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 4 -

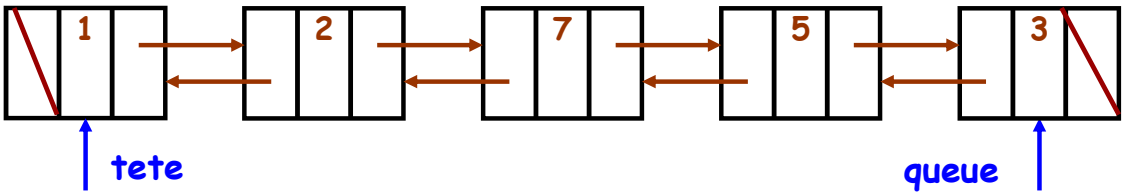
LES LISTES DOUBLEMENT CHAÎNÉES

I. DÉFINITION

- Les Listes Doublement Chaînées sont des structures de données semblables aux listes simplement chaînées :
  - L'allocation de la mémoire est faite au moment de l'exécution
  - La liaison entre les éléments se fait grâce à deux pointeurs (un qui pointe vers l'élément précédent et un qui pointe vers l'élément suivant)
  - Le pointeur precedent du premier élément doit pointer vers NULL (le début de la liste)
  - Le pointeur suivant du dernier élément doit pointer vers NULL (la fin de la liste)

I. DÉFINITION

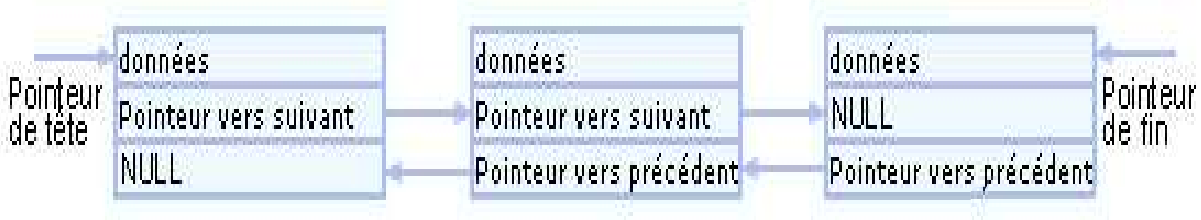
- Pour accéder à un élément de la liste doublement chaînée :
  - en commençant avec la tête : le pointeur suivant permettant le déplacement vers le prochain élément
  - en commençant avec la queue : le pointeur precedent permettant le déplacement vers l'élément précédent



- La liste doublement chaînée peut être parcourue dans les deux sens, du premier vers le dernier élément et/ou du dernier vers le premier élément

II. 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*, un pointeur *precedent* et un pointeur *suivant*
- Les pointeurs *precedent* et *suivant* doivent être du même type que l'élément, sinon ils ne pourront pas pointer vers un élément de la liste
- Le pointeur *precedent* permettra l'accès vers l'élément précédent tandis que le pointeur *suivant* permettra l'accès vers le prochain élément

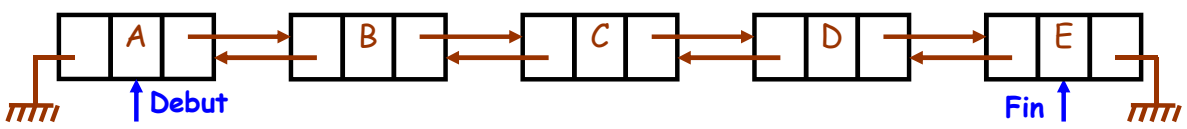


II. CONSTRUCTION DU PROTOTYPE D'UN ÉLÉMENT DE LA LISTE

- Exemple 1 : Représentation d'une liste de 5 éléments: 'A', 'B','C','D' et 'E'

```
typedef struct ElementListe {
    char donnee ;
    struct ElementListe *precedent ;
    struct ElementListe *suivant ;
} Element;
```

- Pour avoir le contrôle de la liste il est préférable de sauvegarder certains éléments : *Debut, Fin, Taille*
  - Le pointeur *debut* contiendra l'adresse du premier élément de la liste. Element *Debut ;
  - Le pointeur *Fin* contiendra l'adresse du dernier élément de la liste. Element *Fin ;
  - La variable *Taille* contient le nombre d'éléments. int Taille ;



III. OPÉRATIONS SUR LES LISTES DOUBLEMENT CHAÎNÉES

- Nous allons travailler par la suite avec les structures de données et les déclarations suivantes :

```
typedef struct ElementListe {
    char *info ;
    struct ElementListe *precedent ;
    struct ElementListe *suivant ;
} Element;

int Taille ;
Element *Debut, *Fin;
```


III-1) 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;
}
```

III-2) Insertion d'un élément dans la liste

- Algorithme d'insertion et de sauvegarde des éléments
  - 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 l'élément précédent et l'élément suivant
  - Mettre à jour les pointeurs vers le 1er et le dernier élément si nécessaire
    - Cas particulier : dans une liste avec un seul élément, le 1er est en même temps le dernier
  - Mettre à jour la taille de la liste

III-2) Insertion d'un élément 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 avant un élément

5) Insertion après un élément

III-2) Insertion d'un élément dans la liste

1) Insertion dans une liste vide :

Étapes :

- Allocation de la mémoire pour le nouvel élément
- Remplir les champs de données du nouvel élément
- Le pointeur precedent du nouvel élément pointera vers NULL
- Le pointeur suivant du nouvel élément pointera vers NULL
- Les pointeurs Debut et Fin pointeront vers le nouvel élément
- La Taille est mise à jour

III-2) Insertion d'un élément dans la liste

1) Insertion dans une liste vide :

```
int Inserer_liste_vide (char *info) {
    Element *nouveau;
    nouveau = (Element*) malloc (sizeof(Element));
    if(nouveau == NULL)  return -1;
    nouveau->info = (char *) malloc (50 * sizeof(char));
    strcpy (nouveau->info, info);
    nouveau->precedent = NULL;
    nouveau->suivant = NULL;
    Debut = nouveau;
    Fin = nouveau;
    Taille++;
    return 0;
}
```

III-2) Insertion d'un élément dans la liste

2) Insertion au début de la liste :

Étapes :

- Allocation de la mémoire pour le nouvel élément
- Remplir le champ de données du nouvel élément
- Le pointeur precedent du nouvel élément pointe vers NULL
- Le pointeur suivant du nouvel élément pointe vers le 1er élément
- Le pointeur precedent du 1er élément pointe vers le nouvel élément
- Le pointeur Debut pointe vers le nouvel élément
- Le pointeur Fin ne change pas
- La Taille est incrémentée

III-2) Insertion d'un élément dans la liste

2) Insertion au début de la liste :

```
int Inserer_debut (char *info) {
    Element *nouveau;
    nouveau = (Element*) malloc (sizeof (Element));
    if (nouveau == NULL)  return -1;
    nouveau->info = (char *) malloc (50 * sizeof(char));
    strcpy (nouveau->info, info);
    nouveau->precedent = NULL;
    nouveau->suivant = Debut;
    Debut->precedent = nouveau;
    Debut = nouveau;
    Taille++;
    return 0;
}
```

III-2) Insertion d'un élément dans la liste

3) Insertion à la fin de la liste :

Étapes :

- Allocation de la mémoire pour le nouvel élément
- Remplir le champ de données du nouvel élément
- Le pointeur suivant du nouvel élément pointe vers NULL
- Le pointeur precedent du nouvel élément pointe vers le dernier élément (le pointeur Fin)
- Le pointeur suivant du dernier élément va pointer vers le nouvel élément
- Le pointeur Fin pointe vers le nouvel élément
- Le pointeur Debut ne change pas
- La Taille est incrémentée

III-2) Insertion d'un élément dans la liste

3) Insertion à la fin de la liste :

```
int Inserer_fin (char *info) {
    Element *nouveau;
    nouveau = (Element*) malloc (sizeof (Element));
    if (nouveau == NULL)  return -1;
    nouveau->info = (char *) malloc (50 * sizeof(char));
    strcpy (nouveau->info, info);
    nouveau->suivant = NULL;
    nouveau->precedent = Fin;
    Fin->suivant = nouveau;
    Fin = nouveau;
    Taille++;
    return 0;
}
```

III-2) Insertion d'un élément dans la liste

4) Insertion avant un élément de la liste :

- *L'insertion s'effectuera avant une certaine position passée en argument à la fonction.*
- *La position indiquée ne doit pas être le 1er élément. Dans ce cas il faut utiliser les fonctions d'insertion au début de la liste.*

III-2) Insertion d'un élément dans la liste

4) Insertion avant un élément de la liste :

Étapes :

- 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'élément courant
- Le pointeur precedent du nouvel élément pointe vers l'adresse sur la quelle pointe le pointeur precedent d'élément courant
- Si le pointeur precedent de l'élément courant est NULL alors le pointeur Debut pointe vers le nouvel élément
- Sinon le pointeur suivant de l'élément qui précède l'élément courant pointera vers le nouvel élément
- Le pointeur precedent d'élément courant pointe vers le nouvel élément
- Le pointeurs Fin ne change pas
- La Taille est incrémentée d'une unité

III-2) Insertion d'un élément dans la liste

4. Insertion avant un élément de la liste :

```
int Inserer_avant (char *info, int pos) {
    int i; Element *nouveau, *courant;
    nouveau = (Element*) malloc (sizeof (Element));
    if (nouveau ===NULL)  return -1;
    nouveau->info = (char *) malloc (50 * sizeof(char));
    strcpy (nouveau->info, info);
    courant = Debut;
    for (i = 1 ; i < pos ; ++i)  courant = courant->suivant;
    nouveau->suivant = courant;
    nouveau-> precedent = courant->precedent;
    if(courant->precedent == NULL)  Debut = nouveau;
    else courant->precedent->suivant = nouveau;
    courant->precedent = nouveau;
    Taille++;
    return 0;
}
```


III-2) Insertion d'un élément dans la liste

5. Insertion après un élément de la liste :

Étapes:

- 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 la quelle pointe le pointeur suivant d'élément courant
- Le pointeur precedent du nouvel élément pointe vers l'élément courant.
- Si le pointeur suivant de l'élément courant est NULL alors le pointeur Fin pointe vers le nouvel élément
- Sinon le pointeur precedent de l'élément qui succède l'élément courant pointera vers le nouvel élément
- Le pointeur suivant d'élément courant pointe vers le nouvel élément
- La Taille est incrémentée d'une unité

III-2) Insertion d'un élément dans la liste

5. Insertion après un élément de la liste :

```
int Inserer_apres (char *info, int pos) {
    int i; Element *nouveau, *courant;
    nouveau = (Element*) malloc (sizeof (Element));
    if (nouveau == NULL)  return -1;
    nouveau->info = (char *) malloc (50 * sizeof(char));
    courant = Debut;
    for (i = 1; i < pos; ++i) courant = courant->suivant;
    nouveau->suivant = courant->suivant;
    nouveau->precedent = courant;
    if(courant->suivant == NULL) Fin = nouveau;
    else courant->suivant->precedent = nouveau;
    courant->suivant = nouveau;
    Taille++;
    return 0;
}
```


III-3) Suppression d'un élément dans la liste

- La suppression au début et à la fin de la L.D.C ainsi qu'avant ou après un élément revient à la suppression à la position 1 ou à la position N (nombre d'éléments de la liste) ou ailleurs dans la liste.
- La suppression dans la L.D.C à n'importe quelle position ne pose pas des problèmes grâce aux pointeurs precedent et suivant, qui permettent de garder la liaison entre les éléments de la liste.
- C'est la raison pour la quelle nous allons créer une seule fonction.
- Si nous voulons supprimer :
  - l'élément au début de la liste nous choisirons la position 1
  - l'élément à la fin de la liste nous choisirons la position N
  - un élément quelconque alors on choisit sa position dans la liste

III-3) Suppression d'un élément dans la liste

- La position choisie est 1 (suppression du 1er élément de la liste)
  - Le pointeur supp_element contiendra l'adresse du 1er élément
  - Le pointeur Debut contiendra l'adresse contenue par le pointeur suivant du 1er élément que nous voulons supprimer
    - Si ce pointeur vaut NULL alors nous mettons à jour le pointeur Fin (liste avec un seul élément)
    - Sinon nous faisons pointer le pointeur precedent du 2ème élément vers NULL)
- La position choisie est égale au nombre d'éléments de la liste
  - Le pointeur supp_element contiendra l'adresse du dernier élément
  - Nous faisons pointer le pointeur suivant de l'avant dernier élément vers NULL
  - Nous mettons à jour le pointeur Fin

III-3) Suppression d'un élément dans la liste

- La position choisie est aléatoire dans la liste :
  - Le pointeur supp_element contiendra l'adresse de l'élément à supprimer
  - Le pointeur suivant de l'élément qui précède l'élément à supprimer pointe vers l'adresse contenu par le pointeur suivant d'élément à supprimer
  - Le pointeur precedent d'élément qui succède l'élément à supprimer pointe vers l'adresse contenu par le pointeur precedent d'élément à supprimer
  - La Taille de la liste sera décrémentée d'une unité

III-3) Suppression d'un élément dans la liste

```
int Supprimer(int pos) {
    int i; Element *supp_element,*courant;
    if(Taille == 0) return -1;
    if(pos == 1) { /* suppression de 1er élément */
        supp_element = Debut;
        Debut = Debut->suivant;
        if(Debut == NULL)  Fin = NULL;
        else  Debut->precedent = NULL;
    }
    else if(pos == Taille) { /* suppression du dernier élément */
        supp_element = Fin;
        Fin->precedent->suivant = NULL;
        Fin = Fin->precedent;
    }
```


III-3) Suppression d'un élément dans la liste

```
    else {  /* suppression ailleurs */
        courant = Debut;
        for(i=1;i<pos;++i) courant = courant->suivant;
        supp_element = courant;
        courant->precedent->suivant = courant->suivant;
        courant->suivant->precedent = courant->precedent;
    }
    free(supp_element->info);
    free(supp_element);
    Taille--;
    return 0;
}
```

III-4) Affichage de la liste

- Pour afficher la liste entière :
  - Se positionner au début (Debut) de la liste ou à la fin (Fin) de la liste ;
  - Parcourir la liste du 1er vers le dernier élément ou du dernier vers le 1er élément en utilisant le pointeur suivant ou precedent de chaque élément ;
  - La condition d'arrêt est donnée par le pointeur suivant du dernier élément qui vaut NULL ou le pointeur precedent du 1er élément qui vaut NULL .

III-4) Affichage de la liste

- Pour afficher la liste entière

```
Afficher_Liste() {  /* affichage en avançant */
    Element *courant;
    courant = Debut;   /* point du départ le 1er élément */
    printf("[");
    while(courant != NULL) {
        printf("%s ", courant->info);
        courant = courant->suivant;
    }
    printf("]\n");
}
```

III-5) Destruction de la liste

- Pour détruire la liste entière :
  - On doit supprimer élément par élément ,
  - La suppression peut commencer par la position 1 tant que la Taille est plus grande que 0 .
- La fonction :

```
Detruire () {

    while (Taille > 0)  Supprimer(1) ;

}
```