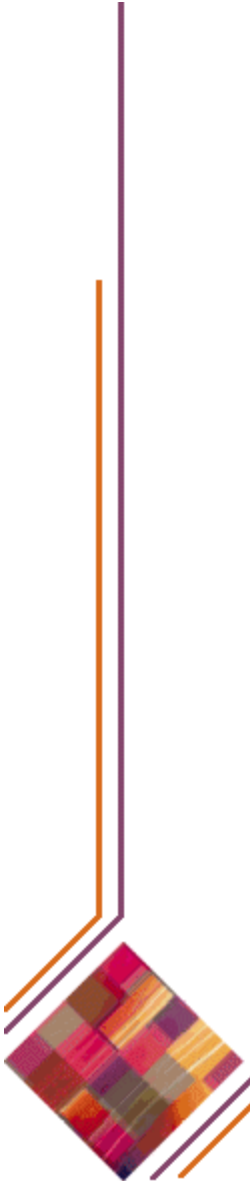


# Structures de données en langage C

**Abdellatif HAIR**

Université Sultan Moulay Slimane  
Faculté des Sciences et Techniques  
B.P. 523, Béni-Mellal, MAROC



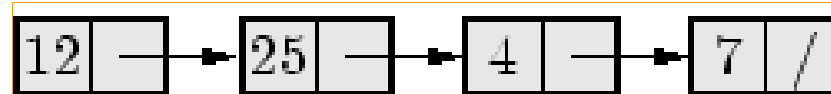
## 2. LES LISTES SIMPLEMENT CHAÎNÉES

- DÉFINITION ET REPRESENTATION
- CONSTRUCTION DU PROTOTYPE D'UN ÉLÉMENT DE LA LISTE
- OPÉRATIONS SUR LES LISTES SIMPLEMENT CHAÎNÉES
- UN EXEMPLE COMPLET



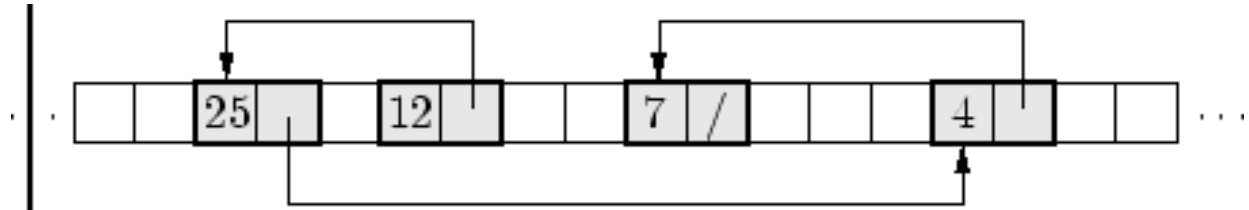
# 1. DÉFINITION ET REPRÉSENTATION

- ◆ Les **Listes Simplement Chaînées (LSC)** sont des structures de données semblables aux tableaux sauf que l'accès à un élément ne se fait pas par **index** mais à l'aide d'un **pointeur**
- ◆ L'allocation de la mémoire est faite au moment de l'exécution
- ◆ Dans une liste les éléments sont contigus en ce qui concerne l'enchaînement
- ◆ Chaque élément est lié à son successeur et il n'est donc pas possible d'accéder directement à un élément quelconque de la liste



# 1. DÉFINITION ET REPRÉSENTATION

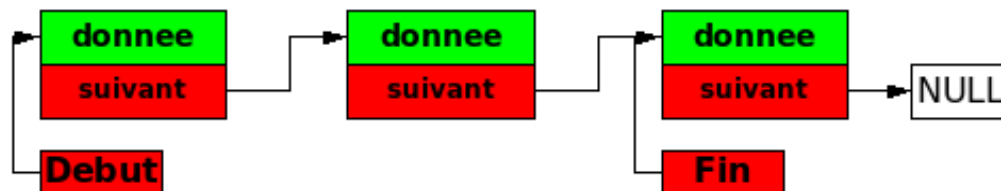
- ◆ Les tableaux éléments sont contigus dans la mémoire, tandis que les éléments d'une liste sont éparpillés dans la mémoire



- ◆ La liaison entre les éléments se fait grâce à un **pointeur**
- ◆ Le pointeur **suivant** du dernier élément doit pointer vers **NULL** (la fin de la liste)
- ◆ Pour accéder à un élément, la liste est parcourue en commençant avec le **début** (tête), le pointeur **suivant** permettant le déplacement vers le prochain élément
- ◆ Le déplacement se fait dans une seule direction, du premier vers le dernier élément

## 2. CONSTRUCTION DU PROTOTYPE D'UN ÉLÉMENT DE LA LISTE

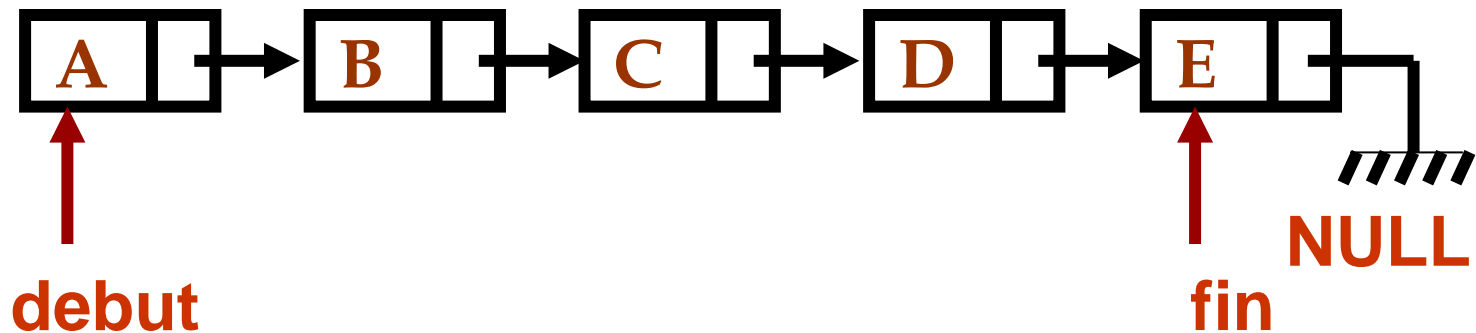
- ◆ Pour définir un élément de la liste le type **struct** sera utilisé
- ◆ L'élément de la liste contiendra un champ **donnee** et un pointeur **suivant**
- ◆ Le pointeur suivant doit être du même type que l'élément, sinon il ne pourra pas pointer vers l'élément
- ◆ Le pointeur **suivant** permettra l'accès vers le prochain élément
- ◆ Pour avoir le contrôle de la liste, il est préférable de sauvegarder certains éléments : le premier élément **debut**, le dernier élément **fin**, le nombre d'éléments **taille**



## 2. CONSTRUCTION DU PROTOTYPE D'UN ÉLÉMENT DE LA LISTE

◆ **Exemple 1** : représentation d'une liste de 5 éléments  
'A', 'B', 'C', 'D' et 'E'

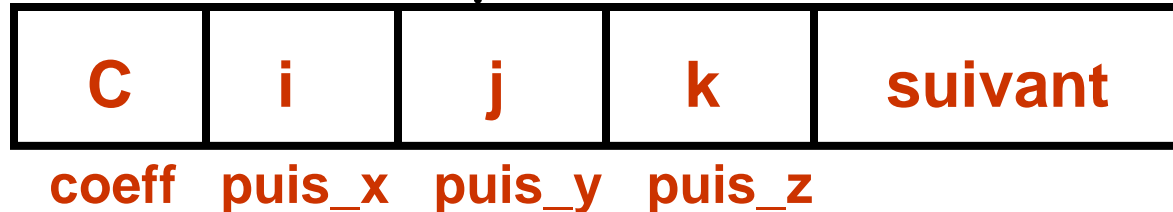
```
struct ElementListe {  
    char donnee ;  
    struct ElementListe *suivant ; } ;
```



## 2. CONSTRUCTION DU PROTOTYPE D'UN ÉLÉMENT DE LA LISTE

◆ **Exemple 2** : représentation d'un polynôme en  $x, y, z$   
 $5x^2 + 3xy + y^2 + yz$

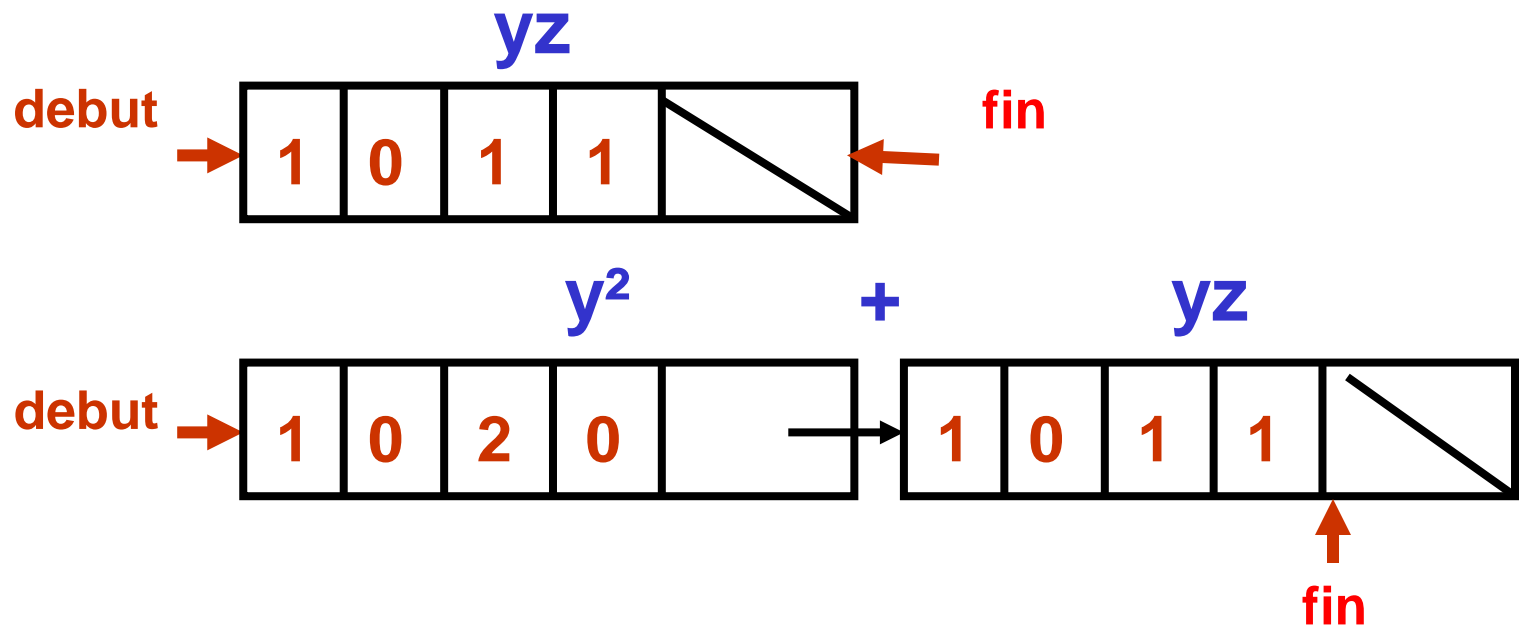
◆ Un élément de liste représente un monôme  $C x^i y^j z^k$



◆ `typedef struct ElementListe {  
    float coeff;  
    int px, py, pz;  
    struct ElementListe *suivant ; } Element;  
  
Element * debut, *fin;  
int taille;`

## 2. CONSTRUCTION DU PROTOTYPE D'UN ÉLÉMENT DE LA LISTE

◆ Construction du polynôme :





# 3. OPÉRATIONS SUR LES LSC

◆ Parmi les opérations nécessaires pour manipuler une liste simplement chaînée on trouve :

- Initialisation
- Insertion d'un élément dans la liste
- Suppression d'un élément dans la liste
- Affichage de la liste
- Destruction de la liste

# 3. OPÉRATIONS SUR LES LSC

## Initialisation

- ◆ Prototype de la fonction

`initialisation ( );`

- ◆ Cette opération doit être faite avant toute autre opération sur la liste

- ◆ Elle initialise le pointeur `debut` et le pointeur `fin` avec le pointeur `NULL`, et la `taille` avec la valeur `0`

- ◆ La fonction

```
initialisation ( ) {  
    debut = NULL;  
    fin = NULL;  
    taille = 0;  
}
```



# 3. OPÉRATIONS SUR LES LSC

## Insertion d'un élément dans la liste

Pour ajouter un élément dans une LSC il y a plusieurs situations :

1. Insertion dans une liste vide
2. Insertion au début de la liste
3. Insertion à la fin de la liste
4. Insertion ailleurs "dans la liste"



# 3. OPÉRATIONS SUR LES LSC

## Insertion d'un élément dans la liste

### 1. Insertion dans une liste vide

#### Étapes :

- déclaration et allocation de la mémoire pour le nouvel élément
- remplir le champ de données du nouvel élément
- le pointeur suivant du nouvel élément pointera vers **NULL**
- les pointeurs **debut** et **fin** pointeront vers le nouvel élément
- la mise à jour de la **taille**

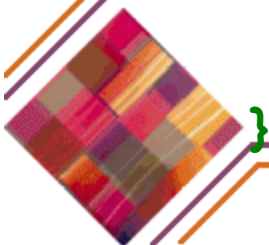


# 3. OPÉRATIONS SUR LES LSC

## Insertion d'un élément dans la liste

### 1. Insertion dans une liste vide

```
int ins_dans_liste_vide (float c, int i, int j, int k) {  
    Element *element;  
    if ((element=(Element *) malloc(sizeof(Element)))==NULL) return -1;  
    element->coeff = c ;  
    element->px = i ;  
    element->py = j ;  
    element->pz = k ;  
    element->suivant = NULL;  
    debut = element;  
    fin = element;  
    taille++;  
    return 0  
}
```



# 3. OPÉRATIONS SUR LES LSC

## Insertion d'un élément dans la liste

### 2. Insertion au début de la liste

Étapes :

- déclaration et allocation de la mémoire pour le nouvel élément
- remplir le champ de données du nouvel élément
- le pointeur **suivant** du nouvel élément pointe vers le 1er élément
- le pointeur **debut** pointe vers le nouvel élément
- le pointeur **fin** ne change pas
- la **taille** est incrémentée



# 3. OPÉRATIONS SUR LES LSC

## Insertion d'un élément dans la liste

### 2. Insertion au début de la liste

```
int ins_debut_liste (float c, int i,int j,int k) {  
    Element *element;  
    if ((element=(Element *) malloc(sizeof(Element)))==NULL) return -1;  
    element->coeff = c ;  
    element->px = i ;  
    element->py = j ;  
    element->pz = k ;  
    element->suivant = debut;  
    debut = element;  
    taille++;  
    return 0  
}
```

# 3. OPÉRATIONS SUR LES LSC

## Insertion d'un élément dans la liste

### 3. Insertion à la fin de la liste

Étapes :

- déclaration et allocation de la mémoire pour le nouvel élément
- remplir le champ de données du nouvel élément
- le pointeur suivant du nouvel élément pointe NULL
- le pointeur suivant du dernier élément pointe vers le nouvel élément
- le pointeur fin pointe vers le nouvel élément
- le pointeur debut ne change pas
- la taille est incrémentée





# 3. OPÉRATIONS SUR LES LSC

## Insertion d'un élément dans la liste

### 3. Insertion à la fin de la liste

```
int ins_fin_liste (float c; int i, int j, int k) {  
    Element *element;  
    if ((element=(Element *) malloc(sizeof(Element)))==NULL) return -1;  
    element->coeff = c ;  
    element->px = i ;  
    element->py = j ;  
    element->pz = k ;  
    element->suivant = NULL ;  
    fin->suivant = element ;  
    fin = element;  
    taille++;  
    return 0;  
}
```



# 3. OPÉRATIONS SUR LES LSC

## Insertion d'un élément dans la liste

### 4. Insertion ailleurs "dans la liste"

*L'insertion s'effectuera après une certaine position passée en argument à la fonction*

#### Étapes :

- déclaration et allocation de la mémoire pour le nouvel élément
- remplir le champ de données du nouvel élément
- choisir une position dans la liste
- le pointeur suivant du nouvel élément pointe vers l'adresse sur laquelle pointe le pointeur suivant d'élément courant
- le pointeur suivant de l'élément courant pointe vers le nouvel élément
- les pointeurs debut et fin ne changent pas
- la taille est incrémentée



# 3. OPÉRATIONS SUR LES LSC

## Insertion d'un élément dans la liste

### 4. Insertion ailleurs "dans la liste"

```
int ins_liste (float c; int i, int j, int k, int pos) {  
    Element *courant;  
    Element *element;  
    int t ;  
    if ((element=(Element *) malloc(sizeof(Element)))==NULL) return -1;  
    element->coeff = c ;  
    element->px = i ; element->py = j ; element->pz = k ;  
    courant = debut;  
    for (t = 1; t < pos; t++)    courant = courant->suivant;  
    element->suivant = courant->suivant;  
    courant->suivant = element;  
    taille++;  
    return 0 ;  
}
```



# 3. OPÉRATIONS SUR LES LSC

## Suppression d'un élément dans la liste

◆ Pour supprimer un élément dans la liste il y a deux situations :

1. Suppression au début de la liste
2. Suppression ailleurs "dans la liste"  
(y inclus la fin de la liste)



# 3. OPÉRATIONS SUR LES LSC

## Suppression d'un élément dans la liste

### 1. Suppression au début de la liste

#### Étapes :

- déclarer le pointeur `supp_elem`
- le pointeur `supp_elem` contiendra l'adresse du 1er élément
- le pointeur `debut` pointera vers le 2ème élément
- la `taille` de la liste sera décrémentée d'un élément



# 3. OPÉRATIONS SUR LES LSC

## Suppression d'un élément dans la liste

### 1. Suppression au début de la liste

/\*La fonction renvoie -1 en cas d'échec sinon elle renvoie 0 \*/

```
int supp_debut ( ) {  
    Element *supp_element;  
    if (taille == 0)    return -1;  
    supp_element = debut;  
    debut = debut-> suivant;  
    if (debut==NULL) fin = NULL;    // taille == 1  
    free(supp_element);  
    taille--;  
    return 0;  
}
```



# 3. OPÉRATIONS SUR LES LSC

## Suppression d'un élément dans la liste

### 2. Suppression ailleurs "dans la liste"

supprimer un élément après la position demandée

Étapes :

- le pointeur `supp_elem` contiendra l'adresse vers laquelle pointe le pointeur suivant d'élément courant
- le pointeur suivant de l'élément courant pointera vers l'élément sur lequel pointe le pointeur suivant de l'élément qui suit l'élément courant dans la liste
- Si l'élément courant est l'avant dernier élément, le pointeur `fin` doit être mis à jour
- la taille de la liste sera décrémentée d'un élément



# 3. OPÉRATIONS SUR LES LSC

## Suppression d'un élément dans la liste

### 2. Suppression ailleurs "dans la liste"

/\*La fonction renvoie -1 en cas d'échec sinon elle renvoie 0 \*/

```
int supp_dans_liste (int pos) {  
    int i; Element *courant , *supp_element;  
    if (taille <= 1 || pos < 1 || pos >= taille)    return -1;  
    courant = debut;  
    for (i = 1; i < pos; i++)    courant = courant->suivant;  
    supp_element = courant->suivant;  
    courant->suivant = courant->suivant->suivant;  
    if(courant->suivant == NULL) fin = courant;  
    free (supp_element);  
    taille--;  
    return 0;  
}
```



# 3. OPÉRATIONS SUR LES LSC

## Affichage de la liste

### ◆ Pour afficher la liste entière

- il faut se positionner au début de la liste (le pointeur debut le permettra)
- en utilisant le pointeur suivant de chaque élément la liste est parcourue du 1er vers le dernier élément
- la condition d'arrêt est donnée par le pointeur suivant du dernier élément qui vaut NULL



# 3. OPÉRATIONS SUR LES LSC

## Affichage de la liste

La fonction /\* affichage de la liste \*/

```
affiche () {  
    Element *courant;  
    courant = debut;  
    while (courant != NULL) {  
        printf ("%0.1f ", courant->coeff);  
        if (courant->px !=0) printf ("x(%d)" , courant->px);  
        if (courant->py !=0) printf ("y(%d)" , courant->py);  
        if (courant->pz !=0) printf ("z(%d)" , courant->pz);  
        if (courant !=fin) printf (" + ");  
        courant=courant->suivant;  
    }  
}
```



# 3. OPÉRATIONS SUR LES LSC

## Destruction de la liste

### ◆ Pour détruire la liste entière :

- on doit supprimer élément par élément
- la suppression commence par le début de la liste tant que la taille est plus grande que zéro

### La fonction

```
destruire () {  
while (taille > 0)    supp_debut();  
}
```



# 3. OPÉRATIONS SUR LES LSC

## Exemple et teste

```
main() {  
  initialisation();  
  ins_dans_liste_vide(3,1,1,0);  
  ins_debut_liste(5, 2, 0, 0);  
  ins_fin_liste(1, 0, 1, 1);  
  ins_liste(1, 0, 2, 0, 2);  
  affiche();  
  system("pause");  
}
```



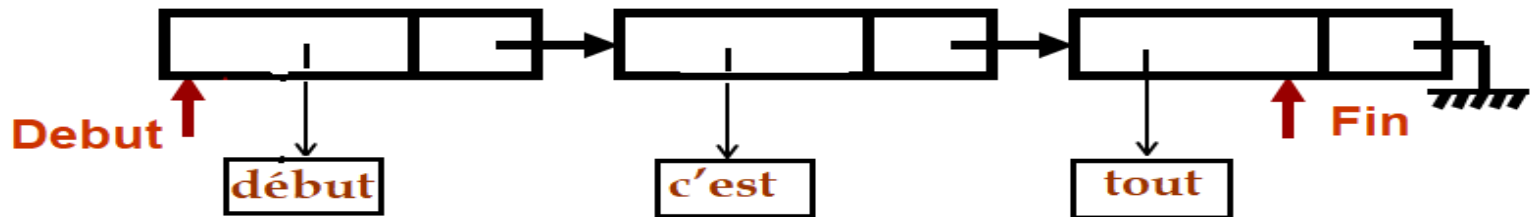
## 4. UN EXEMPLE COMPLET

Exercice : En utilisant la structure de donnée suivante :

```
typedef struct Element {  
    char *mot ;  
    struct Element *suivant ; } noeud;
```

- Réaliser les opérations suivantes :
  - Créer la liste chaînée (initialisation du premier élément)
  - Insérer de nouveaux noeuds (à la fin)
  - Parcourir la liste en affichant le texte
  - Afficher les nœuds (mots) dans le sens inverse.

Exemple :



L'affichage à l'envers doit nous donner : **tout c'est début**

## 4. UN EXEMPLE COMPLET

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

```
#include <string.h>
```

```
typedef struct Element {
```

```
    char *mot ;
```

```
    struct Element *suivant ; } noeud;
```

```
noeud *debut, *Fin ; int taille ;
```

```
initialisation() ;
```

```
ins_dans_liste_vide(noeud *element);
```

```
ins_dans_fin_liste(noeud *element);
```

```
lire_chaine();
```

```
afficher();
```

```
afficher_envers();
```

```
main() {
```

```
    initialisation() ;
```

```
    if (lire_chaine() == -1) printf("pas d'espace mémoire \n");
```

```
    else {
```

```
        afficher();
```

```
        printf("***affichage à l'envers *** \n ");
```

```
        afficher_envers();
```

```
    }
```

```
    system("pause");
```

```
}
```

## 4. UN EXEMPLE COMPLET

```
int lire_chaine() {  
    noeud *element;  
    printf("Entrer des mots et taper le mot 'Fin' pour terminer \n");  
    do {  
        if ((element = (noeud *) malloc (sizeof (noeud)))==NULL) return -1;  
        if ((element->mot = (char *) malloc (50 * sizeof(char))) == NULL) return -1;  
        gets(element-> mot);  
        if (strcmp(element->mot,"Fin") ==0) break;  
        if (Debut == NULL) ins_dans_liste_vide(element);  
        else ins_dans_fin_liste(element) ;  
    }  
    while (1) ; // condition toujours vraie  
    return 0;  
}
```



## 4. UN EXEMPLE COMPLET

```
initialisation () {Debut = NULL; Fin = NULL; taille = 0; }
```

```
ins_dans_liste_vide(noeud *element) {  
    element->suivant = NULL;  
    Debut = element;  
    Fin = element;  
    taille++; }
```

```
ins_dans_fin_liste (noeud *element) {  
    element->suivant = NULL ;  
    Fin-> suivant = element;  
    Fin = element;  
    taille++;}
```

```
afficher() {  
    noeud *courant;  
    courant = Debut;  
    if (courant == NULL) printf("liste vide \n");  
    else while (courant != NULL) { puts(courant->mot);  
                                     courant = courant->suivant;}
```

```
}
```



## 4. UN EXEMPLE COMPLET

```
afficher_envers() {
```

```
noeud *p, *q, *r;
```

```
p= Debut ; q= Debut->suivant; r= q->suivant;
```

```
p->suivant=NULL ; q->suivant=p;
```

```
do {p=q ; q=r ; r= r->suivant; q->suivant=p; }
```

```
while (r!=NULL);
```

```
Fin=Debut ; Debut=q;
```

```
afficher();
```

```
}
```

```
afficher_envers_recursive(noeud *p) {
```

```
if (p!=NULL) {
```

```
    afficher_envers_recursive(p->suivant);
```

```
    puts(p->mot);
```

```
}
```

```
}
```

