Chapitre 7: 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 à la 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 :

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.

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)

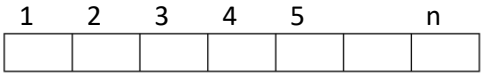
La concaténation de ll1 et ll2 dans l'ordre (ordre significatif) donne :

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

3. Représentation physique

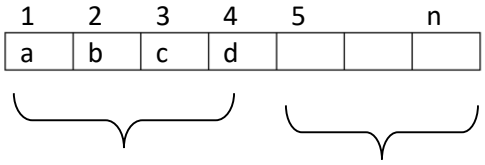
On distingue : représentation contiguë et représentation chaînée.

3.1 Représentation contiguë

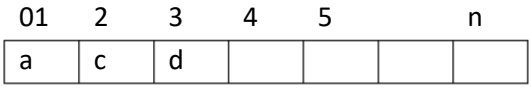


avec n est estimé a

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

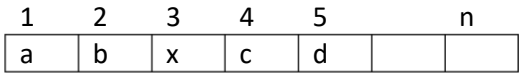


suppression : elle exige des décalages à gauche pour récupérer la position devenue disponible.
Exemple : supprimer b



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.

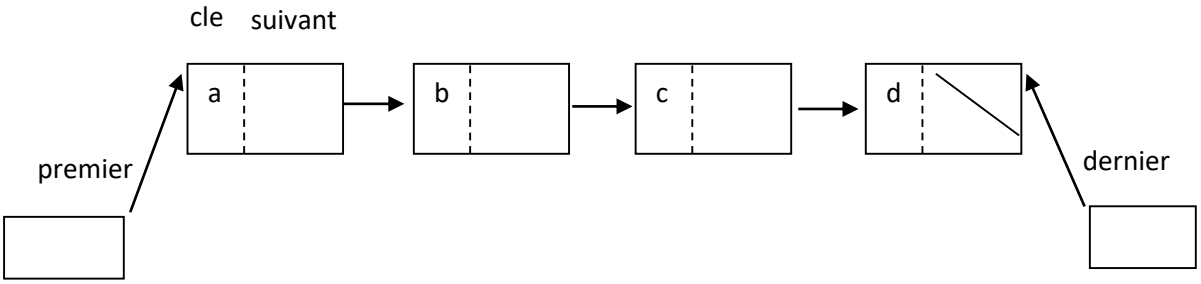


visiter : balayer tous les éléments d'un tableau dans la partie garnie.

3.2 Représentation chaînée

ll=(a, b, c, d)

variante 1 :



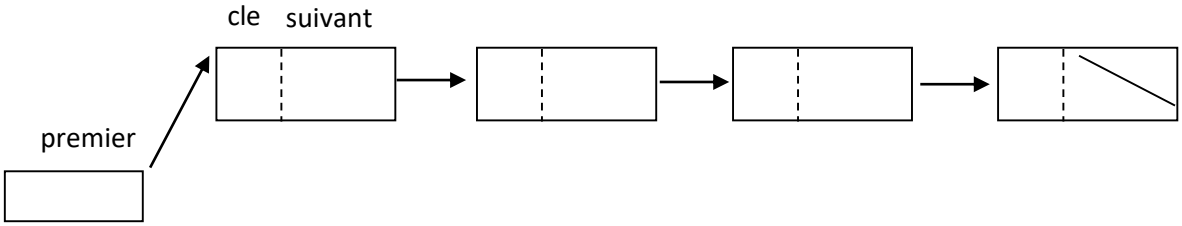
Représentation chainée ≠ liste linéaires

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.

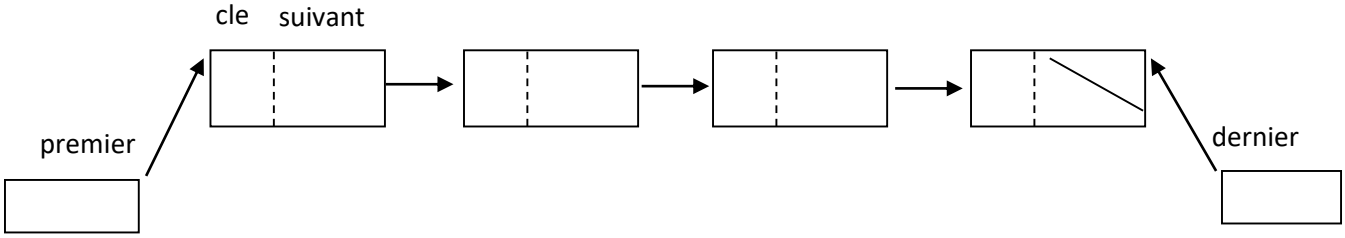
On peut dire que la représentation contiguë de la SDLL n'est pas recommandée à cause des décalages impliqués par les deux opérations fondamentales ajouter et supprimer notamment pour les listes linéaires de taille plus au moins importante.

4. Variantes de la SD liste linéaire On distingue :

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

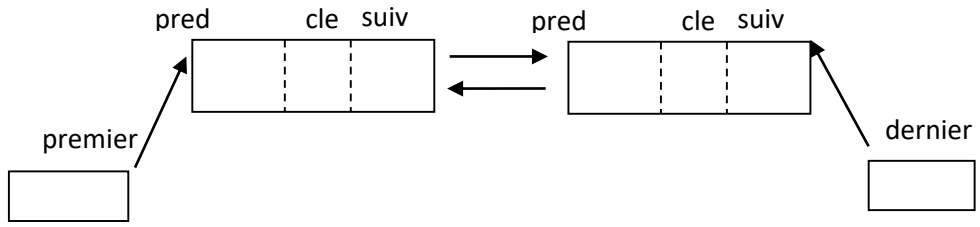


4.2 Liste linéaire unidirectionnelle avec deux points d'entrée (premier et dernier)



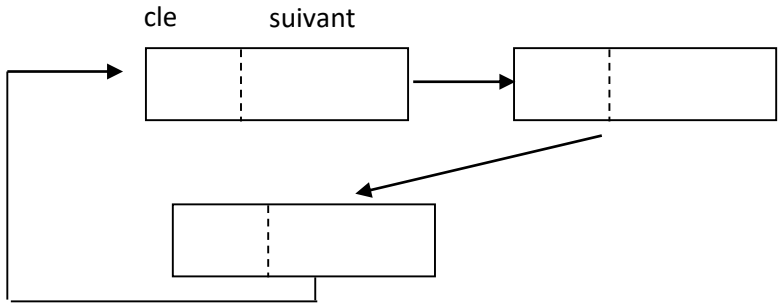
Pour les deux variantes (1) et (2), la liste linéaire est unidirectionnelle. À partir d'un élément donné on peut passer à son successeur. Ceci est possible grâce au champ de chaînage suivant : dans la variante (1), le premier élément est privilégié (accès direct) dans la variante (2) le premier et le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

```
Types
        Cellule = Struct
                cle : entier
                Suiv : ^Cellule
        FinStruct

        Liste = Struct
                premier: ^Cellule
                dernier : ^Cellule
        FinStruct
```

5.1.1. Création d'une liste chaînée

La procédure suivante permet de créer une liste chaînée de n éléments de type entier.

/*création d'une liste linéaire*/

Solution1 : sous forme d'une procédure

```
Procédure creer_liste (Var L : Liste)
Début
        L.premier←Nil
        L.dernier←Nil

Fin proc
```

Solution2 : sous forme d'une fonction

```
Function creer_liste (): Liste
Var
        Ll : Liste
début
        L1.premier←Nil
        L1.dernier←Nil
        creer_liste←L1
fin Fn
```

5.1.2. Tester la vacuité d'une liste linéaire

Solution (1)

```
Fonction liste_vide ( ll : Liste) : boolean
Debut
        Si (ll.premier=Nil)  alors
              liste_vide← vrai
        Sinon liste_vide← faux
        finsi
fin Fn
```

Solution (2)

```
Fonction liste_vide ( ll :Liste) : boolean
debut
        Si ((ll.premier=Nil) et (ll.dernier=Nil)  ) alors
                liste_vide← vrai
        Sinon liste_vide← faux
        finsi
fin Fn
```

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

5.1.3 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

/*insertion après élément référencée*/

```
procedure inserer_apres (info:entier, var p:^Cellule)
var
        q:^Cellule
debut
        Allouer(q)
        q^.cle←info
        q^.suiv←p^.suiv

        /*mise à jour du successeur de p*/
        p^.suiv←q
fin proc
```

/*insertion avant un élément référencé*/

```
procedure inserer_avant (info:entier, var p:^Cellule)
//le problème : la liste est unidirectionnelle à partir de p, on ne peut pas passer à son
//prédécesseur
var
        q:^Cellule
debut
        Allouer(q)
        q^←p^
        /*mise à jour l'espace référencé par p*/
        p^.cle←info
        p^.suiv←q
Fin proc
```

/*insertion avant le premier élément*/

```
procedure inserer_avant_premier (info:entier, var ll:Liste)
var
        q:^Cellule
debut

        si (liste_vide (ll)) alors
                Allouer(q)
                q^.cle←info
                q^.suiv←Nil
                ll.premier←q
                ll.dernier←q
        sinon
                Allouer(q)
                q^.cle←info
                q^.suiv←ll.premier
                ll.premier←q
        finsi
fin proc
```

/*insertion après le dernier*/

```
procedure inserer_apres_dernier(info:entier, var ll:Liste)
var
    q:^Cellule
debut
        si (liste_vide (ll)) alors
                Allouer(q)
                q^.cle←info
                q^.suiv←Nil

                ll.premier←q
                ll.dernier←q
        sinon
                Allouer(q)
                q^.cle←info
                q^.suiv←Nil
                (ll.dernier)^.suiv←q
                ll.dernier←q
       finsi
fin proc
```

5.1.4 Parcours d'une liste chaînée

La procédure itérative suivante permet de parcourir et afficher les éléments d'une liste chaînée.

```
Procédure AffichListe_itér (ll : Liste)
Var
        P : ^Cellule
Début
        p^ ←(ll.premier)^
        TantQue (P # Nil) Faire
                Ecrire(P^.cle)
                P^← (P^.Suiv)^
         FinTQ
Fin Proc
```

5.1.5 Recherche d'un élément dans une liste chaînée

```
Fonction recherche(x : Entier ; ll: Liste) : Booléen
Var
        P :^Cellule
        trouve : Booléen
Début
        Trouve←Faux
        P^ ←(ll.premier)^
        TantQue (P # Nil) ET (trouve = Faux) Faire
            Trouve←(P^.cle = x)
            P^←(P^.suiv)^
        FinTQ
        recherche ←trouve
Fin Fn
```

5.1.6 Processus de suppression

On distingue trois types de suppression:

1) supprimer un élément référencée, 2) supprimer le premier et 3) supprimer le dernier

/* suppression d'un élément référencée */

```
procedure supprimer_elem ( var p:^Cellule)
var
        q :^Cellule
debut
                /* on suppose que p admet un successeur*/
        Assure (p^.suiv != Nil) /* la méthode assert de la bibliothèque assert.h en C */
        q←p^.suiv
        p^←q^
        liberer (q)
fin proc
```

/* suppression du premier élément */

```
procedure supprimer_premier (var ll :Liste)
var q :^Cellule
debut
        q←ll.premier
        ll.premier←q.suiv
        liberer (q)
        si (ll.premier= Nil) alors
                ll.dernier←Nil
fin proc
```

/* suppression du dernier élément */

```
procedure supprimer_dernier (var ll :Liste)
var
        q :^Cellule
Debut
        Si (ll.premier = ll.dernier) alors suppressimer_premier (ll)
        sinon
        q←ll.premier
        TantQue (q^.suiv != ll.dernier)
                q←q^.suiv
        Fin TQ
        q^.suiv← Nil
        liberer (ll.dernier)
        ll.dernier←q
        finsi
fin proc
```

5.2. Liste linéaire bidirectionnelle avec 2 points d'entrée

```
Types
        Cellule = Struct
                Pred :^Cellule
                cle : entier
                Suiv : ^Cellule
        FinStruct

        Liste = Struct
                 premier: ^Cellule
                 dernier : ^Cellule
        FinStruct
```

/*création*/

```
Procédure creer_liste (Var ll : Liste)
Début
        ll.premier←Nil
        ll.dernier←Nil

Fin proc
```

/*insertion après élément référencée*/

```
procedure inserer_apres (info:entier, var p:^Cellule)
var
        q:^Cellule
debut
        Allouer(q)
        q^.cle←info
        q^.suiv←p^.suiv
        q^.pred←p

        (p^.suiv)^.pred←q
         p^.suiv←q
fin proc
```

/*insertion avant un élément référencé*/

```
procedure inserer_avant (info:entier, var p:^Cellule)
var
        q:^Cellule
debut
         Allouer(q)
         q^.cle←info
         q^.suiv←p
         q^.pred←p^.pred

        (p^.pred)^.suiv←q
         p^.pred←q
Fin proc
```

5.2.1 Parcours inverse d'une liste à chaînage double

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

```
Procédure AffichInvListe (ll : Liste )
Var
        P : ^Cellule
Début
        P ←ll.dernier
        TantQue (P # Nil) Faire
                Ecrire(P^.cle)
                P←P^.Pred
        FinTQ
Fin
```