Cours: ASD 2 & Complexité

Chapitre 7: Les Listes Linéaires

Réalisé par:
Dr. Sakka Rouis Taoufik

1

Ch 7: Les Listes Linéaires

I. Introduction

- Une liste linéaire (LL) est la représentation informatique d'un ensemble fini, de taille variable et éventuellement nul, d'éléments de type T. Un tel ensemble est ordonnée.

- Mathématiquement, on peut présenter une LL en énumérant ses éléments dans l'ordre.

- Pour la SD pile, les adjonctions (empiler), les suppressions (depiler) et les recherches (dernier) sont faites par rapport au sommet. On dit que la SD pile est une structure à un seul point d'accès.

- Pour la SD file, les adjonctions (enfiler) sont faites par rapport au queue, les suppressions (défiler) et les recherches (premier) sont faites par rapport à la tête. On dit que la SD file est une structure à deux points d'accès.

- Pour la SD LL, les adjonctions, les suppressions et les recherches ne sont pas faites systématiquement ni par rapport à la tête, ni par rapport à la queue.

2

Ch 7: Les Listes Linéaires

II. Opérations sur la SD LL

2.1 Les opérations élémentaires

- **creerliste** : permettant de créer une liste linéaire vide.
- **listevide** : permettant de voir si la liste linéaire est vide ou non ?
- **ajouter** ou opération d'**adjonction :** - en tête : avant le premier
  - en queue : après le dernier
  - quelque part au milieu
- **supprimer** un élément de la liste : - premier élément
  - dernier élément
  - quelque part au milieu
- **recherche** : permettant de voir si un élément appartient dans une liste linéaire. Le point de départ peut être fourni comme paramètre.
- **Visiter** : permettant de visiter tous les éléments de la SD LL en effectuant pour chaque élément visité une action donnée (paramètre). Cette action est connue sous le nom de **traversée**.

3

Ch 7: Les Listes Linéaires

II. Opérations sur la SD LL

2.2 Les opérations élaborés

- **Inversion** permettant d'inverser une liste linéaire

ll=(a, b, c, d) ll_{inv}=(d, c, b, a)

- **supprimer_tous** : permet de supprimer tous les éléments d'une liste donnée.

- **concaténation** permet de concaténer deux listes données.

ll_1=(a, b, c, d)

ll_2=(x, y, z, w, k)

La concaténation de ll1 et ll2 dans l'ordre (ordre significatif) donne :

ll=(a, b, c, d, x, y, z, w, k)

4

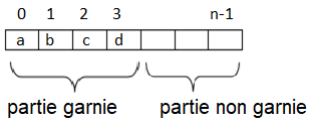
Ch 7: Les Listes Linéaires

III. Représentation physique

3.1 Représentation contiguë

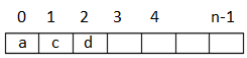
- **Illustration :** ll=(a, b, c, d)

représentation abstraite



- **suppression** : elle exige des décalages à gauche pour récupérer la position devenue disponible.

  Exemple : supprimer b



5

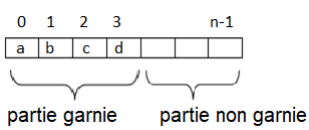
Ch 7: Les Listes Linéaires

III. Représentation physique

3.1 Représentation contiguë

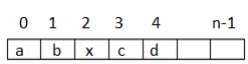
- **Illustration :** ll=(a, b, c, d)

représentation abstraite



- **visiter** : balayer tous les éléments d'un tableau dans la partie garnie
- **recherche** ➔ recherche dans un tableau, on fait appel aux algorithmes connus soit recherche séquentielle soit dichotomique
- **adjonction** : elle exige des décalages à droite.

  Exemple : ajouter x après b.



6

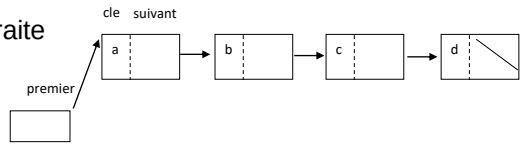
Ch 7: Les Listes Linéaires

III. Représentation physique

3.2 Représentation chaînée

- **Illustration :** ll=(a, b, c, d)

représentation abstraite



- L'adjonction dans une liste linéaire (LL) concrétisée à l'aide d'une représentation chaînée n'exige pas de **décalages** contrairement à la représentation contiguë.

➔ On peut dire que la représentation contiguë de la SDLL n'est pas recommandée à cause des décalages impliqués par les deux opérations fondamentales ajouter et supprimer notamment pour les listes linéaires de taille plus au moins importante.

7

Ch 7: Les Listes Linéaires

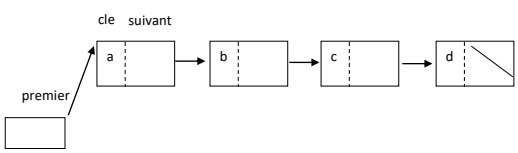
IV. Variantes de la SD liste linéaire

4.1 LL uni directionnelle avec un seul point d'entrée

- **Illustration :** ll=(a, b, c, d)

Représentation abstraite

Variante 1:



8

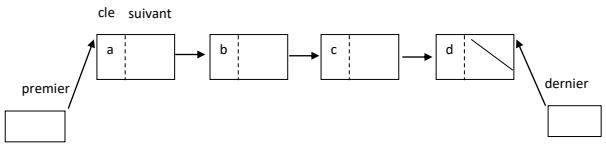
Ch 7: Les Listes Linéaires

IV. Variantes de la SD liste linéaire
4.2 LL uni directionnelle avec deux points d'entrée

- **Illustration :** ll=(a, b, c, d)

Représentation abstraite

Variante 2:



- Pour les deux variantes (1) et (2), la liste linéaire **est unidirectionnelle**. À partir d'un élément donné on peut passer à son successeur. Ceci est possible grâce au champ de chaînage suivant : dans la variante (1), le premier élément est privilégié (accès direct) dans la variante (2) le premier et le dernier élément sont privilégiés (accès directe).

9

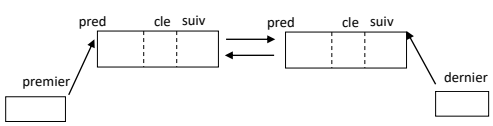
Ch 7: Les Listes Linéaires

IV. Variantes de la SD liste linéaire
4.3 LL bidirectionnelle avec 2 points d'entrée

- **Illustration :** ll=(a, b, c, d)

Représentation abstraite

Variante 3:



- À partir d'un élément donné, on peut passer soit à son successeur soit à son prédécesseur.

10

Ch 7: Les Listes Linéaires

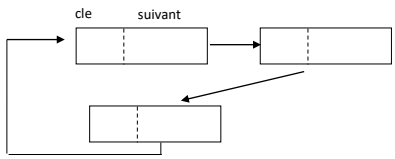
IV. Variantes de la SD liste linéaire

4.4 LL Circulaire ou Anneau

- **Illustration :** ll=(a, b, c, d)

Représentation abstraite

Variante 4:



- Les notions de premier et dernier disparaissent, c'est-à-dire ces notions n'ont pas de sens dans un anneau. Un anneau est doté uniquement d'un point d'entrée quelconque.

11

Ch 7: Les Listes Linéaires

V. Matérialisation de liste chainée à 2 points d'entrée

- En supposant que les éléments de la liste sont des entiers, celle-ci se déclare de la façon suivante :

```
/*représentation physique*/

struct noeud {
   int cle;
   struct noeud *suivant;
};

struct liste {
   struct noeud *premier;
   struct noeud *dernier;
};
```

12

Ch 7: Les Listes Linéaires

V. Matérialisation de liste chainée à 2 points d'entrée
V.1 Création d'une liste chaînée

Solution1 : sous forme d'une procédure

```
void creer_liste (struct liste *ll) {
   ll->premier = NULL;
   ll->dernier = NULL;
}
```

Solution 2 : sous forme d'une fonction

```
        struct liste * creer_liste (void) {
        struct liste *ll ;
        ll=(struct liste*) malloc (sizeof (struct liste) ) ;
        ll-> premier=NULL;
        ll-> dernier=NULL;
        return ll ;
};
```

13

Ch 7: Les Listes Linéaires

V. Matérialisation de liste chainée à 2 points d'entrée
V.2 Tester la vacuité d'une liste linéaire

Solution (1)

```
unsigned liste_vide (struct liste *ll) {
        return (ll->premier==NULL) ;
}
```

Solution (2)

```
unsigned liste_vide (struct liste*ll) {
        return ((ll -> premier==NULL)&&(ll -> dernier==NULL)) ;
}
```

La solution (2) est cohérente par rapport à la réalisation de créer liste

14

V. Matérialisation de liste chainée à 2 points d'entrée
V.3 Processus d'adjonction ou d'insertion

- On distingue quatre types d'insertions :
  - insérer après un élément référencée
  - insérer avant un élément référencée
  - insérer avant premier
  - insérer après dernier

V. Matérialisation de liste chainée à 2 points d'entrées
V.3 Processus d'adjonction ou d'insertion

/*insertion après élément référencée*/

```
void inserer_apres (int info, struct noeud *p) {
        struct noeud *q;
        q=(struct noeud *) malloc (sizeof (struct noeud));
        q->cle=info;
        q->suivant=p->suivant;
        /*mise à jour du successeur de p*/
        p->suivant=q ;
}
```

V. Matérialisation de liste chainée à 2 points d'entrée
V.3 Processus d'adjonction ou d'insertion

/*insertion avant un élément référencé*/

```
void inserer_avant (int info, struct nœud*p) {
    /*le problème : la liste est unidirectionnelle à partir de p,
       on ne peut pas passer à son prédécesseur */

      struct noeud * q;
      q=(struct  noeud*) malloc (sizeof (struct noeud ));
      *q=*p;    /*  q->cle=p->cle; q->suivant=p->suivant ; */
  /*mise à jour l'espace référencé par p*/

      p->cle=info ;
      p->suivant=q ;
}
```


V. Matérialisation de liste chainée à 2 points d'entrée
V.3 Processus d'adjonction ou d'insertion

/*insertion avant le premier élément*/

```
void  inserer_avant_premier (int info, struct liste *ll) {
      struct noeud * q;
      q=(struct noeud*) malloc (sizeof(struct noeud));
      if (liste_vide (ll)) {
           q->cle=info;
           q->suivant=NULL ;
           ll->premier=q ;
           ll->dernier=q ;      }
     else {
          q->cle = info;
          q->suivant = ll->premier ;
          ll->premier = q ;     }
}
```

V. Matérialisation de liste chainée à 2 points d'entrée
V.3 Processus d'adjonction ou d'insertion

```
/*insertion après le dernier*/
void  inserer_apres_dernier (int info, struct liste *ll) {
        struct noeud * q ;
        q=(struct noeud*) malloc (sizeof (struct noeud)) ;
        q->cle=info ;
        q->suivant=NULL ;
        if (liste_vide (ll)) {
            ll->premier=q ;
            ll->dernier=q ;
       }
       else {
            ll->dernier->suivant=q  ;
            ll->dernier=q ;

      }
}
```

19

V. Matérialisation de liste chainée à 2 points d'entrée
V.4 Recherche d'un élément dans une lise chaînée

```
struct nœud *cherche (int info, struct nœud* p )
/* elle rend une référence non nulle si info appartient à la liste à partir de l'élément référencé par p sinon elle rend une référence nulle*/
/*point commun :algorithme de recherche séquentiel dans un tableau, dans un contexte d'une liste linéaire*/

struct noeud * chercher (int info, struct noeud* p) {
        while(p && (p->cle !=info))  /* l'ordre des deux sous expressions est significatif*/
                    p=p->suivant ;
       /*à la sortie de la boucle !p || p->clé==info
                  échec : !p  ⇔ (p== NULL)  ➔ return p
                  succès :p->cle==info    ➔ return p  */
        return (p) ;
}
```

20

V. Matérialisation de liste chainée à 2 points d'entrée

V.5 Parcours d'une lise chaînée

Visiter tous les éléments d'une liste linéaire en effectuant un traitement donné.

Rappel sur les paramètres finis en C :

- **Paramètre non fonctionnel** : il s'agit d'un paramètre qui mémorise une valeur (ordinaire, adresse).
- **Paramètre fonctionnel** : il mémorise un comportement (ou un traitement) traduit par un sous programme. Les sous programmes qui admettent des paramètres fonctionnels sont dits des sous programmes **d'ordre supérieur**

Quels sont les paramètres nécessaires à l'opération de parcours ?

- Un paramètre non fonctionnel : qui indique le point de départ pour traverser la liste **struct nœud *p**
- Un paramètre fonctionnel : qui indique le traitement à effectuer sur les nœuds visités **void (*oper) (struct nœud*) ;** avec **Oper** est un pointeur sur une procédure exigeant un paramètre de type **struct nœud ***

RQ. À ne pas confondre avec **void* oper (struct nœud *)** ; Il s'agit d'une fonction appelée oper qui rend un pointeur sur n'importe quoi et qui exige un paramètre de type struct nœud*.

V. Matérialisation de liste chainée à 2 points d'entrée

V.5 Parcours d'une lise chaînée

```
void parcours (struct noeud* p, void (*oper) (struct noeud*) ) {
      while (p) {
         /*  appliquer à l'élément porté par p le traitement fourni par oper    */
         (*oper) (p) ;
         /*  passer à l'élément suivant */
         p=p->suivant ;
      }
}
```

Ch 7: Les Listes Linéaires

V. Matérialisation de liste chainée à 2 points d'entrée
V.5 Parcours d'une lise chaînée

Utilisation de la procédure parcours :

```
struct noeud * point_de_depart ;
…

void afficher (struct noeud * q) {
    printf ("%d \n", q->cle) ;       /* afficher l'élément visité */

}
void incrementer (struct noeud *q)    {
    q->cle++ ;                        /* incrémenter l'élément visité  */

}

/* activation de parcours */
parcours (point_de_depart, afficher) ;
parcours (point_de_depart, incrementer) ;
parcours (point_de_depart, afficher) ;
```

En c, le nom d'un sous programme est considéré comme un pointeur. Toute référence à (*oper) au sein de la P.E du sous programme parcours est une référence au paramètre effectif correspond : c'est le principe d'indirection.

23

Ch 7: Les Listes Linéaires

V. Matérialisation de liste chainée à 2 points d'entrée
V.6 Processes de suppression

- On distingue trois types de suppression:
  - supprimer un élément référencée,
  - supprimer le premier
  - supprimer le dernier

24

V. Matérialisation de liste chainée à 2 points d'entrée

V.6 Processes de suppression

```
/* suppression d'un élément référencée */

void supprimer_elem (struct noeud * p ) {
    struct noeud* q ;
   /* on suppose que p admet un successeur*/
    assert (p->suivant != NULL) ;
    q=p->suivant ;
    *p=*q ;
    free (q) ;
}
```


V. Matérialisation de liste chainée à 2 points d'entrée

V.6 Processes de suppression

```
/* suppression du premier élément */

void supprimer_premier (struct liste *ll){
     struct nœud* q ;
     assert (!liste_vide (ll))  ;
     q=ll->premier ;
     ll->premier = q->suivant;
     free (q) ;
     if (ll->premier== NULL)
        ll->dernier= NULL ;
}
```

V. Matérialisation de liste chainée à 2 points d'entrée

V.6 Processes de suppression

```
/* suppression du dernier élément */
void supprimer_dernier (struct liste *ll) {
    struct noeud* q;
    if (ll->premier == ll->dernier)
        supprimer_premier(ll);
    else{
        q = ll->premier;
        while (q->suivant != ll->dernier)
            q = q->suivant;
        q->suivant = NULL;
        free(ll->dernier);
        ll->dernier = q;
    }
}
```

27

VI. Matérialisation de L. L. bidirectionnelle à 2 points d'entrée

```
/*représentation physique*/
struct noeud {
        struct noeud * pred ;
        int cle ;
        struct noeud * succ ;
};
struct liste {
        struct noeud* premier ;
        struct noeud * dernier ;
};

/* création : sous forme d'une procédure */
void creer_liste (struct liste * ll) {
        ll->premier=NULL ;
        ll->dernier=NULL ;
}
```

28

Ch 7: Les Listes Linéaires

VI. Matérialisation de L. L. bidirectionnelle à 2 points d'entrée

```
/*insertion après élément référencée*/
void inserer_apres (int info, struct noeud *p) {
        struct noeud *q ;
        q=(struct noeud*) malloc (sizeof (struct noeud)) ;
        q→cle=info;
        q->succ=p→succ ;
        q->pred=p ;
        p->succ->pred=q ;
        p→succ=q ;
}
```

29

Ch 7: Les Listes Linéaires

VI. Matérialisation de L. L. bidirectionnelle à 2 points d'entrée

```
/*insertion avant un élément référencé*/

void inserer_avant (int info, struct noeud*p) {
      struct noeud * q ;
      q=(struct noeud*) malloc (sizeof (struct noeud)) ;
      q->cle=info ;
      q->succ=p ;
      q->pred=p->pred ;
      p->pred->suiv=q ;
      p->pred=q ;

}
```

30

Ch 7: Les Listes Linéaires

VI. Matérialisation L. L. bidirectionnelle à 2 points d'entrée

La procédure suivante permet de parcourir et afficher les éléments d'une liste à chaînage double en commençant par le dernier élément.

```
void AffichInvListe (struct liste * ll ) {
        struct noeud * p ;
        p=ll->dernier
        while (p) {
        printf("%d \n ", p->cle);
        p = p->pred ;
        }
}
```

31

Ch 7: Les Listes Linéaires

VI. Exercices d'application

Exercice 1: Matrices creuses

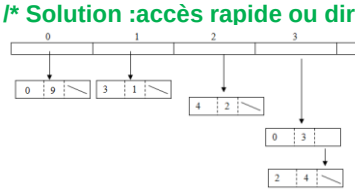
Trouver un représentation informatique des matrices creuses. Une matrice creuse est une matrice comportant un nombre important d'éléments nuls.
➔La représentation souhaitée doit stocker uniquement les éléments non nuls.
exemple

9	0	0	0	0
0	0	0	1	0
0	0	0	0	2
3	0	4	0	0
0	0	0	0	0

/* Solution classique* /

```
#define n 100
float m [n] [n] ;
```

/* Solution :accès rapide ou directe sur la ligne */



➔Tableau de listes, chaque ligne est représente par une liste linéaire dont chaque nœud comporte :
col : colonne
inf : valeur m[i][col]
suivant : successeur non nul relative à la ligne i

32

VI. Exercices d'application

Exercice 1: Matrices creuses

La colonne appartenant à une ligne i est obtenu par recherche séquentielle au sein de la liste ligne[i]
On a intérêt à trier les listes sur le champ colonne.

Traduction en c

```
#define n 100
struct element {
   unsigned col ;
   float info ;
   struct element * suivant ;
} ;
struct element * ligne [n] ;
```

Une matrice creuse représentée ci- dessus peut être assimilée à une SD dotés des opérations suivantes :

- Création d'une matrice creuse vide : tous les éléments sont à zéro.
- Accès à un élément identifié par le couple (ligne :i, colonne : j)
- Modification d'un élément donnée

33

VI. Exercices d'application

Exercice 2: paramètre fonctionnel

a) Réaliser en C un sous-programme permettant d'afficher les éléments d'une liste linéaire répondant à un critère de sélection donné. Un tel critère peut exprimer une condition quelconque.

b) Donner un exemple d'utilisation du sous-programme établi en a.

34

VI. Exercices d'application

Exercice 2: paramètre fonctionnel

Correction :

```
struct noeud {
        int cle ;
        struct noeud *suivant ;
} ;
void afficheSelectif (struct noeud *p, unsigned (*cond) ( struct nœud *)) {
        while (p) {
                if ((*cond)(p))
                printf("%d\n", p->cle) ;
                p=p->suivant ;
        }
 }      /*question b*/
unsigned critere (struct noeud *p) {
        return p->cle >= 10 ;
}
void main (void) {
        struct nœud *debut ;
        …     /*créer la liste et ajouter des éléments  à la liste*/
         afficheSelectif (debut, critere) ;
}
```


VI. Exercices d'application

Exercice 3: Inversion

a) Écrire une méthode AfficheInvListe1 permettant d'afficher inversement les éléments d'une liste linéaire unidirectionnelle.

b) Écrire une méthode AfficheInvListe2 permettant d'afficher inversement les éléments d'une liste linéaire bidirectionnelle.

c) Écrire une méthode InverserListe1 permettant d'inverser une liste linéaire unidirectionnelle.

d) Écrire une méthode InverserListe2 permettant d'inverser une liste linéaire bidirectionnelle.

Ch 7: Les Listes Linéaires

VI. Exercices d'application

Exercice 4: Tri Fusion

A- Proposer une méthode qui fait la concaténation de deux listes (en entrée) retournant une liste (en sortie). Par exemple, si l'on a les listes (3, 7, 2) et (5, 4) on obtient la liste (3, 7, 2, 5, 4).

B- Proposer ensuite une méthode séparant une liste en deux. Etant donné une liste en entrée, un élément sur deux va dans la première liste et un élément sur deux va dans la deuxième liste.

C- Faire une action de fusion de listes supposées triées par ordre croissant en une troisième liste triée.

D- Faire, en utilisant au choix les trois actions précédentes une action de tri fusion qui étant donné une liste en entrée retourne une liste triée.

37

Ch 7: Les Listes Linéaires

VI. Exercices d'application

Correction Exercice 4: Tri Fusion

```
struct liste * concatenerListes (struct liste *L1, struct liste *L2) ; /* à implémenter */
void Separer (struct liste *L , struct liste *L1, struct liste *L2) ; /* à implémenter */
struct liste * Fusion (struct liste * L1, struct liste *L2) ; /* à implémenter */
struct liste * TriFusion (struct liste * L) {
   struct liste *L1= (struct liste*) malloc (sizeof(struct liste));
   struct liste *L2= (struct liste*) malloc (sizeof(struct liste));
   struct liste *L1b;
   struct liste *L2b;
   assert (! liste_vide(L) );
   if (L->premier->suivant)  { /* contient plus qu'un élément */
      Separer (L, L1, L2 );
      L1b= TriFusion (L1);
      L2b= TriFusion (L2);
      return Fusion (L1b, L2b);    }
   else {
      creer_liste (L1);
      inserer_avant_premier (L->premier->cle, L1);
      return L1;                    }
}
```

38

VI. Exercices d'application

Correction Exercice 4: Tri Fusion

```
/*Solution 1  */
struct liste * concatenerListes (struct liste *L1, struct liste *L2) {
    struct noeud *P1, *P2;
    struct liste * nouvelleListe = (struct liste *) malloc(sizeof(struct liste));
    creer_liste (nouvelleListe) ;
    P1= L1-> premier;
    while (P1){
        inserer_apres_dernier(P1->cle, nouvelleListe);
        P1=P1-> suivant;
    }
    P2= L2-> premier;
    while (P2){
        inserer_apres_dernier(P2->cle, nouvelleListe) ;
        P2=P2-> suivant;
    }
    return nouvelleListe ;
}
```


VI. Exercices d'application

Correction Exercice 4: Tri Fusion

```
/*Solution 2 */
struct liste *concatenerListes (struct liste *L1, struct liste * L2) {
    struct liste *nouvelleListe = (struct liste *) malloc(sizeof(struct liste));
    creer_liste (nouvelleListe);
    while(! liste_vide(L1)){
        inserer_apres_dernier(L1->premier->cle, nouvelleListe);
        supprimer_premier(L1);
    }
    while(! liste_vide(L2)){
        inserer_apres_dernier(L2->premier->cle, nouvelleListe);
        supprimer_premier(L2);
    }


    return nouvelleListe;
}
```

VI. Exercices d'application

Correction Exercice 4: Tri Fusion

```
/*Solution 3  (la bonne solution)*/
struct liste *concatenerListes (struct liste *L1, struct liste *L2) {
    struct liste *nouvelleListe = (struct liste *) malloc(sizeof(struct liste));
    if (liste_vide (L1)) {
        nouvelleListe->premier = L2->premier;
        nouvelleListe->dernier  = L2->dernier;
    } else {
        nouvelleListe->premier = L1->premier;
        nouvelleListe->dernier  = L2->dernier;
        L1->dernier->suivant = L2->premier;
    }

    return nouvelleListe;
}
```


VI. Exercices d'application

Correction Exercice 4: Tri Fusion

```
/*Solution 4 (la bonne solution)*/ */
struct liste * concatenerListes (struct liste *L1, struct liste *L2) {
   if (liste_vide (L1)) {
          return L2 ;
    } else {
             L1->dernier->suivant = L2->premier;
             L1->dernier = L2->dernier;
             return L1;

    }

}
```

VI. Exercices d'application

Correction Exercice 4: Tri Fusion

```
/*solution 1*/
void Separer (struct liste *L , struct liste * L1, struct liste * L2) {
   struct noeud* P ;
   P = L->premier;
   int  B = 1;
   creer_liste (L1);
   creer_liste (L2);

   while (P) {
      if (B>0)
         inserer_apres_dernier(P->cle, L1);
       else
         inserer_apres_dernier(P->cle, L2);

      B *= -1;
      P = P->suivant;
   }
}
```


VI. Exercices d'application

Correction Exercice 4: Tri Fusion

```
/*solution 2 (la bonne solution)*/
void Separer (struct liste *L , struct liste * L1, struct liste * L2) {

   int  B = 1;
   creer_liste (L1); /*obligatoire */
   creer_liste (L2); /*obligatoire */
   while (!liste_vide(L)) {
      if (B>0)
         inserer_apres_dernier(L->premier->cle, L1);
      else
         inserer_apres_dernier(L->premier->cle, L2);

      supprimer_premier(L);
      B *= -1;
   }
}
```

Ch 7: Les Listes Linéaires

VI. Exercices d'application

Correction Exercice 4: Tri Fusion

```
/*solution 1*/
struct liste * Fusion (struct liste * L1, struct liste * L2){
   struct liste * L = (struct liste *) malloc (sizeof(struct liste));
   struct noeud *P1 , *P2;
   P1= L1->premier;
   P2= L2->premier;
   creer_liste(L);
   while (P1 && P2 ) {
      if (P1->cle < P2->cle) {
         inserer_apres_dernier(P1->cle, L);
         P1 = P1->suivant;
         supprimer_premier(L1);
      } else {
         inserer_apres_dernier (P2->cle, L);
         P2 = P2->suivant;
         supprimer_premier(L2);
      }
   }
   while (P1 ) {
         inserer_apres_dernier (P1->cle, L);
         P1 = P1->suivant;
      }
      while (P2) {
         inserer_apres_dernier (P2->cle, L);
         P2 = P2->suivant;
      }
      return L;
   }
```

45

Ch 7: Les Listes Linéaires

VI. Exercices d'application

Correction Exercice 4: Tri Fusion

```
/*solution 2 (la bonne solution)*/
struct liste * Fusion (struct liste * L1, struct liste * L2) {
   struct liste *L =(struct liste *) malloc(sizeof(struct liste));
   creer_liste(L);

   while (!liste_vide(L1) && !liste_vide(L2) ) {

      if (L1->premier->cle < L2->premier->cle) {
         inserer_apres_dernier(L1->premier->cle, L);
         supprimer_premier(L1);
      } else {
         inserer_apres_dernier(L2->premier->cle, L);
         supprimer_premier(L2);
      }
   }
   return (liste_vide(L1) ? concatenerListes (L, L2) : concatenerListes (L, L1));
}
```

46

VI. Exercices d'application

Correction Exercice 4: Tri Fusion

```
void main() {
    struct liste *liste1 = (struct liste*) malloc (sizeof(struct liste));
    struct liste *liste2 ;
    creer_liste (liste1);
    inserer_apres_dernier(-10, liste1);
    inserer_apres_dernier(500, liste1);
    inserer_apres_dernier(300, liste1);
    inserer_apres_dernier(111, liste1);
    inserer_apres_dernier(888, liste1);
    inserer_apres_dernier(-99, liste1);

    printf("\n \t initialement la liste contient \n");
    parcours(liste1->premier, afficher);

    liste2=TriFusion(liste1);
    printf("\n \t finalement la liste contient \n");
    parcours(liste2->premier, afficher);
}
```

47