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 5 -

LES LISTES CHAÎNÉES CIRCULAIRES

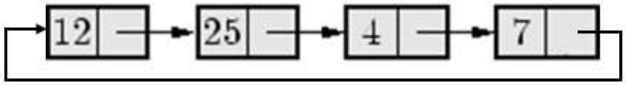
1) Listes Simplement Chaînées Circulaires

2) Listes Doublement Chaînées Circulaires

3) Exercices

1. Listes Simplement Chaînées Circulaires

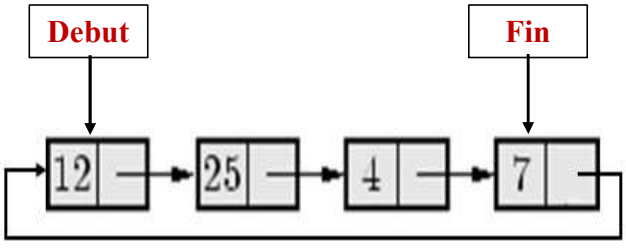
- Les Listes Simplement Chaînées Circulaires (LSCC) sont des Listes Simplement Chaînées, sauf que le dernier élément de la liste pointe sur le premier élément de la liste.



- La liaison entre les éléments se fait grâce à un pointeur.
- Le pointeur suivant du dernier élément doit pointer vers le début de la liste (le premier élément 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.

1. Listes Simplement Chaînées Circulaires

- 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.



Taille = 4

1. Listes Simplement Chaînées Circulaires

- Parmi les opérations nécessaires pour manipuler une liste simplement chaînée circulaire, on trouve :
  - **Initialisation**
  - **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**
  - **Suppression d'un élément dans la liste**
    - **Suppression au début de la liste**
    - **Suppression ailleurs dans la liste**
  - ***Affichage de la liste***
  - **Destruction de la liste**

1.1) Prototype d'un élément et initialisation d'une LSCC

- **Prototype d'un élément de la liste :**

```
typedef struct ElementListe {
    int donnee;
    struct ElementListe *suivant ;
} Element;
```

- **Initialisation :**

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

1.2) Insertion d'un élément dans la liste

➢ **Insertion dans une liste vide :**

```
inserer_liste_vide (int d) {
        Element *nouveau;
        nouveau = (Element *) malloc (sizeof (Element));
        nouveau -> donnee = d ;
        nouveau -> suivant = nouveau;  // liste circulaire
        Debut = nouveau;
        Fin = nouveau;
        Taille++;
 }
```


1.2) Insertion d'un élément dans la liste

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

```
inserer_debut (int d) {
        Element *nouveau;
        nouveau = (Element *) malloc (sizeof (Element));
        nouveau -> donnee = d ;
        nouveau -> suivant = Debut;
        Debut = nouveau;
        Fin -> suivant = Debut;  // liste circulaire
        Taille++;
}
```

1.2) Insertion d'un élément dans la liste

➢ **Insertion à la fin de la liste :**

```
inserer_fin (int d) {
        Element *nouveau;
        nouveau = (Element *) malloc (sizeof (Element));
        nouveau -> donnee = d ;
        Fin -> suivant = nouveau;
        nouveau -> suivant = Debut; // liste circulaire
        Fin = nouveau;
        taille++;
}
```


1.2) Insertion d'un élément dans la liste

➢ **Insertion ailleurs de la liste :**

```
inserer_ailleurs (int d, int pos) {
        Element *nouveau, *courant;  int i ;
        nouveau = (Element *) malloc (sizeof (Element));
        nouveau -> donnee = d ;
        nouveau -> suivant = NULL ;
        courant = Debut;
        for (i = 1; i < pos; i++)   courant = courant -> suivant;
        nouveau -> suivant = courant -> suivant;
        courant -> suivant = nouveau;
        Taille ++;
}
```

1.3) Suppression d'un élément dans la liste

➢ Suppression au début de la liste :

```
int supprimer_debut ( ) {
        Element *supp_element;
        if (Taille == 0)    return -1;
        supp_element = Debut;
        Debut = Debut -> suivant;
        Fin -> suivant = Debut ;   //Circulaire
        if (Taille == 1)    Fin = NULL;
        free(supp_element);
        Taille--;
        return 0;
}
```


1.3) Suppression d'un élément dans la liste

➢ Suppression ailleurs dans la liste :

```
int supprimer_ailleurs (int pos) {
    int i;  Element *courant, *precedent, *supp_element;
    if (Taille <= 1 || pos < 1 || pos > Taille)    return -1;
    courant = Debut;
    for (i = 1; i < pos; i++) {
        precedent = courant; //Mémoriser l'élément avant courant
        courant = courant -> suivant;
    }
    supp_element = courant; //élément à supprimer
    precedent->suivant = courant->suivant; //Suppression de courant
```

1.3) Suppression d'un élément dans la liste

➢ **Suppression ailleurs dans la liste (suite):**

```
    if(pos == Taille) {  //élément supprimé existe à la fin
        precedent -> suivant = Debut; //Circulaire
        Fin = precedent;
    }
    free (supp_element);
    Taille--;
    return 0;
}
```


1.4) Affichage de la liste

➢ ***Affichage du contenu de la liste :***

```
void afficher_liste () {
    Element *courant;
    courant = Debut;
    int i=1;
    while (i <= Taille) {
        printf ("%d \t", courant -> donnee);
        i++;
        courant = courant -> suivant;
    }
}
```

1.5) Destruction de la liste

➢ **Destruction des éléments de la liste :**

```
void detruire_liste () {
    while (Taille > 0) {
        Supprimer_debut();
    }
}
```


1.6) Exercice d'application

On considère une liste simplement chainée circulaire contenant des entiers.

1) Donner les déclarations et les fonctions nécessaires pour manipuler une liste de type (LSCC).
2) En se basant sur le principe de manipulation d'une liste de type (LSCC), ajouter une fonction qui permet de supprimer les entiers négatifs.
3) Rédiger la fonction main() pour le test.

1.6) Exercice d'application (Solution)

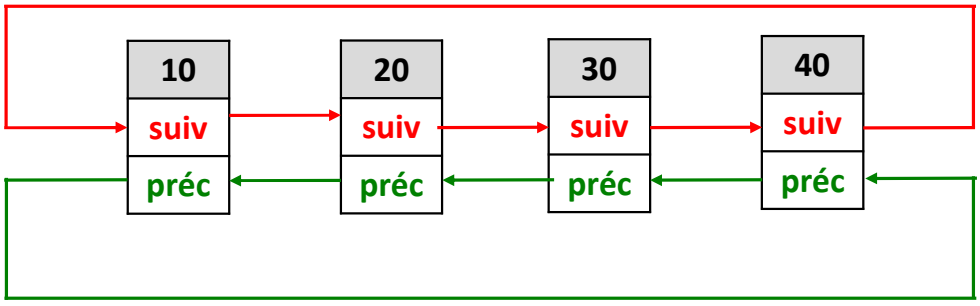
```
//Question 2 :
int rech() {
    Element *courant;
    int i = 1, r = -1;
    courant = Debut;
    while (i <= taille  &&  r == -1) {
        if(courant -> donnee < 0)  r = i;
        else { courant = courant -> suivant; i++; }
    }
    return r;
}
```


1.6) Exercice d'application (Solution)

```
//Question 2 :
void supp_negatif() {
    int pos, i, N;
    N = Taille;
    for (i=1; i <= N; i++) {
        pos = rech();
        if (pos != -1) {
            if( pos == 1) supprimer_debut();
            else  supprimer_ailleurs(pos);
        }
}
```

2) Listes Doublement Chaînées Circulaires

- Les Listes Doublement Chaînées Circulaires (LDCC) sont des Listes Doublement Chaînées, sauf que le champ suivant du dernier élément de la liste pointe sur le premier élément et le champ précédent du premier élément de la liste pointe sur le dernier élément.



2) Listes Doublement Chaînées Circulaires

- Parmi les opérations nécessaires pour manipuler une liste Doublement chaînée circulaire, on trouve :
  - Initialisation
  - 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 avant un élément de la liste
    - Insertion après un élément de la liste
  - Suppression d'un élément de la liste
  - Affichage de la liste
  - Destruction de la liste

2.1) Prototype d'un élément et initialisation

- **Prototype d'un élément de la liste :**

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

- **Initialisation :**

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


2.2) Insertion d'un élément dans la liste

- **Insertion dans une liste vide :**

```
int inserer_liste_vide (int d) {
    Element *nouveau;
    nouveau = (Element*) malloc (sizeof(Element));
    if(nouveau == NULL)  return -1;
    nouveau -> donnee = d;
    nouveau -> precedent = nouveau;   // Liste Circulaire
    nouveau -> suivant = nouveau;   // Liste Circulaire
    Debut = nouveau;
    Fin = nouveau;
    Taille++;
    return 0;
}
```

2.2) Insertion d'un élément dans la liste

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

```
int inserer_debut (int d) {
    Element *nouveau;
    nouveau = (Element*) malloc (sizeof(Element));
    if(nouveau == NULL)  return -1;
    nouveau -> donnee = d;
    nouveau -> suivant = Debut;
    Debut->precedent = nouveau;
    nouveau -> precedent = Fin;  // Liste Circulaire
    Fin-> suivant= nouveau;    // Liste Circulaire
    Debut = nouveau;
    Taille++;
    return 0;
}
```


2.2) Insertion d'un élément dans la liste

➢ **Insertion à la fin de la liste :**

```
int inserer_fin (int d) {
    Element *nouveau;
    nouveau = (Element*) malloc (sizeof(Element));
    if(nouveau == NULL)  return -1;
    nouveau -> donnee = d;
    nouveau -> precedent = Fin;
    Fin -> suivant = nouveau;
    nouveau -> suivant = Debut;   // Liste Circulaire
    Debut -> precedent = nouveau;  //Liste Circulaire
    Fin = nouveau;
    Taille++;
    return 0;
}
```

2.2) Insertion d'un élément dans la liste

➢ **Insertion avant un élément de la liste :**

```
int inserer_avant (int d, int pos) {
    Element *nouveau, *courant;
    nouveau = (Element*) malloc (sizeof(Element));
    if(nouveau == NULL)  return -1;
    nouveau -> donnee = d;  courant = Debut;
    for (int i = 1; i < pos; i++) courant = courant->suivant;
    courant -> precedent -> suivant = nouveau;
    nouveau -> precedent = courant -> precedent;
    nouveau -> suivant = courant; courant -> precedent = nouveau;
    if(pos==1) { Debut = nouveau;
            Debut -> precedent = Fin; // Liste Circulaire
            Fin -> suivant = Debut;     // Liste Circulaire
    }
    Taille++;   return 0;
}
```


2.2) Insertion d'un élément dans la liste

➢ **Insertion après un élément de la liste :**

```
int inserer_apres (int d, int pos) {
    Element *nouveau, *courant;
    nouveau = (Element*) malloc (sizeof(Element));
    if(nouveau == NULL)  return -1;
    nouveau -> donnee = d;  courant = Debut;
    for (int i = 1; i < pos; i++) courant = courant->suivant;
    nouveau -> suivant = courant -> suivant;
    courant -> suivant -> precedent = nouveau;
    nouveau -> precedent = courant;
    courant -> suivant = nouveau;
    if(pos==Taille) {  Fin = nouveau;
            Fin -> suivant = Debut;   // Liste Circulaire
            Debut -> precedent = Fin;  // Liste Circulaire
    }
    Taille++;  return 0;
}
```

2.3) Suppression d'un élément de la liste

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


2.3) Suppression d'un élément de la liste (suite)

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

2.4) Affichage de la liste entière

```
afficher_liste () { // affichage en avançant
    Element *courant;
    int i;
    courant = Debut; // point du départ le 1er élément
    for(i=1; i<=Taille; i++) {
        printf("%d  ", courant -> donnee);
        courant = courant->suivant;
    }
}
```


2.5) Destruction de la liste

```
detruire_liste ()     // suppression élément par élément
{
    while(Taille > 0)
        supprimer(1);    // ou supprimer(Taille);
}
```

2.6) Exercice d'application

On considère une Liste Doublement Chainée Circulaire contenant des entiers.

1) Donner les déclarations et les fonctions nécessaires pour manipuler une liste de type (LDCC).
2) Ajouter une fonction permettant la recherche du minimum et du maximum.
3) Rédiger la fonction main() pour le test.

2.6) Exercice d'application (solution)

```
void min_max() {
    int i, Min, Max;
    Element *courant;
    courant = Debut;
    Min = courant -> donnee;
    Max = courant -> donnee;
    for(i=2; i <= taille; i++) {
        courant=courant->suivant;
        if(courant -> donnee < Min)  Min = courant -> donnee;
        if(courant -> donnee > Max)  Max = courant -> donnee;
    }
    printf("Maximum: %d \n", Max);
    printf("Minimum: %d \n", Min);
}
```