Université Sultan Moulay Slimane
Ecole Supérieure de Technologie
- Beni Mellal -

STRUCTURES DE DONNEES

Filières : DUT **GI** / DUT **IDIA**
Semestre : S3
Année Universitaire : 2025/2026

Pr. M. OUTANOUTE
m.outanoute@usms.ma

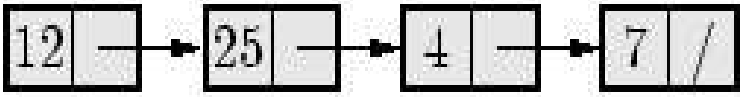
- Chapitre 3-

LES LISTES SIMPLEMENT CHAÎNÉES

1) DÉFINITION ET REPRESENTATION
2) CONSTRUCTION DU PROTOTYPE D'UN ÉLÉMENT DE LA LISTE
3) OPÉRATIONS SUR LES LISTES SIMPLEMENT CHAÎNÉES
4) UN EXEMPLE COMPLET

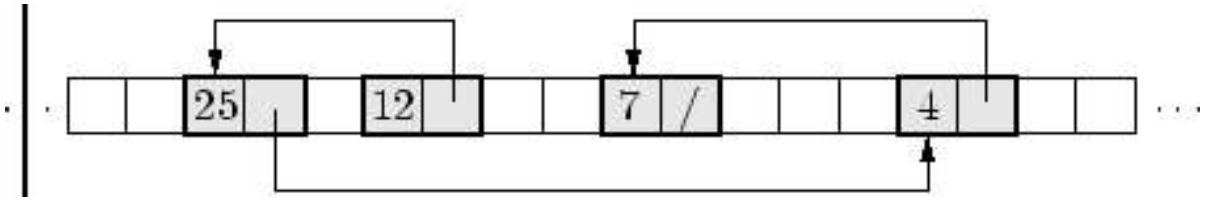
1. DÉFINITION ET REPRÉSENTATION

- 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

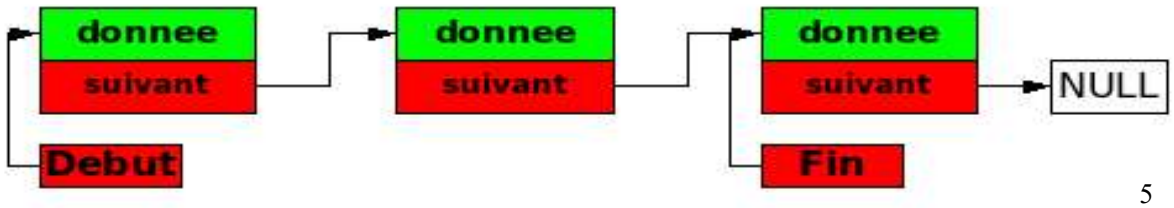
- Les éléments d'un tableau sont contigus dans la mémoire, tandis que les éléments d'une liste sont éparpillés dans la mémoire.



- 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.



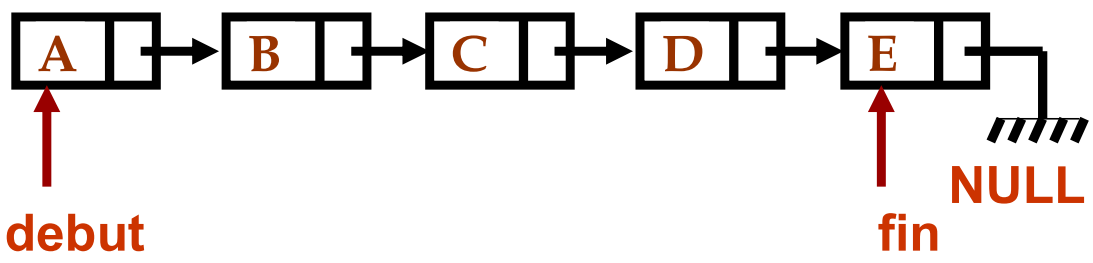
2. CONSTRUCTION DU PROTOTYPE D'UN ÉLÉMENT DE LA LISTE

- **Exemple 1 :**

Représentation d'une liste de 5 éléments:

'A', 'B', 'C', 'D' et 'E'.

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

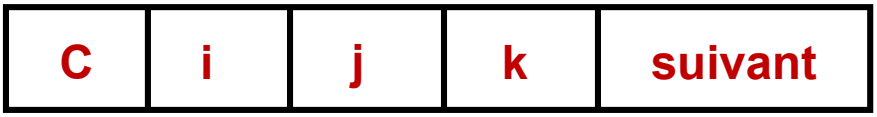


2. CONSTRUCTION DU PROTOTYPE D'UN ÉLÉMENT DE LA LISTE

➢ **Exemple 2 :**

Représentation d'un polynôme : $5x^2 + 3xy + y^2 + yz$

- **Un élément de la liste représente un monôme** $C\ x^i\ y^j\ z^k$



coeff puis_x puis_y puis_z

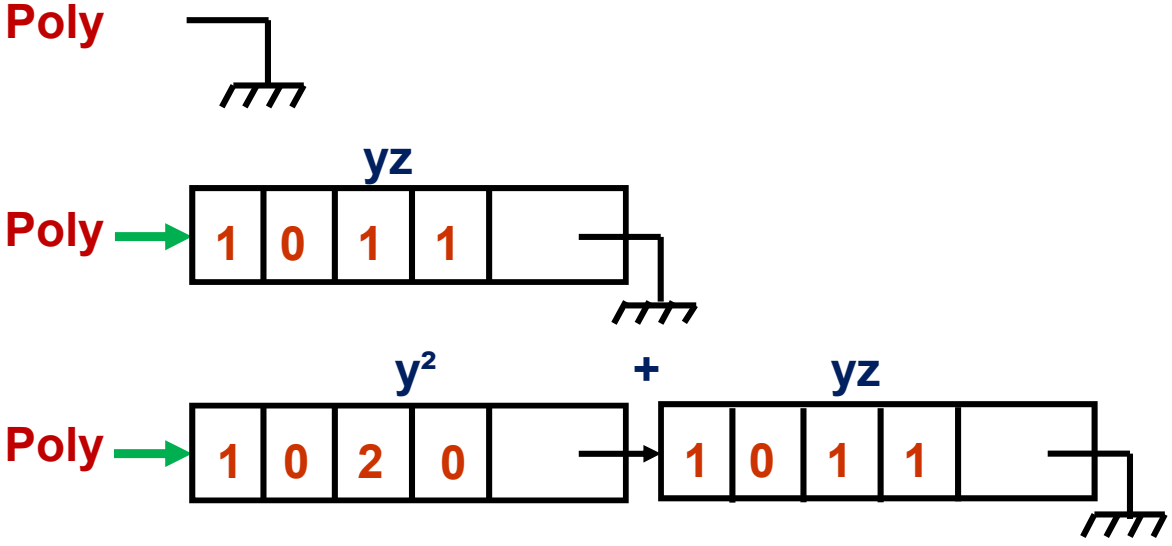
```
typedef struct ElementListe {
        float coeff;
        int px, py, pz;
        struct ElementListe *suivant ;
} Element;
```


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

➢ **Exemple 2 (suite):**

Construction du polynôme :

```
Element Poly ;
```



3. OPÉRATIONS SUR LES LISTES SIMPLEMENT CHAÎNÉES

- **Parmi les opérations nécessaires pour manipuler une liste simplement chaînée on trouve :**
  1) **Initialisation**
  2) **Insertion d'un élément dans la liste**
     - **Insertion dans une liste vide**
     - **Insertion au début de la liste**
     - **Insertion à la fin de la liste**
     - **Insertion ailleurs dans la liste**
  3) **Suppression d'un élément dans la liste**
     - **Suppression au début de la liste**
     - **Suppression ailleurs dans la liste**
  4) **Affichage de la liste**
  5) **Destruction de la liste**

3.1) INITIALISATION

- **Prototype de la fonction:** `void initialiser ( );`
- **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 :**

```
void initialiser ( ) {
   debut = NULL;
   fin = NULL;
   taille = 0;
   Poly = NULL;
}
```

3.2) INSERTION D'UN ELEMENT DANS LA LISTE

- **Algorithme d'insertion et de sauvegarde des éléments:**
  - ✓ **Déclarer l'élément à insérer ;**
  - ✓ **Allouer de la mémoire pour le nouvel élément ;**
  - ✓ **Remplir le contenu du champ de données ;**
  - ✓ **Mettre à jour les pointeurs vers le premier et le dernier élément si nécessaire ;**

    Cas particulier : dans une liste avec un seul élément, le premier est en même temps le dernier.
  - ✓ **Mettre à jour la taille de la liste.**

3.2) INSERTION D'UN ELEMENT DANS LA LISTE

- **Pour ajouter un élément dans la liste, 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

3.2) INSERTION D'UN ELEMENT DANS LA LISTE

➢ Insertion dans une liste vide :

❖ Étapes :

- ✓ Allouer de la mémoire pour le nouvel élément ;
- ✓ Remplir le champ de données du nouvel élément ;
- ✓ Pointer le pointeur suivant du nouvel élément vers NULL ;
- ✓ Pointer les pointeurs debut et fin vers le nouvel élément ;
- ✓ Mettre à jour la taille de la liste (incrémentation) .

3.2) INSERTION D'UN ELEMENT DANS LA LISTE

➢ Insertion dans une liste vide :

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

3.2) INSERTION D'UN ELEMENT DANS LA LISTE

➢ **Insertion au début de la liste :**

❖ **Étapes :**

- ✓ Allouer de la mémoire pour le nouvel élément ;
- ✓ Remplir le champ de données du nouvel élément ;
- ✓ Pointer le pointeur **suivant** du nouvel élément vers le premier élément ;
- ✓ Pointer le pointeur **debut** vers le nouvel élément ;
- ✓ Le pointeur **fin** ne change pas ;
- ✓ Incrémenter la **taille** de la liste .

3.2) INSERTION D'UN ELEMENT DANS LA LISTE

➢ **Insertion au début de la liste :**

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

3.2) INSERTION D'UN ELEMENT DANS LA LISTE

➢ Insertion à la fin de la liste :

❖ Étapes :

- ✓ Allouer de la mémoire pour le nouvel élément ;
- ✓ Remplir le champ de données du nouvel élément ;
- ✓ Pointer le pointeur **suivant** du dernier élément vers le nouvel élément ;
- ✓ Pointer le pointeur **fin** vers le nouvel élément ;
- ✓ Le pointeur **debut** ne change pas ;
- ✓ Incrémenter la **taille** de la liste .

3.2) INSERTION D'UN ELEMENT DANS LA LISTE

➢ Insertion à la fin de la liste :

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

3.2) INSERTION D'UN ELEMENT DANS LA LISTE

➢ Insertion ailleurs dans la liste :

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

❖ Étapes :

- ✓ Allouer de la mémoire pour le nouvel élément ;
- ✓ Remplir le champ des données du nouvel élément ;
- ✓ Choisir une position dans la liste ;
- ✓ Pointer le pointeur suivant du nouvel élément vers le pointeur suivant de l'élément courant ;
- ✓ Pointer le pointeur suivant de l'élément courant vers le nouvel élément ;
- ✓ Les pointeurs debut et fin ne changent pas ;
- ✓ Incrémenter la taille de la liste .

3.2) INSERTION D'UN ELEMENT DANS LA LISTE

➢ Insertion ailleurs dans la liste :

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

3.3) SUPPRESSION D'UN ELEMENT DANS LA LISTE

➢ **Algorithme de suppression d'un élément de la liste :**

- ✓ Utiliser un pointeur temporaire pour sauvegarder l'adresse de l'élément à supprimer ;
- ✓ L'élément à supprimer se trouve après l'élément courant ;
- ✓ Faire pointer le pointeur suivant de l'élément courant vers l'adresse du pointeur suivant de l'élément à supprimer ;
- ✓ Libérer la mémoire occupée par l'élément supprimé ;
- ✓ Mettre à jour la taille de la liste (décrémentation) .

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

1) Suppression au début de la liste

2) Suppression ailleurs dans la liste

3.3) SUPPRESSION D'UN ELEMENT DANS LA LISTE

➢ **Suppression au début de la liste :**

❖ **Étapes :**

- ✓ Pointer le pointeur supp_element vers le premier élément ;
- ✓ Pointer le pointeur debut vers le deuxième élément ;
- ✓ Détruire l'élément pointé par supp_element ;
- ✓ Décrémenter la taille de la liste.

3.3) SUPPRESSION D'UN ELEMENT DANS LA LISTE

➢ Suppression au début de la liste :

```
// La fonction renvoie -1 en cas d'échec sinon elle renvoie 0.
int supprimer_debut ( ) {
    Element *supp_element;
    if (taille == 0)    return -1;
    supp_element = debut;
    debut = debut-> suivant;
    Poly == debut;
    if (taille == 1)    fin = NULL;
    free(supp_element);
    taille--;
    return 0;
}
```


3.3) SUPPRESSION D'UN ELEMENT DANS LA LISTE

➢ Suppression ailleurs dans la liste :

Supprimer un élément après la position demandée.

❖ Étapes :

- ✓ Pointer le pointeur supp_element vers le pointeur suivant de l'élément courant ;
- ✓ Pointer le pointeur suivant de l'élément courant 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 (suppression à la fin) ;
- ✓ Décrémenter la taille de la liste.

3.3) SUPPRESSION D'UN ELEMENT DANS LA LISTE

➢ **Suppression ailleurs dans la liste :**

```
// La fonction renvoie -1 en cas d'échec sinon elle renvoie 0.
int supprimer_ailleurs (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;
}
```


3.4) AFFICHAGE DE LA LISTE

➢ **Pour afficher la liste entière :**

- ✓ Se positionner au début de la liste (le pointeur **debut** le permettra) ;
- ✓ Parcourir la liste du premier vers le dernier élément, en utilisant le pointeur **suivant** de chaque élément ;
- ✓ La condition d'arrêt est donnée par le pointeur **suivant** du dernier élément qui vaut NULL .

3.4) AFFICHAGE DE LA LISTE

➢ **La fonction affichage de la liste :**

```
afficher () {
    Element *courant;
    courant = debut;
    while (courant != NULL) {
        printf ("%.2f ", 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;
    }
}
```


3.5) 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 de destruction de la liste :**

```
detruire () {
    while (taille > 0)
        supprimer_debut(debut);
}
```

3. OPÉRATIONS SUR LES LISTES SIMPLEMENT CHAÎNÉES

➢ **Exemple et test des fonctions :**

```
main() {
    initialiser();
    inserer_liste_vide(3,1,1,0);
    inserer_debut(5, 2, 0, 0);
    inserer_fin(1, 0, 1, 1);
    inserer_ailleurs(1, 0, 2, 0, 2);
    afficher(); // 5x(2) + 3x(1)y(1) + 1y(2) + 1y(1)z(1)
    detruire();
}
```


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 et insertion du premier élément);
- Insérer de nouveaux nœuds (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 ;
void initialiser() {
    debut = NULL;
    fin = NULL;
    taille = 0;
 }
```


4. UN EXEMPLE COMPLET

```
inserer_liste_vide(Noeud *nouveau) {
    nouveau->suivant = NULL;
    debut = nouveau;
    fin = nouveau;
    taille++;
}
inserer_fin(Noeud *nouveau) {
    nouveau->suivant = NULL ;
    fin-> suivant = nouveau;
    fin = nouveau;
    taille++;
}
```

4. UN EXEMPLE COMPLET

```
lire_chaine() {
    Noeud *nouveau;
    printf("Entrer des mots et taper le mot 'Fin' pour terminer:\n");
    do {
        nouveau = (Noeud *) malloc (sizeof (Noeud));
        nouveau->mot = (char *) malloc (50 * sizeof(char));
        gets(nouveau-> mot); // scanf
        if (strcmp(nouveau->mot, "Fin") ==0)  break;
        if (debut == NULL) inserer_liste_vide(nouveau);
        else inserer_fin(nouveau) ;
    }while(1);  // while (1)  une condition toujours vraie
}
```


4. UN EXEMPLE COMPLET

```
afficher() {
    Noeud *courant;  courant = debut;
    if (debut == NULL) printf("\n Liste vide ! \n");
    else  while (courant != NULL) {
          puts(courant->mot); // printf("%s ", courant->mot) ;
          courant =  courant->suivant;
    }
}
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();
}
```

4. UN EXEMPLE COMPLET

```
afficher_envers_recursive(noeud *p) {
    if (p!=NULL) {
        afficher_envers_recursive(p->suivant);
        puts(p->mot); // scanf("%s ", p->mot) ;
    }
}
main() {
  initialiser() ;
  lire_chaine();
  afficher();
  printf("\nAffichage à l'envers :\n ");
  afficher_envers();
}
```