité au second passage après le parcours du sous arbre gauche et avant le parcours du sous arbre droit, c'est-à-dire, il consiste à :

- Parcourir le sous arbre gauche.
-Visiter la racine.
- Parcourir le sous arbre droite.

La procédure (récursive) qui affiche les valeurs en parcours infixe d'un arbre A est :

```
void Infixe(Arbre A){
        if(A !=NULL){
            Infixe(A->FilsG) ;
            printf(A->Val) ;
            Infixe(A->FilsD) ;}}
```

Le résultat de parcours en infixe de l'arbre précèdent est : 4, 3, 5, 2, 7, 6, 1, 9, 8, 10

Dans le cas des arbres binaires de recherche, le parcours préfixe donne des nœuds dans un ordre croissant.

- Parcours postfixe (fils Gauche, fils Droit, Racine) : le noeud racine est traité au dernier passage après le parcours des sous arbres, c'est-à-dire, il consiste à :

- Parcourir le sous arbre gauche.
- Parcourir le sous arbre droite.
-Visiter la racine.

La procédure (récursive) qui affiche les valeurs en parcours postfixé d'un arbre A est :

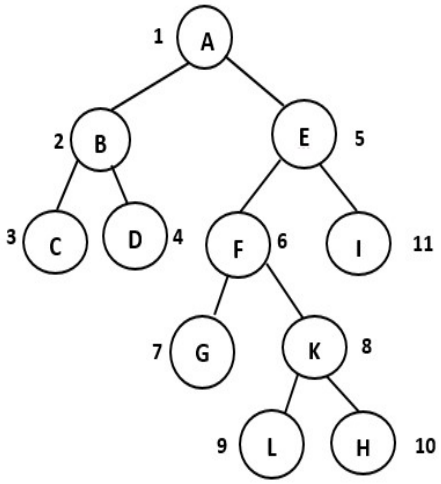
```
void Postfixe(Arbre A){
        if(A !=NULL){
            Postfixe(A->FilsG) ;
            Postfixe(A->FilsD) ;
            printf(A->Val) ;
          }}
```

Le résultat de parcours postfixe de l'arbre précèdent est : 4, 5, 3, 7, 6, 2, 9, 10, 8, 1

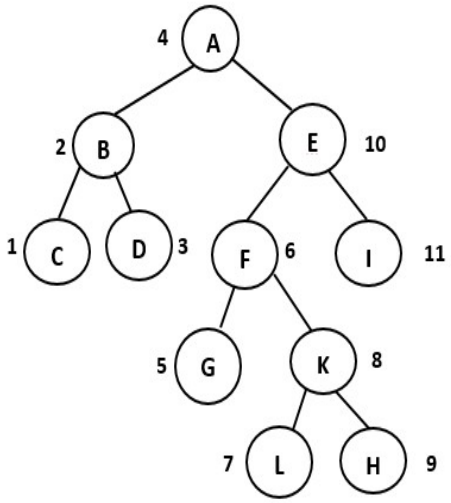
Parcours postfixe est utilisé pour supprimer l'arbre. Parcours postfixe est également utilisé pour obtenir l'expression postfixe d'un arbre d'expression.

Exemple

Parcours Prefixe : A ,B,D,E,F,G,K,L,H,I



Parcours infixe :C,B,D,A,G,F,L,K,H,E,I



Parcours Postfixe : C,D,B,G,L,H,K,F,I,E,A

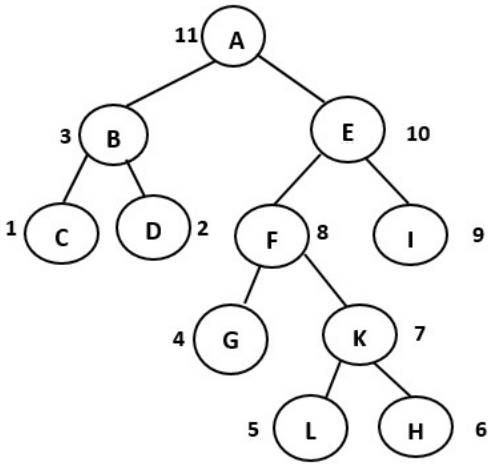


Tableau 03 : *Exemple de parcours en profondeur.*

VII.2 Parcours en largeur ou hiérarchique

Il permet d'explorer l'arbre niveau par niveau. C'est à dire que l'on va parcourir tous les nœuds du niveau 1 puis ceux du niveau deux et ainsi de suite jusqu'à l'exploration de tous les nœuds.

Contrairement au précédent, ce parcours essaie toujours de visiter le nœud le plus proche de la racine qui n'a pas déjà été visité.

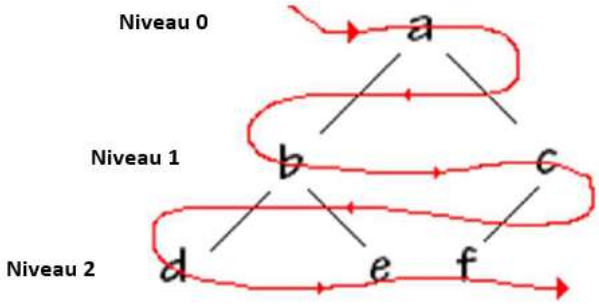


Figure 32 : *Parcours d'un arbre en largeur*

La procédure (récursive) qui affiche les valeurs en parcours en largeur d'un arbre A est :

```
void largeur(Arbre A){
if(A !=NULL){
   File F ;
   Init(F) ;
   Enfiler(F ,A) ;
   while( !FileVide(F)){
          Defiler(F,A) ;
          printf(A->Val) ;
          if(A->FilsG !=NULL) Enfiler(F,A->FilsG) ;
          if(A->FilsD !=NULL) Enfiler(F,A->FilsD) ;}}}
```

Le parcours en largeur de l'arbre précèdent (Figure 31) est : 1, 2, 8, 3, 6, 9, 10, 4, 5, 7

VIII. Arbre binaire de recherche

Un arbre binaire de recherche (ABR) est un arbre binaire ordonné tel que pour tout noeud n :

- ✓ Toutes les valeurs du sous arbre gauche de n sont inférieures à la valeur de n.
- ✓ Toutes les valeurs du sous arbre droit de n sont supérieures à la valeur de n.
- ✓ Et, il n'y a pas de nœuds égaux.

Un arbre binaire de recherche est intéressant puisqu'il est toujours possible de connaître dans quelle branche de l'arbre se trouve un élément et de proche en proche le localiser dans l'arbre. Les propriétés ci-dessus de l'arbre binaire de recherche permettent de classer les clés afin que les opérations telles que et maximum recherche**,** minimum puissent être effectuées rapidement. S'il n'y a pas d'ordre, il se peut que nous devions comparer chaque clé pour rechercher une clé donnée.

Exemple

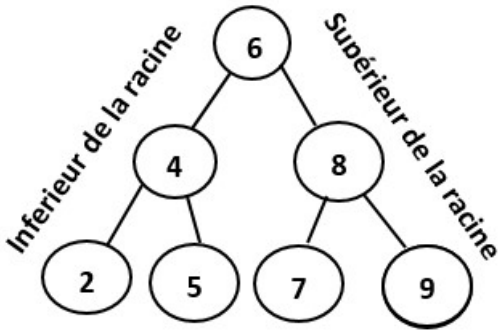


Figure 33 : *Arbre binaire de recherche*

VIII.1 Rechercher un élément

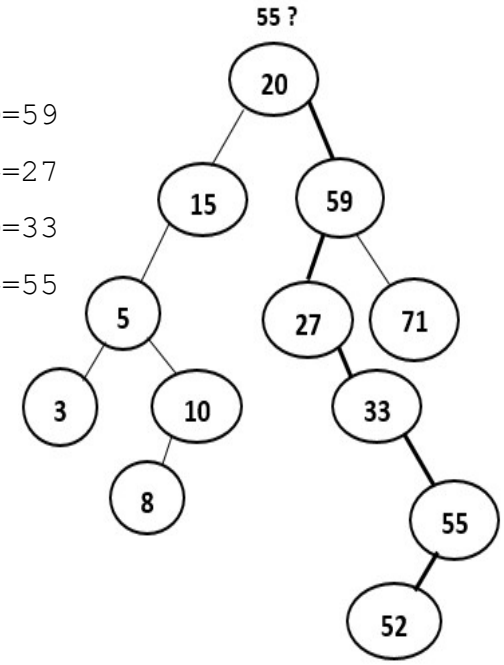
Pour rechercher une clé donnée dans l'arbre binaire de recherche, nous la comparons d'abord avec racine, si la clé est présente à racine, nous retournons racine. Si la clé est supérieure à la clé de la racine, on recommence pour le sous arbre droit du noeud racine. Sinon, le sous arbre gauche. Pour savoir si un élément existe, il faut parcourir seulement une branche de l'arbre. Ce qui fait que le temps de recherche est directement proportionnel à la hauteur de l'arbre.

L'algorithme se base directement sur les propriétés de l'arbre, si l'élément que l'on cherche est plus petit que la valeur du noeud alors on cherche dans le sous arbre de gauche, sinon, on cherche dans le sous arbre de droite. Ce qui se traduit de la façon suivante :

```
bool Trouver(Arbre A , int elt)
{
if ( A==NULL )
    return false;
else if ( A->Val == elt )
         return true;
else if ( A->Val > elt )
       return Trouver(A->FilsG, elt);
else
       return Trouver(A->FilsD, elt);
}
```

Exemple :

- Trouver(A,55)=>la racine 20
- Trouver(A->FilsD,55)=> A->FilsD=59
- Trouver(A->FilsG,55)=> A->FilsG=27
- Trouver(A->FilsD,55)=> A->FilsD=33
- Trouver(A->FilsG,55)=> A->FilsG=55
- Elément Trouver



VIII.2 Insertion d'un élément dans un ABR

Pour insérer un nouvel élément dans un ABR il faut d'abord repérer sa place dans l'arbre, il faut donc le comparer aux éléments déjà existants dans l'ABR. Enfin l'insérer comme fils du dernier noeud visité. Le nouveau nœud est ajouté en tant qu'enfant de la feuille, ce qui se traduit de la façon suivante :

```
void AjouterElt(Arbre A, int elt)
{    if ( A == NULL )
       { A = Create(elt,NULL,NULL);}
     else
        if ( A->Val > elt)
             AjouterElt(A->FilsG,elt);
        else if(A->Val<elt)
               AjouterElt(A->FilsD,elt);
             else printf("element existe déja");
}
```

Remarque :

La fonction Create est la suivant :

```
Arbre *Create(int valeur,Arbre *G,Arbre* D)
{ Arbre A;
A=malloc(sizeof(Arbre));
A->Val=valeur ;
A->FilsG=NULL ;
A->FilsD=NULL ;
   return(A);}
```

Exemple

Ajouter les deux éléments 1 et 15 à un arbre binaire de recherche.

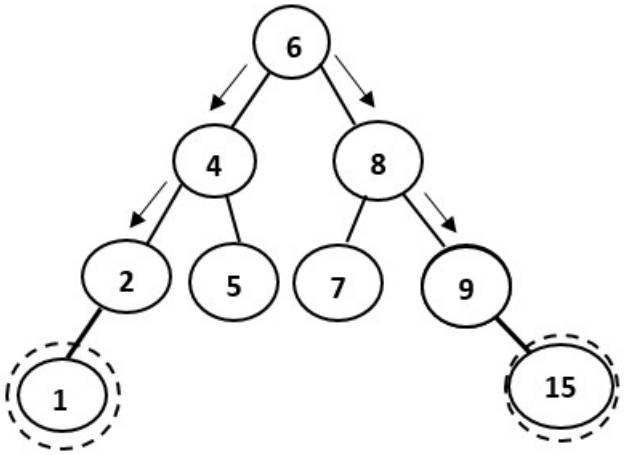


Figure 34 : *exemple d'ajout des éléments à un ABR*

VIII.3 Suppression d'un élément d'un ABR

La suppression dans un ABR est assez compliquée, c'est pour cela que nous allons détailler tous les cas possibles.

- Pour supprimer un noeud dans un ABR, plusieurs cas de figure peuvent se présenter. Il est toutefois nécessaire d'obtenir un ABR à l'issue de la suppression.
- D'abord il faut chercher l'élément à supprimer, une fois trouvé on se trouve dans l'une des situations suivantes, soit « x » le noeud à supprimer :
  - ✓ *1er cas :* x est une feuille : on la supprime et on la remplace par Nil.
  - ✓ 2eme cas : x est un noeud qui a un seul fils : on supprime x et on le remplace par ce fils.
  - ✓ *3eme cas :* x est un noeud qui a deux fils : on supprime x et on le remplace par l'élément minimum se trouvant dans son sous arbre droit (le noeud le plus à gauche du sous arbre

droit) ou par l'élément maximum se trouvant dans son sous arbre gauche (le noeud le plus à droite du sous arbre gauche).

VIII.3.a. 1er cas : x est feuille

Exemple : Supprimer la feuille 3.

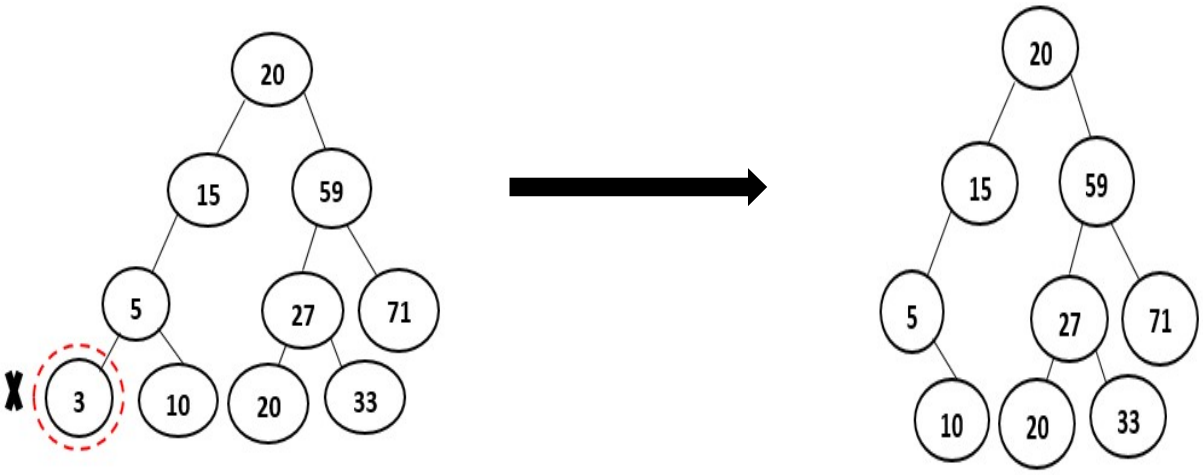


Figure 35 : *Exemple de suppression d'une feuille à un ABR*

VIII.3.b. 2eme cas : x est un noeud qui a un seul fils

Exemple :

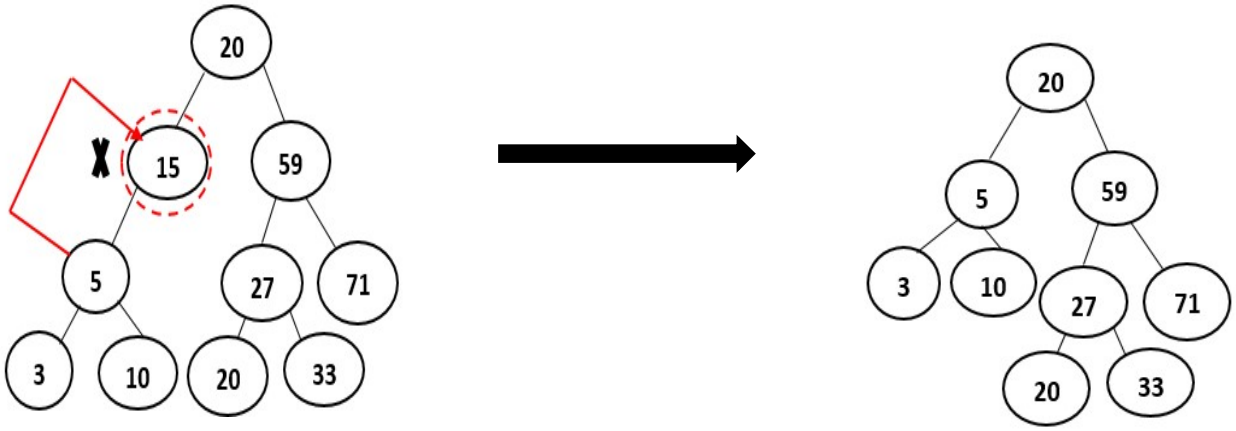


Figure 36 : *Exemple de suppression d'un nœud qui a un seul fils à un ABR.*

VIII.3.c. 3eme cas : x est un noeud qui a deux fils

Exemple :

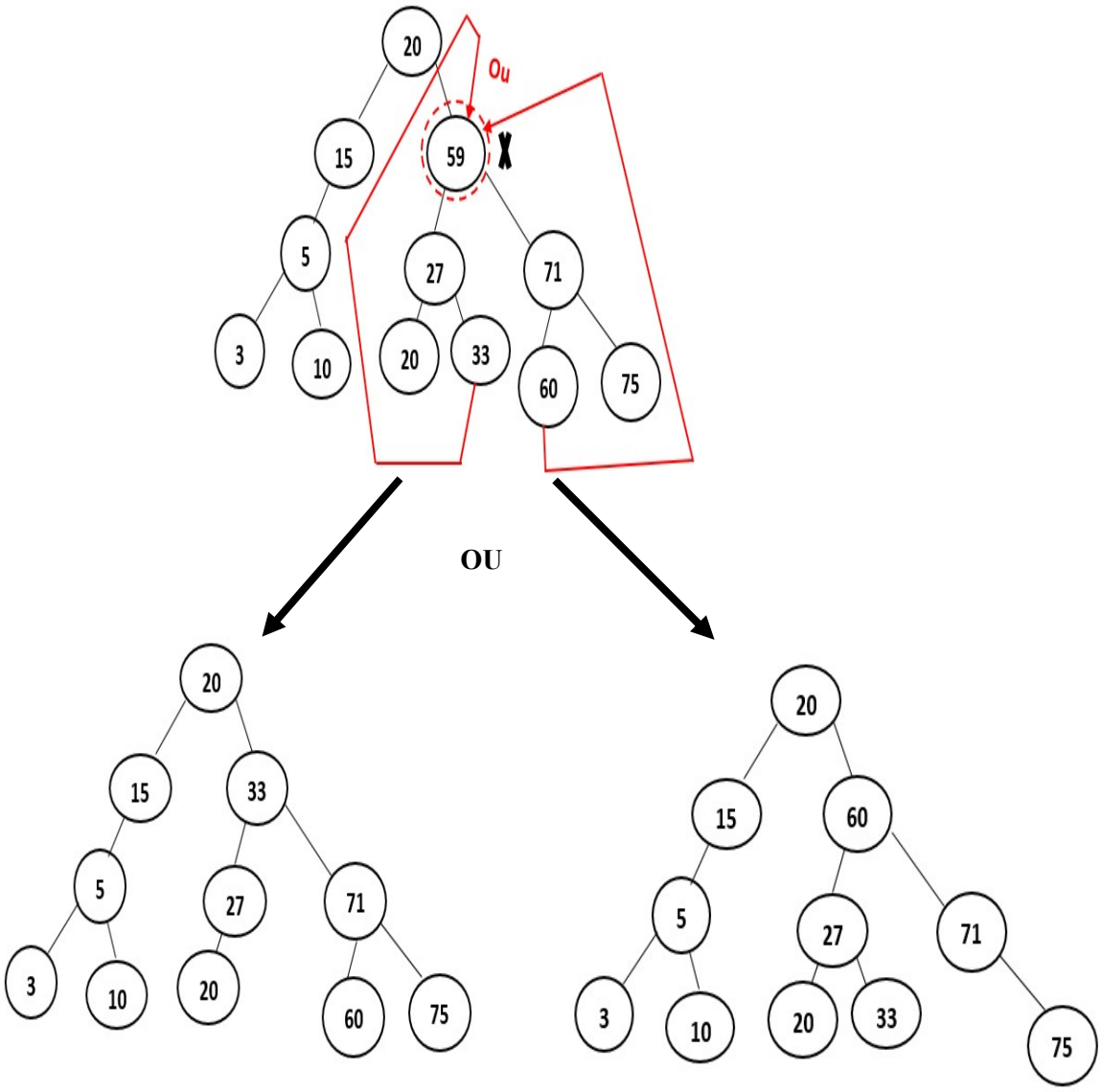


Figure 37 : *Exemple de suppression d'un nœud qui a deux fils à un ABR.*

Partie A

Chapitre 06 : Technique de hachage

V. Introduction

On utilise souvent en informatique une structure appelée "Table" ou dictionnaire pour ranger des informations en mémoire. Une table est un ensemble de couples <clé, information> où chaque clé n'apparaît qu'une seule fois dans la table.

Exemples :

1. Annuaire téléphonique
2. Dictionnaire
3. Codage

Le problème posé est : comment organiser la table pour que les accès (recherche, insertion, suppression) soient les plus rapides possibles.

VI. Type d'organisation des clés dans la table

Il existe en générale trois façons de les ranger

II.1 Accès séquentiel

Il consiste à ranger les clés dans la table dans l'ordre de leur arrivée, les une à la suite des autres, c'est-à-dire que l'ajout se fait toujours à la fin de la table. La recherche d'une clé consiste à tester les éléments l'un après l'autre jusqu'à la trouver c'est-à-dire passer par tous les éléments placés avant. L'insertion se fait à la fin du tableau. La recherche est alors en moyenne de n/2 (O(n)).

II.2 Table triée

La table est triée selon les valeurs des clés donc l'insertion des éléménts provoque un décalage des éléments du tableau : la recherche est en log2(n) c'est à dire dichotomique.

II.3 Hachage (HashCoding)

C'est une technique très utilisée en informatique, elle se base sur une fonction h appelée fonction de hachage ou de hashcoding, qui appliquée à la clé fournit l'indice correspondant dans la table. Cette troisième façon donne la possibilité de ranger une donnée dans un tableau :
- Ranger la donnée « x » à un emplacement « y » calculé par une fonction « h» tel que y=h(x).
On parle ici de table de rangement dispersé ou technique de HACHAGE. Dans ce type de rangement, que ce soit pour insérer ou rechercher une donnée, on procédera toujours aussi rapidement (O(c)).

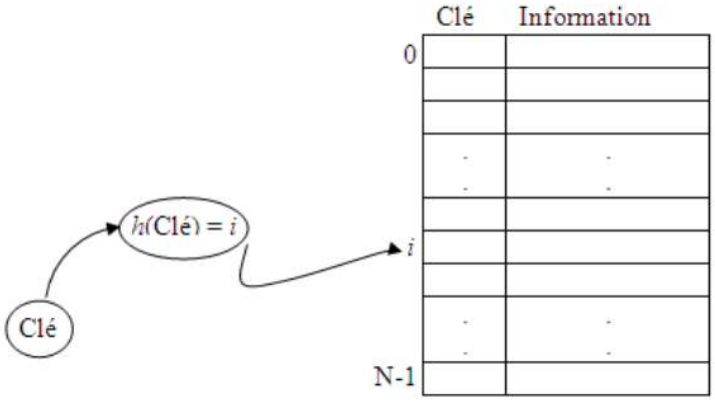


Figure 38 : *schéma de table de hachage*

Exemple : Codage

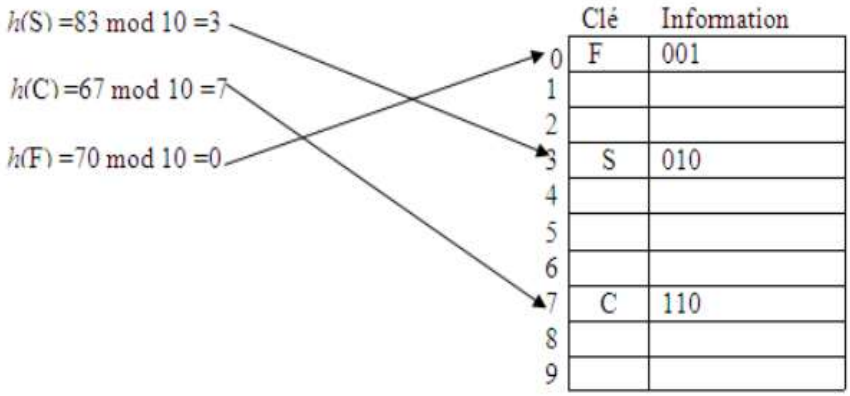


Figure 39 : *exemple de Codge*

VII. Fonction de hachage

Nous allons donner maintenant quelques principes de construction de fonction de hachage. Ceux-ci sont modulables entre eux. Nous supposerons pour les exemples que les clés sont des mots représentés en mémoire par une suite de bits interprétable comme un entier.

Remarque : L'intérêt est qu'il n'y a pas besoin de calcul pour l'interprétation de la valeur.

Nous devons ensuite réduire les valeurs obtenues à l'intervalle [1,m] ou bien [0,m-1] (il est souvent plus facile de travailler en fonction de 0).

On trouve plusieurs types de fonctions utilisées en Hashcoding :

Méthode principales de hachage	
Pour avoir une valeur de e	Pour avoir un indice dans T
Extraction	Division
Compression	Multiplication

Tableau 04 : *Méthode principales de hachage.*

III.1 Hachage par Extraction

Principe : On extrait une partie de la valeur (de la chaîne de bits) de l'élément e

Par exemple :

- Seulement certains bits.
- Seulement certaines zones de bits.

*Exemple : E*xtraction des bits 1, 2,7 et 8 de l'élément.

élément	représentation	h (élément)$_2$	h (élément)$_{10}$
ET	0010110100	1000	8
OU	0111110101	1101	13
NI	0111001001	1101	13
IL	0100101100	0000	0

Tableau 05 : *Exemple d'Hachage par extraction des bits.*

Avantages :

calcul facile à mettre en œuvre : Si N = 2P − 1 alors h → un indice dans T

Inconvénients :

adaptée seulement à des cas particuliers :

- Quand on connaît la valeur des éléments a priori
- Quand on sait que certains bits ne sont pas significatifs

En général, méthode par extraction → pas de bons résultats car ne dépend pas de la totalité de la valeur de l'élément.

III.2 Hachage par Compression

Principe : On utilise tous les bits de e pour calculer son indice dans la table.

Par exemple :

- On découpe la chaîne de bits de l'élément en morceaux d'égale longueur.
- On additionne les morceaux. Pour éviter les débordements on peut utiliser, à la place de l'addition, l'opération booléenne "ou exclusif" (xor).

Exemple :

Elément	Calcul	h (élément)$_2$	h (élément)$_{10}$
ET	00101 xor 10100	10001	17
OU	01111 xor 10101	11010	26
NI	01110 xor 01001	00111	7
CAR	00011 xor 00001 xor 10010	10000	16

Tableau 06 : *Exemple d'Hachage par Compression des bits.*

Avantages :

- Valeur de e = taille d'un mot mémoire
- Combinaison possible avec les méthodes qui suivront

Inconvénients :

- Hache de la même façon toutes les permutations d'un même mot → h(CAR) = h(ARC)
- Vient du fait que toutes les sous chaines de bits sont des représentations de caractères → plusieurs sous-chaines peuvent être identiques

Par exemple décaler circulairement les sous-chaînes de bits :

1iere sous-chaîne → 1 bit vers la droite

2ieme sous-chaîne → 2 bits vers la droite

3ieme sous-chaîne → 3 bits vers la droite. . .

Dans ce cas

Elément	Calcul	h (élément)$_2$	h (élément)$_{10}$
ARC	10000 xor 10100 xor 01100	01000	8
CAR	10001 xor 01000 xor 01010	10011	19

Tableau 07 : *Exemple 2 d'Hachage par Compression des bits.*

Exemple :

- Le hachage par compression partage la représentation binaire de la clé et opère une suite d'opérations.
- Partage de la clé en mots de 5 bits puis sommation bit-a bit par ou-exclusif :

h(ET) = 00101 xor 10100 = 10001

h(OU) = 01111 xor 10101 = 11010

h(NI) = 01110 xor 01001 = 00111

h(IL) = 01001 xor 01100 = 00101

III.3. Hachage par division

principe : Le hachage par division divise la clé par la taille du tableau. h(e) = e mod N , et il doit tenir compte de l'uniformité des données par rapport aux diviseurs de la taille du tableau.

exemple : Taille de tableau : 23

h(ET = 0010110100) = 180 mod 23 = 19

h(OU = 0111110101) = 501 mod 23 = 18

h(NI = 0111001001) = 457 mod 23 = 20

h(IL = 0100101100) = 300 mod 23 = 1

Exemple :

Taille de la table N = 37

Elément	h (élément)$_{10}$	Calcul	h (élément)$_{10}$
ET	180	180 mod 37	32
OU	501	501 mod 37	20

Tableau 08 : *Exemple d'Hachage par division.*

Pour éviter des phénomènes d'accumulation il faut choisir un diviseur premier.

Avantages : facile et rapide à calculer.

Inconvénients : dépend trop de N.

Par exemple :

Si N est pair alors tous les éléments vont aller dans les indices pairs de T.

Si N est impair alors tous les éléments vont aller dans les indices impairs de T.

Si N a des petits diviseurs . . .

- h n'est pas uniforme.

- il faudrait que la taille N de T soit un nombre premier (comme dans l'exemple) . . .

Mais il peut tout de même y avoir des phénomènes d'accumulation.

III.4 Hachage par multiplication

Principe : Le hachage par multiplication multiplie la clé par un réel θ (0 < θ < 1), garde la partie décimale puis multiplie par la taille du tableau et prendre la partie entière

Algorithme :

1. r ← e ∗ θ
2. On garde la partie décimale de r
3. r ← r × N : on multiplie par la taille du tableau
4. On garde la partie entière de r

$$h(e) = [((e \times \theta) \bmod 1) \times N]$$

Exemple 01 :

Supposons que :

$\theta = 0.6125423371$ et $N = 30$

Alors $h(ET) = [((180 \times \theta) \bmod 1) \times N]$

$= [(110.87016302 \bmod 1) \times 30]$

$= [0.87016302 \times 30]$

$= 26$

Exemple 02 :

Taille de tableau : 20 et $\theta = 0.123456$.

$180 * 0.123456 = 22.222080$ h(ET) = 4

$501 * 0.123456 = 61.851456$ h(OU) = 17

$457 * 0.123456 = 56.419390$ h(NI) = 8

$300 * 0.123456 = 37.036800$ h(IL) = 0

Avantages :

La taille du tableau est sans importance

Inconvénients :

La valeur de θ doit être choisie avec soin → pas trop près de 0 ni de 1 pour éviter les accumulations aux extrémités de la table.

- Des études théoriques montrent que les valeurs de θ qui repartissent uniformément les éléments sont :

  $\theta = (\sqrt{5} - 1)/2 \approx 0.6180339887$

  $\theta = 1 - (\sqrt{5} - 1)/2 \approx 0.3819660113$

VIII. Méthode de résolution de collision

Un problème sérieux se pose avec les fonctions de hachage si deux clés différentes donnent lieu à une même adresse lorsqu'on leur applique la fonction de hachage, c'est à dire

$$h(cle1) = h(cle2).$$

Une telle situation est appelée « collision » et plusieurs solutions existent pour sa résolution. Nous présentons deux grandes techniques pour résoudre les collisions :

- Par chaînage (méthode indirecte).
- Par calcul (méthode directe).

IV.1 Par chaînage (méthodes indirectes)

Dans ce type de méthodes, les éléments en collision sont chaînés entre eux. Soit à l'extérieur du tableau (hachage avec chaînage séparé), soit dans une zone de débordement (hachage coalescent) si l'allocation dynamique n'est pas possible (ce qui est rarement possible ;). Je veux dire que ce ne soit pas possible).

IV.1. a Hachage avec chaînage séparé

Comme nous le disions, cette méthode chaîne les éléments entre eux à l'extérieur du tableau de hachage. Chaque case du tableau de hachage contient un pointeur sur une liste chaînée d'éléments dont la valeur de hachage primaire correspond à l'indice de la case du tableau.
Pour les valeurs d'exemple suivantes :

éléments	X_1	X_2	X_3	X_4	X_5	X_6	X_7	X_8	X_9	X_{10}	X_{11}	X_{12}	X_{13}	X_{14}	X_{15}
Valeurs de hachage	1	3	5	2	1	3	1	7	4	5	3	6	3	9	2

Tableau 09 : *Exemple tableau des valeurs de hachage.*

Nous aurions le tableau de hachage TabH suivant :

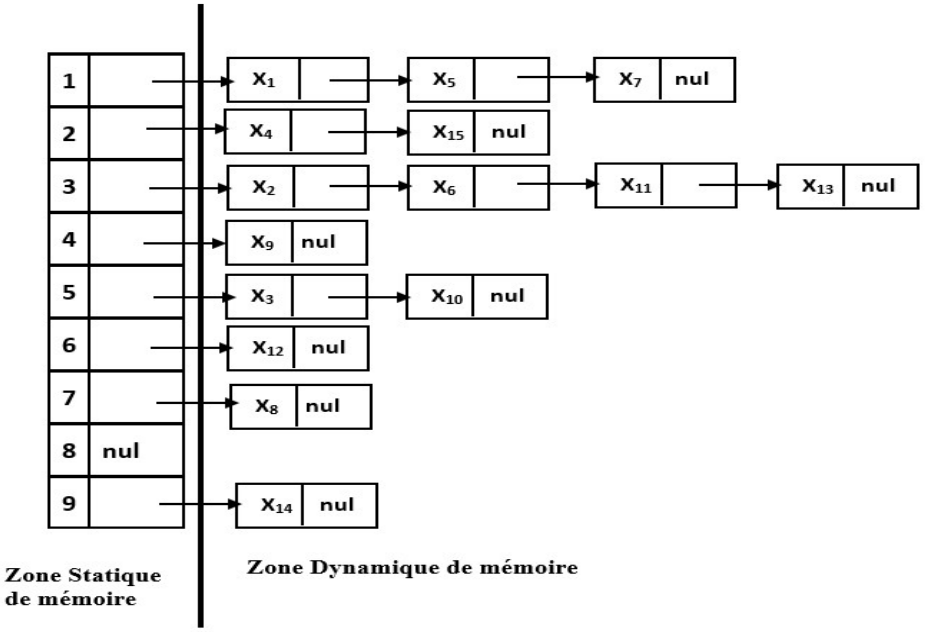


Figure 40 : *Exemple d'Hachage avec chaînage séparé*

Remarque :

Les éléments en collision pourraient être représentés sous d'autres formes (arbres, etc.), mais cela ne présente pas d'intérêt dans la mesure où : Si la fonction de répartition est uniforme et adaptée (notre postulat de départ) à la collection de données, le nombre de collision ne devrait

pas excéder 5, ce qui en terme de recherche (même séquentielle :D) est tout à fait "tolérable/acceptable".

Recherche

La recherche est simple à implémenter, il suffit pour l'élément de déterminer sa valeur de hachage, et ensuite de comparer chaque élément de la liste des éléments en collision (sur cette valeur) pour déterminer si celui que l'on cherche existe ou non.

Ajout

Pour l'ajout, il y a deux possibilités :

- Recherche d'appartenance de l'élément à ajouter, et s'il n'existe pas, ajout de celui-ci en fin de liste de collision (nous y sommes déjà après la recherche). L'intérêt est de maintenir des listes courtes,
- Ajout en première place de la liste de collision. L'avantage est qu'il n'y a pas de recherche préalable, donc gain de temps. L'inconvénient est de rallonger les listes de collision.

Suppression

Comme pour l'ajout, il y a deux possibilités dont le choix dépend de celle choisie pour l'ajout :

- Recherche de l'élément à supprimer et s'il existe, suppression,
- Recherche de toutes les occurrences possibles de l'élément à supprimer et, le cas échéant, suppression. Le principal inconvénient est de devoir systématiquement parcourir toute la liste de collision.

IV.1.b Hachage coalescent

Si l'allocation dynamique de la mémoire n'est pas possible, il est alors nécessaire de chaîner les éléments entre eux à l'intérieur du tableau de hachage. Chaque indice de ce dernier référence alors deux champs : un élément et un lien (généralement entier) représentant l'indice du tableau où se trouve l'éventuel élément en collision primaire avec celui-ci.
Le principe est de séparer le tableau de hachage (de taille ***m***) en deux zones :

- une ***zone de hachage primaire*** de ***p*** éléments,
- une ***zone de réserve*** de ***r*** éléments permettant de gérer les collisions.

Les contraintes sont d'avoir ***m=p+r*** supérieur ou égal au nombre d'éléments de la collection à "hacher" et un élément de lien, pour chaque valeur d'indice à l'intérieur du tableau de hachage. Lorsqu'un élément est à placer et que sa valeur de hachage primaire atteint une case vide de la zone principale, il est placé là, sinon, il est placé dans la zone de réserve et le lien de collision est mis à jour.

Remarque : La zone de réserve est toujours utilisée en ordre décroissant d'indices.

Avec ***p=6*** et ***r=3***, pour les valeurs d'exemple suivantes :

éléments	X_1	X_2	X_3	X_4	X_5	X_6	X_7	X_8
Valeurs de hachage	1	3	1	3	4	2	1	6

Tableau 10 : *Exemple 2 tableau des valeurs de hachage.*

Nous aurions le tableau de hachage TabH suivant :

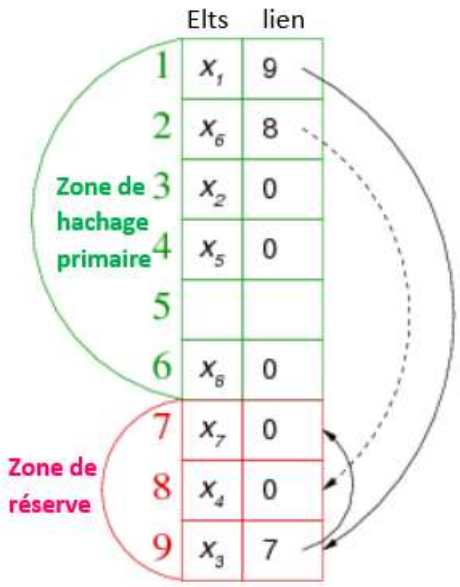


Figure 41 : *Exemple d'Hachage* coalescent

Remarque : Dans ce cas, la valeur de hachage n'est plus calculée sur **m**, mais sur **p**.

Comme nous pouvons le constater, la difficulté réside dans le choix de la taille de la réserve. Sur l'exemple précédent, si nous voulions ajouter un élément dont la valeur de hachage primaire est ***1, 2, 3, 4*** ou ***6***, il serait en collision avec un autre élément déjà présent dans le tableau et

nous serions dans l'impossibilité de lui trouver une place dans la mesure où la réserve est déjà pleine.

Le problème est le suivant :

- Si la réserve est trop petite, elle se remplie trop vite et l'utilisation du tableau est incomplète,
- Si la réserve est trop grande, l'effet de dispersion est perdu et une extrapolation donnerait une liste chaînée (ce qui ne présente aucun intérêt).

Dans ce cas, une solution est de ne pas considérer de zone de réserve, d'utiliser ***m*** en valeur maximale de hachage primaire et de gérer les collisions comme s'il existait une zone de réserve (en partant de l'indice maximum et en remontant).

L'inconvénient, c'est que cela crée des collisions qui ne sont pas dues à la coïncidence des valeurs de hachage primaire. On les appelle des ***collisions secondaires***.

En utilisant ce principe et ***m=17***, pour les valeurs d'exemple suivantes :

éléments	X_1	X_2	X_3	X_4	X_5	X_6	X_7	X_8	X_9	X_{10}	X_{11}	X_{12}	X_{13}	X_{14}	X_{15}
Valeurs de hachage	1	3	5	2	1	3	1	17	4	5	3	14	3	9	2

Nous aurions le tableau de hachage TabH suivant :

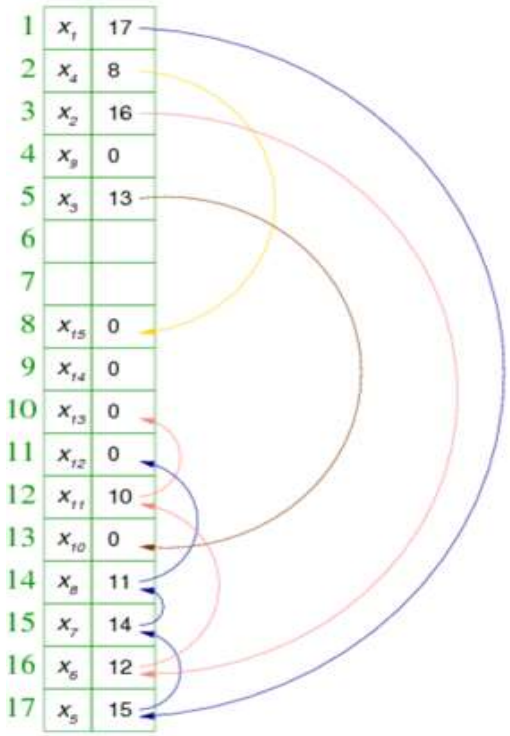


Figure 42 : *Exemple d'Hachage* coalescent *secondaires*.

Remarque : Nous utilisons directement les valeurs d'indice pour le lien des collisions.

Comme on peut le voir sur l'exemple précédent, au moment d'ajouter x_8 de valeur de hachage primaire ***17***, on constate que sa place est déjà utilisée pour gérer la collision de x_5 sur x_1. En effet, ces deux éléments ont même valeur de hachage primaire (***1***). On résout le problème en posant x_8 au premier emplacement libre en remontant (***14***) et on le lie avec sa case théorique (***17***). Dans ce cas, les listes de valeurs de hachage ***1*** et ***17*** fusionnent, comme d'autres par la suite. C'est ce phénomène qui est à l'origine du nom hachage coalescent.

IV.2 Par calcul (méthodes directes)

Dans ce type de méthodes, les liens n'existent plus. L'avantage est de pouvoir utiliser leur place mémoire pour d'autres éléments. Le problème est de pouvoir gérer les collisions entre les éléments. La résolution se fait à l'aide de calcul à l'intérieur du tableau. Comme pour le hachage coalescent, ***m*** doit être supérieur ou égal au nombre d'éléments à placer.

La solution repose sur la conception d'une *fonction d'essais successifs* :

essai : $E \rightarrow \{1,2,\ldots,m\}^m$

$: x \rightarrow (essai_1(x), essai_2(x), \ldots, essai_m(x))$

Avec

$\forall i \in \{1,\ldots,m\} \quad essai_i(x) \in \{1,\ldots,m\}$

Et

$Si\ i \neq j\ alors\ essai_i(x) \neq essai_j(x)$

Pour un tableau de m valeurs de hachage, m fonctions de hachage h1 (hachage primaire), h2,.., hm.

- Pour tout élément x de l'univers des clés, h1(x), . . ., hm(x) doit être une permutation de [1...m].
- Les fonctions de hachage sont testées dans l'ordre jusqu'à ce que ce soit la clé (recherche positive) ou une place vide ou qu'il n'y ait plus de fonction de hachage (recherche négative).

IV.2.a Hachage linéaires

Calcul de la fonction de hachage hi suivant la fonction de hachage primaire :

$$hi(x) = h1(x) + (i - 1).$$

Le hachage linéaire fonctionne de la façon suivante : Lorsqu'il y a collision sur une case d'indice i, on essaie la case d'indice i+1. Si l'on est sur la m^{ième} case (la dernière), on repasse à la 1^{ière}. Appelons ***Modulo plus*** cette progression circulaire et notons-la $a \oplus b$.

A l'aide d'une fonction de hachage uniforme ***h***, la suite d'essais successifs serait :

$$hi\ (x) = h1(x) \oplus m\ (i - 1)$$

IV.2.b Double Hachage

Un des problèmes soulevés par le hachage linéaire est la formation de groupements d'éléments. La probabilité qu'un groupement de ***k*** éléments sur un tableau de taille ***m*** voit sa taille augmenter au prochain ajout est de $\frac{k+2}{m}$

Si collision à la position i, on essaie k × i + 1 mod m (avec k nombre fixe), Ceci pour éviter les regroupements aux alentours de i + 1 mod m.

MAIS regroupement quand même

- Il faut que k dépende de e.
- Calcul de k par une seconde fonction de hachage h'(e) (d'où le double hachage)

- Si N premier, il suffit que h′(e) → [1..N − 1].
- Si N = 2p , il suffit que h′(e) → indice impair.

Partie A

Chapitre 07 : Les Graphes

V. Définition

Un graphe définit une relation binaire sur un ensemble d'éléments. Plus précisément, un graphe est défini par un couple G = (S, A) tel que S est un ensemble fini de sommets (aussi appelés nœuds), et A ⊆ S × S est un ensemble de couples de sommets définissant une relation binaire sur S qu'on nomme arêtes(ou arc dans un graphe orienté).L'ordre d'un graphe est le nombre de ses sommets. Un graphe peut être orienté ou non, selon que la relation binaire est asymétrique ou symétrique.

I.1 Définition Graphe Orienté

Un graphe orienté G = (S;A) :

• S est un ensemble fini S d'éléments appelé Sommets

• A ⊆ S×S dont les éléments sont appelés arcs.

• Un arc (x ; y) représente une liaison orientée entre l'origine x et l'extrémité y.

• Si (x; y) est un arc, y est un successeur de x, x est un prédécesseur de y.

• Si x = y l'arc (x; x) est appelé boucle.

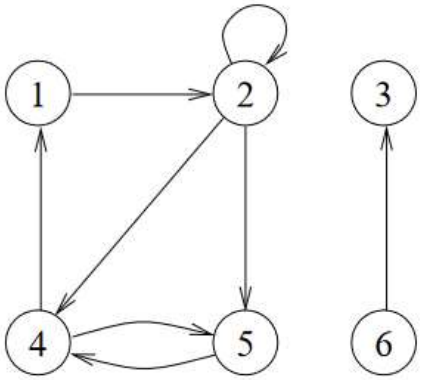


Figure 43 : *Exemple de graphe orienté*

I.2 Définition Graphe non Orienté

Un graphe non oriente G = (S;A) :

• S est un ensemble fini d'éléments appelés sommets

• A une famille de paires de S dont les éléments sont appelés arêtes.

• Etant donne un graphe oriente, sa version non orientée est obtenue en supprimant les boucles et en substituant à chaque arc restant (x; y) la paire {x; y}.

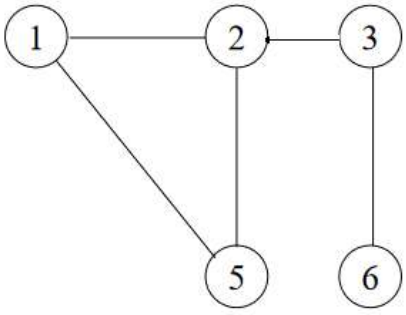


Figure 44 : *Exemple de graphe non orienté*

VI. Utilisation des graphes

Les graphes permettent de modéliser plusieurs problèmes :

- Problème d'ordonnancement de tâches, S= l'ensemble des tâches, A= l'ordre de précédence ou de succession.
- Réseau d'ordinateurs, S c'est les ordinateurs, A la connexion entre ces ordinateurs.
- Réseau social, S c'est des individus et A c'est leurs liaisons (mariages, amitié, client etc).
- Cartographie, routes, S représente les villes, A c'est les routes qui peuvent exister entre ces villes.
- Réseau d'articles dans un magasin, S c'est les articles, A c'est une liason .
- Gestion des réseaux de communication
- Plus généralement, les graphes interviennent chaque fois que l'on veut représenter et étudier un ensemble de liaisons (orientées ou non) entre les éléments d'un ensemble ni d'objets.

Exemple

La Figure 43 est une représentation graphique du graphe orienté G = (S, A) avec l'ensemble de sommets S = {1, 2, 3, 4, 5, 6} et l'ensemble d'arcs A = {(1, 2), (2, 2), (2, 4), (2, 5), (4, 1), (4, 5), (5, 4), (6, 3)} ; les sommets étant représentés par des cercles et les arcs par des flèches. On notera que les boucles — une boucle étant un arc qui relie un sommet à lui-même — sont ici possibles.

VII. Représentation

En algorithmique, un graphe peut être représenté selon deux stratégies c'est-à-dire deux façons classiques de représenter un graphe.

- Par un ensemble de listes d'adjacences (Dynamique).
- Par une matrice d'adjacences (Statique).

III.1 Liste d'adjacence

On peut représenter un graphe en utilisant : Une liste qui contient les sommets du graphe. Chaque élément de cette liste peut contenir un lien vers la liste de ses voisins (liste adjacente). Stratégie dynamique avec une liste contenant les adresses des voisins. La représentation par listes d'adjacences est souvent préférée, car elle fournit un moyen peu encombrant de représenter les graphes peu denses Liste d'adjacence.

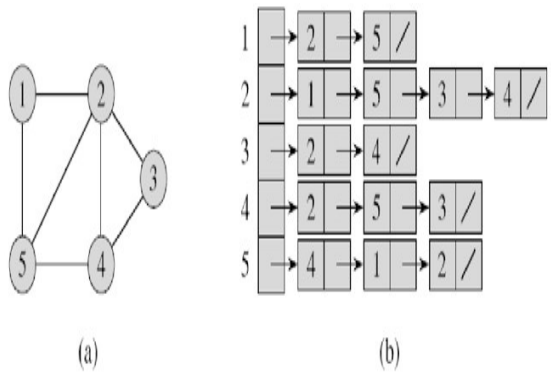


Figure 45.a : *Liste d'adjacence (Graphe non orienté)*

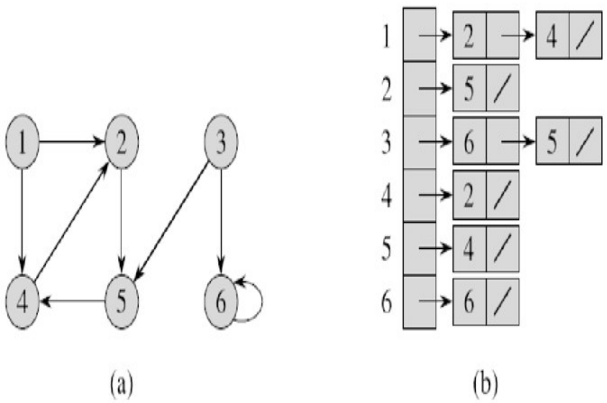


Figure 45.b : *Liste d'adjacence (Graphe orienté)*

```
type struct Sommet {
    char val ;
    int visiter ;
    Struct Sommet *suivant ;
 } Sommet ;
struct Sommet *Graphe ;
```

III.2 Matrice d'adjacences

On stocke les sommets dans un tableau. On stocke les arêtes dans une matrice, dont chaque cellule [i,j], représente une arête (ou un arc)entre les sommets d'indice i et j. On déclare un enregistrement qui représente un sommet du graphe comme décrit dans le listing suivant,

```
struct Sommet
{
char val ;
int visité ;
}
Sommet  S[N] ;
int A[N][N] ;
```

VIII. Parcours de graphes (en profondeur et largeur)

Le parcours des graphes se révèle être un peu plus compliqué que celui des arbres. En effet, les graphes peuvent contenir des cycles et nous voulons éviter de parcourir indéfiniment ces cycles !

Pour remédier au problème cité, les algorithmes de parcours de graphes maintiennent en général un bit de marquage pour chaque sommet du graphe. Initialement, le bit est à 0 pour tous les sommets du graphe. Le bit de marquage est mis à 1 quand un sommet est visité pendant le parcours. Si un sommet marqué est rencontré à nouveau pendant le parcours, il n'est pas visité une deuxième fois. Ceci permet d'éviter les cycles. Une fois l'algorithme de parcours terminé, on peut vérifier si tous les sommets ont été visités en consultant le tableau de marquage. La stratégie de parcours d'un graphe est alors comme suit : Nous allons discuter, dans ce qui suit, deux procédures : parcours en largeur d'abord et parcours en profondeur d'abord.

IV.1 Parcours en profondeur

Dans le cas du parcours en largeur on "découvrait" tous les sommets situés à une distance k de l'origine avant de s'intéresser aux sommets situés à une distance k+1 de l'origine. Dans le cas du parcours en profondeur, on va chercher à aller "le plus loin possible" dans le graphe : A -> B -> C -> E -> I -> D, quand on tombe sur "un cul-de-sac" (dans notre Figure 46, D est un "cul-de-sac", car une fois en D, on peut uniquement aller en B, or, B a déjà été découvert...), on revient "en arrière" (dans notre exemple, on repart de B pour aller explorer une autre branche : G -> F -> H).

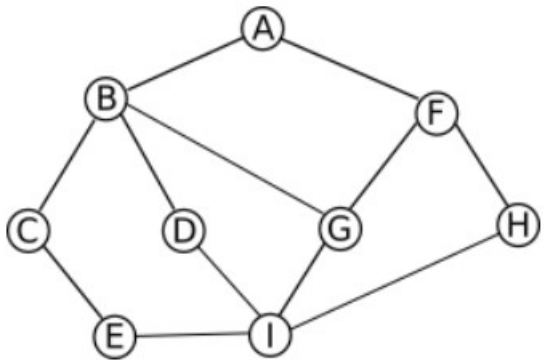


Figure 46 : *Exemple de graphe*

La stratégie de cette méthode est comme suit :

On commence par v qu'on marque comme visitée. Ensuite, un sommet adjacent w adjacent à v est choisi et un parcours en profondeur d'abord est effectué à partir de w. Quand un sommet u est atteint tel que tous les sommets qu'ils lui sont adjacents sont visités, alors on choisit un sommet adjacent du dernier sommet visité et on recommence la procédure jusqu'à ne plus avoir de sommets adjacents non visité. En terme algorithmiques, on obtient ce qui suit :

```
void ParcoursProfondeur(int v) // v étant le sommet de départ
{
pour i allant de 1 à n faire
initialiser marquer [i] = 0;
marquer [v] = 1
Pour chaque sommet w adjacent à v faire
si marquer [w] = 0 alors ParcoursProfondeur(w)
finpour }
```

À noter que l'utilisation d'un algorithme récursif n'est pas une obligation pour le parcours en profondeur : on va ici retrouver le même système de couleur (blanc : sommet non visité, noir : sommet déjà visité)

```
VARIABLE
s : noeud (origine)
G : un graphe
u : noeud
v : noeud
p : pile (pile vide au départ)
//On part du principe que pour tout sommet u du graphe G,
u.couleur = blanc à l'origine
DEBUT
s.couleur ← noir
piler(s,p)
tant que p n'est pas vide :
  u ← depiler(p)
  pour chaque sommet v adjacent au sommet u :
    si v.couleur n'est pas noir :
      v.couleur ← noir
      piler(v,p)
    fin si
  fin pour
```

```
fin tant que
FIN
```

IV.2 Parcours en largeur

Nous allons travailler sur un graphe G(V,E) avec V l'ensemble des sommets de ce graphe et E l'ensemble des arêtes de ce graphe. Un sommet u sera adjacent avec un sommet v si u et v sont reliés par une arête (on pourra aussi dire que u et v sont voisins). La stratégie de cette exploration est comme suit : on commence au sommet v et le marquer comme étant visité. L'ensemble des sommets adjacents à v sont visités. Ensuite, les sommets non visités de ces éléments sont à leur tour visités, et ainsi de suite, jusqu'à visiter tous les sommets.

```
void Parcourslargeur (int debut) // debut étant le sommet de
départ
{enfiler (debut) ;
marquer[debut] = 1;
tant que (file non vide)
{ s = defiler () ;
// traitement nécessaire sur le sommet s
pour tout w adjacent à s faire
si (Marque[w] == 0 )
{
Marquer[w] = 1;
enfiler (w);
}
}}
```

Dans notre Figure 46 le parcours en largeur donne les sommets dans l'ordre suivant : A, B, F, C, D, G, H, E et I (ce n'est pas la seule solution possible) .

Partie A

Chapitre 08 : Récursivité

II. Introduction

La notion de récursivité est avant tout un problème algorithmique plus qu'au niveau du langage lui-même. Que ce soit en C, C++, Java, VB, Python, etc.., l'implémentation d'une fonction récursive se fera toujours plus ou moins de la même manière. Ici nous allons traiter de la récursivité avec le Langage C, telle est notre rubrique !

Mais qu'est-ce que la récursivité ? Et bien en fait d'un point de vue théorique cela reste assez simple; il s'agit de procédure ou de fonctions d'un programme qui ont la faculté de s'appeler eux-mêmes (on entend également le terme d'auto appel ce qui est logique). La récursivité est une manière simple et élégante de résoudre certains problèmes algorithmiques, notamment en mathématique, mais cela ne s'improvise pas, il convient donc de savoir comment ce principe fonctionne.

II. Définition

La programmation récursive est une technique de programmation qui remplace les instructions de boucles (while, for, etc.) par des appels de fonction. L'idée est de diviser un problème P en sous-problèmes (certains de la même forme), de résoudre ces sous-problèmes de manière directe si c'est possible sinon de la même manière que P (le problème initial), puis de combiner le résultat de ces sous-problèmes pour résoudre le problème initial P. Autrement dit, on traite le problème P en le ramenant à un problème de la même forme mais plus petit.

Comme pour les définitions récursives, les fonctions récursives sont des fonctions qui sont appelées depuis leur propre corps de fonction, soit directement soit indirectement, à travers une ou plusieurs fonctions relais. Si la fonction P appelle directement P, on dit que la récursivité est directe. Si P appelle une fonction P1, qui appelle une fonction P2 et qui enfin appelle P, on dit qu'il s'agit d'une récursivité indirecte.

Si la fonction P appelle dans son propre corps la fonction P, il est logique de penser que l'exécution ne s'arrêtera jamais. Il est donc primordial qu'une des branches de la fonction P permette de stopper la récursivité.

Un algorithme est dit, récursif, s'il fait un ou plusieurs appels à soit même. Il est défini en fonction d'un objet ou d'une action de même nature mais de complexité moindre.

Exemple 01 :

On veut calculer la somme des n premiers entiers positifs. Une définition récursive de cette somme serait :

$$\begin{cases} Somme(n)=n + somme\ (n-1) \\ Somme(1)=1 \end{cases}$$

Ce qui se traduit par la fonction récursive suivante :

```
Fonction somme (n : entier) : entier
Var
   m : entier
Debut
     Si (n=1) alors
        Somme<-1
     Sinon
       m<-somme (n-1)
       Somme<-n+m

Finsi

Fin
```

Remarque : Il est clair que sans le test « si (n=1) »cette fonction ne s'arrêterait jamais.

Exemple 02 :

Par exemple, on peut définir le factoriel d'un nombre « n »positif de deux manières :

- *Définition non récursive(ou itérative)*

  N !=n*n-1*n-2*……2*1

- *Définition récursive :*

  n !=n*(n-1) !

  0 !=1

On remarque qu'une définition récursive est composée de deux parties :

- Une partie strictement récursive et
- Une partie non récursive (base) servant de point d'arrêt d'utilisation de la définition récursive.

III. Type de récursivité

D'une façon générale on peut identifier deux types de récursivité :

III.1. Récursivité simple ou linéaire

Un algorithme récursif est *simple* ou *linéaire* si lorsque un sous-programme (procédure, fonction) s'appelle lui-même, si on effet le cas générale de récursivité comme on a déjà vu dans les exemples précédent.

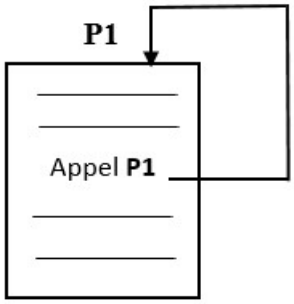


Figure 47 *: Schéma explicatif de la récursivité simple*

III.2. Récursivité croisée ou mutuelle

La définition des algorithmes récursifs donnée plus haut qui les caractérise comme étant les algorithmes faisant appel à eux-mêmes masque le phénomène des algorithmes mutuellement récursifs. Deux algorithmes sont *mutuellement récursifs (croisée)* si l'un fait appel à l'autre et l'autre à l'un. On peut étendre cette définition à un nombre quelconque d'algorithmes.

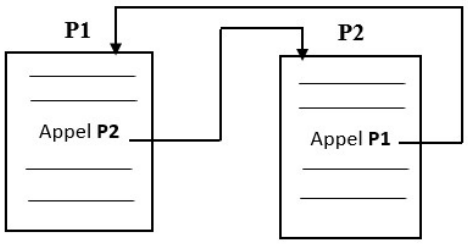


Figure 48 *: Schéma explicatif de la récursivité croisée*

Exemple 03

En voici dans l'exemple suivant deux fonctions nommées pair et impair déterminant la parité ou l'imparité d'un entier.

Prédicat de parité (pair)

Entrée : $n \in \mathbb{N}$
Sortie : *Vrai* si *n* est pair, *Faux* sinon
si $n=0$
Renvoyer *Vrai*
sinon
Renvoyer impair($n-1$)
fin si

Prédicat d'imparité (impair)

Entrée : $n \in \mathbb{N}$
Sortie : *Vrai* si *n* est impair, *Faux* sinon
si $n=0$
Renvoyer *Faux*
sinon
Renvoyer pair($n-1$)
fin si

IV. Conception d'algorithmes récursifs

Pour construire l'algorithme récursif, il faut suivre les étapes suivantes :

- Décomposer le problème en un problème plus simple, réduire la taille du problème considéré.

- Pour la récursivité sur des entiers : la taille du problème est définie par un entier, on réduit la valeur de cet entier à chaque appel récursif.

- Pour la récursivité sur les tableaux : soit on considère la taille du tableau, on réduit la taille du tableau considéré à chaque appel récursif ou bien on utilise un ou des indices qui varient à chaque appel pour tendre vers la condition d'arrêt (dépendant des valeurs des indices)

V. Construction de la récursivité

Une fonction récursive est définie par : au moins un cas de base et, au moins un cas général.

Cas de base : on décrit les cas pour lesquels le résultat de la fonction est simple à calculer ; la valeur retournée par la fonction est directement définie.

Cas général : la fonction est appelée récursivement et le résultat retourné est calculé en utilisant le résultat de l'appel récursif. A chaque appel récursif, la valeur d'au moins un des paramètres (effectifs) de la fonction doit changer.

Attention : Il faut toujours s'assurer que chaque cas général converge vers un cas de base.

VI. Déroulement d'un algorithme récursif

Lorsqu'on lance l'exécution d'un programme, le système d'exploitation crée une zone mémoire pour ce programme, dans cette zone on stocke les variables, il lui affecte aussi une pile, qu'on appelle pile d'exécution. Dans cette pile on stocke les appels ainsi que les paramètres associés à ces appels. Les appels récursifs dans un algorithme sont stockés dans cette pile, jusqu'à ce que la condition d'arrêt soit atteinte. Si la condition n'est jamais réalisée, le système déclenche une erreur de débordement de la pile, car il empile les appels sans fin.

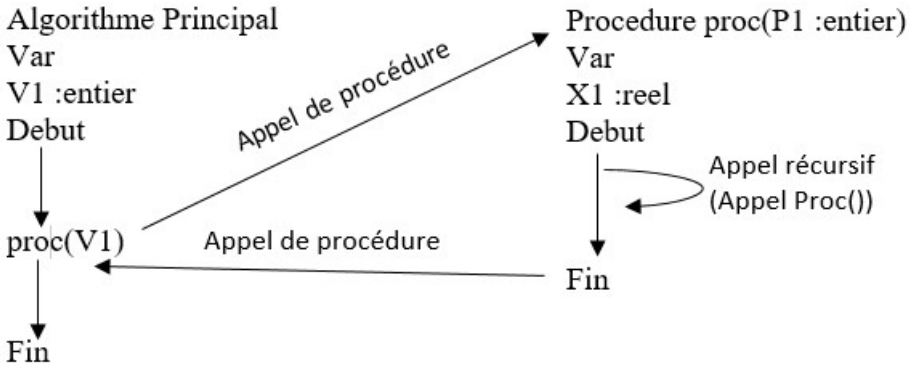


Figure 49 *: Schéma explicatif des appels récursif*

Factorielle

La fonction factorielle qui calcule le produit des n premiers entiers positifs $(n!=\prod_{k=1}^{n} k)$ est simplement définie par la relation de récurrence :

Fonction factorielle $\begin{cases} 0\,!=1 \\ n*(n-1)\,! \;\forall\; n\in N^{*} \end{cases}$

VI.1 Version itérative

```
Fonction FactorielleIteratif(n: entier):entier
Debut
Var i,k,res: entiers
res   n
Si(n=0 ou n=1)alors
retourner 1
Sinon
Pour(i de 1 `a n)
res   res*(n-i)
retourner res
Fin
```



```
int FactorielleIterative(int n)
{
int i,k,res;


res=n;
if(n==0jjn==1)
return 1;
else
for(i=1;i<n;i++)
res=res*(n-i);
return res;}
```

On retrouve ici la version itérative bien connue.

VI.2 Version récursive

```
Fonction FactorielleRecursif(n: entier):entier
Debut
Si(n=0 ou n=1)alors
   retourner 1;
Sinon
   retourner n*FactorielleRecursif(n-1);

Fin
```



```
int FactorielleRecursive(int n)
{
if(n==0 jj n==1)
   return 1;
else
   return n*FactorielleRecursive(n-1);

}
```

Cette version est dite « récursive » parce que dans le corps de la fonction factorielle, on appelle la fonction factorielle elle-même. Cette version récursive est la traduction directe de la formulation mathématique.

Dans la version récursive, le processus nécessite que l'interpréteur garde une trace des multiplications à réaliser plus tard. Le processus croît puis décroît : la croissance se produit lorsque le processus construit une chaîne d'opérations différées (ici, une chaîne de multiplications différées) et la décroissance intervient lorsqu'on peut évaluer les multiplications. Ainsi, la quantité d'information qu'il faut mémoriser pour effectuer plus tard les opérations différées croît linéairement avec *n* on parle de processus récursif linéaire.

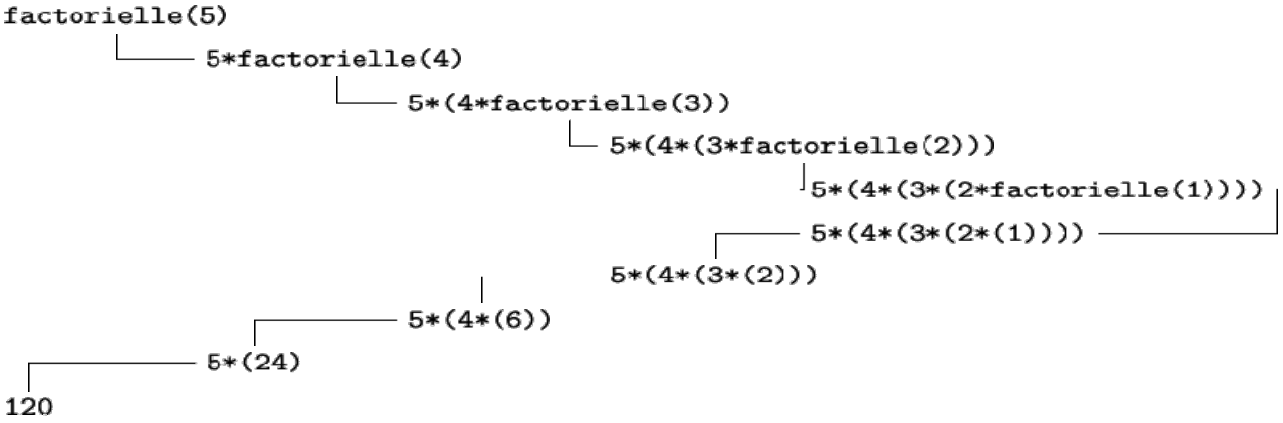


Figure 50 *: Récursivité-factorielle*

L'interprétation d'une fonction récursive passe donc par une phase d'expansion dans lesquels les appels récursifs sont « empilés » jusqu'à arriver à un appel de la fonction pour lequel une condition d'arrêt sera vérifiée, puis par une phase de contraction dans laquelle les résultats des appels précédemment empilés sont utilisés. Par contre, dans la version itérative, le processus de calcul ne croît ni ne décroît : à chaque étape, seules les valeurs courantes des variables u et i sont nécessaires (il n'y a pas d'opérations différées) et le temps requis pour calculer $n!$ augmente linéairement avec n (on passe $(n-2)$ fois dans la boucle).

VII. **Conclusion**

Certains problèmes peuvent être résolus plus logiquement en utilisant la récursivité. Les programmes sont plus compacts, plus faciles à écrire et à comprendre. Son usage est naturel quand le problème à traiter peut se décomposer en deux ou plusieurs sous-problèmes identiques au problème initial mais avec des valeurs de paramètres différentes. Refuser la récursivité dans ce dernier cas oblige l'utilisateur à gérer lui-même une pile des différentes valeurs des variables, ce que le système fait automatiquement lors de l'utilisation de la récursivité.

Partie B
Chapitre 01 : Les Fichiers

II. Raisons de l'utilisation des mémoires secondaires

Les variables n'existent que dans la mémoire vive. Une fois votre programme est arrête, toutes vos variables sont Supprimées de la mémoire et il n'est plus possible de retrouver ensuite leur valeurs. Comment faire pour pouvoir continuer votre travaille, la prochaine fois, son perdre vos données ?

Solution : lire et écrire dans des fichiers. Ces fichiers seront écrits sur la mémoire secondaire(le disque dur, des clés USB…) : l'avantage est donc qu'ils sont enregistrés, même si vous arrêtez le programme ou l'ordinateur.

II. Définition

Un fichier (en anglais : file) est un ensemble de données stockées en général sur un support externe (disque dur, disquette, CD-ROM, bande magnétique, …). Un fichier structuré contient une suite d'enregistrements homogènes, qui regroupent le plus souvent plusieurs composantes appartenant à un ensemble (champs).

En C, Un fichier est en général considéré comme une suite d'octets qu'on peut gérer par un ensemble de fonctions prédéfinies, Les informations contenues dans le fichier ne sont pas forcément de même type (un char, un int, une structure ...). Les fonctions prédéfinies sont caractérisées par l'usage d'un tampon mémoire propre au programme (à l'intérieur de son segment de données) pour le stockage temporaire de données. Elles ne font appel au système que pour transmettre les données en mémoire secondaire lorsque le tampon est totalement rempli. Ces fonctions sont prédéfinies dans la bibliothèque **stdio.h**

III. Fichiers binaires et Fichiers textes

Il existe, en général, deux types de fichiers :

- *Les fichiers textes :* sont constitués d'une suite de caractères, qu'ils sont considérés comme une collection de lignes de texte. Lorsqu'il s'agit de ce type de fichiers, le système réagit à certains caractères comme le caractère de fin de ligne ou de fin de texte.
- *Les fichiers binaires :* sont des suites d'octets qui peuvent faire l'objet, par exemple, de collections d'entités constituées d'un nombre précis d'octets (enregistrements ou structures). Dans ce cas, le système n'attribue aucune signification aux caractères transférés depuis ou vers le périphérique.

	Fichiers textes	**Fichiers binaires**
Taille	Peu compacte	Compacte
Programme courant	Oui	Non
Lisible avec un programme spécifique	Oui	Oui
Ecriture/Lecture par blocs	Non	Oui
Fonctions	fopen(mode r,w ou a) fclose() fprint() fscanf()	fopen(mode rb,wb ou ab) fclose() fwrite() fread()

Tableau 11 : *Comparaison entre fichiers binaires et fichiers textes*

IV. Déclaration de fichiers

Pour pouvoir manipuler un fichier, un programme a besoin d'un certain nombre d'informations : l'adresse de l'endroit de la mémoire-tampon où se trouve le fichier, la position

de la tête de lecture, le mode d'accès au fichier (lecture ou écriture) ... Un fichier possède un nom physique (nom externe ou nom Dos de type chaîne de caractères) et un nom logique (nom interne) de type pointeur sur FILE (FILE*), est défini dans stdio.h (C'est à dire mettre un `#include<stdio.h>` au début du programme). Attention aux majuscules pour 'FILE'. Contrairement à Turbo Pascal, le C fait la distinction entre minuscules et majuscules.

Syntaxe :

```
FILE *nom_interne_fichier ;
```

Exemple :

```
FILE *fiche_source;     /*fiche_source est déclaré comme pointeur sur FILE. Il pourrait
                          alors faire l'objet de nom logique d'un document fichier.*/
```

V. Fichiers à accès séquentiel et fichier à accès direct

On distingue deux techniques d'accès aux supports magnétiques :

- L'accès séquentiel qui consiste à traiter les informations dans l'ordre où elles apparaissent sur le support (bandes). Le lecteur physique avance avec la lecture, et se positionne sur le début de l'enregistrement suivant.
- L'accès direct qui consiste à se placer directement sur l'information sans parcourir celles qui la précèdent (disques). Le lecteur physique reste sur le même enregistrement après une lecture.

En langage C, l'accès est séquentiel mais il est possible de déplacer le "pointeur de fichier" c'est à dire sélectionner l'indice du prochain octet à lire ou écrire.

Comme nous venons de le voir dans les modes d'ouverture, le pointeur de fichier peut être initialement placé en début ou fin de fichier.

Les quatre fonctions d'entrée-sortie (`fgetc`, `fputc`, `fscanf`, `fprintf`) travaillent séquentiellement à partir de cette origine fixée par `fopen`.

V.1 Fichiers en organisation séquentielle

Les enregistrements forment une séquence, c'est à dire qu'ils sont arrangés de manière adjacente les uns à la suite des autres. Sur la plupart des ordinateurs l'entrée sur clavier et la sortie sur l'écran sont assimilés à des fichiers séquentiels.

Remarque : L'organisation séquentielle convient pour les deux types de fichiers (textes et binaires)

V.2 Fichiers en organisation relative (ou fichiers à accès direct)

Les enregistrements sont identifiés par un numéro d'ordre (relatif par rapport au début du fichier). L'enregistrement de numéro d'ordre n occupe la n^{ème} position dans le fichier.

Deux type d'accès possible :

- Séquentiel (idem fichiers séquentiels)
- Direct, à condition de connaître le numéro d'ordre de l'enregistrement.

Avantages accès direct :

- Rapidité de recherche
- Lecture et écriture n'importe où dans le fichier

Remarque : Les fichiers relatifs sont de type texte ou binaire (mais par utilisation limitée de l'accès direct pour les fichiers de textes.)

VI. Fichiers physique et fichiers logique

Le système d'exploitation offre à l'utilisateur une unité de stockage indépendante des propriétés physiques des supports de conservation : le fichier. Ce concept de fichier recouvre deux niveaux. D'une part, le *fichier logique* représente l'ensemble des données incluses dans le fichier telles qu'elles sont vues par l'utilisateur. D'autre part, le *fichier physique* représente le fichier tel qu'il est alloué physiquement sur le support de masse. Le système d'exploitation gère ces deux niveaux de fichiers et assure notamment la correspondance entre eux, en utilisant une structure de répertoire.

- *Le fichier logique* : correspond à la vue que l'utilisateur de la machine a de la conservation de ses données. Plus précisément, le fichier logique est, d'une part, un type de données standard défini dans les langages de programmation, sur lequel un certain nombre d'opérations spécifiques peuvent être réalisées et d'autre part, un ensemble d'enregistrements ou articles. Un enregistrement est un type de données regroupant des données de type divers liées entre elles par une certaine sémantique inhérente au programme qui les manipule.
- *Le fichier physique* : correspond à l'entité allouée sur le support permanent et contient physiquement les enregistrements définis dans le fichier logique. Les enregistrements composant le fichier logique doivent être écrits dans les secteurs composant les blocs du disque, pour former ainsi le fichier physique correspondant au fichier logique. Le système de gestion de fichiers effectue la correspondance entre fichiers logiques et fichiers physiques par le biais d'une table appelée répertoire qui contient des informations

de gestion des fichiers, dont notamment, pour chaque fichier existant sur le disque, le nom logique du fichier et son adresse physique sur le disque.

VII. Fonctions sur les fichiers textes

Un fichier est déclaré par son nom logique (nom interne). Pour manipuler les données d'un fichier (en lecture ou en écriture), on notifie son accès par la commande `fopen`. Cette fonction prend comme argument le nom physique du fichier (nom externe), négocie avec le système d'exploitation et initialise un flot de données, qui sera ensuite utilisé lors de l'écriture ou de la lecture. Après les traitements, on annule la liaison entre le fichier et le flot de données grâce à la fonction `fclose`.

VII.1 Ouverture des fichiers (La fonction `fopen`)

Cette fonction, de type FILE* ouvre un fichier et lui associe un flot de données.

Syntaxe :

```
nom_interne_fichier=fopen("nom_externe_fichier","modes d'ouverture de fichier" );
```

La fonction `fopen` ouvre le fichier physique (fichier Dos) et retourne un pointeur sur `FILE`, si l'exécution de cette fonction ne se déroule pas normalement, la valeur retournée est le pointeur `NULL`. Il est donc recommandé de toujours tester si la valeur renvoyée par la fonction `fopen` est égale à `NULL` afin de détecter les erreurs (lecture d'un fichier inexistant...). Le paramètre modes d'ouverture de fichier permet d'indiquer le mode de fichier texte et le mode d'ouverture (création, lecture, écriture ou mise à jour). Il s'agit des paramètres :

w Ouvre un fichier texte en écriture. Il le crée s'il n'existe pas et l'écrase s'il existe déjà.
r Ouvre un fichier texte en lecture seule.
a Ouvre un fichier texte en écriture, positionnement à la fin (compléter).Si le fichier n'existait pas, il est créé.
r+ Ouvre un fichier texte en lecture/écriture (modifier des éléments).
w+ Crée un fichier avec accès en lecture et écriture.
a+ Ouvre un fichier texte en lecture/écriture, écriture à partir de la fin.

Remarque :

- Lorsqu'on ouvre un fichier, son pointeur pointe sur son premier élément.

- On peut ouvrir un fichier en deux façons :
    1. Avec le *chemin relatif* au « répertoire courant », En général, au début, c'est le répertoire contenant l'exécutable.
    2. Avec le *chemin absolu*, On indique tous les répertoires depuis la racine mais attention à la syntaxe du langage C : doubler les \\.

Exemple

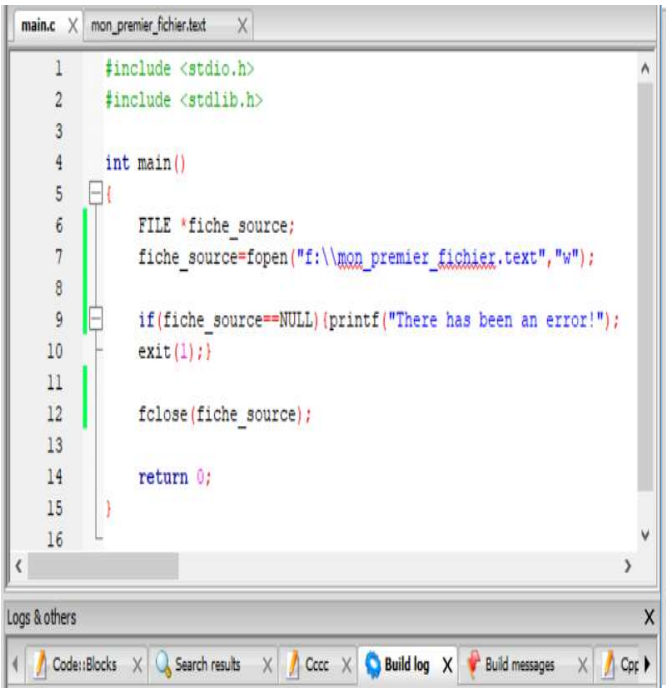


Figure 51 : *Ouverture d'un fichier avec chemin absolu*

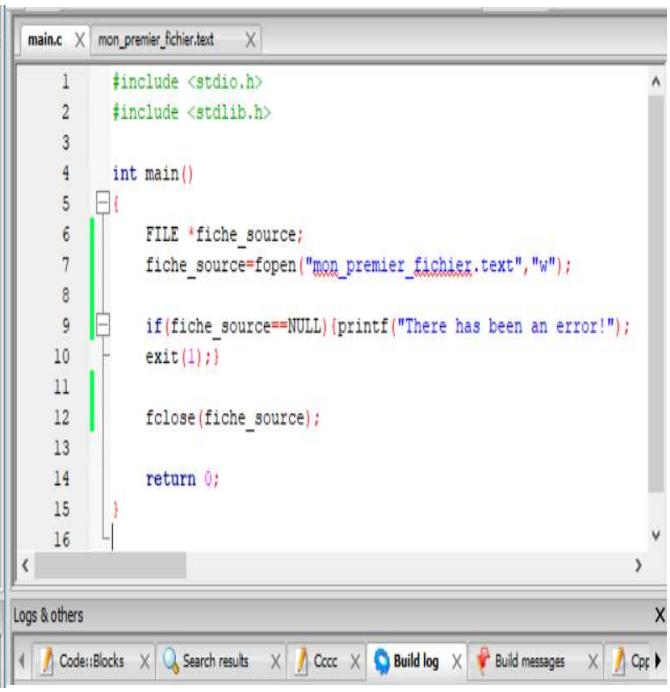


Figure 52 : *Ouverture d'un fichier avec chemin relatif*

VII.2 Fermeture de fichiers (la fonction `fclose`)

Elle permet de fermer le flot qui a été associé à un fichier par la fonction `fopen`.

Syntaxe :

```
fclose(nom_fichier_interne);
```

La fonction `fclose` ferme le fichier dont le nom interne est indiquée en paramètre. Elle retourne 0 si l'opération s'est bien déroulée, -1 en cas d'erreurs.

Il est indispensable de fermer un fichier après utilisation, pour au moins deux raisons :

- Les écritures dans un fichier ne sont pas immédiates, elles sont regroupées en gros blocs, pour des raisons d'efficacité, donc il reste souvent des données en attente d'écriture, et la fermeture du fichier déclenche l'écriture de ces données

- Il y a souvent des limites au nombre de fichiers qu'un programme peut avoir ouvert simultanément.

VII.3 Lecture de fichiers (les fonctions `fgetc, fgets, fscanf`)

Il y'a plusieurs fonctions de lecture dans un fichier texte :

- Lecture d'un caractère (`fgetc`)

```
C=fgetc(nom_fichier_interne) ;
```

Exemple

*main.c

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

int main()
{
    FILE* fichier = NULL;
    int caractereActuel = 0;

    fichier = fopen("test.txt", "r");

    if (fichier != NULL)
    {
        // Boucle de lecture des caractères un à un
        do
        {
            caractereActuel = fgetc(fichier);
            printf("%c", caractereActuel);
        } while (caractereActuel != EOF);

        fclose(fichier);
    }

    return 0;
}
```

Figure 53 : *Lecture d'un fichier avec la fonction fgetc.*

- Lecture d'une ligne (`fgets`)

```
fgets(buffer, max_count, nom_fichier_interne);
```

• buffer : un buffer permettant de recueillir la chaîne à lire.

• max_count : permet de spécifier le nombre de caractères maximal pouvant être utilisé au sein du buffer (uniquement dans le cas d'une utilisation de la fonction fgets).

• nom_fichier_interne : ce paramètre permet d'indiquer le fichier à utiliser pour la lecture.

Exemple :

*main.c

```c
#include <stdio.h>
#include <stdlib.h>
#define TAILLE_MAX 1000 // Tableau de taille 1000

int main(int argc, char *argv[])
{
    FILE* fichier = NULL;
    char chaine[TAILLE_MAX] = ""; // Chaine vide de taille TAILLE_MAX

    fichier = fopen("test.txt", "r");

    if (fichier != NULL)
    {
        fgets(chaine, TAILLE_MAX, fichier);
        printf("%s", chaine); // On affiche la chaine

        fclose(fichier);
    }

    return 0;
}
```

Figure 54 : *Lecture d'un fichier avec la fonction fgets.*

- Lecture d'une information (`fscanf`)

```
fscanf (nom_fichier_interne, % format, &buffer);
```

Lira dans le fichier pointé par nom_fichier_interne la valeur selon le format (%format) qui sera stocké à l'adresse de buffer (&buffer).elle fait la lecture selon une spécification de format, elle fonctionne comme scanf, mais lit depuis un fichier.

Exemple :

main.c

```c
#include <stdio.h>
#include <stdlib.h>
int main()
{
    FILE* fichier = NULL;
    int score[3] = {0}; // Tableau des 3 meilleurs scores

    fichier = fopen("test.txt", "r");

    if (fichier != NULL)
    {
        fscanf(fichier, "%d %d %d", &score[0], &score[1], &score[2]);
        printf("Les meilleurs scores sont : %d, %d et %d", score[0], score[1], score[2]);

        fclose(fichier);
    }

    return 0;
}
```

Figure 55 : *lecture d'un fichier avec la fonction fscanf.*

VII.4 Ecriture dans un fichier

Il existe plusieurs fonctions capables d'écrire dans un fichier texte :

- Ecriture d'un caractère (`fputc`)

```
fputc(C, nom_interne_fichier);
```

• Ecrira le caractère C dans le fichier pointé par nom_interne_fichier.

- Le caractère à écrire (de type int, ce qui comme je vous l'ai dit revient plus ou moins à utiliser un char, sauf que le nombre de caractères utilisables est ici plus grand). Vous pouvez donc écrire directement 'C'par exemple.
- Le pointeur sur le fichier dans lequel écrire. L'avantage de demander le pointeur de fichier à chaque fois, c'est que vous pouvez ouvrir plusieurs fichiers en même temps et donc lire et écrire dans chacun de ces fichiers. Vous n'êtes pas limités à un seul fichier ouvert à la fois.

Exemple :

main.c

```c
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[])
{
    FILE* fichier = NULL;

    fichier = fopen("test.txt", "w");

    if (fichier != NULL)
    {
        fputc('A', fichier); // Écriture du caractère A
        fclose(fichier);
    }

    return 0;
}
```

Figure 56 : *écriture d'un fichier avec la fonction fputc.*

- Ecriture d'une ligne de texte et un saut de ligne (`fputs`)

```
fputs(''texte'', nom_interne_fichier);
```

• Retourne une valeur non négative en cas de succès, EOF si échec.

• Ecrit une ligne et un saut de ligne dans le fichier.

Exemple :

main.c

```c
#include <stdio.h>
#include <stdlib.h>
int main()
{
    FILE* fichier = NULL;

    fichier = fopen("test.txt", "w");

    if (fichier != NULL)
    {
        fputs("Salut mes étudiants\n Comment allez-vous ?", fichier);
        fclose(fichier);
    }

    return 0;
}
```

Figure 57 : *Lecture d'un fichier avec la fonction fputs.*

- Ecriture selon une spécification de format (`fprintf`)

```
fprintf(nom_interne_fichier, format, ...);
```

• Fonctionne comme printf, mais écrit dans un fichier, d'ailleurs, excepté le fait que vous devez indiquer un pointeur de FILE en premier paramètre.

Exemple :

main.c

```c
#include <stdio.h>
#include <stdlib.h>
int main()
{
    FILE* fichier = NULL;
    int age = 0;

    fichier = fopen("test.txt", "w");

    if (fichier != NULL)
    {
        // On demande l'âge
        printf("Quel age avez-vous ? ");
        scanf("%d", &age);
        // On l'écrit dans le fichier
        fprintf(fichier, "L'étudiant qui utilise le programme, il a %d ans", age);
        fclose(fichier);
    }
    return 0;
}
```

Figure 58 : *écriture d'un fichier avec la fonction fprintf.*

VII.5 Détection de fin de fichier (`feof`)

Cette fonction permet de déterminer s'il reste (ou non) des octets à lire dans le flux.

Syntaxe :

```
feof (nom_fichier_interne)
```

La fonction feof retourne 0 si la fin de fichier n'a pas été détectée, une valeur différente de 0 sinon. En fait, elle n'indique la fin de fichier qu'après une lecture effective qui n'aboutit pas.

VII.6 Accès aux éléments d'un fichier

Chaque fois que vous ouvrez un fichier, il existe en effet un curseur qui indique votre position dans le fichier. Vous pouvez imaginer que c'est exactement comme le curseur de votre éditeur de texte (tel Bloc-Notes). Il indique où vous êtes dans le fichier, et donc où vous allez écrire.

En résumé, le système de curseur vous permet d'aller lire et écrire à une position précise dans le fichier. Existe trois fonctions à connaître :

- `ftell` : indique à quelle position vous êtes actuellement dans le fichier ;
- `fseek` : positionne le curseur à un endroit précis ;
- `rewind` : remet le curseur au début du fichier (c'est équivalent à demander à la fonction fseek de positionner le curseur au début).

➢ `ftell` : position dans le fichier

Cette fonction est très simple à utiliser. Elle renvoie la position actuelle du curseur sous la forme d'un long :

```
A=ftell(nom_interne_fichier);
```

Le nombre renvoyé A indiqué donc la position du curseur dans le fichier.

Exemple :

*main.c

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

int main()
{long c;
    FILE *fiche_source;
    fiche_source=fopen("test.txt","a+");
    if(fiche_source==NULL){printf("There has been an error!");
    exit(1);}

    fprintf(fiche_source,"bonjour ");
    c=ftell(fiche_source);
    printf("la position est: %ld",c);
    fclose(fiche_source);

    return 0;
}
```

Figure 59 *: Position du curseur dans le fichier.*

➢ `fseek` : se positionner dans le fichier

La syntaxe de `fseek` est la suivante :

```
fseek(nom_interne_fichier, deplacement, origine);
```

La fonction `fseek` permet de déplacer le curseur d'un certain nombre de caractères (indiqué par deplacement) à partir de la position indiquée par origine.

- Le nombre deplacement peut être un nombre positif (pour se déplacer en avant), nul (= 0) ou négatif (pour se déplacer en arrière).
- Quant à l'origine, vous pouvez mettre comme valeur l'une des trois constantes (généralement des define) listées ci-dessous :
  - `SEEK_SET` : indique le début du fichier ;
  - `SEEK_CUR` : indique la position actuelle du curseur ;
  - `SEEK_END` : indique la fin du fichier.

Voici quelques exemples pour bien comprendre comment on utilise avec déplacement et origine

- Le code suivant place le curseur trois caractères après le début :

```
fseek(fichier, 3, SEEK_SET);
```

- Le code suivant place le curseur cinq caractères avant la position courante :

```
fseek(fichier, -5, SEEK_CUR);
```

Remarquez que deplacement est négatif car on se déplace en arrière.

- Le code suivant place le curseur cinq caractères avant la fin du fichier :

```
fseek(fichier, -5, SEEK_END);
```

➢ `rewind` **:** retour au début

Cette fonction est équivalente à utiliser `fseek` pour nous renvoyer à la position 0 dans le fichier.

```
rewind(nom_interne_fichier);
```

L'utilisation est aussi simple que le prototype. Vous n'avez pas besoin d'explication supplémentaire !

Remarque :

- ➢ La fonction `rewind` remet le pointeur au début de fichier :

```
rewind (nom_fichier_interne)
```

- ➢ La fonction `ftell` retourne la position actuelle de type long (exprimée en octets) par rapport au début du fichier.

```
ftell (nom_fichier_interne)
```

VII.7 Détection d'erreur

Syntaxe :

```
ferror (nom_fichier_interne)
```

Vérifie si le statut d'erreur est levé pour ce flux (suite à l'utilisation d'une fonction de manipulation de flux du module `<stdio.h>`) . Si une erreur a effectivement été déclenchée, une valeur non nulle vous sera retournée. Pour de plus amples informations sur l'erreur précise rencontrée, il vous faudra ensuite interroger la valeur de la variable globale errno (un entier) définie dans l'entête `<errno.h>`. Il est aussi possible d'afficher le message d'erreur correspondant en invoquant la fonction `perror`.

Exemple

Afin de produire une erreur, nous allons ouvrir un fichier en lecture pour y écrire dedans : bien entendu, il n'y a aucune chance que cela puisse fonctionner.

*main.c

```c
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>

int main() {

    FILE * myFile = fopen( "test.txt", "r" );
    int rc = fputc( 'A', myFile );

    printf( "Results == %d %d %d\n", rc, ferror( myFile ), errno );
    perror( "My message" );

    rc = fclose( myFile );

    return 0;
}
```

Figure *60 : exemple de détection d'erreur.*

Le résultat produit par l'exécution de ce programme est le suivant :

```
$> sample
Results == -1 1 9
My message: Bad file descriptor
$>
```

La première valeur, -1, correspond au code retour de la fonction `fputc` (valeur équivalente à la macro `EOF`). La seconde valeur, 1, est le résultat renvoyé par `ferror` : le status d'érreur est donc bien mis pour le flux associé à la variable `myFile`. Le code erreur produit (stocké dans la variable `errno`) est donc le code 9. En voici sans définition.

```
#define  EBADF        9  /* Bad file descriptor */
```

Partie B

Chapitre 02 : Les méthodes d'indexation

III. Introduction

La méthode souvent la plus efficace pour l'utilisation de fichiers séquentiels est l'indexation (elle est également utilisable pour les autres types de données, stockées en mémoire). Les composantes sont stockées dans le fichier dans l'ordre de leur création. On utilise alors un tableau d'index, donnant en première position le numéro de la première composante, puis de la seconde,... Un index est construit sur une *clé*, et sert à accélérer les recherches pour lesquelles cette clé sert de critère.

Le principe de base d'un index est de construire une structure permettant d'optimiser les *recherches par clé* sur un fichier. Le terme de « clé » doit être compris ici au sens de « critère de recherche », ce qui diffère de la notion de clé primaire d'une table. Les recherches par clé sont typiquement les sélections de lignes pour lesquelles la clé a une certaine valeur.

IV. Index primaire

Avec les structures de fichiers simples, quand le fichier de données devient volumineux, les opérations d'accès (recherche, insertion, ...) deviennent très inefficaces. Les méthodes d'index permettent d'améliorer, dans une certaine mesure, les performances en gérant une structure auxiliaire (table d'index) accélérant la recherche. Un index est (généralement) une table ordonnée contenant les couples <clé, adr > utilisée pour accélérer la recherche des enregistrements d'un fichier. Si le champ clé ne contient de valeurs en double, l'index est alors « primaire ». Un index est dit « dense » s'il contient toutes les clés du fichier de données. Dans ce cas le fichier n'a pas à être gardé ordonné. Un index est dit « non dense » s'il ne contient pas toutes les clés du fichier de données (par exemple on garde une clé par bloc). Dans ce cas le fichier doit être ordonné.

II.1 Index à un niveau

Généralement, le fichier de données n'est pas ordonné pour simplifie l'insertion et la suppression. On maintient en MC une table ordonnée, contenant toutes les clés du fichier de données (index dense). A chaque clé est alors associée l'adresse de l'enregistrement dans le fichier de données. L'adresse est un couple de nombres : <num_bloc, déplacement>. Pour ne pas avoir à reconstruire la table d'index à chaque démarrage, on sauvegarde la table dans un fichier en fin de traitement.

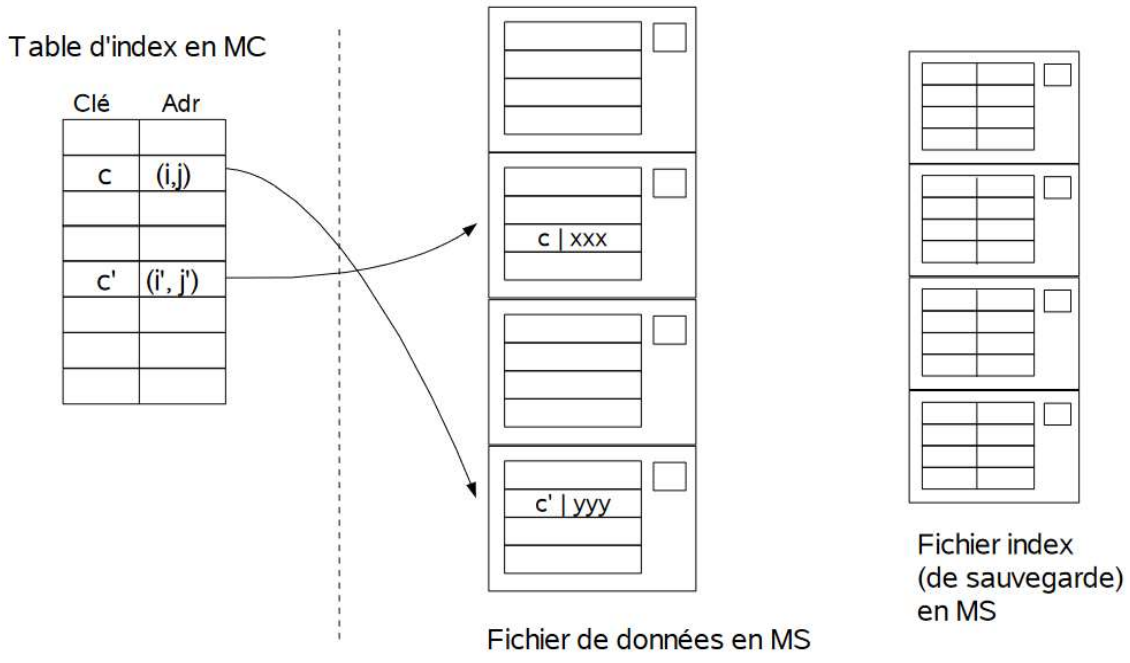


Figure 61 *: Schéma explicatif d'Index à un niveau*

Les fichiers de données et d'index peuvent être de n'importe quelle structure (blocs contigus, blocs chaînés, ...). De même que les enregistrements peuvent être à format fixe ou variable (avec ou sans chevauchement).

II.1.a Les opérations de base

- *La recherche d'un enregistrement* consiste à faire une recherche dichotomique de sa clé dans la table d'index, si elle existe, on récupère l'enregistrement à partir du fichier de donnée avec un seul accès disque. => Coût de l'opération : 1 accès disque (au max).
- *La requête à intervalle* consiste à rechercher tous les enregistrements dont la clé appartient à un intervalle de valeurs donné [a,b] :
    1- On commence par rechercher la plus petite clé >= 'a' dans l'index (recherche dichotomique en MC).
    2- Puis on continue séquentiellement dans la table jusqu'à trouver une clé > 'b'.
    3- Pour chaque clé, on accède au fichier de données pour récupérer l'enregistrement. Ce dernier point peut être amélioré si on tri les numéros de blocs trouvés avant d'accéder au fichier de données (afin de ne pas lire 2 fois le même bloc) => coût de l'opération en nombre d'accès : le nombre de clés vérifiant la requête (au max).
- *L'insertion d'un nouvel enregistrement* se fait en fin de fichier de données. Sa clé est insérée dans la table d'index (en MC) avec décalages pour garder l'ordre des clés. => Coût de l'opération : 2 accès disques.
- *La suppression* dans le cas d'un format variable avec chevauchement sera logique s'il n'y a pas de gestion de trous. => Coût : 0 accès si le bit d'effacement se trouve dans la table d'index, 1 à 2 accès sinon. Dans les autres cas, la suppression peut être physique en déplaçant par exemple le dernier enregistrement du fichier à la place de celui effacé.

II.1.b Cas particuliers

Si le fichier de données est ordonné, on ne garde qu'une clé par bloc (par exemple la plus grande du bloc) dans la table d'index. C'est alors un index non dense (car il ne contient pas toutes les clés). Cela permet de diminuer la taille de la table d'index tout en gardant pratiquement les mêmes performances de recherche qu'une méthode d'index avec fichier non ordonné.

La requête à intervalle est beaucoup plus performante, car dans un même bloc on peut récupérer plusieurs enregistrements vérifiant la condition de la requête.
L'insertion est par contre coûteuse, à cause des décalages dans le fichier de données. Une solution serait alors de maintenir une zone de débordement dédiée aux enregistrements issus des décalages inter-blocs.

II.1.c Les suppressions sont logiques.

On peut aussi représenter l'index en mémoire centrale sous forme d'un arbre de recherche binaire (au lieu d'une table ordonnée). L'avantage est d'éviter les décalages lors de l'insertion de nouvelles clés dans l'index. L'inconvénient est le risque du déséquilibre de l'arbre ou alors le surcout associé au maintien de l'équilibrage par des algorithmes appropriés.

II.2 Index multiniveaux

Si l'index est trop grand pour résider en MC, on construit un deuxième index sur le fichier index (ordonné). Dans ce cas, on choisira une seule clé pour chaque bloc du fichier index (index non dense) pour construire le 2e index. Si le 2e index est encore trop grand pour résider en MC, on le stocke sur disque (fichier 2e index) et on construit un 3e index en choisissant une clé par bloc du 2e fichier index. On peut répéter ce procédé au tant que nécessaire.
Dans la figure ci-dessous, un index à 2 niveaux :

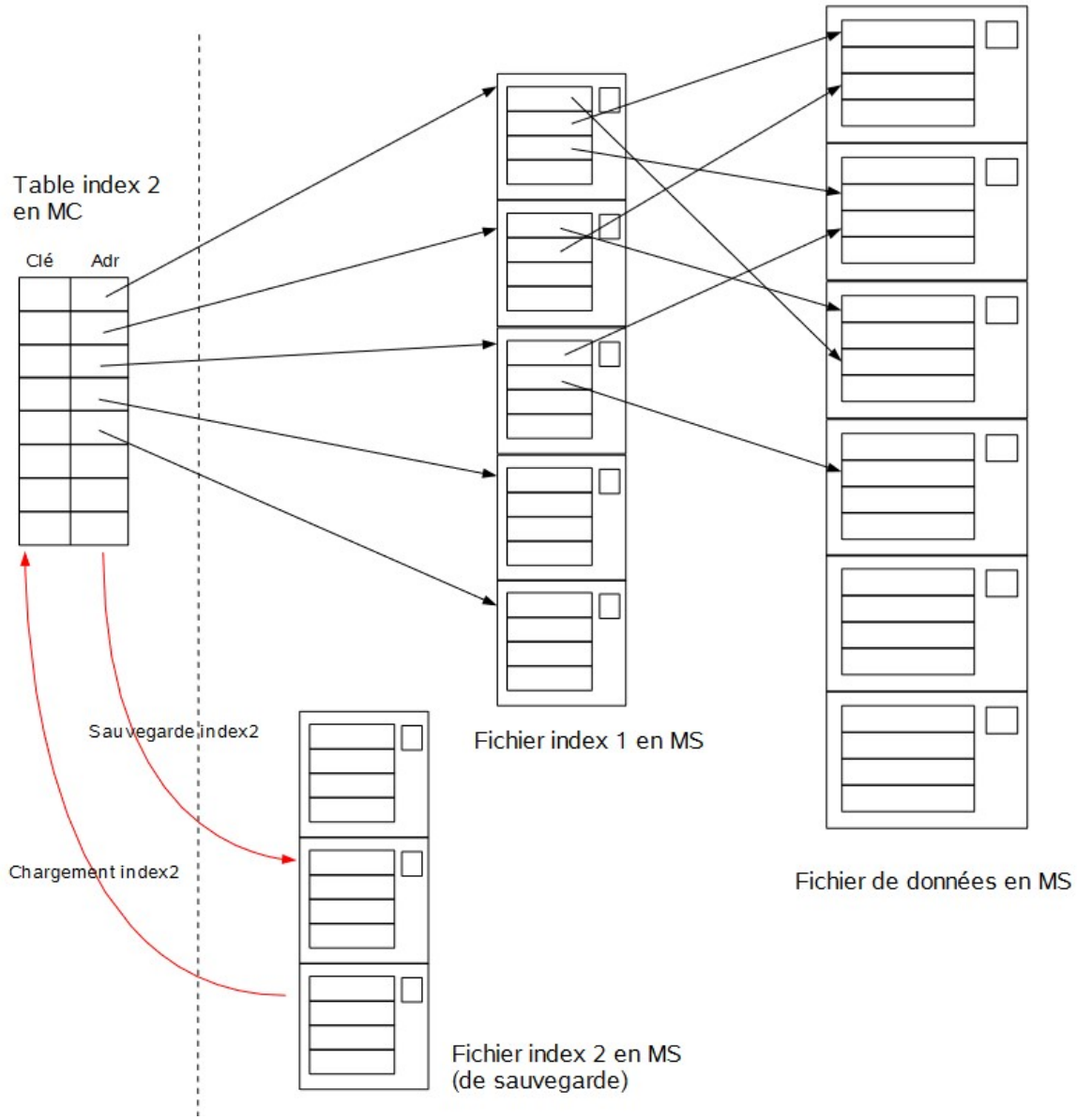


Figure 62 *: Schéma explicatif d'Index à 2 niveaux*

V. Index secondaire

Pour améliorer les recherches basées sur des champs non clés (appelés aussi clés secondaires), on peut construire des index secondaires sur ces champs.

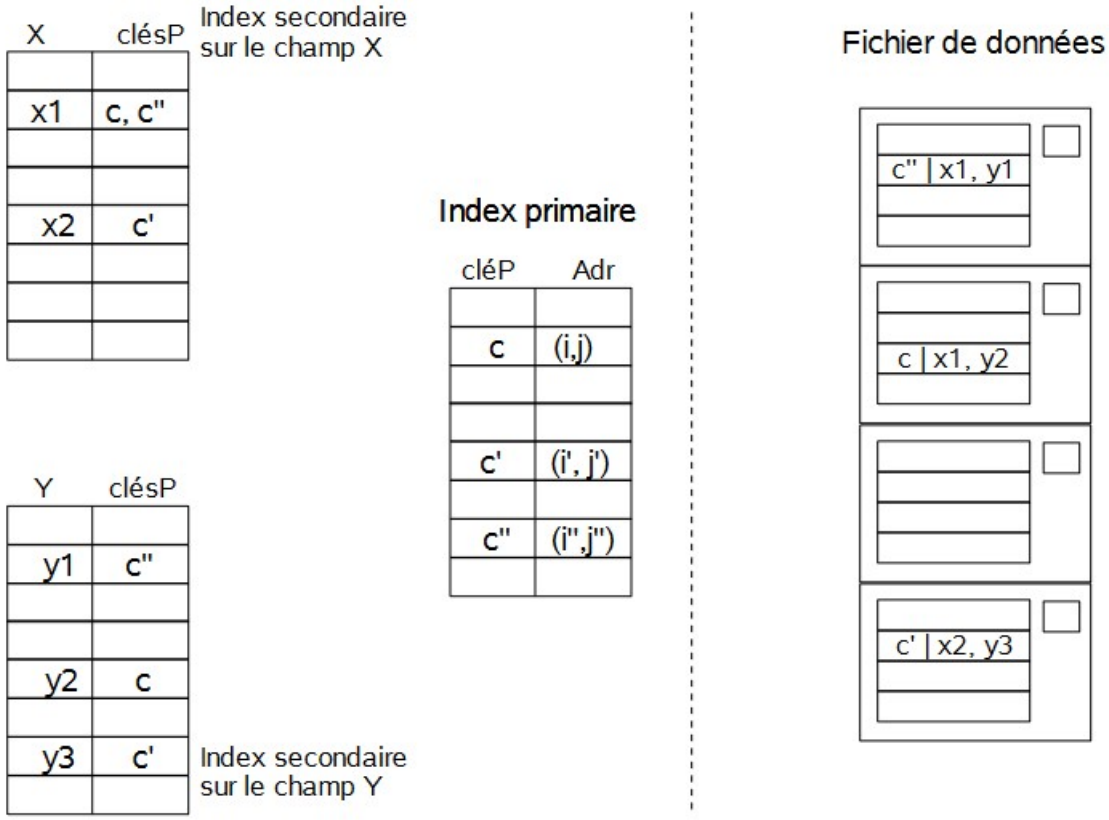
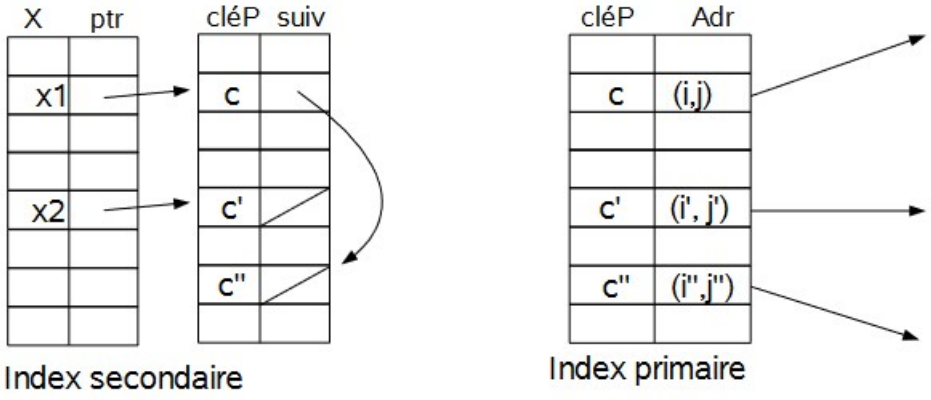


Figure 63 *: Schéma explicatif d'Index secondaire*

Le problème avec les clés secondaires, est qu'il peut exister plusieurs enregistrements pour une valeur du champ indexé. On implémente généralement cette multiplicité à travers des listes de clés primaires



Quand on recherche les enregistrements suivant une clé secondaire (par exemple X=a), on utilise l'index secondaire sur ce champ pour récupérer la ou les clés primaires associées à la valeur cherchée (a). Pour chaque clé primaire trouvée, on utilise l'index

primaire pour localiser l'enregistrement sur le fichier de données (numéro de bloc et déplacement). C'est la méthode des listes inversées.

Exemple 01 :

Pour les recherches multicritères de la forme « trouver tous les enregistrements dont la valeur de X = a ET la valeur de Y = b ET ... » avec X, Y, ... des clés secondaires, on procède comme suit:

1- En utilisant l'index secondaire X, trouver la liste Lx de clés primaires associées à la valeur de X (a).

2- Refaire la même action pour chaque clé secondaire mentionnée dans la requête...

3- Faire l'intersection des listes de clés primaires Lx, Ly, ... pour trouver les clés primaires associées avec chaque valeur de clé secondaire mentionnée dans la requête.

4- Utiliser alors l'index primaire pour retrouver les enregistrements du fichier de données

Exemple 02 :

Pour insérer un enregistrement <c,x,y,...> avec c sa clé primaire et x,y,... ses clés secondaires, on procède comme suit :

1- recherche c dans l'index primaire pour vérifier qu'elle n'existe pas déjà et pour trouver l'indice ip où doit être insérée cette clé (recherche dichotomique).

2- Insérer l'enregistrement à la fin du fichier de données. Soit (i,j) son adresse.

3- Insérer le couple <c, (i,j)> dans la table d'index primaire, à l'indice ip (en procédant par décalages).

4- rechercher la valeur x dans l'index secondaire X,

si x existe, rajouter c à la liste pointée par x

si x n'existe pas, insérer x (par décalages) dans la table X. La nouvelle entrée x, pointe une liste formée par une seule clé primaire (c).

5- refaire l'étape 4) pour chaque clé secondaire restante.

Exemple 03 :

Pour supprimer un enregistrement de clé primaire c, il suffit de positionner un bit d'effacement au niveau de la table d'index primaire, au niveau de l'entrée c. Cela permet de ne pas avoir à mettre à jour tous les index secondaires. Au niveau du fichier de données, l'enregistrement peut être supprimé physiquement si la structure du fichier le permet (comme par exemple T~OF).

Les méthodes d'index comme celles présentées dans ce chapitre, sont dédiées aux fichiers statiques (c'est à dire dans les cas où le nombre d'insertions et de suppressions est relativement faible).

VI. Avantage des méthodes d'index

L'avantage de cette méthode est que l'ajout de composantes est optimal : on rajoute la valeur en fin de fichier, on met à jour le tableau d'index. Tout déplacement d'une composante sera donc remplacé par une modification du tableau d'index, sans déplacement réel de la valeur dans le fichier. En général, ce tableau peut tenir en mémoire, ce qui permet une modification rapide, en général on préfère le sauver également sur support magnétique avant de quitter le programme, ce qui évitera de le recréer (par exemple refaire un tri) à la prochaine utilisation. On peut également utiliser une liste d'index si les déplacements sont fréquents (mais alors l'accès devient séquentiel). Le second avantage de cette méthode est que l'on peut utiliser simultanément plusieurs index : par exemple pour une liste de personnes, on peut créer un index pour le classement alphabétique des noms, un autre sur les villes, on accédera donc plus rapidement à tous les champs indexés, alors que les champs non indexés devront se satisfaire d'une recherche séquentielle, et ce sans modification dans le fichier (un tri par nom puis par ville auraient été nécessaires sans indexation). Par contre toute modification nécessitera la mise à jour de tous les tableaux d'index.

La suppression, par contre, pose problème. En général, toujours pour éviter les décalages dans les fichiers, on préfère marquer d'un signe distinctif les champs supprimés (par exemple un nom non alphabétique ou vide), puis remettre à jour les index qui ne pointeront plus sur ce champ. Le retassage, assez long, n'est effectuée que sur ordre de l'utilisateur ou lorsqu'il quitte le programme. On peut aussi (comme dans la méthode du super-tableau) créer une liste des champs vides, ce qui permettra d'y accéder, plus rapidement que par une recherche séquentielle, lors de la prochaine insertion.

Sur un fichier indexé, on peut à nouveau se permettre des algorithmes utilisant l'insertion, puisque celle-ci n'affecte que l'index (à accès rapide). Pour un tri par exemple, on pourra utiliser le tri par insertion, à condition d'optimiser la recherche de la position d'insertion (par dichotomie pondérée par exemple), puisque celle-ci nécessite des lectures de champs dans le fichier alors que l'insertion n'entraîne que des décalages dans un tableau, d'une durée généralement négligeable devant le temps pris par la recherche. On peut également utiliser une liste d'index plutôt qu'un tableau si nécessaire.

Annexe
(Séries des TD)

Ecole Normale Supérieure
Constantine
Département de l'informatique
et des mathématiques

2eme année Informatique
Module : Algorithme 2
Enseignante : Dr. F. Bettou

TD N°1 : Les pointeurs

Exercice 01 :

A l'aide de pointeurs, écrivez un programme qui calcule la somme, le produit, la différence et la division de deux nombres réels.

Exercice 02 :

A l'aide de pointeur, écrire un programme qui vérifier si un nombre est pair ou impair.

Exercice 03 :

A l'aide de pointeur, écrire un programme qui affiche les diviseurs d'un entier positif n non nul.

Exercice 04 :

L'objectif de cet exercice est de représenter en mémoire les données déclarées dans un programme, ainsi que leurs différentes valeurs, à un moment d'exécution donné. Pour ce faire. Vous représenterez l'occupation des données en mémoire dans un tableau à 3 colonnes.

Exercice 05 :

Ecrire un programme pour entrer et affiche les éléments d'un tableau à l'aide d'un pointeur.

Exercice 06 :

Ecrire un programme qui lit un entier x et un tableau Tab du type int au clavier et élimine toutes les occurrences de x dans Tab en tassant les éléments restants. Le programme utilisera les pointeurs P1 et P2 pour parcourir le tableau.

Ecole Normale Supérieure
Constantine
Département de l'informatique
et des mathématiques

2eme année Informatique
Module : Algorithme 2
Enseignante : Dr. F. Bettou

TD N°2 : Les listes chaînées

Exercice 01 :

L'objectif est d'écrire les deux fonctions suivantes : la première permet de transformer en liste chainée un tableau dynamique de nb éléments. La seconde transforme à l'inverse une liste chainée de nb éléments en un tableau dynamique. Faire un programme de test.

Exercice 02 :

Ecrire une fonction qui prend en paramètre une liste chainée et renvoie une autre liste ayant les mêmes éléments mais dans l'ordre inverse. Tester dans un programme.

Exercice 03 :

Ecrire une fonction de concaténation de deux listes chainées. Il y a deux versions de la fonction :
une destructrice des deux listes données au départ et l'autre qui préserve ces deux listes.

Exercice 04 :

Ecrire une fonction qui détermine si une liste chainée d'entiers est triée ou non en ordre croissant. Ecrire une fonction qui insère un élément dans une liste chainée triée.

Exercice 05 :

Ecrire une fonction qui permet de fusionner deux listes chainées. La fusion se faite en alternant un élément d'une liste avec un de l'autre liste. Tous les éléments des deux listes doivent trouver leur place dans la liste résultante même en cas de différence de taille. Faire deux versions, une qui conserve les deux listes de départ et une autre qui les détruit. Tester dans un programme.

Exercice 06 :

Ecrire une fonction qui prend en entrée une liste chainée d'entiers et qui ressort deux listes chainées, une pour les nombres pairs et une autre pour les nombres impairs. La liste initiale est détruite. Tester dans un programme.

Ecole Normale Supérieure
Constantine
Département de l'informatique
Et des mathématiques

2eme année Informatique
Module : Algorithme 2
Enseignante : Dr. F. Bettou

TD N°3 : les Piles

Exercice 01 :
On vent avoir l'image miroir d'une phrase terminer par un point.
Exemple : cela est un exemple
Elpmexe nu tse alec
Utiliser la pile statique pour réaliser ce travail.

Exercice 02 :
On vent avoir l'image miroir de chaque mot de la phrase de l'exercice 01
Exemple : cela est un exemple
Alec tse nu elpmexe

Exercice 03 :
Calculer la factorielle n ! en utilisant une pile n !=n(n-1)(n-2)…..2.

Exercice 04 :
Ecrire une fonction stack_copy(s) recevant une pile (s) comme argument et renvoyant une copie s2 de s. Attention, la pile s doit (bien sûr) être conservée !
Evaluer le coût en mémoire et le nombre d'opérations de la fonction.

Exercice 05 :
Soit une Pile d'entiers on désire supprimer toutes les valeurs négatives. Ecrire la fonction correspondante.

Exercice 06 :
Définir les fonctions suivantes qui travaillent sur une pile (on stocke des animaux : Nom et Nombre de Pattes) on écrira ces fonctions en utilisant uniquement les fonctions empiler, depiler, estVide, estPleine :

- La fonction **afficher** qui affiche le contenu de P.
- La fonction **regarder** renvoie l'animal de P qui se trouve en haut de la pile (mais en conservant cet animal en haut de la pile).
- La fonction **dupliquer** recopie en haut de la pile l'animal qui s'y trouve.
- La fonction **echanger** échange les deux animaux situés en haut de la pile.
- La fonction **suprimerBipedes suprime** les animaux ayant 2 pattes.

Ecole Normale Supérieure
Constantine
Département de l'informatique
et des mathématique

2eme année Informatique
Module : Algorithme 2
Enseignante : Dr. F. Bettou

TD N°4 : File d'attente

Exercice 01 :
Soit une file de nombre réels. Ecrire un programme en C qui permet :

1. De déclarer le type FILE de nombre réel
2. D'implémenter les opérations suivantes sur une FILE
   - **Taille** : fonction entière qui retourne la taille d'une file
   - **Est_Vide :** fonction booléenne qui indique si une file est vide ou non.
   - **Enfiler :** ajouter un élément à la queue de la file
   - **Défiler :** enlever un élément de la file.
   - **Supprimer :** toutes les occurrences d'une valeur X dans une file.

Exercice 02 :
Soit F une File d'entiers. Écrire les fonctions pour déterminer :

1. Le nombre d'éléments.
2. La valeur maximale.
3. La valeur minimale.

Exercice 03 :
On dispose d'une file de nombres entiers, ordonnés suivant l'ordre décroissant des valeurs. Écrire une procédure qui insère une valeur val à la place qu'il faut (pour garder l'ordre décroissant)
dans les deux cas suivants :

1. Si elle n'existe pas ;
2. Même si elle existe, si la même chose avec le premier cas, mais, cette fois, au lieu de faire "si x > Val",
vous faites "si x ≥ Val".

Exercice 04 :
Soit F une File d'entiers. Écrire une procédure qui permet de trier ses éléments selon un ordre croissant.

Exercice 05 :
Soit à gérer une file de patients qui attendent dans une salle. Chaque patient qui arrive est enregistré et se met en queue de la file d'attente.

Les informations concernant le patient sont :

- Nom du patient
- Prénom du patient
- Numéro d'ordre

Sachant que la file d'attente ne dépasse pas 100 patients :

1. Ecrire une procédure qui permet d'ajouter un patient à la file.
2. Ecrire une procédure qui permet de supprimer un patient de la file.
3. Ecrire le programme.

Exercice 06 :

Soit à gérer une file de patients qui attendent dans une salle. Chaque patient qui arrive est enregistré et se met en queue de la file d'attente.

Les informations concernant le patient sont (voir exercice 05).

1. Quel est l'avantage d'une représentation par les listes chaînées dans cet exemple ?
2. Ecrire une procédure qui permet d'ajouter un patient à la file.
3. Ecrire une procédure qui permet de supprimer un patient de la file.
4. Ecrire le programme.

Ecole Normale Supérieure
Constantine
Département de l'information et des mathématiques

2eme année Informatique
Module : Algorithme 2
Enseignante : Dr. F. Bettou

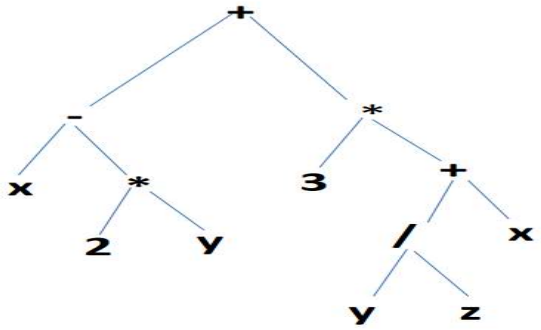
TD N°5 : Arbre

Exercice 01:

1. Construire l'arbre Binaire correspondant à l'expression arithmétique suivant : A=B+(C+2)*E/(B+1)
2. Donnez le résultat des parcours Préfixé, Postfixé, infixé de l'arbre précédent.

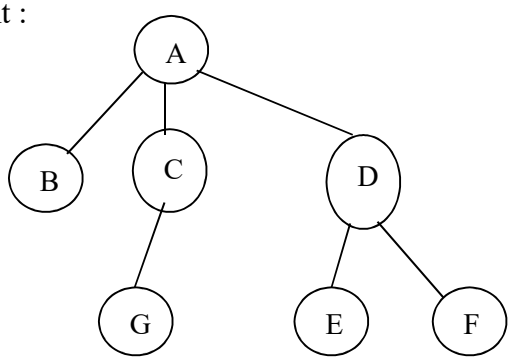
Exercice 02 :

Une expression arithmétique peut être représentée par un arbre binaire.
Soit l'expression arithmétique représentée par l'arbre A :
Donner les expressions représentées par A lorsqu'A est parcouru en ordre préfixe, infixe et post-fixe.



Exercice 03 :

Soit l'arbre de degré qq suivant :



Donnez les mesures suivantes :

- La taille de l'arbre, le degré de cet arbre, la profondeur de l'arbre.
- Transformer cet arbre en un arbre binaire.

Exercice 04 :

Donnez la déclaration d'un arbre binaire et une fonction Feuille(P) qui teste si un nœud P de l'arbre est une feuille et retourne 1 sinon 0.

Exercice 05 :

Sachant qu'un arbre est par définition de nature récursive, alors proposez une fonction récursive Nbre-Feuille(Racine) qui retourne le nombre de feuille d'un arbre binaire.
Pour un traitement récursif, il faut :

- Décomposer le problème en sous problèmes (ici l'arbre binaire à décomposer en sous arbres binaires)
- Déterminer le cas trivial(le test d'arrêt de l'appel récursif et sa solution)
- Définir l'appel récursif avec les paramètres d'appel de taille plus réduite (celle du sous problèmes)
- Dans la récursivité, n'utilise jamais un compteur

Exercice 06 :

Soit un ABR ayant comme éléments, les Etudiants.

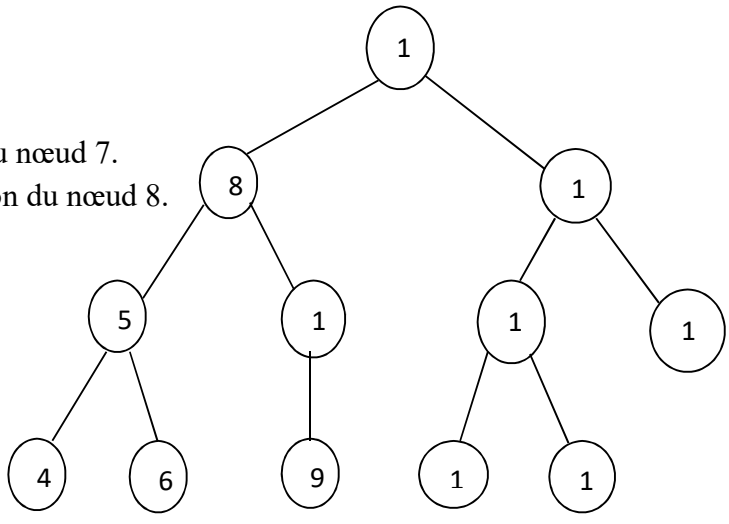
```
ArbreE=^Etud;
Etud = Record
       NumCart:integer;
       NomE,PrenomE : String[30];
       EtudG: ArbreE;
       EtudD: ArbreE;
      End;
```

1. Ecrire une procédure qui permet d'insérer un nouveau mot dans l'arbre.
2. Ecrire une procédure qui permet de supprimer un mot de l'arbre.

Exercice 07 :

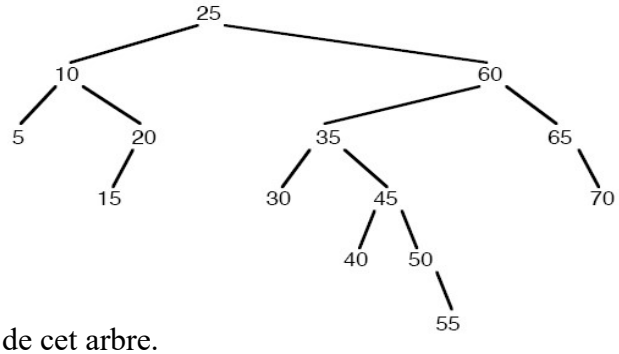
Soit l'arbre lexicographique suivant :

1. Donnez le résultat après l'insertion du nœud 7.
2. Donnez le résultat après la suppression du nœud 8.



Exercice 08 :

Voici un arbre de 14 éléments :



On s'intéresse aux ABRs

a. Rappelez les propriétés des ABRs.
b. Donner les listes infixé, préfixé et postfixé de cet arbre.
c. Donner la déclaration de cet arbre
d. Ecrire une fonction qui permet de compter le nombre des feuilles de cet arbre.
e. Ecrire une fonction qui permet de chercher un élément n dans cet arbre.

Ecole Normale Supérieure
Constantine
Département de l'informatique
et des mathématique

2eme année Informatique
Module : Algorithme 2
Enseignante : Dr. F. Bettou

TD N°6 : Technique de hachage

Exercice 01 :

Considérer la fonction de hachage h(c) = c mod 13 et une table de hachage avec m = 13 adresses.

1. Insérer les clés 26, 37, 24,30, et 11 dans la table de hachage ci-dessus en utilisant la résolution des collisions par adressage ouvert et sondage linéaire avec la fonction hi(c) = (h(c)+i) mod m

12	24
11	37
10	
9	
8	
7	
6	
5	
4	30
3	
2	
1	11
0	26

2. Rajouter maintenant les clés 1, 2, 3, 4, 5, 6, 7, 8, 9,10. Quel problème rencontrez-vous ? Quelles solutions proposez-vous ?

Exercice 02 :

On dispose d'une table de hachage de taille m dont les collisions sont résolues par liste chaînée. On suppose que l'on a effectué un hachage uniforme de n clés dans la table par une fonction h.

1. Quel est le nombre attendu de collisions, c'est-à-dire l'espérance du cardinal de $\{(x, y) \mid h(x) = h(y)\}$?

2. Quel est le nombre moyen de cases vides dans le tableau, c'est-à-dire le cardinal moyen de $\{i \in J0, m - 1K \mid \forall k \in K, h(k) /= i\}$ où K est l'ensemble des clés qui ont été insérées dans la table ?

Exercice 03 :

Écrire une fonction table_de_hachage_t cree_table_de_hachage(int taille) qui crée une table de hachage vide avec un tableau de la taille indiquée (mais qui ne contient que des listes vides puisqu'il n'y a pas encore d'élément dans la table de hachage).

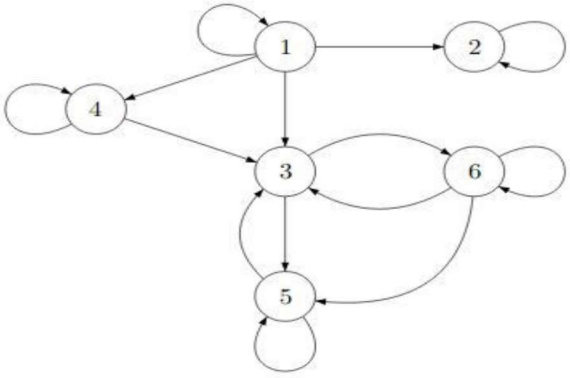
Ecole Normale Supérieure
Constantine
Département de l'informatique
et des mathématiques

2eme année Informatique
Module : Algorithme 2
Enseignante : Dr. F. Bettou

TD N°7 : Graphe

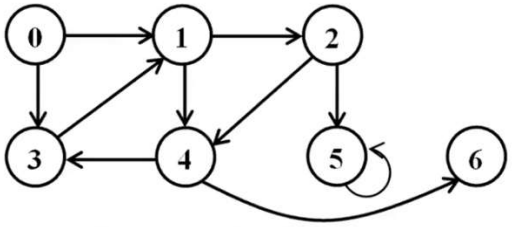
Exercice 01 :

Soit le graphe illustré dans la figure 1, donner la représentation de ce graphe en utilisant les deux approches vues en cours (statique et dynamique).



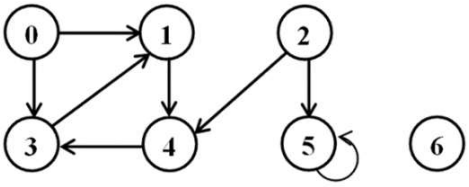
Exercice 02 :

Donnez le résultat du parcours en largeur d'abord, ensuite, profondeur d'abord, du graphe ci-dessous.



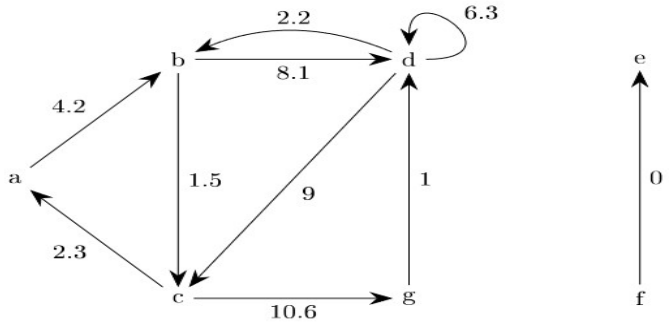
Exercice 03 :

Ecrire un programme C permettant de créer le graphe ci-dessous en utilisant une liste d'adjacence.



Exercice 04 :

La figure ci-dessous représente un graphe G. L'ensemble de ses sommets est {a, b, c, d, e, f , g}. Les arcs sont étiquetés par des réels, comme indiqué sur la figure.



Question 01 : illustrez chacun des termes suivants par un exemple pris dans ce graphe (s'il en existe un) : chemin, chemin de longueur 3, circuit, boucle, composante fortement connexe.

Question 02 : Proposez une structure de donnée permettant de mémoriser un type Graphe représentant un graphe orienté avec arcs et sommets étiquetés. Dessinez le contenu de cette structure de données pour le graphe G. On examinera les deux cas suivants :

- une structure de données basée sur des tableaux de taille fixe ;
- une structure de données basée sur des chaînages explicites par pointeurs.

Pour chacune de ces structures :

Question 03 : Écrire un algorithme calculant le nombre d'arcs du graphe ;

Question 04 : Écrire l'algorithme de parcours générique d'un graphe.

Exercice 05 :

Question 01 : écrire un algorithme qui vérifie si deux sommets i et j sont voisins en utilisant l'approche statique.

Question 02 : Même question 01, en utilisant l'approche dynamique (liste adjacente d'adresse des voisins).

Ecole Normale Supérieure
Constantine
Département de l'informatique
et des mathématiques

2eme année Informatique
Module : Algorithme 2
Enseignante : Dr. F. Bettou

TD N°8 : Récursivité

Exercice 01 : Somme de n entiers

On considère la fonction Somme(n) qui donne la somme de n premiers entiers positifs, on vous propose un algorithme itératif.

```
Algorithme Itérative (N : entier)
  I, S : entier
  Début
     S ← 0
     Pour I ← 1 à  N faire
          S ← S + I
          Somme ← S
  Fin.
```

Ecrire une fonction Somme(n) récursive

Exercice 02 : Puissance d'un nombre

Ecrire une fonction Puissance(x,n) récursive qui calcule la puissance d'un nombre réel x^n avec n entier positif.

Exercice 03 : Plus Grand Diviseur Commun

On veut calculer le Plus Grand Diviseur Commun (PGCD) de deux entiers N et M, par la méthode suivante :

- N > M
- On divise N par M tel que R = N mod M
- Si R = 0 alors PGCD=M
- Si R ≠ 0 alors Diviser M sur R

Et ainsi de suite jusqu'à retrouver R=0, le PGCD est égal au dernier diviseur qui donne comme reste zéro. Ecrire une fonction PGCD(N,M) récursive.

Exercice 04:

On appelle "palindrome" un mot ou une phrase qui se lit de la même façon à l'endroit comme à l'envers, sans tenir compte des espaces. Exemple : le mot "abcba" est un palindrome.

Ecrire une fonction récursive permettant de vérifier si une chaîne de caractères ch est un palindrome.

Exercice 05:

On considère la fonction d'Ackermann F de deux variables x, y réelles définie comme suit:

- F(x, y) = y+1 si x=0 ;
- F(x, y) = F(x-1, 1) si x≠0 et si y=0 ;
- F(x, y) = F(x-1, F(x, y-1)) si x et y sont différents de 0.

Ecrire une fonction récursive.

Ecole Normale Supérieure
Constantine
Département de l'informatique
et des mathématique

2eme année Informatique
Module : Algorithme 2
Enseignante : Dr. F. Bettou

TD N°9 : les Fichiers

Exercice 01 :

Ecrire un programme qui permet de copier un fichier texte vers un autre.

Exercice 02 :

Ecrire un programme qui permet de concaténer deux fichiers dans un autre.

Exercice 03 :

Soit un fichier qui comporte un texte formé de lettres en minuscule non accentuées et d'espaces. Ecrire un programme en C qui permet de recopier les voyelles dans un fichier sans les espaces et les consonnes dans un autre fichier.

Exercice 04 :

Ecrire le programme qui insère tous les nombres multiples de 5, compris entre 1 et 1000, dans un fichier appelé Multiple5

Exercice 05 :

Soit deux fichiers F1 et F2 contenant une séquence de nombres entier on ordre croissance, on veut crier un troisième fichier F qui contient les deux séquences classer en ordre croissance.

Exercice 06 :

Ecrire la fonction NombreMots (nomf : text) : integer; qui renvoie le nombre de mots présents dans le fichier nomf.

Exercice 07 :

Il existe un fichier « F_Personnel » contenant les informations concernant des personnes : le Nom, le prénom, le sexe (M/F) et l'Age.

Ecrire un programme qui permet de donner la femme et l'homme (le Nom, le prénom et l'Age) les plus âgés.

Références bibliographiques

- D. Beauquier, J. Berstel, Ph. Chretienne. Elements d'algorithmique, Masson, 2005.
- Thomas Cormen, Charles Leiserson, Ronald Rivest et Clifford Stein. Introduction à l'algorithmique, Cours et exercices, Dunod
- M. C. Gaudel, M. Soria et C. Froidevaux. Types de Données et Algorithmes, Vol1. Analyse d'Algorithmes, Définition des Types de Données, INRIA, Collection Didactique
- https://docplayer.fr/
- T. Cormen, C. Leiserson, R. Rivest, C.Stein, INTRODUCTION À L'ALGORITHMIQUE, Cours et exercices, 2001.