Algorithmique et Programmation : en langage C

Abdellatif HAIR

Université Sultan Moulay Slimane

Faculté des Sciences et Techniques

B.P. 523, Béni-Mellal, MAROC

4. LES LISTES DOUBLEMENT CHAÎNÉES

- DÉFINITION
- CONSTRUCTION DU PROTOTYPE D'UN ÉLÉMENT DE LA LISTE
- OPÉRATIONS SUR LES LISTES DOUBLEMENT CHAÎNÉES
- EXERCICE D'APPLICATION

1. DÉFINITION

◆ Les Listes Doublement Chaînées sont des structures de données semblables aux listes simplement chaînées :

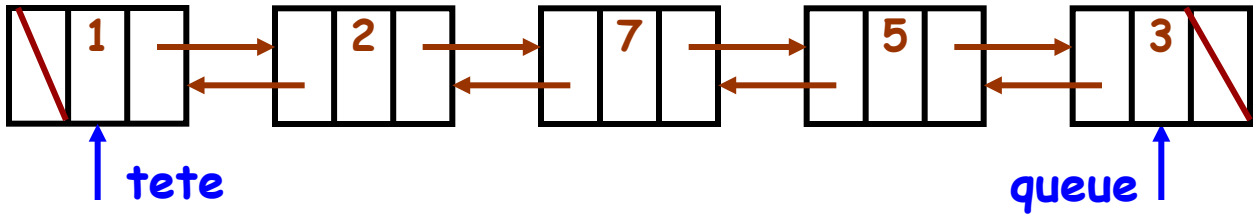
- L'allocation de la mémoire est faite au moment de l'exécution
- La liaison entre les éléments se fait grâce à deux pointeurs (un qui pointe vers l'élément précédent et un qui pointe vers l'élément suivant)
- Le pointeur precedent du premier élément doit pointer vers NULL (le début de la liste)
- Le pointeur suivant du dernier élément doit pointer vers NULL (la fin de la liste)

1. DÉFINITION

◆ Pour accéder à un élément de la Liste Doublement Chaînée (LDC) :

➢ en commençant par la tête (début): le pointeur suivant permettra le déplacement vers le prochain élément

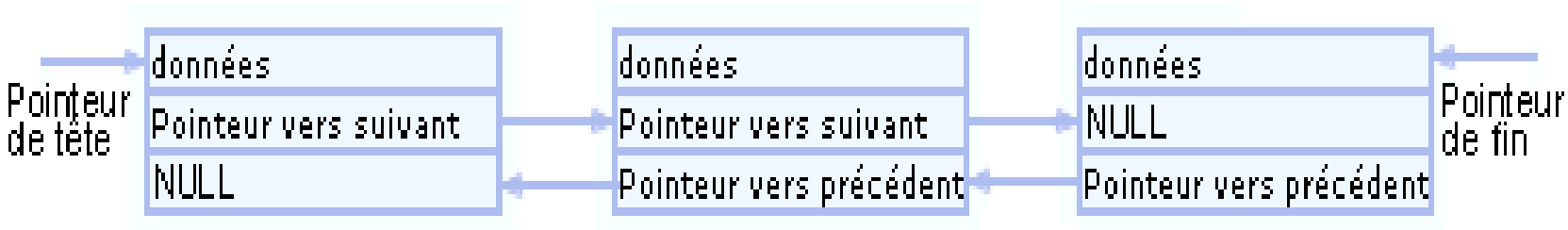
➢ en commençant par la queue (fin) : le pointeur precedent permettra le déplacement vers l'élément précédent



◆ La LDC peut être parcourue dans les deux sens, du premier vers le dernier élément et/ou du dernier vers le premier élément

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

- 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



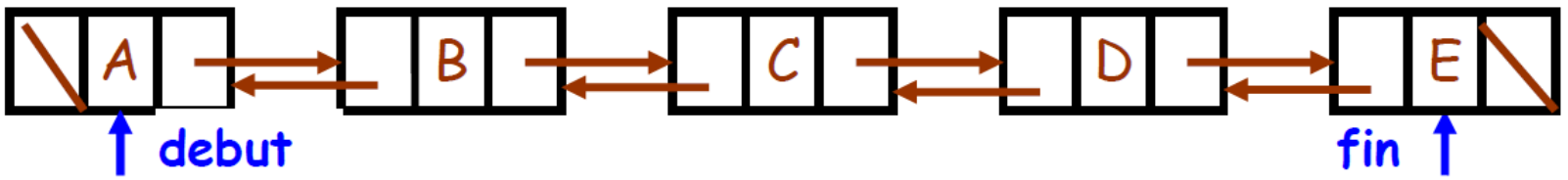
2. CONSTRUCTION DU PROTOTYPE D'UN ÉLÉMENT DE LA LDC

◆ 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 ; } Liste;
```

◆ 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. Liste *debut ;
- Le pointeur *fin* contiendra l'adresse du dernier élément de la liste. Liste *fin ;
- La variable *taille* contient le nombre d'éléments. int taille ;



3. OPÉRATIONS SUR LES LISTES DOUBLEMENT CHAÎNÉES

◆ Nous allons travaillé 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;
```

3. OPÉRATIONS SUR LES LDC

Initialisation

- Prototype de la fonction `initialisation ( );`
- Cette opération doit être exécutée 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 LDC

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

3. OPÉRATIONS SUR LES LDC

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 mise à jour de la Taille

3. OPÉRATIONS SUR LES LDC

Insertion d'un élément dans la liste

1. Insertion dans une liste vide

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

3. OPÉRATIONS SUR LES LDC

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

3. OPÉRATIONS SUR LES LDC

Insertion d'un élément dans la liste

2. Insertion au début de la liste

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

3. OPÉRATIONS SUR LES LDC

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

3. OPÉRATIONS SUR LES LDC

Insertion d'un élément dans la liste

3. Insertion à la fin de la liste

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

3. OPÉRATIONS SUR LES LDC

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.

3. OPÉRATIONS SUR LES LDC

Insertion d'un élément dans la liste

4. 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

3. OPÉRATIONS SUR LES LDC

Insertion d'un élément dans la liste

4. Insertion avant un élément de la liste

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

3. OPÉRATIONS SUR LES LDC

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

3. OPÉRATIONS SUR LES LDC

Insertion d'un élément dans la liste

5. Insertion après un élément de la liste

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

3. OPÉRATIONS SUR LES LDC

Suppression d'un élément dans la liste

- La suppression au début et à la fin de la LDC 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 LDC 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 écrire 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

3. OPÉRATIONS SUR LES LDC

Suppression d'un élément dans la liste

Étapes:

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)

Sinon 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

3. OPÉRATIONS SUR LES LDC

Suppression d'un élément dans la liste

Étapes:

Sinon 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'un élément

3. OPÉRATIONS SUR LES LDC

Suppression d'un élément dans la liste

```
int supp(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 = Fin->precedent;
                    Fin->suivant = NULL;
                  }
```

3. OPÉRATIONS SUR LES LDC

Suppression d'un élément dans la liste

```
else {     /* suppression ailleurs  (pos != 1) && (pos != Taille) */
                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;
}
```

3. OPÉRATIONS SUR LES LDC

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

3. OPÉRATIONS SUR LES LDC

Affichage de la liste

- Pour afficher la liste entière

```
affiche() { /* 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");
}
```

3. OPÉRATIONS SUR LES LDC

Destruction de la liste

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

La fonction

```
detruire () {
while (Taille > 0)     supp(1);
}
```

4. EXERCICE

L'objectif de ce problème est d'écrire un programme C qui permet d'ajouter des nombres entiers strictement positifs en ordre croissant dans une liste doublement chainées.

Pour définir un élément de la liste le type *struct* sera utilisé.

Un élément de la liste contient les trois champs suivants :

***info* de type entier**
***precedent* un pointeur de même type qu'un élément de la liste**
***suivant* un pointeur de même type qu'un élément de la liste**

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

4. EXERCICE

Pour avoir un bon contrôle la LDC , on sauvegarde les éléments suivants :

pointeur *Debut* : contient l'adresse du premier élément de la liste.

pointeur *Fin* : contient l'adresse du du dernier élément de la liste

variable *NElements* contient le nombre d'éléments de la liste

Les variables globales du programme sont :

```
Element *Debut;
Element *Fin;
int NElements;
```

4. EXERCICE

1- Ecrire la fonction qui permet d'initialiser les pointeurs Debut et Fin à NULL et le NElements avec la valeur 0.

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

4. EXERCICE

2- Ecrire la fonction qui permet d'insérer un élément dans une liste vide

```
int ins_dans_liste_vide(int i) {
Element *element;
if ((element = (Element *)malloc(sizeof(Element)))==NULL) return -1;
element-> info = i ;
element-> suivant = NULL;
element-> precedent = NULL;
Debut = element;
Fin = element;
NElements++;
return 0;
}
```

4. EXERCICE

3- Ecrire la fonction qui permet d'insérer un élément à la fin d'une liste non vide

```
int ins_fin_liste (int i) {
Element *element;
if ((element = (Element *)malloc(sizeof(Element)))==NULL) return -1;
element-> info = i ;
element-> suivant = NULL;
Fin->suivant = element;
element->precedent = Fin;
Fin = element;
NElements++;
return 0 ;
}
```

4. EXERCICE

4- Ecrire la fonction qui permet d'insérer un entier dans la liste en ordre croissant

```
int Insertion_Ordre_Croissant(int i) {
Element *courant, * e;
if (NElements==0) return ins_dans_liste_vide(i);
else {   courant=Debut;
         while ((courant !=NULL) && (courant->info <=i)) courant=courant->suivant;
         if (courant ==NULL) return ins_fin_liste(i);
         else     {   if ((e = (Element *)malloc(sizeof(Element)))==NULL) return -1;
                      e-> info = i ;
                      e->suivant=courant ;
                      e->precedent=courant->precedent;
                      if (courant->precedent==NULL)  Debut=e;
                      else courant->precedent->suivant=e;
                      courant->precedent = e;
                      NElements++;
                      return 0;
                  }
}
}
```

4. EXERCICE

5- Ecrire la fonction qui permet d'afficher la suite des nombres en ordre croissant

```
Afficher_Ordre_Croissant() {
Element *p ;
p=Debut;
while (p != NULL) {
        printf("%d" , p->info);
        p = p->suivant;
}
printf("\n");
}
```

4. EXERCICE

6- Ecrire la fonction qui permet d'afficher la suite des nombres en ordre décroissant

```
Afficher_Ordre_Decroissant() {
Element *p ;
p=Fin;
while (p != NULL) {
        printf("%d" , p->info);
        p = p->precedent ;
}
printf("\n");
}
```

4. EXERCICE

7- Ecrire la fonction main() pour tester toutes les fonctions

```
main(){
int donnee;
initialisation ( );
do {
     printf("saisir un entier et pour terminer taper 0 : ");
     scanf("%d", &donnee);
     if (donnee != 0) Insertion_Ordre_Croissant(donnee);
     else break;
} while (1) ;
printf("Ordre Croissant  \n"); Afficher_Ordre_Croissant ( );
printf("Ordre Décroissant  \n"); Afficher_Ordre_Deroissant ( );
system("pause");
}
```