Programmation: en langage C

LES LISTES DOUBLEMENT CHAÎNÉES

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

1. DÉFINITION

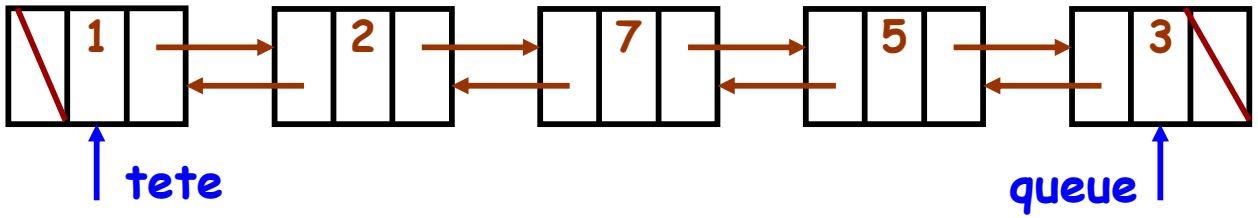
◆ Les Listes Doublement Chaînées sont des structures de données semblables aux listes simplement chaînées :

- 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 la liste doublement chaînée :

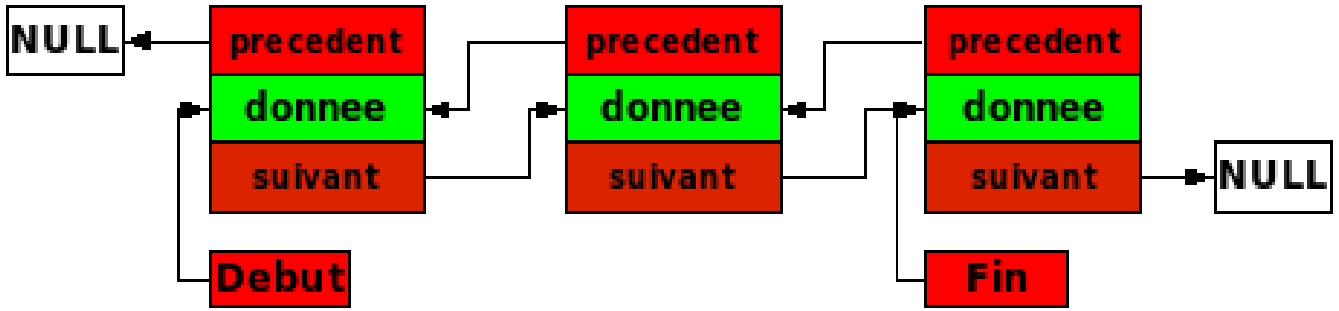
- en commençant avec la tête : le pointeur suivant permettant le déplacement vers le prochain élément
- en commençant avec la queue : le pointeur precedent permettant le déplacement vers l'élément précédent



◆ 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

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*, 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



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;
```

3. OPÉRATIONS SUR LES L.D.C.

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 le pointeur vers le 1er élément si nécessaire

3. OPÉRATIONS SUR LES L.D.C.

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 L.D.C.

Insertion d'un élément dans la liste

1. Insertion dans une liste vide

Étapes :

- allocation de la mémoire pour le nouvel élément
- remplir le champ de données du nouvel élément
- le pointeur precedent du nouvel élément pointera vers NULL
- le pointeur suivant du nouvel élément pointera vers NULL
- le pointeur Debut pointera vers le nouvel élément

3. OPÉRATIONS SUR LES L.D.C.

Insertion d'un élément dans la liste

1. Insertion dans une liste vide

```
Element*ins_dans_liste_vide (char*info) {
Element *nou_element;
nou_element=(Element*)malloc(sizeof(Element));
if (nou_element==NULL) exit(1);
nou_element->info = (char *) malloc (50 * sizeof(char));
strcpy (nou_element-> info, info);
nou_element->precedent = NULL;
nou_element->suivant = NULL;
return nou_element;}
```

3. OPÉRATIONS SUR LES L.D.C.

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 Debut pointe vers le nouvel élément

3. OPÉRATIONS SUR LES L.D.C

Insertion d'un élément dans la liste

2. Insertion au début de la liste

```
Element* ins_debut_liste (Element*Lis, char *info) {
Element *nou_element=ins_dans_liste_vide (info) ;
if(Lis==NULL) Lis=nou_element;
else{ nou_element->suivant = Lis;
      Lis->precedent=nou_element;}
Lis=nou_element;
return Lis;}
```

3. OPÉRATIONS SUR LES L.D.C

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 suivant du dernier élément va pointer vers le nouvel élément
- le pointeur Debut ne change pas

3. OPÉRATIONS SUR LES L.D.C

Insertion d'un élément dans la liste

3. Insertion à la fin de la liste

```
Element*ins_fin_liste (Element*Lis, char *info){
Element *nou_element= ins_dans_liste_vide (info);
if(Lis==NULL) Lis= nou_element;
else{
Element*temp=Lis;
        while(temp->suivant!=NULL){
      temp=temp->suivant; }
nou_element->precedent = temp;
temp->suivant=nou_element;}
return nou_element;}
```

3. OPÉRATIONS SUR LES L.D.C

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 L.D.C

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

3. OPÉRATIONS SUR LES L.D.C

Insertion d'un élément dans la liste

4. Insertion avant un élément de la liste

```
Element*rech_position(Element*Lis, int pos) {int i;
Element*courant = Lis;
for (i = 1; i < pos; i++) courant = courant->suivant;
return courant; }
 Element* ins_avant (Element*Lis, char *info, int pos) {
Element *nou_element, *courant;
courant=rech_position(Lis, pos);
nou_element= ins_dans_liste_vide (info);
nou_element->suivant = courant;
nou_element-> precedent = courant->precedent;
if(courant->precedent == NULL) Lis= nou_element;
else courant->precedent->suivant = nou_element;
courant->precedent = nou_element;
return Lis; }
```

3. OPÉRATIONS SUR LES L.D.C

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 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

3. OPÉRATIONS SUR LES L.D.C

Insertion d'un élément dans la liste

5. Insertion après un élément de la liste

```
Element*ins_apres (Element*Lis, char *info, int pos){
Element *nou_element, *courant;
nou_element= ins_dans_liste_vide (info);
courant = rech_position(Lis, pos);
nou_element->precedent = courant;
nou_element->suivant = courant->suivant;
if(courant->suivant != NULL)
            courant->suivant->precedent = nou_element;
courant->suivant = nou_element;
return Lis;
}
```

3. OPÉRATIONS SUR LES L.D.C

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 au 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

3. OPÉRATIONS SUR LES L.D.C

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 faisons pointer le pointeur precedent du 2ème élément vers NULL)

La position choisie est égale au nombre d'éléments (taille(Lis)) 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

3. OPÉRATIONS SUR LES L.D.C

Suppression d'un élément dans la liste

Étapes:

La position choisie est aléatoire dans la liste

- le pointeur supp_element contiendra l'adresse de l'élément à supprimer
- le pointeur suivant d'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

3. OPÉRATIONS SUR LES L.D.C

Suppression d'un élément dans la liste

```
Element*supp(Element*Lis, int pos) { int i;
          Element *supp_element,*courant;
if(taille(Lis) == 0) exit(1);
if(pos == 1) { /* suppression de 1er élément */
    supp_element = Lis;
    Lis= Lis->suivant;
    if(Lis!== NULL) Lis->precedent = NULL;
    }
else if(pos == taille(Lis) { /* suppression du dernier élément */
                  supp_element = dernier(Lis);
                  dernier(Lis)->precedent->suivant = NULL;
                  }
```

3. OPÉRATIONS SUR LES L.D.C

Suppression d'un élément dans la liste

```
else {  /* suppression ailleurs */
                courant = Lis;
                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);
return Lis
}
```

3. OPÉRATIONS SUR LES L.D.C

Affichage de la liste

- Pour afficher la liste entière
  - se positionner au début de la liste ou à la 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 L.D.C

Affichage de la liste

- Pour afficher la liste entière

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

3. OPÉRATIONS SUR LES L.D.C

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 (Element*Lis) {
while (taille(Lis)> 0) Lis=supp(Lis, 1) ;
}
```