Structure de Données Avancées

Support du Cours

Driss El Ouadghiri

Email : dmelouad@gmail.com

Université My Ismail
Faculté des Sciences
Département de Mathématiques et d'Informatique
Meknès

Partie 1

Rappels :

- **Structures (enregistrements)**
- **Pointeurs**
- **Allocation dynamique de la memoire**

Les structures

Les structures permettent de rassembler des valeurs de type différent. Par exemple pour une adresse, on a besoin d'un numéro (int) et un nom de la rue (char).

Déclaration

```
struct  adresse {
        int  numero ;
        char  rue[50] ;
      } ;
```

- Chaque élément déclaré à l'intérieur de la structure est appelé un champ.
- Le nom donné à la structure est appelé étiquette de la structure.

Ici, on a déclaré un type de structure et non pas une variable structure.

On déclare une variable associée à une structure de la manière suivante :

```
struct  adresse adr1, adr2 ;
```

on peut initier une structure lors de sa déclaration :

```
struct  adresse adr1 = {15, ''Avenue Saghro''}  ;
```

Manipulation

On accède aux données contenues dans un champs d'une structure en faisant suivre le nom de la structure par un point '.' et le nom du champ voulu :

```
adr2.numero = 19 ; strcpy(adr2.rue , ''Rue  Annajah'') ;
```

Si deux structures on le même type, on peut effectuer :

```
adr1 = adr2 ; /* la personne qui habitait adr1 a déménagé à adr2 */
```

Mais on ne peut pas comparer deux structures (avec = = ou !=). La comparaison se fait champ par champ.

Tableau de structure

on déclare un tableau de structure de la même façon qu'un tableau de variables simples :

```
struct  adresse pers[50] ;
```

Attention: La structure adresse est déjà déclarée avant le tableau pers qui est un tableau dont chaque élément est une structure de type adresse.

pers[i].rue fait référence au champ rue de la i^{ème} personne du tableau pers.

Structure de structure:

On peut utiliser une structure comme champ d’une autre structure:

```
struct adresse {
        int numero ;
        rue  char[50] ;  } ;
struct employe {
        char nom[20] ;
        char prenom[20] ;
        struct adresse domicile ;  } ;
struct employe rep_empl[50] ;

strcpy(rep_empl[0].nom ; ‘’Amalou’’) ;
strcpy(rep_empl[0].prenom ; ‘’Ider’’) ;
rep_empl[0].maison.numero = 19 ;
strcpy(rep_empl[0].maison.rue ; ‘’Avenue Zaid Ouhmad’’) ;
```

On peut accéder aux valeurs de ce tableau par :

```
char  ch1;
ch1 = rep_empl[0].prenom ;
```

Les pointeurs

Lors de la compilation d'un programme, l'ordinateur réserve dans sa mémoire une place pour chaque variable déclarée. C'est à cette place que la valeur de la variable est stockée. Le compilateur associe à la variable l'adresse de stockage (adresse de début de la place mémoire réservée). Lors de l'exécution du programme, à chaque rencontre d'un nom d'une variable, le programme va chercher à l'adresse correspondante la valeur de la mémoire.

Exemple:

```
int a = 0xf ; /* entier codé sur 4 octets */
short b = 0x0 ; /* entier codé sur 2 octets */
int c = 0x9 ; /* entier codé sur 4 octets */
```

Ceci, est stocké en mémoire de la manière suivante :

Non	Adresse hexa	Valeur en hexadécima
a	bfbff000	00 00 00 0f
b	bfbff004	00 00
c	bfbff006	00 00 00 09

Ici on suppose que l'espace servant à stocker les données commence à l'adresse (bfbff000).

a est stocké sur 4 octets et son adresse est (bfbff000), donc l'adresse de b sera (bfbff000)+(4) = (bfbff004).
b est codé sur 2 octets et son adresse est (bfbff004), donc l'adresse e c sera (bfbff004)+(2) = (bfbff006).

Définition et déclaration d'un pointeur :

Un pointeur est une variable qui a pour valeur l'adresse d'une variable : celle sur laquelle elle pointe.

Un pointeur est toujours associé à un type de variable et un seul.

Au moment e la déclaration, on détermine le type de la variable qui sera pointé par le pointeur, en écrivant le type concerné, puis le non du pointeur avec une * devant.

```
int *ptn ; /* ptn est une variable pointeur sur un entier */
int n ; /* la varaible n est un entier */
```

pour afficher l'adresse de la variable n au pointeur ptn on utilise l'**opérateur adresse &** (qui signifie adresse de) : ptn := &n ;

Exemple2 :

```
#include<include.h>
main() {
        float *px ; /*déclaration d'un pointeur sur un réel */
        float x = 45.9 ;
        px = &x ; /* px pointe sur x */
        printf(''Adresse de x : %x \n'', &x) ;
        printf(''Valeur de px : %x \n'', px) ;
        printf(''Valeur de x : %4.1f \n'', x) ;
        printf(''Valeur pointée par px : %5.1f \n'', *px) ;
        return 0;
}
```

Le programme affichera ceci à l'écran :

```
Adresse de x : bfbffa4c      (par exemple)
Valeur de px : bfbffa4c
Valeur de x : 45.9
Valeur pointée par px : 45.9
```

Pointeurs et fonctions :

Une variable globale est une variable connue dans toutes les fonctions d'un programme.

Une variable locale n'est connue qu'a l'intérieur d'une fonction.

Ainsi, il peut exister une variable float a ; dans une fonction et une variable int a ; dans une autre fonction sans qu'il y ait conflit.

Exemple : la fonction suivante n'a pas d'effet lors de son appel depuis main

```
void permut(int a, int b) {
int c ;
c = a ;    a = b ;    b = c;
return ; }
```

Lors de l'appel de cette fonction depuis main, les valeurs des arguments réels vont être copiés dans les variables de permut et ce sont ces variables locales qui vont être modifiées, pas celle de main.

Ainsi, dans main, un appel de type permut(n, p) ; laissera n et p inchangés.

On dit que ces arguments sont passé **par valeur**.

Pour que permut puisse changer les valeurs de ses arguments lors de son appel depuis un main, il faut que ses arguments soient les adresses des variables a et b et donc des pointeurs.

```
void permut (int *a, int *b) {
int c ;
c = *a ;   * a = *b ;    *b = c;
return ; }
```

Lors de l'appel de la fonction, les pointeurs locaux reçoivent les adresses des variables i et j qui sont des variables de main. Donc travailler sur ces pointeurs revient à travailler sur les variables i et j de main.

L'appel de cette fonction se fait ainsi : permut(&a, &b) ;

D'une façon générale, on utilise des pointeurs avec les fonctions quand on veut qu'une fonction modifie des variables du programme appelant.

Pointeurs et tableaux :

soit la déclaration suivante : int tab[10] ;

Le nom seul du tableau est une constante qui contient l'adresse du premier élément du tableau. Ainsi, tab est égal à &tab[0] et donc *tab est égal tab[0]. L'élément tab[i] est équivalent à *(tab+i). On a donc les correspondances suivantes :

tab → tab[0], tab+1 → tab[1], … tab+9 → tab[9]

Exemple :

```
int tab[6] ;
int pta, ptb ;
pta = tab ; ptb = pta + 2 ;
```

L'ordinateur a réservé en mémoire 6 fois par 4 octets pour le tableau tab de trois entiers. La variable constante tab et donc pta contient l'adresse du premier élément du tableau. L'opération ptb = pta + 5 n'ajoute pas 5 à la valeur de pta, mais ajoute 5 fois le nombre d'octets correspondant à un int. Donc ptb est n pointeur qui pointe sur le dernier élément du tableau. Il contient donc l'adresse de tab[5].

L'opérateur sizeof : Cet opérateur fournit la dimension d'un objet en octet.

```
int i ; sizeof(i) donne 4
double x ; sizeof(x) donne 8
char c ; sizeof(c) donne 1
int *pe ; sizeof(pe) donne 4
double pr ; sizeof(pr) donne 4
char *pc ; sizeof(pc) donne 4
double t[4] ; sizeof(t) vaut 32
int m[4][5] ; sizeof(mat) vaut 80
```

```
sizeof(int) donne 4
sizeof(double) donne 8
sizeof(char) donne 1
sizeof(int *) donne 4
sizeof(double *) donne 4
sizeof(char *) donne 4
```

Remarque : Le nom d'un tableau ne peut pas être assimilé à un pointeur. En effet, l'adresse que représente le nom du tableau est imposé par le compilateur au moment de la déclaration et ne peut pas être changé par la suite.

```
int t[10], *pe ;
pe = t ;
```

sizeof(t) et sizeof(pe) ne donne pas la même chose.

sizeof(t) donne la taille du tableau c'est 40 et sizeof(pe) donne 4 la taille d'une adresse.

Pointeurs et tableaux à plusieurs dimensions:

Un tableau à plusieurs dimensions est un tableau dont les éléments sont eux même des tableaux.

Ainsi, le tableau défini par int t[4][5] ; contient 4 tableaux de 5 entiers chacun. t donne l'adresse de 1er sous tableau {t[0][0], t[0][1], t[0][2], t[0][3], t[0][4]}, t+1 celle de 2ème sous tableau {t[1][0], t[1][1], t[1][2], t[1][3], t[1][4]} et ainsi de suite.

Ici, l'opération t+3 n'ajoute pas 3 à la valeur de t mais ajoute 3 fois le nombre d'octets correspondant à un tableau de 5 entiers ; à savoir 3*4*5 = 60 octets.

Tableaux de pointeurs:

```
int tab[3] ;
int    *ptab[3] = {tab, tab+1, tab+2} ; /* ptab est un tableau de 3 pointeurs d'entiers */
```

Les valeurs de ptab sont des adresses de données. ptab est donc un pointeur de pointeur car ptab pointe sur l'adresse de son 1er élément, qui est lui même une adresse.

Allocation dynamique de mémoire

Int tab[4][3] ; /* déclaration d'un tableau de 4*3 entiers */

Si on veut que le tableau change du taille d'une exécution à une autre, cela nous oblige à modifier le programme et le recompiler à chaque fois. Sinon, on déclare un tableau 1000*1000 entiers et n'utiliser que les premières cases. C'est du gâchis!.

Pour éviter cela, on fait appel à l'**allocation dynamique du mémoire** : au lieu de réserver de la place lors de la compilation, on la réserve pendant l'exécution du programme.

La fonction malloc( )

Pour l'utiliser il faut inclure la bibliothèque **<stdlib.h>**.

malloc(n) ; renvoie l'adresse d'un bloc de mémoire de n octets libres ou la valeur 0 s'il n'y a pas assez de mémoire.

```
int *p ;
p = malloc(50) ; /* fournit l'adresse d'un bloc de 50 octets libres */
                 /* et l'affecte au pointeur p */
```

Utilisation de l’opérateur sizeof( ):

D’une machine à une autre, la taille réservée pour un type change. Nous avons toujours besoin de la taille effective d’une donnée de ce type. C’est l’opérateur sizeof qui nous fournit ce renseignement.

```
int a[8] ;
printf(‘’taille de a : %d \n’’, sizeof a) ;
printf(‘’taille de 4.52 : %d \n’’, sizeof 4.52) ;
printf(‘’taille de Bonjour! : %d \n’’, sizeof  ‘’Bonjour!’’) ;
printf(‘’taille d’un float : %d \n’’, sizeof(float)) ;
```

Allocation dynamique pour un tableau à 1 dimension :

On veut réserver de la place mémoire pour un tableau de n entiers ou n est lu au clavier :

```
int n, *tab ;
printf(‘’taille du tableau : ‘’) ; scanf(‘’%d’’, &n) ;
tab = (int *)malloc(n*sizeof(int)) ;
```

malloc(n*sizeof(int)) : renvoie juste l’adresse d’un bloc de n fois sizeof(int). L’adresse retourné n’a pas de type. malloc() est dite de type générique.

C’es pourquoi il est nécessaire de préciser le type de données pour lesquelles l’adresse retournée est réservée.

(int *)malloc(n*sizeof(int)) est une adresse de type int.

tab = (int *)malloc(n*sizeof(int)) ; /* initialisation de tab par cette adresse*/

tab contient donc l’adresse de début d’un bloc de n entiers et on accède à la $i^{ième}$ valeur du tableau par tab[i] ou *(tab+i).

*(tab+i) est une variable de type int.

<u>Remarque :</u> Pour libérer l’espace mémoire réservé quand on en n’a plus besoin on utilise simplement l’instruction : free(p) ; Si on ne le fait pas cet espace mémoire reste inutilisable jusqu’à la fin d’exécution du programme. Ce qui peut conduire à une saturation de mémoire.

<u>Allocation dynamique pour un tableau à plusieurs dimensions :</u>

On veut réserver de la place mémoire pour un tableau de n*m entiers, ou n et m sont lus au clavier: On va utiliser des tableaux de pointeurs (i.e. des pointeurs de pointeurs).

```
int i, j, m, n ;
    float  **tab ;                  /* (a) */
    scanf(‘’%d%d’’, n, m) ;
    tab = (float **)malloc(n*sizeof(float *)) ;            /* (b) */
    for (i=0 ; i<n ; i++){
          tab[i] = (float *)malloc(m*sizeof(float)) ;                  /* (c) */
     for (i=0 ; i<n ; i++){
          for (j=0 ; i<m ; j++){
                   tab[i][j] =  5*i+j ;              /* (d) */
          }
    }
```

Explication :

(a) un tableau de pointeur est un pointeur de pointeurs. On peut déclarer au choix un tableau de pointeur : int *tab[6] ; ou un pointeur de pointeur : int **tab ;. Dans le cas de l’allocation dynamique de mémoire, comme on ne connaît pas la taille de du tableau dont on aura besoin, on déclare un ointeur de pointeurs.

(b) tab étant un tableau de pointeur de réels de type float, on réserve un bloc pouvant contenir n pointeur de type float. tab contient alors l'adresse de ce bloc.

(c) les tab[i] sont des sous tableaux de tab. On réserve donc pour chacun d'eux de la place pour m réels de type float. Au total, on a bien réservé de la place pour n*m réels de type float.

(d) Manipulation des éléments de tab comme ceux d'un tableau ''normal''.

Exemple1 :

```
/* Fonction qui affiche une matrice quelconque */
void  aff_mat(double **A,int l,int c)
{
 int i,j;
 for(i=0; i<l; i++){
    for(j=0; j<c; j++) { printf("%lf ",A[i][j]); }
    printf("\n");  }
 printf("\n"); }
```

Exemple2:

/* Fait le produit de la matrice a(m,n) par la matrice b(n,p) */
/* Le résultat est dans ab(m,p) */

```
void  mat_prod(double **a,double **b,double **ab,int m,int n,int p)
    {
      int i,j,k;
     for(i=0; i<m; i++) {
     for(k=0; k<p; k++) {
            ab[i][k]=0.;
            for(j=0; j<n; j++) { ab[i][k]+=a[i][j]*b[j][k]; }
                              }
                            }
}
```

```
double  **Mat_alloc_double( int nl,   int nc){
 /* nl : nombre de lignes, nc : nombre de colonnes */
 double  **mat;
 int i;
 mat = (double **)malloc(nl*sizeof(double*));
 if(mat==NULL){
   printf(‘’Mémoire insuffisante\n");
   exit;
                               }
 for(i=0 ; i<nl ; i++){
   mat[i]=(double*)malloc(nc*sizeof(double));
   if(mat[i]==NULL){
     fprintf(stderr,"Mémoire insuffisante\n’’);
     exit;                  }
                            }
 return mat;
}
```

```
void   mat_free(double **mat,  int nl, int nc ){
/*  **mat : pointeur sur la matrice, nl : nombre de lignes /*
/*       nc : Nombre de colonnes  */
  int i;
  for(i=0 ; i<nl ; i++) { free(mat[i]); }
  free(mat);
  return;
}
```

Allocation de la mémoire pour un tableau de réels à trois dimensions

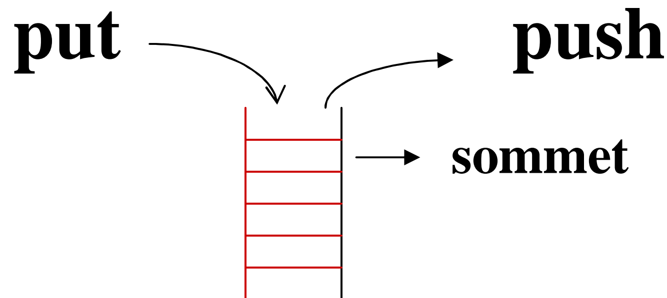
```
double    ***Td3_alloc( double ***mat, int n1,    int n2,    int n3   ) {
 int i,j;
 mat=(double***)malloc(n1*sizeof(double**));
 if(mat==NULL) {
    printf( ‘’Mémoire insuffisante\n");
    exit; }
 for(i=0;i<n1;i++){
    mat[i]=(double**)malloc(n2*sizeof(double*));
    if(mat[i]==NULL) {
    printf(‘’Mémoire insuffisante\n");
    exit ; }
    for (j=0;j<n2;j++){
     mat[i][j]=(double *)malloc(n3*sizeof(double));
       if(mat[i][j]==NULL){
         printf(‘’Mémoire insuffisante\n");
           exit ; }
                         }
                    }
 return mat;
}
```

Partie 2

Structures de Données Avancées

1) Définitions et Exemples

Une pile est une liste linéaire dont une seule extrémité (le sommet) est accessible (Visible).



Exemples : Pile d'assiettes.

Pile de dossiers.

2) Caractéristiques

- **L'extraction ou dépilement puis l'ajout se font uniquement au sommet de la pile.**
- **Une pile est en théorie un objet dynamique (en opposition à un tableau qui est statique).**

Représentation statique :

a) un tableau + une variable globale indiquant le sommet.

b) Un enregistrement avec deux champs { tableau, sommet }

Représentation dynamique : liste

Les piles suivent une discipline LIFO (Last In / First Out) (Le dernier entré est le premier servi).

Tout Problème utilisant cette démarche peut être donc simulé, dans sa résolution, par des piles.

De ce fait la manipulation est bien simplifiée puisqu'elle ne nécessite que deux fonctions :

- Une fonction pour ajouter un élément au sommet de la pile
- Une seconde pour le retirer

3) Les opérations de bases

On suppose que la pile est déclarée de la façon suivante :

```
typedef  type  elt ;
typedef   struct    pile  {
                          elt tab[MAX];
                          int sommet;
                         };
pile   p ;
```

cette représentation amène des restrictions supplémentaires à la fonction Insertion : empiler ne pas empiler si la pile est pleine!

Les fonctions sur les piles

```
#define MAX 100   /*hauteur de pile*/
```

Création d'une pile vide :

```
pile creer(){
pile p;
p.sommet = -1;
return p;
}
```

Tester si une pile est vide ou non :

```
int vide(PILE * pp){
* renvoie vrai si la pile pointée par pp est vide*/
return (pp->sommet == -1);
                                                    }
```

Insertion d'un élément :

```
void empile(elt e, pile * pp) {
/*empile e sur la pile pointée par pp */

        if (pp->sommet = = MAX-1) { printf (‘’Stack Overflow \n’’);
                                      exit;   }
        else  { (pp->sommet = pp->sommet+1;
                pp->tab[pp->sommet] = e ;}
        return ;
}
```

<u>Extraction d'un élément</u> : extrait (dépile) un élément du sommet et le retourne comme valeur de la fonction.

```
elt  depiler( pile  *pp ) {
 /* tester si la pile est vide */
      elt c;
      if (vide(pp)) { printf('' Stack Underflow \n'');
                     halt;  }
     else   {  c = pp->tab[pp->sommet] ;
                 pp-> sommet = pp->sommet – 1 ; }
    return( c ) ;
   }
```

4) Exemples d'applications

<u>Traitement des expressions mathématiques</u>

a+b)　　　,　　　)a+b(-c　　　,　　　((a+b)-c)

compteur : +1 si on ouvre et – 1 si on ferme

((a+b)-c) expression correcte

122221110

Une expression est correcte si et seulement si :

- **Le compteur est toujours >= 0 (on ne commence pas par un délimiteur fermant)**
- **Le compteur est à zéro à la fin de l'expression.**

Problème :

Cas ou on a plus d’un délimiteur.

Par exemple : (, {, [, …

On peut utiliser la méthode précédente en gardant la trace du nombre d’ouvrants et de fermants pour chaque délimiteur.

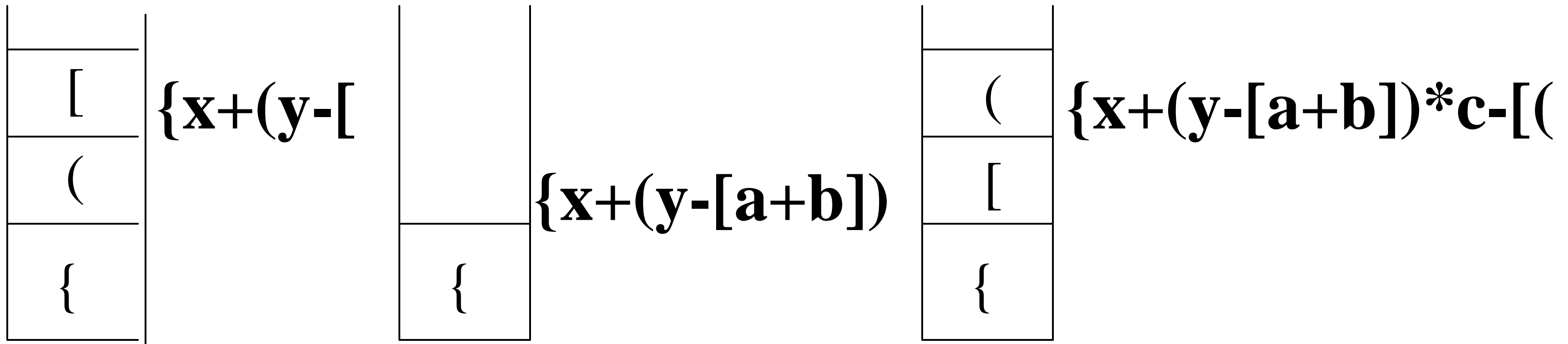
Dans ce problème :

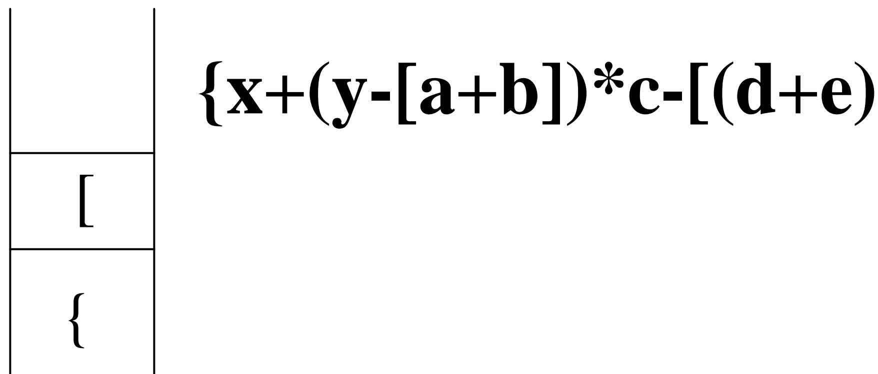
ouvrir correspond à empiler et fermer correspond à dépiler.

Le dernier délimiteur ouvert est le premier à être fermé, d’où la politique LIFO.

Ce problème peut donc être simulé par une structure de pile.

Exemple : {x+(y-[a+b])*c-[(d+e)]}





Ainsi l'expression est correcte.

Remarque :

- **Le problème d'Underflow (pile vide et on essaie de dépiler) correspond à une fermeture en plus.**

- **Il faut s'assurer toujours que le délimiteur qu'on ferme correspond bien (est de même type) à celui qu'on vient d'ouvrire.**

Choix de la structure de donnée

```
const max = 20 ;

type element = char ;
     pile = array[1..max] of element ;

var   p : pile ;
      s, symb : char ;
      valid : boolean ;
      sommet, pos : integer ;
      expr : string[80] ;
```

Fonction qui associe à chaque symbole fermant le symbole ouvrant correspondant en le retournant.

```
function ouvrant(symbf: char): char;
begin
        case symbf of
                ‘]’ : ouvrant :=‘[‘;
                ‘)’ : ouvrant :=‘(‘;
                ‘}’ : ouvrant :=‘{‘;
        end;
end;
```

```
Begin { Programme principal}

  valide := true;

  initialise ;

  pos:=1;

  write('Donner une expression arithmétique : ');

  readln(expr);

  symb:=expr[pos];
```

Driss EL Ouadghiri, FSMEK
dmelouad@gmail.com

```
repeat   { début de la boucle repeat }
   if    ( symb in [‘(‘,’[‘,’{‘] )   then
         begin
                 if pile_pleine(p) then write(‘Pile pleine’);
                 halt;
         end
   else  empiler(p,symb);
   if symb in [‘]’,’)’,’}’] then
        begin
                 if pile_vide(p) then write(‘ Pile vide’);
                 halt;
        end
    else   begin
                 s:= depiler(p);
                 if s <> ouvrant(symb) then valide := false;
           end;
```

```
    pos:=pos+1;
    symb := expr[pos];
until ( (pos > length(expr)) or not(valide)); { fin de repeat}
If (not(pile_vide(p)) or (not(valide))) then
    write('expression incorrecte')
else writeln('expression correcte');
readln;
end. {fin du programme}
```

TP1 et devoir à rendre
Traitement des expressions arithmétiques :

Le but de TP est de déterminer si une expression arithmétique contenant les délimiteurs {, (, [,],), } est correcte ou non. Une expression est correcte si à chaque délimiteur ouvrant {, (, [correspond un délimiteur fermant],), } de même type. Le dernier symbole ouvert doit être le premier à être fermé. Ce qui peut être simulé par une pile avec les opérations :

empiler : ouvrir un délimiteur d'un type donné

dépiler : fermer le délimiteur ayant le même type que celui au sommet de la pile.

Énoncé : Écrire un programme faisant appel à des fonctions pour résoudre ce problème (traduire le programme Pascal en C).

Chapitre 2 : La récursivité

<u>Définition et exemples</u>

Un objet est dit récursif s'il est défini en fonction de lui même.

Une fonction est dite récursive si elle a la faculté de s'appeler elle même.

La récursivité est une manière simple de résoudre un certains nombre de problèmes et surtout en mathématiques.

<u>Remarque:</u>

Deux modes de récursivité :

- **Direct : la fonction fait appel à elle même d'une façon directe.**
- **Indirect (croisé) : la fonction fait référence à une procédure ou une fonction qui lui fait référence.**

```
type  F ( arguments )
    debut
                initialisation ;
                appel à F avec une condition d'arrêt ;
                instruction ;
    Fin
```

<u>Exemple</u> : Factoriel

n! = n*(n-1)!

0! = 1 (condition d'arrêt)

```
unsigned   long   fact( int  n)
{
    if ( n < 0) {
       printf('' Un entiers négatif n'a pas de factoriel \n'');
       exit;
   }
   else  if  (n == 0 || n == 1) {
            return (1) ;
   }
   return(n* fact(n-1)) ;
}
```

Driss EL Ouadghiri, FSMEK
dmelouad@gmail.com

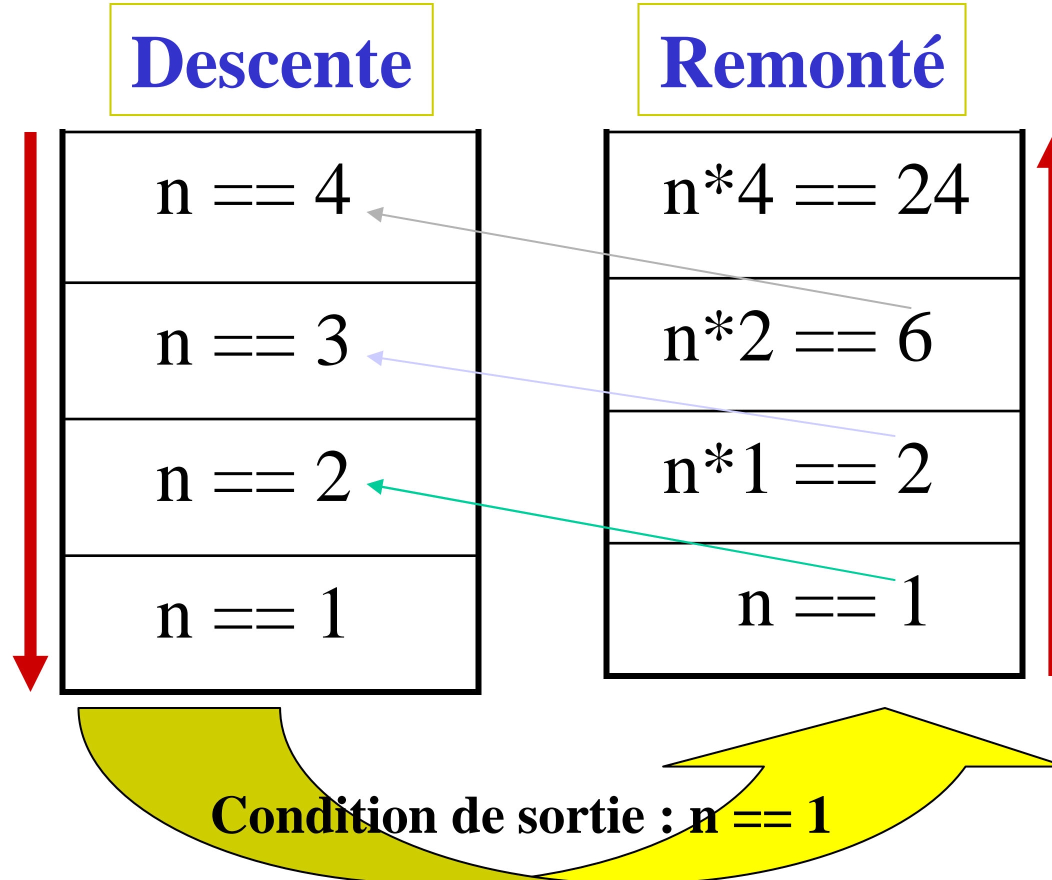
Le dernier **return** est un appel récursif.

A chaque appel, le paramètre de la fonction est diminué de 1 Jusqu'à ce que n = = 1, ce qui est la condition de sortie.

Ce n'est pas fini là. Lorsque la fonction rencontre la condition de sortie, elle remonte dans tous les appels précédents pour calculer *n* avec la valeur précédemment trouvée !

Les appels des fonctions récursives sont en fait *empilées* dans une pile système. Une fonction de ce type possède donc deux parcours: la *phase de descente* et la *phase de remontée*.

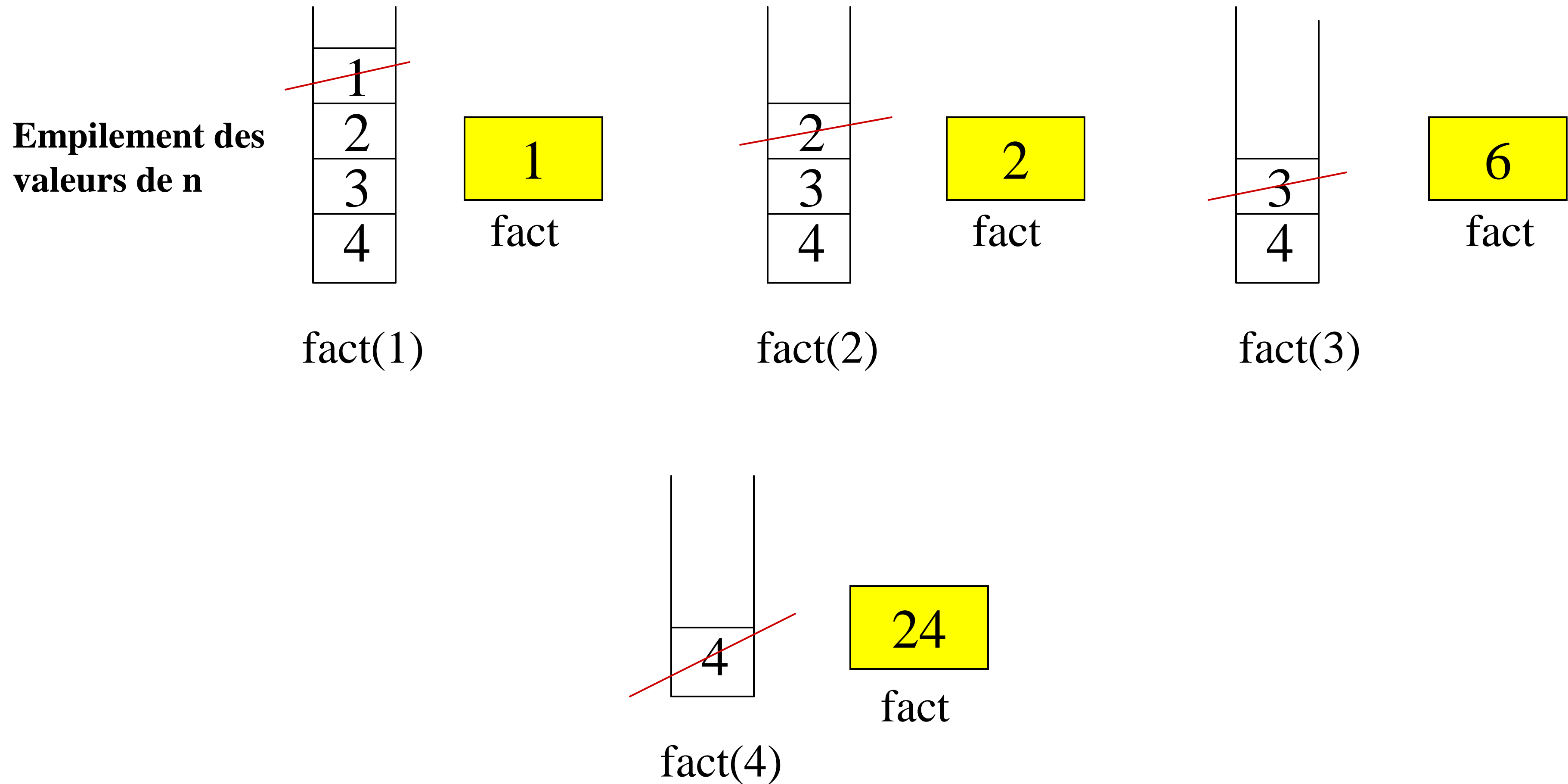
Le schéma suivant illustre cela :
On suppose que la fonction est appelée par n = = 4.



C’est au moment où la condition de sortie est vraie, que les appels empilés sont dépilés au fur et à mesure de la remontée.

Autrement, voyant de près ce qui se passe dans la pile système.

On considère toujours l'exemple de factoriel avec n = 4



Précautions :

Les fonctions récursives reste un moyen assez puissant pour résoudre certains problèmes et de façon élégante. Aussi, elles pressentent des dangers et ce pour plusieurs raisons :

- Dépassement de capacité : Essayer de stocker un nombre plus grand que ce que peut contenir le type de votre variable.

- Débordement de pile (Stack Overflow) : Ceci l'une des causes les plus souvent rencontrés dans le plantage de programmes avec des fonctions récursives. En effet, les appels récursifs de fonctions sont placés dans la pile du programme. Cette pile est d'une taille assez limité car elle est fixée une fois pour toutes lors de la compilation.

Dans la pile sont non seulement stockés les valeurs des variables de retour mais aussi les adresses des fonctions. les données sont nombreuses et un débordement de la pile peut très vite arriver ce qui provoque des sorties anormales du programme.

Qu’on rencontre souvent le problème de « ‘Stack Overflow’, une approche itérative sera préférable qu’une approche récursive. L’approche récursive demande beaucoup de moyens en ressources alors l’approche itérative, telle une boucle **for,** est bien moins coûteuse en terme de ressources et est bien plus sûre, sauf dans le cas d'un dépassement de capacité bien sûr.

Voyons l’approche itératif pour calculer de factoriel d’un entier.

```
unsigned long fact_iter (int n) {
unsigned long f = 1;
short int i ;
   for (i = 1; i <= n; i++)
   {
   f *= i;
   }
return f;
}
```

Cet approche est moins lisible que celle récursive. Un dépassement de capacité est aussi possible. Le seul avantage ici réside dans le fait qu’on risque jamais le débordement de la pile.

Devoir et TP

Écrire deux procédures, une itérative et l'autre récursive, permettant de donner la représentation binaire d'un entier positif.

<u>Remarque :</u>

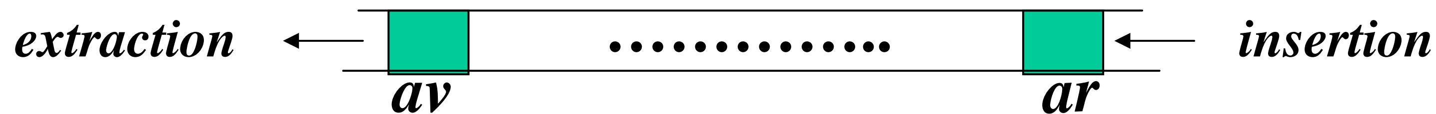
la représentation binaire d'un entier *n* positif est celle de *n* div 2 suivi par 0 si n est paire et de 1 si n est impaire.

Chapitre 3 : Les Files

<u>Définition</u> :

Une file est une liste linéaire ou toute les insertions se font par une extrémité (queue, arrière) et toutes les extractions se font par l'autre extrémité (avant, tête).

Principe: ajout à un bout et retrait à un autre



Nombre d'éléments dans la file = = *ar – av + 1*

<u>Exemple</u> :

File d'attente devant un bus, devant un guichet automatique, etc …

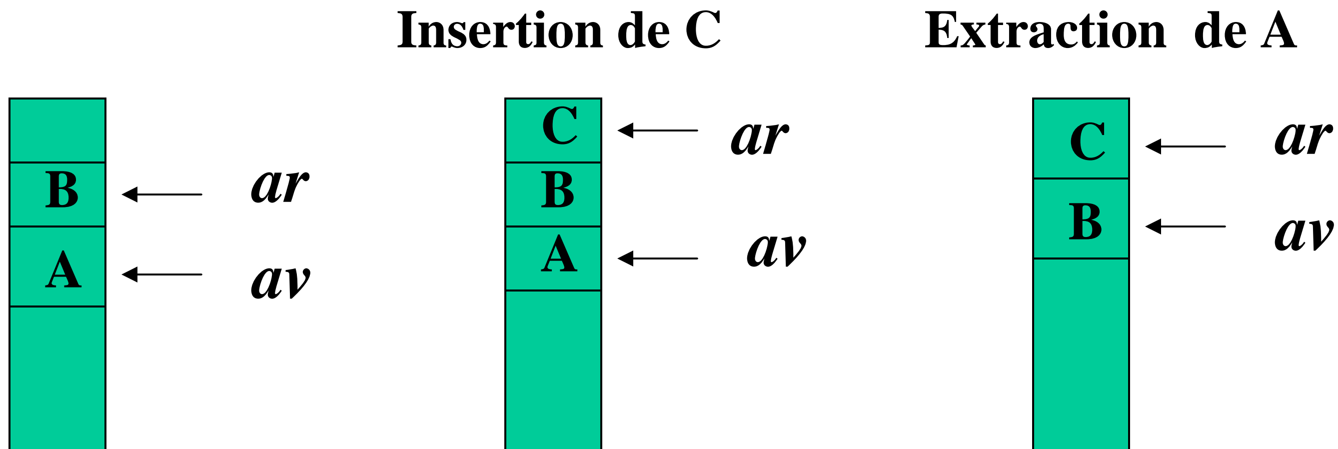
Une file suit la discipline FIFO (First In, First Out), le premier arrivé est Le premier servi.

Une file est représentée par un enregistrement contenant les champs suivants :

- **un tableau**
- **indice du premier élément**
- **indice du dernier élément**

ou bien par un tableau et deux variables globales indiquant les indices du premier élément et de dernier élément de la file (la tête et la queue de la file).

Opérations sur une file :



Au départ quand la file la file est vide on a : av = = 0 et ar = = -1.

En général file vide est exprimé par : ***ar < av***

```
#define MAX 100   /*hauteur de la file*/
```

Création d'une file vide :

```
file créer_file_vide(){
file f ;
f.av = 0 ;
f.ar = -1 ;
return f ;
}
```

Tester si une file est vide ou non :

```
int file_vide(file * pf){
/* renvoie vrai si la file pointée par pf est vide*/
return (pf->ar  <  pf->av );
}
```

Insertion d’un élément dans la file :

```
void  insert_file (elt e, file * pf) {
    if (pf->ar = = MAX-1) { printf (‘’On ne peut pas insérer dans la file \n’’);
                                      exit;   }
    else if (file_vide(* pf)) {  pf->tab[0]=e;
                                      pf->ar=0;  }
         else {  pf->ar  =  pf->ar + 1 ;
                    pf->tab[pf->ar] = e ; }
    return;
}
```

<u>Extraction d’un élément dans la file</u> : extrait (dépile) un élément du sommet et le retourne comme valeur de la fonction.

```
elt  extract_file( pile  *pf ) {
/* tester si la file est vide */
         elt c;
         if (file_vide(pf)) { printf(‘’ File vide \n’’);
                              exit;  }
        else   {  c = pf->tab[pf->av] ;
                  pf-> av = pf->av + 1 ; }
        return( c ) ;
}
```

<u>Remarque</u> : nbr_elt_file == *ar-av*+1

<u>Problème</u> :

Du fait que les deux variables *av* et *ar* sont toujours incrémentées, on arrive à un Overflow (dépassement de capacité) même si la file n'est pas saturée. On ne peut pas insérer un autre élément dans la file.

1) **Une première solution serait de modifier l'opération extraire. Lorsqu'un élément est extrait toute la file est poussée vers l'avant.**

Si on ignore la possibilité d'Underflow, on écrit extraire :

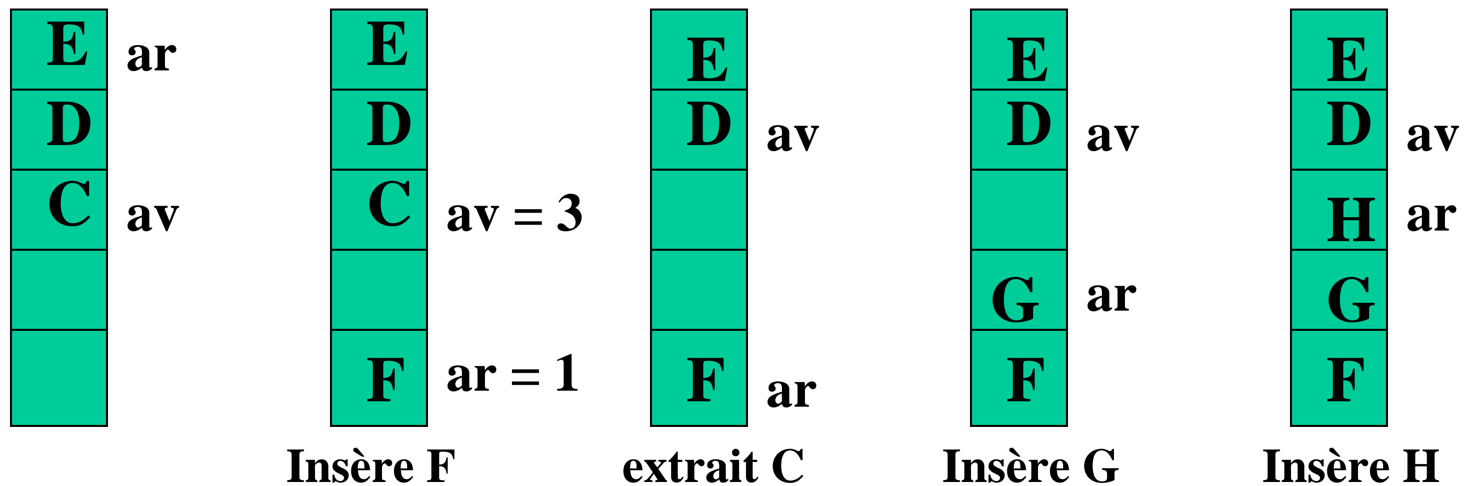
```
c = pf->tab[0] ;
for( i = 0 ; i < ar-1 ; i++) {
        pf->tab[i] = pf->tab[i+1] ; }
        ar := ar-1 ;
```

Remarque : On a plus besoin, dans cette situation, du premier élément (tête). Elle est toujours au début de la file (tableau).

Inconvénient : Lorsqu'il s'agit d'un tableau de grande taille, l'extraction d'un élément nécessite le déplacement de tous les éléments du tableau.

2) Une seconde solution consiste à voir la file (tableau) comme circulaire : tab[0],tab[1], …,tab[MAX-1] , tab[0]…

<u>Exemple</u> :



Il n'y a pas de perte de place, on exploite tout le tableau.

on a un Overflow lorsque, $ar = av-1$

TP N° 3

Problème : Simulation d’une file d’attente

On désir faire des statistiques sur une station de bus, en supposant q’une seule ligne passe par cette station.

Les personnes qui arrivent pour prendre le bus forment ainsi une file d’attente ne dépassant pas 20 personnes.

L’arrivée d’un bus se fait par l’affichage du nombre de places disponibles.

<u>Questions</u> :

1°) Écrire la fonction qui lit l'heure d'arrivée d'une personne et qui l'insère dans la file d'attente?

2°) Écrire la fonction qui enregistre l'arrivée d'un bus et qui met à jour la file d'attente?

3°) Écrire la fonction qui affiche l'état courant de la file?

4°) Écrire un programme qui appelle les fonctions des questions précédentes?

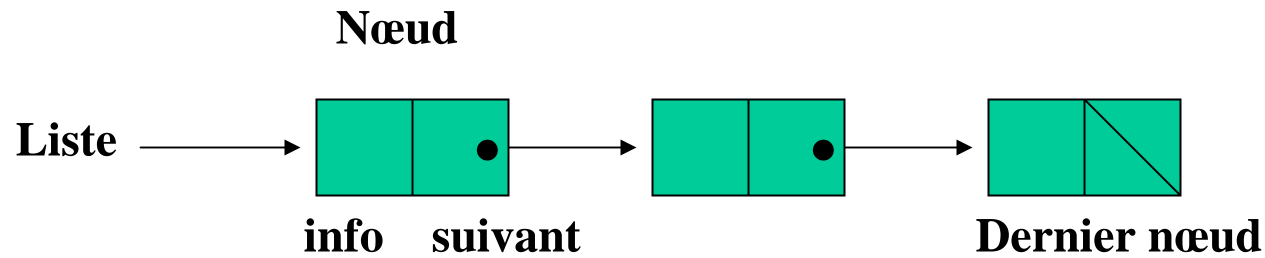
Chapitre 5 : Les listes chaînées

Les files et les piles sont des listes particulières, elles sont ordonnées linéairement. La représentation séquentielle, dans la mémoire, préservait cet ordre.

<u>Inconvénients</u> :

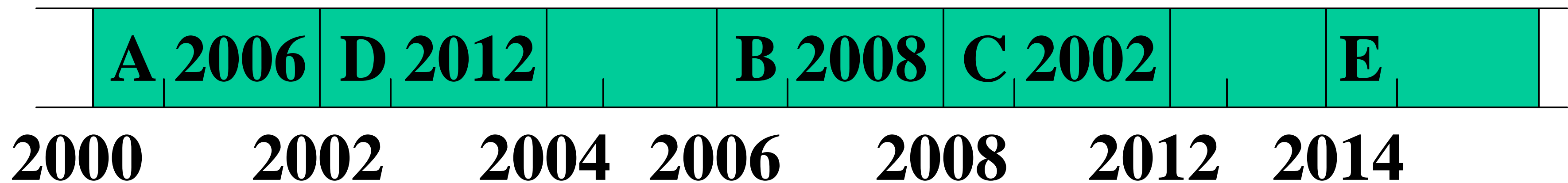
1) **Overflow**
2) **Capacité non utilisée au maximum**

Supposons que l'ordre est gérer par le programmeur. Pour chaque élément de la liste on resserve un champs qui contient l'adresse de l'élément suivant (pointeur).



Liste : est un pointeur externe qui nous permet d'accéder à la liste.

<u>Exemple</u> : Dans le cas d'une liste chaînée, la représentation en mémoire n'est pas ordonnée. Une liste de 5 éléments 'A', 'B', 'C', 'D' et 'E' dans l'ordre pourra avoir, en mémoire, la représentation suivante :

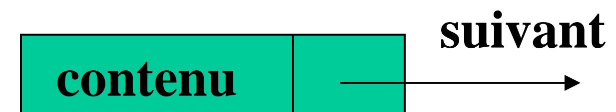


Liste = 2000

Manipulation des listes chaînées

Structure de donnée définissant un nœud :

```
typedef type elt;
typedef struct noeud {
                        elt contenu;
                        noeud *suivant;
                          };
noeud  *liste;  /*  Variable globale : pointeur externe sur la  liste */
```



Création d’une liste vide :

```
noeud  *creer_liste_vide(noeud  *pl)
{
pl->suivant = NULL;
return(pl) ; /* défini dans <stdio.h> */
}
```

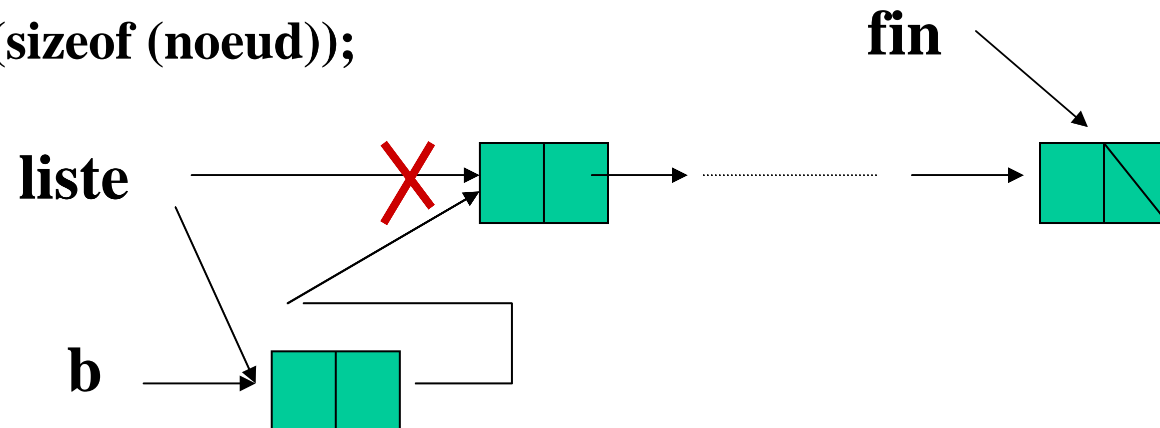
Tester si une liste est vide :

```
int liste_vide(noeud *pl) {
return (pl == NULL) ;
}
```

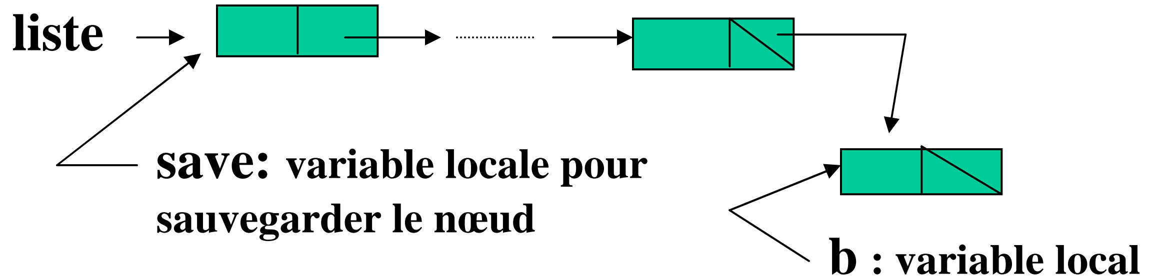
Ajout d'un élément au début de la liste :

```
void ajout_debut(elt x) {
noeud *b;
b = (noeud *)malloc (sizeof (noeud));
b->contenu = x;
b->suivant = liste;
liste = b;
}
```

fin->suivant = = NULL
fin est un pointeur externe sur la fin de la liste



<u>Insertion en fin de la liste</u> :




```
Void  ajout_fin( elt  x){
    noeud  *save, *b;
    b = (noeud *)malloc (sizeof (noeud));  /* construction du nœud à insérer */
    b->contenu = x;
    b->suivant = NULL;
    if (liste == NULL) { liste = p  ;}   /* liste vide*/
    else  {  /*liste non vide donc      recherche de dernier nœud de la liste */
            save = liste ;
            while (save->suivant <> NULL) {
            save = save->suivant ;
            save->suivant := b ;} /* fin de while*/
            }
return ;
}
```

Remarque :

Il est plus pratique et efficace pour des insertions répétées d'utiliser deux pointeurs externes :

- **liste : pointeur sur le début de la liste,**
- **fin : pointeur sur la fin de la liste,**

comme ça on n'a pas à parcourir à chaque fois la liste à la recherche du dernier élément.

Recherche d'un élément x dans une liste :

```
int   recher_liste-rec(elt x) {
    noeud  *a;
    a = liste;
    if (a == NULL) { return 0; }
    else if (a->contenu == x) { return 1; }
         else { return (recher_liste-rec(x)); }    /* appel récursif */
}
```

Longueur d’une liste :

```
int long_liste_rec( noeud *pl) {     /* version récursive */
    if (a = NULL) { return 0; }
    else { return (1 +  long_liste_rec(a->suivant));}
}

int long_liste_iter( ) {  /* version itérative */
     noeud *a;
    int long = 0;
    a = liste;
    while (a != NULL) {
            ++long;
            a = a->suivant;
    }
    return long;
}
```

Manipulation des liste a simple chaînage

On considère la représentation d'un polynôme en x, y et z.

Exemple : $5 x^2 + 3xy + y^2 + yz$

a) Représentation statique (par tableau).

	Y^0 $z^0z^1z^2$	Y^1 $z^0z^1z^2$	Y^2 $z^0z^1z^2$
x^0	000	010	100
x^1	000	300	000
x^2	500	000	000

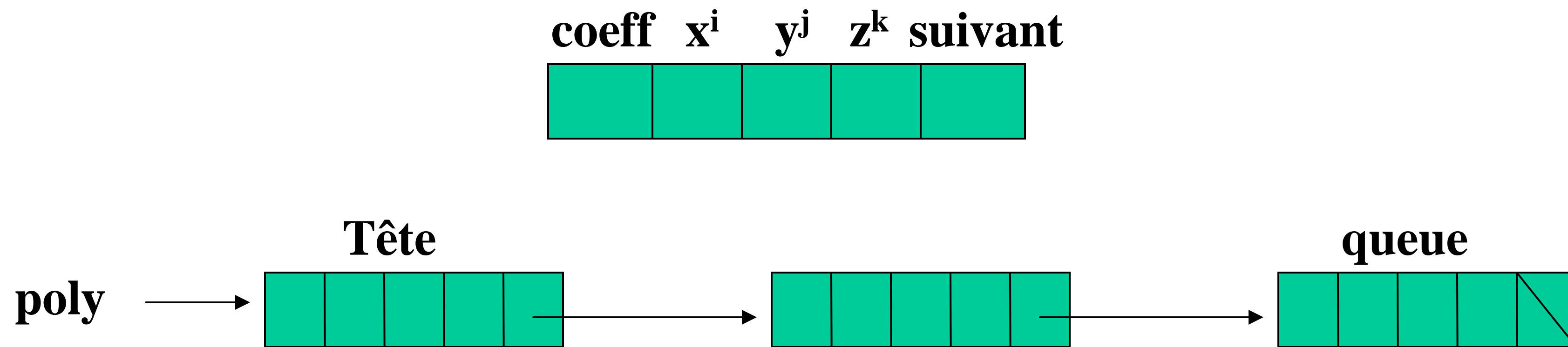
T[i,j,k]

Coefficient de $x^i y^j z^k$

Grosse perte de mémoire (matrice très creuse).

b) Représentation par liste chaînée

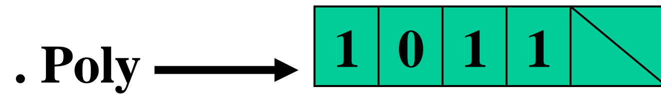
Un nœud représente un monôme :

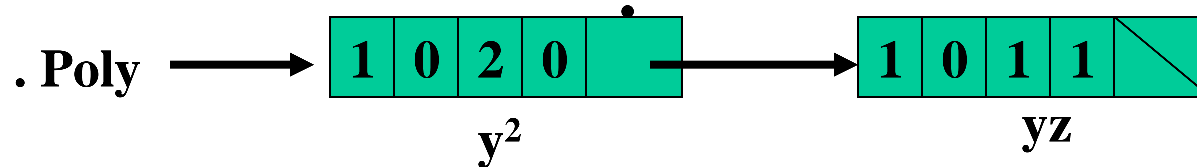



```
typedef    struct    noeud  {
                              float     coeff ;
                               int px, py, pz ;
                              noeud *suivant;
                               };
noeud poly, p;
```

La construction de polynôme peut se faire de la façon suivante :

. Poly → NULL



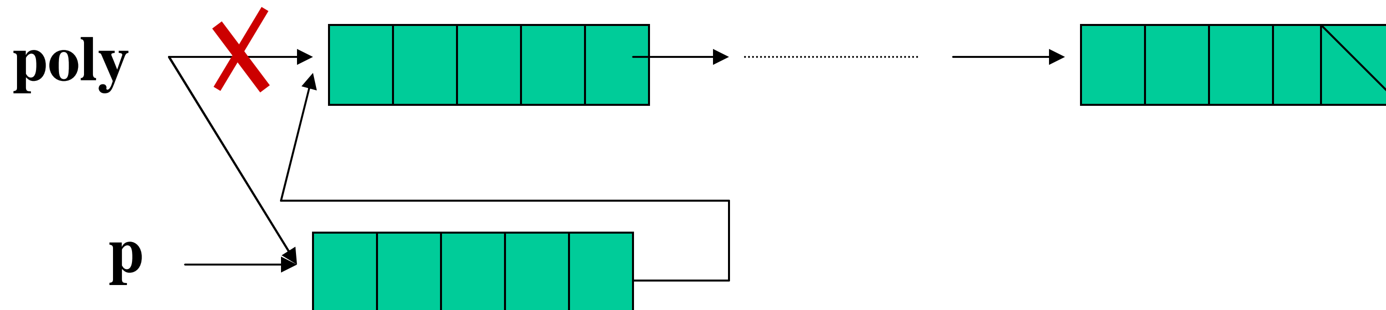


<u>Initialisation</u> :

Poly = NULL;

Poly = (noeud *)malloc (sizeof (noeud));

<u>Insertion en tête d'une liste :</u>



```
Void insert_en_tete(float c, int  i,j,k){
noeud *p;
    p = (noeud *)malloc (sizeof (noeud)); ;
    {charger le nœud pointé par p}
    p->coeff  = c ;   p->px := i ;  p->py = j ;  p->pz = k ;
    p->suivant := poly ;
    poly = p ;
}
void  main(){
    Poly = NULL;
    Poly = (noeud *)malloc (sizeof (noeud));
    { creation du polynome : p(x) = 5 x2 + 3xy + y2 + yz }
    insert_en_tete(1,01,1);
    insert_en_tete(1,02,0);
    insert_en_tete(3,1,1,0);
    insert_en_tete(5,1,0,0);
    return;
}
```

Insertion en fin de la liste :

```
void  inert_fin(float c ;  int i, j, k){
     noeud  *save, *p ;
     p = NULL;
     p = (noeud *)malloc (sizeof (noeud));
     p->px = i ; p->py = j ; p->pz = k ; p->coeff = c ;  p^.suivant = NULL ;
     if (poly == NULL) { poly = p ; }  {liste vide}
     else  {liste non vide, donc recherche de dernier nœud de la liste}
              save = poly ;
              while (save->suivant != NULL) {
                             save = save->suivant ;
                             save->suivant = p ;
             } /* fin while*/
     return;
} /* fin de la focntion insert_fin*/
```

Exercice :

Représentation d'une phrase ou d'un texte par une liste chaînée.

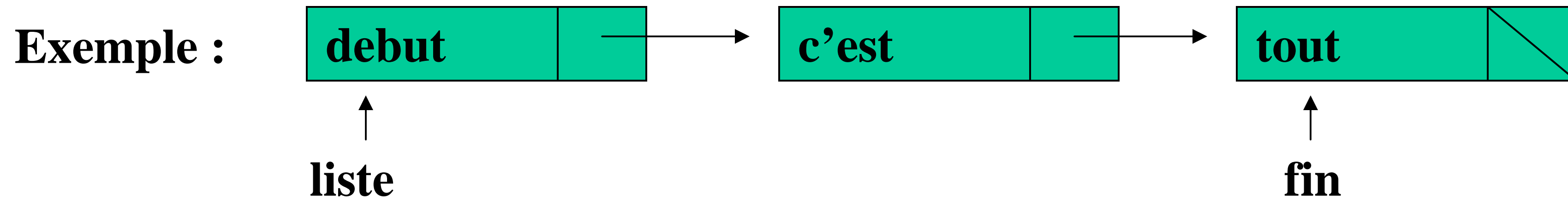
Considérant la structure de données suivantes :

```
typedef    struct      nœud {
                             char  mot[20] ;
                             noeud *suivant ;
                              };
```

Réaliser les opérations suivante :

1) **Créer la liste chaînée (initialisation du premier élément).**
2) **Insérer de nouveaux nœuds à la fin de la liste.**
3) **Parcourir le texte en affichant le texte.**
4) **Afficher les nœuds (mots) dans le sens inverse.**

Exemple :

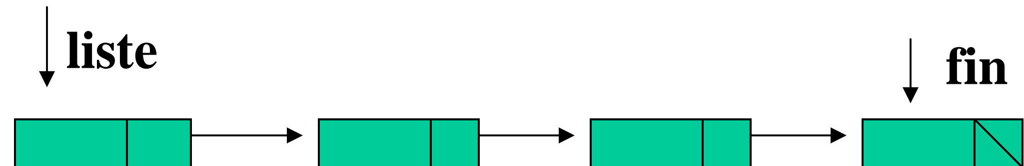


l'affichage à l'envers : **tout c'est debut**

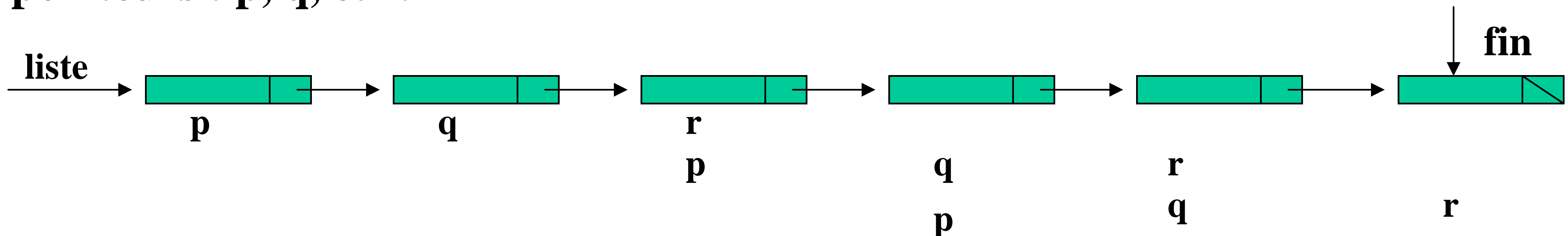
```
noeud      *nouveau, *liste, *fin;
           char   ch[20];
void main() {
        /* initialisation de la liste */
        liste = (noeud *)malloc (sizeof (noeud));
        liste = NULL ;
        fin = liste ;
        printf(‘’Entrer des mots et  « stop »  pour terminer’’);
```

```
            while  (ch !=  ‘’stop’’) {
                        scanf(‘’%s’’, ch) ;
                        insert_fin(ch) ;
                }
            affichage(liste);
            affichage_envers ;
return;
}
 void     insert_fin(char  s[20] ) {
    noeud *nouveau ;
    nouveau = (noeud *)malloc (sizeof (noeud));
    Nouveau->suivant = NULL;
    nouveau->mot = s ;
    fin->suivant = nouveau ;
    fin = nouveau ;
return,
}
```

```
void    affichage(noeud  *p){
        /* affiche tous les mots de la liste à partir du nœud pointé par p */
    while (p != NULL) {
                printf(‘’%s’’, p->mot) ;
                p = p->suivant ;
                }
return;
}
```



Pour écrire une procédure itérative affiche_envers, on a besoin d'au moins 3 pointeurs : p, q, et r.



```
void   affiche_envers() {
 neoud  *p, *q, *r ;
    p = (noeud *)malloc (sizeof (noeud));
    q = (noeud *)malloc (sizeof (noeud));
    r = (noeud *)malloc (sizeof (noeud));
    p = liste ;       q = p->suivant ;         r = q->suivant ;
    p->suivant = NULL ;       q->suivant = p ;
    do {
          p = q ;
          q = r ;
          r = r->suivant ;
          q->suivant = p ;
        }
    while  (r != fin) ;
    affiche(fin) ;
return;
}
```

L'énumération dans une liste

Consiste à passer en revue tous les nœuds de la liste et à effectuer sur chacun d'eux une action spécifique :

```
void   enumeration() {
    noeud  *p ;
     p = (noeud *)malloc (sizeof (noeud));
     p = liste ;
     while  (p != NULL ) {
                    action(p) ;
                    p = p->suivant ;
                    }
return;
}
```

Recherche dans une liste : l'information du nœud à chercher est connue

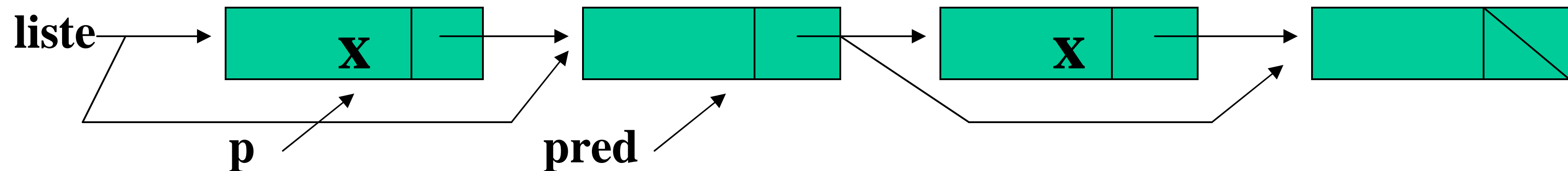
```
noeud   *recherche(info  x) {
    int   trouver ; /* booléen*/
    noeud    *p;
    p = liste ; trouver = 0 ;
    while ((p != NULL) and (trouver == 0)) {
                        if ( p->info = x ) { trouver = 1 ; }
                        else { p  = p->suivant ; }
                                            } /* fin while */
    return(p);
}
```

Remarque :

Cette fonction recherche la première occurrence de x, renvoie un pointeur p sur le nœud trouvé ou NULL sinon.

Retrait d’un élément d’une liste chaînée

1) valeur du nœud connue




```
void   retrait1(info x) {
 noeud  *p, *pred ;
 int trouve ;
    if (liste == NULL )  { printf(‘‘liste vide’’) ; }
    else if (liste->info = x) {            /* x se trouve dans le premier nœud */
                p = liste ; liste = p->suivant ; free(p) ;
                              }
        else {                             /* x se trouve au milieu ou à la fin de la liste */

        pred = liste ; p = pred->suivant ; trouve = 0 ;
                }
```

```
while (p != NULL) and ( trouve == 0 ) {
        if (p->info = x) { trouve = 1 ; }
        else {
                pred = p ; p = p->suivant ;
            }
                                } /* fin de while */
        if (p ==  NULL ) { printf(‘’Information  inexistante’’) ;}
        else {
                pred->suivant = p->suivant ; free(p) ;
            }
return;
}
```

2) Pointeur sur le nœud à enlever connu : vue que le lien de chaînage du prédécesseur de p doit être interrompu, on est obligé de chercher l'adresse du pointeur pred.

```
void   retrait2(noeud  *p){                    /*p pointe sur le nœud à supprimer*/
    noeud *pred ;
    if  (p == liste) { liste = liste->suivant ; }
    else   { pred = liste ; }
    while (pred->suivant != p) {
                    pred = pred->suivant ;
                                }
    pred->suivant = p->suivant;

return;
}
```

Remarque : Dans le Programme principal, si p == NULL ce n'est pas la peine d'appeler la procédure, si non retrait2(p); free(p) ;

TP

Soit le polynôme suivant :

$$2x^2+5xy+y^2+yz$$

Chaque terme du polynôme est constitué de :

- **la puissance de x**
- **la puissance de y**
- **la puissance de z**
- **le coefficient**

Pour représenter un polynôme (en x, y et z) sous forme de liste, on suppose que le terme pointé par p précède le terme pointé par q si :

puiss-x(p) > puiss-x(q) Ou puiss-x(p) = puiss-x(q)

Et

puiss-y(p) > puiss-y(q) Ou puiss-x(p) > puiss-x(q)

Et

puiss-y(p) = puiss-y(q)

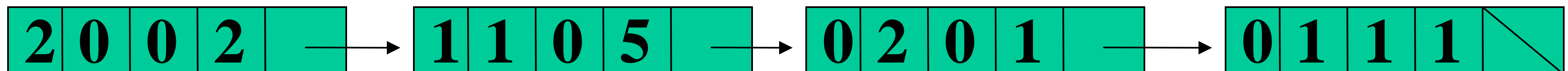
Et

puiss-z(p) > puiss-z(q)

noeud

coefficient	puiss-x	puiss-y	puiss-z	lien

L'exemple précédent est représenté par :



Énoncé :

1- Écrire la fonction d'insertion d'un terme en tête de la liste, pointé par un pointeur de tête liste.

2- Écrire la fonction d'insertion d'un terme en fin de la liste, pointé par un pointeur de tête liste.

3- Écrire la fonction d'insertion d'un terme en fin de la liste, pointé par un pointeur de tête liste et un pointeur de fin fin.

4- Écrire la fonction d'insertion d'un terme dans la liste, pointé par un pointeur de tête liste, en conservant l'ordre de la liste.

5- Écrire la fonction de suppression d'un terme de la liste.

6- Écrire le programme principal faisant appel aux fonctions précédentes.

Chapitre 6 : Les listes à double chaînage et circulaires

1- Liste à double chaînage

Beaucoup d'applications nécessitent le parcourt de la liste dans les sens.

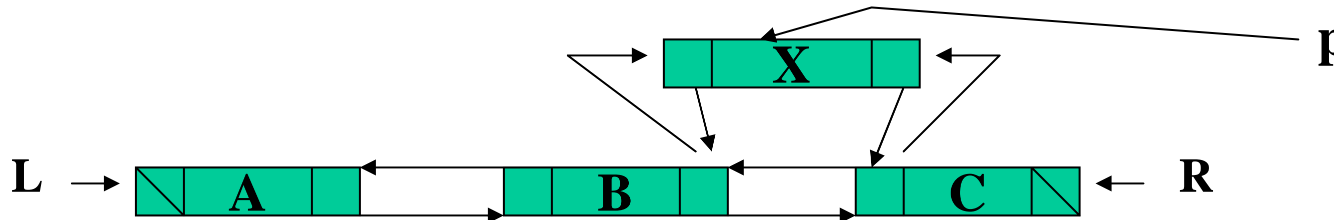
```
typedef   struct  noeud { noued *G ;
                          type : info ;
                          noeud  *D ; };
```

Liste à un élément :

Liste vide L == R == NULL

Liste quelconques :

Insertion d'une info X :



```
void  insert_ch( nœud *N ; info   x) {
    /*insertion d'une information x avant un nœud d'adresse  N*/
    noeud  *pred , *p ;
    p = (noeud *)malloc (sizeof (noeud));
    p->info = x ;
    if ( R == NULL ) and (L == NULL ) {
                                                    L = p ;
                                                    R = p ;
                                                    p->D = NULL ;
                                                    p->G = NULL ;
                                                   }
    else    if (L==N)  {
                              /*insérer avant le premier nœud*/
                              p->G = NULL;  L = p ; p->D = N ; N->G = p ;
                        }
          else    {
                         pred = N->G ; pred->D = p ; p->G = pred ;
                         N->G = p ; p->D = N ;
                     }
}
```

2- Les listes circulaires

• Le dernier nœud contient un pointeur sur le premier plutôt que la valeur NULL.

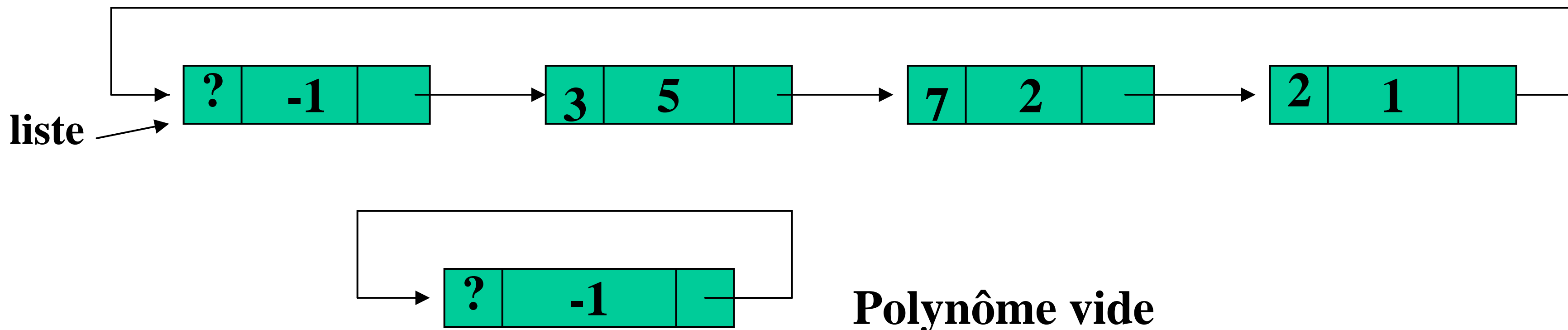
• les listes circulaires permettent comme les listes à double chaînage d'accéder au prédécesseur d'un nœud donné.

• Elles permettent aussi de simplifier les opérations d'insertion et de suppression.

Remarque :

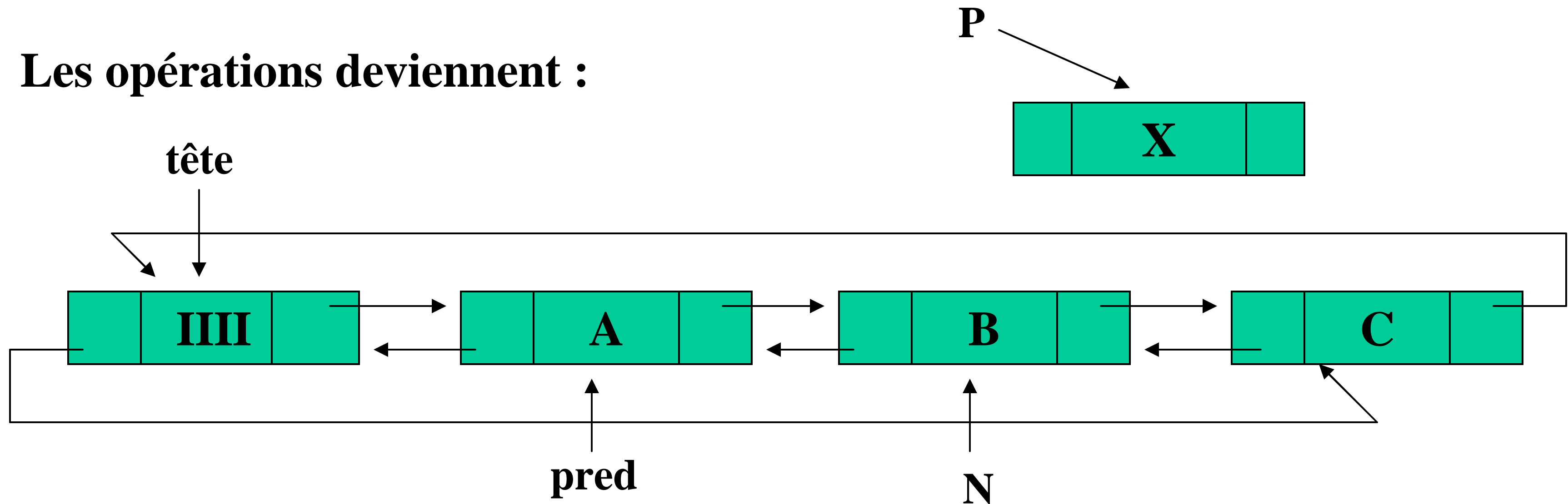
1) Le test de liste vide sera éliminé par l'adjonction d'un nœud en tête.

$$P(x) = 3x^5 + 7x^2 + 2x$$



2) Le cas des nœuds d’extrémité sera éliminé (implantation en liste circulaire)

Les opérations deviennent :



Insertion :

```
{
        pred = N->G ;
        pred->D = p ;
        p->G = pred ;
        p->D = N ;
        N->G = p ;
}
```

Retrait d’un nœud d’adresse M :

(On ne peut pas supprimer le nœud d’entête)

```
if (M !=  tete)  {
                pred = M->G ; succ = M->D ;
                pred->D = succ ;
                succ->G = pred ;
                free(M) ;
              }
else  printf(‘’Suppression du nœud d’entête’’) ;
```

Chapitre 7 : Les structures de données non linéaires (SDNL)

Les structures de données non linéaires permettent de réaliser des liens autre que l'adjacence : arbres, graphes…

I- Les graphes

Un graphe fini est un ensemble de points appelés sommets (nœuds) et d'arrêtes reliant ces sommets.

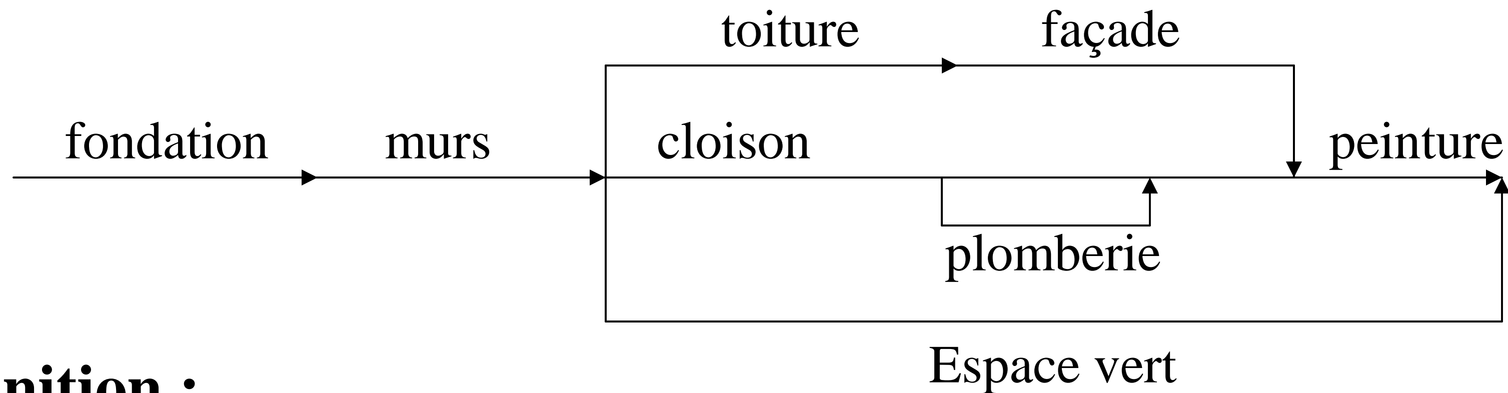
Exemple :

- Réseau Routier

  Sommets : nœuds du réseau (croisements, rond-points)

  Arrêtes : axes de communication.

- Graphe d'ordonnancement qui exprime les contraintes d'antériorité entres les différentes taches d'un projet.



Définition :

- Un graphe est orienté si le sens des arrêtes (appelés arcs) est important.
- Degré d'un nœud V:
  - degré d'entrée : nombre d'arrêtes aboutissant à V.
  - degré de sortie : nombre d'arrêtes partant de V.
- un chemin reliant deux sommets i et j est une suite d'arcs qui commencent en i et qui se terminent en j.
- La longueur d'un chemin est le nombre d'arcs qui le constituent.
- Un cycle est un chemin dont l'extrémité est égale à l'origine.

II- Les Arbres

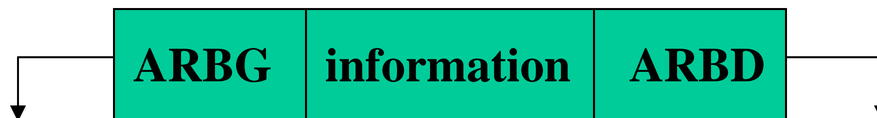
- **Adaptés à la représentation naturelle d'informations homogènes organisés.**
- **On les rencontre :**

- Algorithmique (méthode de tri performant, gestion d'informations en table).

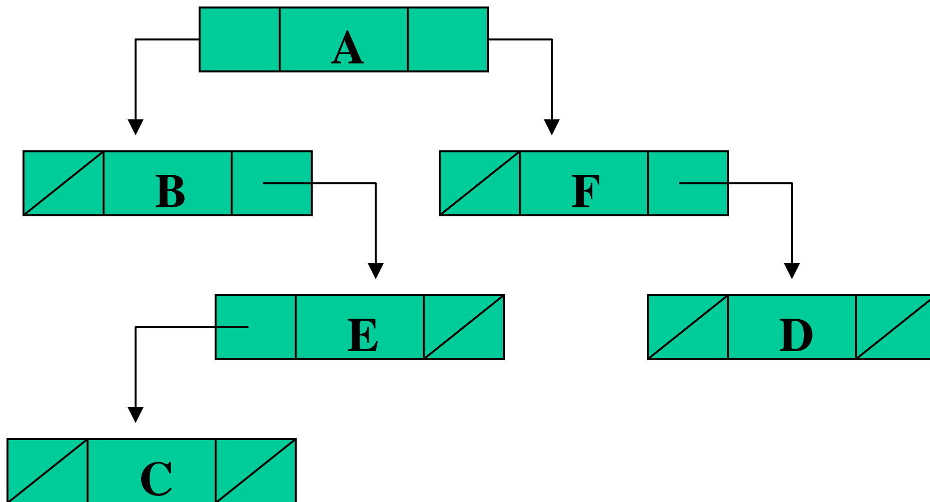
- intelligence artificielle (Décisions, Démonstrations)

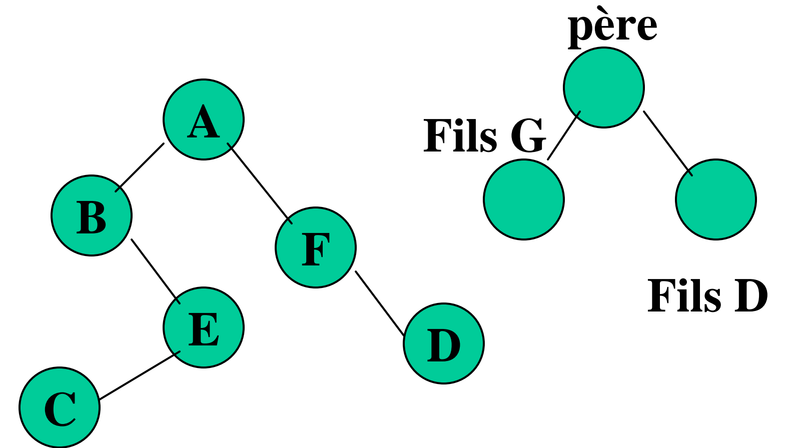
- Compilation (arbres syntaxiques)

Dans un arbre binaire un nœud est représente par :



- **Un arbre binaire est un ensemble d'un nœud particulier appelé racine et d'un certain nombre de nœud qui y sont reliés et ayant la représentation précédente.**





- Nœud terminal (feuille) : tout nœud qui n'as pas de fils gauche et de fils droit.

- les arbres binaires

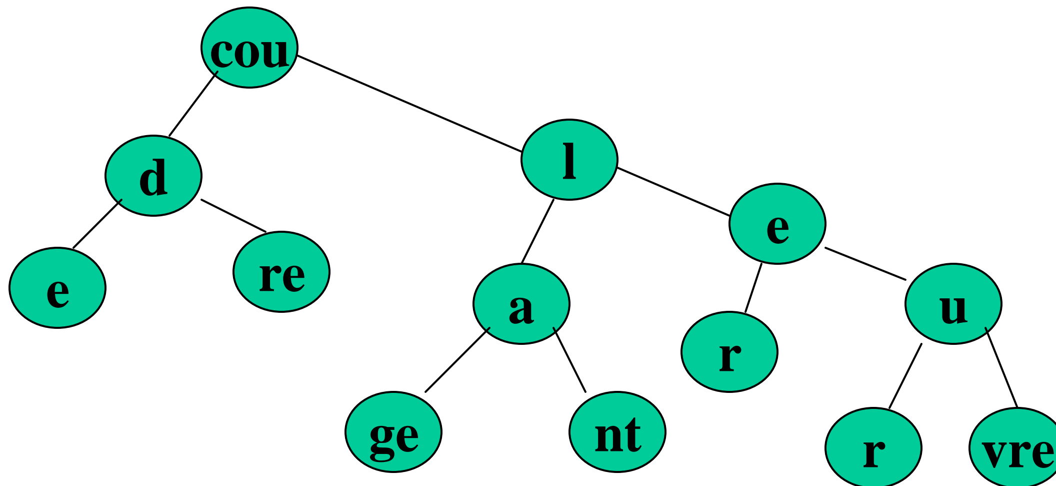
Déclaration :

```
typedef   struct   nœud {
                        nœud  *G ;
                        type info ;
                        noeud  *D  ;
                      }
```

Implantation en tableau

Exemple : Implantation d'un lexique.

Les mots qui commencent par une même chaîne de caractères sont regroupés sous celle-ci.



coude, coudre, coulage, coulant, couler, couleur, couleuvre

	info	G	D
1	Cou	2	4
2	d	3	5
3	e	0	0
4	re	0	0
5	l	6	9
6	a	7	8
7	ge	0	0
8	nt	0	0
9	e	10	11
10	r	0	0
11	u	12	13
12	r	0	0
13	vre	0	0

```
typedef    struct        nœud  {
                                 nœud *G ;
                                 type info;
                                 nœud *D;
                               } ;
int n ;
 nœud  arbre[n] ;
```

on peut utiliser aussi soit un tableau n x 3 soit 3 tableaux de n éléments chacun.

Exercices et TP

Une Association scientifique désire automatiser son fichier ‘adhérant’. Pour cela elle envoie à tous ces adhérents une une fiche de renseignements ayant la structure suivant :

pays	: 10	caractères
ville	: 15	caractères
Nom prénom	: 30	caractères
Adresse	: 40	caractères
Domaine d’intérêt	: 2	caractères

1- Donner une structure de données qui permet de répondre d’une manière performante aux questions suivantes :

a) Édition de la liste classée par ordre alphabétique des noms et prénoms de tous les adhérents d’une ville donnée d’un pays donné.

b) Édition de la liste classée par ordre alphabétique de tous les adhérents ayant un domaine d’intérêt donné.

2- Écrire une procédure permettant l’insertion d’un adhérant dans la structure.

3- Écrire une procédure permettant la suppression d’un adhérant de la structure.

4- Écrire les procédures relatives aux question a) et b).

Codes des domaines d’intérêt :

Informatique	:	1
Automatique	:	2
Physique	:	3
Sciences Naturelles	:	4
Biologie	:	5
Chimie	:	6