Travaux Pratiques N°15

Les listes linéaires

1. Introduction

Une liste linéaire (LL) est la représentation informatique d'un ensemble fini, de taille variable et éventuellement nul, d'éléments de type T. Untel 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. Opérations sur la SD LL

On distingue : les opérations atomiques (ou élémentaires) et les opérations élaborés.

2.1 Les Opérations atomiques

On distingue :

Créer_liste : permettant de créer une liste linéaire vide.
Liste_vide : 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.

2.2 Les Opérations élaborés

On distingue :

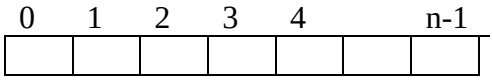
Inversion permettant d'inverser une liste linéaire
ll=(a, b, c, d) ll_{inv}=(d, c, b, a)

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)

3. Représentation physique

On distingue : représentation contiguë et représentation chaînée.

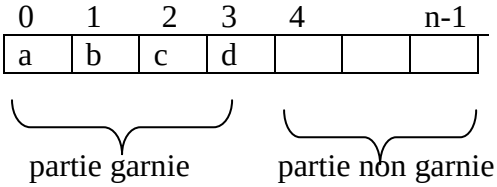
3.1 Représentation contiguë



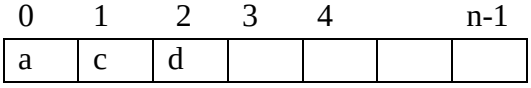
avec n est estimé a

Illustration :

ll=(a, b, c, d) représentation abstraite

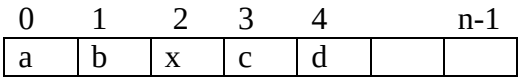


suppression :elle exige des décalages à gauche pour récupérer la position devenue disponible. Exemple : supprimer b



recherche ➔ recherche dans un tableau,on fait appel aux algorithmes connus soit recherche séquentielle soit dichotomique

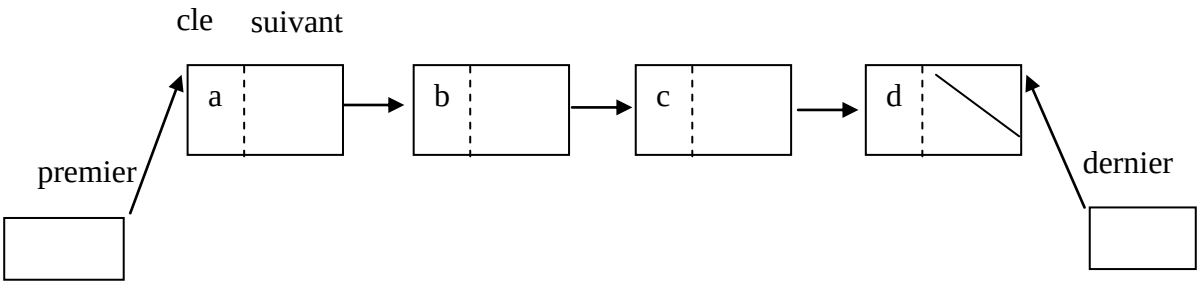
adjonction :elle exige des décalages à droite. Exemple : ajouter x après b.



visiter :balayer tous les éléments d'un tableau dans la partie garnie.

3.2 Représentation chaînée

ll=(a, b, c, d) **variante 1 :**



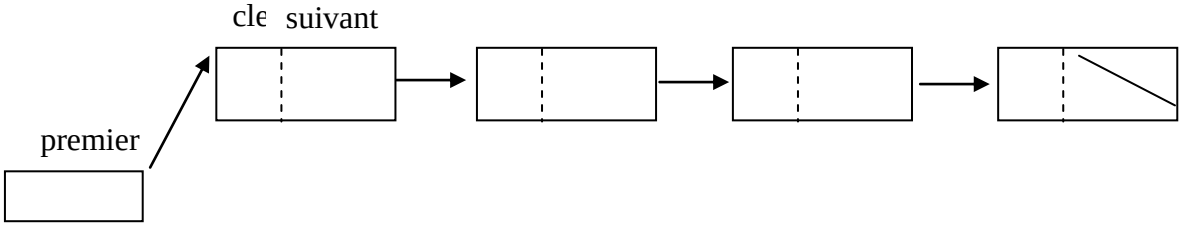
Représentation chainée ≠ liste linéaires

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.

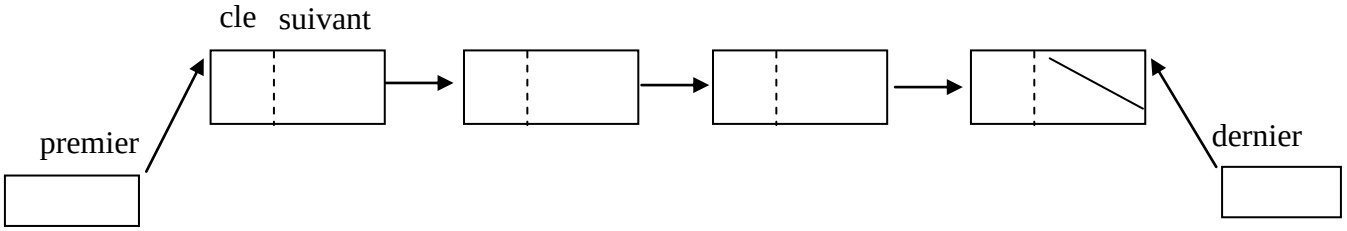
On peut dire que la représentation contiguë de la SDLL n'est pas recommandée à cause des décalages impliqués par les deux opérations fondamentales ajouter et supprimer notamment pour les listes linéaires de taille plus au moins importante.

4. Variantes de la SD liste linéaire On distingue :

4.1 Liste linéaire uni directionnelle avec un seul point d'entrée (premier)

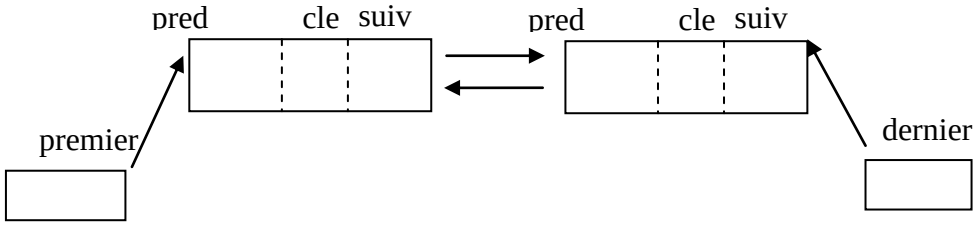


4.2 Liste linéaire unidirectionnelle avec deux points d'entrée (premier et dernier)



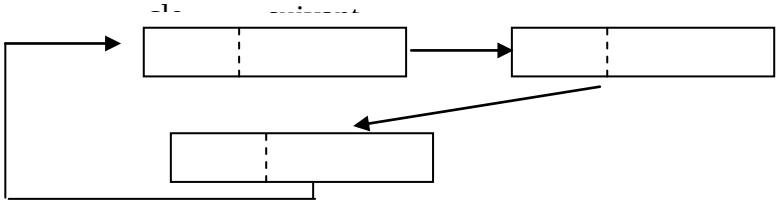
Pour les deux variantes (1) et (2), la liste linéaire est unidirectionnelle. À partir d'un élément donné on peut passer à son successeur. Ceci est possible grâce au champ de chaînage suivant : dans la variante (1), le premier élément est privilégié (accès direct) dans la variante (2) le premier et le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

4.3 Liste linéaire bidirectionnelle avec 2 points d'entrée



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

4.4 Liste linéaire circulaire ou anneau



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.

5 Matérialisation de liste chainée

5.1 Liste chaînée 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 nœud {
        int cle ;
        struct nœud * suivant ;
} ;
struct liste {
        struct nœud* premier ;
        struct nœud * dernier ;
} ;

/*création d'une liste linéaire*/
void creer_liste (struct liste * ll){
        ll->premier=NULL ;
        ll->dernier=NULL ;
}

/*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*/
/*processus d'adjonction ou d'insertion
insertion :insérer après un élément référencée
insérer avant un élément référencée
insérer avant premier
insérer après  dernier*/

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

```
/*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 préd*/
        struct nœud * q ;
        q=(struct nœud*) malloc(sizeof(struct nœud)) ;
        *q=*p;
/*en c l'affectationest définie sur les variables ayant un type à base de struct :
// q->cle=p->cle,q->suivant=p->suivant*/
/*mise à jour l'espace référencé par p*/
        p->cle=info ;
        p->suivant=q ;
}
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 ;
     }
}
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;
   }
}
/* opérations de suppression */
void supprimer_elem (struct noeud * p ) {
    struct noeud* q ;
   /* on suppose que p admet un successeur*/
   assert(p->suivant) ;
    q=p->suivant ;
    *p=*q ;
    free (q) ;
}
```

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

Recherche dans une liste linéaire

Un problème de recherche à deux issues : succès référence non nulle et échec référence nulle

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

Opération de parcours

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

À ne pas confondre !
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*.

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

Utilisation de la procédure parcours :
1- paramètres effectifs :

```
struct nœud * point_de_depart ;
void afficher(struct nœud * q) {
        printf (''%u d\n '', q->cle) ;
}
/* activation de parcours */
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.
2- Traitement incrémenter chaque élément visité :
Le traitement consiste à incrémenter tous les éléments visités.

```
struct nœud *point_de_depart ;
…
void incrementer (struct nœud *q)    {
        q->cle++ ;
}
/*activation */
parcours (point_de_depart, incrementer) ;
```

5.2. Liste linéaire bidirectionnelle avec 2 points d'entrée

```
struct nœud {
        struct nœud * pred  ;
        int cle ;
        struct nœud * succ ;
} ;
struct liste {
        struct nœud* premier ;
        struct nœud * dernier ;
} ;
```

6. Exercices d'application

Exercice 1 : Inversion d'une liste chaînée

Écrire une procédure qui permet d'inverser une liste chaînée Ls dans une liste Ls_Inv.

Exercice 2: Tri par bulles d'une liste chaînée

Écrire une procédure qui permet de trier une liste chaînée Ls dans l'ordre croissant des éléments (utiliser la méthode de tri par bulles).

Exercice 3 : Parcours d'une liste chaînée

Etudiant=struct Nom :chaine Note :réel finStruct	Cellule =Struct Cle :Etudiant Suiv :^Cellule finStruct	Liste=struct Premier :^Cellule Dernier :^Cellule finStruct

On suppose qu'on a en entrée une liste d'étudiants, dont un champ s'appelle note. Calculer la moyenne de la classe (on suppose que la liste est non vide).

Exercice 4 : Insertion dans une liste chaînée

On suppose qu'on a en entrée une liste triée par ordre croissant d'entiers. Ecrivez l'action qui ajoute un entier dans la bonne position de façon que la liste reste encore triée après l'insertion.

Exercice 5 : Tri par fusion d'une liste chaînée

Proposer un algorithme 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).

Proposer ensuite un algorithme 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.

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

Faire, en utilisant au choix les trois actions précédentes une action de tri fusion qui étant donné deux listes en entrée (pas forcément triées) retourne une liste triée.