Les Listes linéaires chainées (LLC)

1

2

1. Introduction

1.1. Notion d'allocations statique et dynamique

Représentation (allocation) statique

L'allocation de l'espace se fait tout à fait au début d'un traitement. En termes techniques, on dit que l'espace est connu à la compilation. C'est donc la notion de tableau.

Représentation dynamique

L'allocation de l'espace se fait au fur et à mesure de l'exécution du programme. Pour pouvoir faire ce type d'allocation, l'utilisateur doit disposer des deux opérations : allocation et libération de l'espace. Si le langage offre ces possibilités, on les utilise directement, sinon on sera dans l'obligation de les simuler, c'est à dire gérer l'espace soi-même dans un grand tableau (super tableau).

1.2. Variables

La Mémoire Centrale (MC) est formée par des cases numérotées. Chaque case peut stocker un octet (8 bits).

Une variable est une zone contigüe en MC (une case ou un ensemble de cases qui se suivent). Sa taille (en nombre de cases) dépend du type de la variable (ex: un entier occupe 4 cases, un réel occupe 8 cases, ...etc). L'adresse d'une variable est le numéro de sa première case.

Exemple:

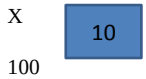
en C on aura:

int x;

Dans ce cas, *x* est le nom donné pour référencer l'emplacement mémoire associé à la variable (la case d'adresse 100)

Quand on affecte une valeur (par exemple 10) à *x*, la case numéro 100 contiendra cette même valeur:

en C : ***x = 10;***



On dit alors que le contenu de l'adresse 100 est 10.

1.3. Pointeurs

Un pointeur est une variable qui peut contenir des adresses de variables

Exemple:
En C, on peut déclarer un pointeur vers un entier comme suit:

int *p;

D'après cette déclaration, on peut affecter à la variable *p* l'adresse d'une variable de type entier.

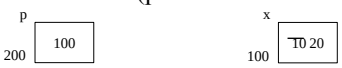
En C, l'expression ***&v*** retourne l'adresse de la variable *v*.

On dit alors que *p* pointe *x*.

Maintenant, on peut par exemple modifier la valeur de *x* indirectement (sans faire référence à *x*). Si on écrit en C :

****p = 20;***

La valeur de *x* sera alors modifiée (par une affectation indirecte)



Une telle utilisation des pointeurs (modifications par indirection) est par exemple utile en C pour écrire des fonctions qui peuvent modifier leurs paramètres (passage par adresse).

Il y a une autre utilisation des pointeurs, utile en C : l'*allocation dynamique* de variables.

1.4. Allocation de variables

Allocation de variables veut dire création de variables. Donc réservation d'espace mémoire en associant à chaque variable l'adresse d'une zone vide en mémoire.
Il existe 2 types d'allocation: ***statique*** (gérée automatiquement par le système) et ***dynamique*** (gérée manuellement par le programmeur).

Toutes les variables déclarées représentent des variables allouées statiquement :

Les variables du programme principal d'un programme C sont automatiquement créées au début de l'exécution et détruites à la fin de l'exécution.

Les variables d'une procédure ou fonction (ainsi que les paramètres d'appels) sont automatiquement créées au début de chaque appel et détruites à chaque retour de procédure ou fonction.

Il existe des fonctions prédéfinies en C pour créer de nouvelles variables durant l'exécution d'un programme et pour les détruire aussi, c'est l'allocation dynamique :

En C, la fonction ***malloc(nb_octets)*** alloue une zone mémoire de taille *nb_octets* et retourne son adresse comme résultat. La fonction ***free(p)*** détruit la variable pointée par *p*.

Exemple en C

```
int main()
{
        char  *p;          /* allocation statique d'une var ( p ) de type pointeur */
        p = malloc( sizeof(char) );   /* allocation dynamique d'une var de même
                                        taille qu'un caractère : sizeof(type) retourne le
                                        nb d'octets nécessaire pour représenter une var
                                        de ce type */
        *p = 'A';                     /* utilisation indirecte de la var dynamique */
        free(p);           /*  destruction de la var dynamique */
        return 0;
}
```

1.5. Remarques

- Les constantes pointeurs NULL ou 0 (en C) indiquent l'absence d'adresse. Donc, par exemple en C, l'affectation *p = NULL*, veut dire que *p* ne pointe aucune variable.

- Il ne faut jamais utiliser l'indirection (* en C) avec un pointeur ne contenant pas l'adresse d'une variable, il y aura alors une erreur de segmentation.

- De même il est interdit de référencer une variable dynamique après l'avoir détruite.

- Les variables dynamiques peuvent être de n'importe quel type, simple ou complexe.

- Une variable dynamique n'a pas de nom, on ne peut la manipuler qu'à travers un pointeur qui contiendrait son adresse.

1.6. Exemple d'introduction

Supposons que l'on veuille résoudre le problème suivant :

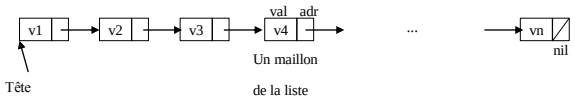
Trouver tous les nombres premiers de 1 à n et les stocker en mémoire

Le problème réside dans le choix de la *structure d'accueil*. Si on utilise un tableau, il n'est pas possible de définir la taille de ce vecteur avec précision même si nous connaissons la valeur de n (par exemple 10000). Ne connaissant pas la valeur de n, on n'a aucune idée sur le choix de sa taille. On est donc, ici, en face d'un problème où la réservation de l'espace doit être ***dynamique***.

Les Listes linéaires chainées (LLC)

2. Définition d'une liste linéaire chaînée

Liste Linéaire Chaînée LLC est une structure de données (le plus souvent dynamique) pour représenter un ensemble de valeurs. Ces valeurs sont chaînées entre elles formant une suite :



Chaque valeur *v* de l'ensemble est stockée dans un maillon (généralement une variable dynamique)
Un élément d'une LLC (maillon) est toujours une structure (objet composé) avec deux champs :

un champ **Valeur** : contenant l'information
un champ **Adresse** : donnant l'adresse du prochain maillon

A chaque maillon est associée *une adresse*. On introduit ainsi une nouvelle classe d'objet: le type POINTEUR en langage algorithmique.

11

Les Listes linéaires chainées (LLC)

Une LLC est caractérisée par :

- l'adresse de son premier élément (la tête de liste). Elle doit toujours être sauvegardée dans une variable pour pouvoir manipuler la liste.

- NIL constitue l'adresse qui ne pointe aucun maillon (indiquant par convention la fin de la liste).

- Si la liste est vide (ne contient aucun maillon), la tête doit alors être positionnée à NIL.

12

Les Listes linéaires chainées (LLC)

Dans le langage algorithmique, on définira le type d'un maillon comme suit :

```
type Maillon = Structure
        val : typeqq
        suiv : Pointeur(Maillon)
fin
```

En C:

```
struct Maillon
        {
                int     val ;
                struct  Maillon  *suiv ;
        };
```

13

Les Listes linéaires chainées (LLC)

3. Modèle sur les listes linéaires chaînées

On définit un ensemble d'opérations que l'on va utiliser pour écrire des algorithmes sur les listes. Cet ensemble d'opérations s'appelle le modèle des listes linéaires chaînées ($\mathcal{M}_{LLC}$) :

Allouer(T, P)	procédure qui alloue (dynamiquement) un nouveau maillon et affecte son adresse dans le pointeur p. Les champs val et adr du nouveau maillon sont indéterminés.
Libérer(P)	procédure qui détruit le maillon pointé par p.
Valeur(P)	consultation du champ valeur du maillon d'adresse p C'est une fonction qui retourne le contenu du champs val du maillon pointé par p
Suivant(P)	consultation du champ adresse du maillon d'adresse p C'est une fonction qui retourne le contenu du champs adr du maillon pointé par p.
Aff_Adr(P, Q)	procédure qui affecte q dans le champs adr du maillon pointé par p.
Aff_Val(P,v)	procédure qui affecte v dans le champs val du maillon pointé par p.

14

Les Listes linéaires chainées (LLC)

Voici l'équivalence en C des fonctions et des procédures du modèle :

```
/*-------------------------------------------------------------------------------------------------*/
/*    Procédures d'implémentation du modèle sur les listes linéaires chaînées      */
/*-------------------------------------- ----------------------------------------------------------*/

struct Maillon *Allouer ( )
        { return   ((struct Maillon *) malloc( sizeof(struct Maillon)));        }

void Aff_Val(struct Maillon *P, int v)
        { P->val =v; }

void Aff_Adr( struct Maillon *P,  struct Maillon *Q)
        { P->Suiv = Q; }

struct Maillon *Suivant(  struct Maillon *P)
        { return( P->Suiv ); }

int Valeur( struct Maillon *P)
        { return( P->Val) ; }
```

15

Les Listes linéaires chainées (LLC)

Solution de l'exemple de l'introduction :

Trouver tous les nombres premiers de 1 à n et les stocker en mémoire

On utilise les deux faits suivants :

- un nombre n'est premier que s'il n'est pas divisible par les nombres premiers qui le précèdent.
- tous les nombres premiers sont de la forme 6m+1 ou 6m-1

16

Les Listes linéaires chainées (LLC)

```
{ Création de maillons pour les nombres premiers 2 et 3 }
Allouer(T,P)
Aff_Val(P,2)
Tête := p
Allouer(T, Q)
Aff_Val(Q,3)
Aff_Adr(Q, NIL)
Aff_Adr(P,Q)
P := Q
M := 1
Continue := VRAI
Aig := VRAI
TANTQUE Continue FAIRE
        Gen_nombre (M, Aig, Nombre)
        Aig := NON Aig
        SI Aig ALORS
                M := M + 1
        FSI
        SI Nombre <= N ALORS
                SI Premier (Tête, Nombre) ALORS
                        Allouer (T, Q)
                        Aff_Val(Q, Nombre)
                        Aff_Adr( Q, NIL)
                        Aff_Adr(P, Q)
                        P := Q
                FSI
                SINON
                        Continue := FAUX
        FSI
FINTANTQUE
```

17

Les Listes linéaires chainées (LLC)

Les modules précédents sont définis comme suit :

```
Procédure Gen_nombre(M, Aig, Var : Nombre )
SI Aig ALORS
    Nombre := 6M – 1
SINON
    Nombre := 6M + 1
FSI
 ******************************************
Fonction Premier (L, N) : Booléen
P:= L
Trouv := FAUX
TANTQUE P <> NIL ET NON Trouv FAIRE
    SI Divisible ( Nombre, Valeur(P) ) ALORS
            Trouv := VRAI
    SINON
            P := Suivant(P)
    FSI
FINTANTQUE
Premier := NON Trouv
```

18

Enfin, le module Divisible (A, B) peut s'écrire comme suit: C'est un prédicat qui est égal à VRAI si B est un diviseur de A , FAUX sinon. Q étant une variable de type ENTIER,

```
Fonction Divisible (A, B) : Booléen
Q := A div B
SI Q.B = A ALORS
    Divisible := VRAI
SINON
    Divisible := FAUX
FSI
```


4. Algorithmes sur les listes

Parcours :

• **accès par valeur** : il s'agit de rechercher (séquentiellement à partir de la tête) une valeur v dans la liste.

• **accès par position** : il s'agit de rechercher (séquentiellement à partir de la tête) le maillon (son adresse) qui se trouve à la position donnée. La position est le numéro d'ordre du maillon dans la liste (entier).

Mises à jour :

• **construction** d'une liste à partir de n valeurs données.

• **insertion** d'une valeur (v) à une position donnée (i) : allouer un nouveau maillon contenant v et l'insérer dans la liste de telle sorte qu'il se retrouve à la $i^{ème}$ position.

• insertion d'une valeur (v) dans une liste ordonnée : allouer un nouveau maillon contenant v et l'insérer dans la liste de telle sorte que la liste reste ordonnée après l'insertion.

• **suppression** du maillon se trouvant à une position donnée (i) : rechercher par position le maillon i et le libérer. Le maillon précédent (s'il existe) doit être mis à jour pour pointer le suivant de i.

• suppression d'une valeur v dans la liste : rechercher par valeur et supprimer le maillon trouvé en mettant à jour le précédent (s'il existe). Si la liste contient plusieurs occurrences de la même valeur, on pourra les supprimer en faisant un seul parcours de la liste.
• destruction de tous les maillons d'une liste.

Tri sur les LLC :
Les mêmes algorithmes de tri utilisable pour les tableaux, en prenant en compte que l'accès par position coûte beaucoup plus cher que dans un tableau.

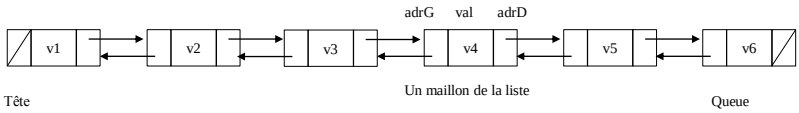
Algorithmes sur plusieurs LLC :
• **fusion** (ou interclassement) de 2 listes ordonnées : à partir de 2 listes ordonnées, construire une liste ordonnée contenant tous leurs éléments.
On peut aussi changer les champs 'adr' des maillons existant dans les 2 listes pour former qu'une seule liste ordonnée (sans allouer de nouveaux maillons).
• **éclatement** d'une liste en 2 listes distinctes : à partir d'une liste L et d'un critère (un prédicat) donné, on construit 2 listes, l'une contenant toutes les valeurs de L vérifiant le critère, l'autre, celles qui ne le vérifient pas.

5. Listes particulières

5.1. Listes bidirectionnelles

Ce sont des listes que l'on peut parcourir dans les deux sens : de gauche à droite et de droite à gauche.

Pour cela on rajoute dans chaque maillon un deuxième pointeur indiquant l'adresse du maillon précédent.



La structure d'un maillon est alors :

```
type Maillon = Structure
        val : typeqq
        adrG : Pointeur(Maillon)
        adrD: Pointeur(Maillon)
fin
```

Les Listes linéaires chainées (LLC)

Souvent on sauvegarde les adresses du 1er et dernier maillon (tête et queue) pour simplifier les parcours dans les deux sens.

Le modèle des listes bidirectionnelles est presque le même que celui des LLC sauf que Aff_Adr(p,q) est remplacée par Aff_AdrG(p,q) et Aff_AdrD(p,q). De plus la fonction de type ptr 'Précédent(p)' est rajoutée pour retourner le contenu du champ 'adrG' alors que la fonction 'Suivant(p)' retourne le champ 'adrD'.

Autrement :

$\mathcal{M}_{LLCb} = \mathcal{M}_{LLC}$ - ***{Aff_Adr} + { Aff_AdrG, Aff_AdrD, Précédent }***

Le modèle des LLC est donc étendu par les opérations suivantes :

Précédent(P)	accès au champ adresse gauche du maillon d'adresse P.
Aff_AdrG(P, Q)	dans le champ Adresse1 (adresse gauche) du maillon d'adresse P, on range l'adresse Q.
Aff_AdrD(P, Q)	dans le champ Adresse2 (adresse droite) du maillon d'adresse P, on range l'adresse Q

23

Les Listes linéaires chainées (LLC)

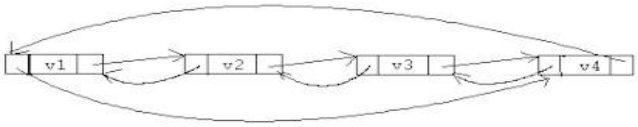
5.2. Listes circulaires

C'est une liste où le dernier maillon pointe le premier, formant ainsi un cercle.
La tête de la liste est l'adresse de n'importe quel maillon de la liste circulaire.

Le même modèle des LLC est utilisé pour écrire des algorithmes sur ce type de listes: $\mathcal{M}_{LLCc} = \mathcal{M}_{LLC}$.

5.3. Listes bidirectionnelles circulaires

C'est une LLC à double sens et dont le dernier (premier) pointe sur le premier (dernier).



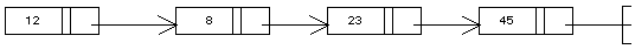
$\mathcal{M}_{LLCbc} = \mathcal{M}_{LLCb}$

24

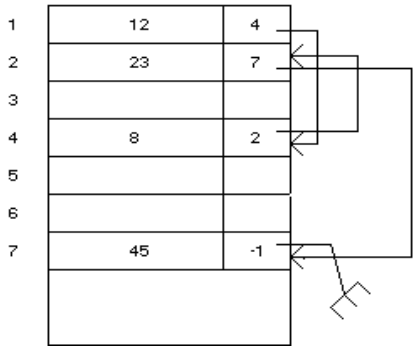
Les Listes linéaires chainées (LLC)

6. Implémentation des listes linéaires chaînées avec la représentation contigüe

On peut représenter les LLC dans un même tableau où chaque élément renferme au moins 2 champs : l'information et le lien. Par exemple la liste suivante



peut être représentée dans un tableau comme suit :



25

Les Listes linéaires chainées (LLC)

Définition de la structure :

```
type element = Structure
        info : typeqq
        Lien : Entier
        Vide : Booleen    /*pour les positions vides du tableau */
fin

var     T : tableau ( 1 : Max ) de element
```

Initialement les champs Vide de tous les éléments sont à vrai (pour indiquer des positions disponibles).

Une LLC est définie par l'indice de son premier élément dans le tableau T.

L'implémentation du modèle de LLC au moyen de tableau revient à traduire les opérations du modèle au moyen de la seule opération d'indexation définie sur les tableaux.

26

Les Listes linéaires chainées (LLC)

La fonction Allouer(P) rend l'indice d'une position vide sinon la valeur -1 (s'il n'y a pas de place). Ceci revient à chercher une entrée libre P dans le tableau T. Une solution simple consiste à balayer le tableau linéairement du début vers la fin jusqu'à la rencontre d'une entrée libre.

```
Fonction Allouer(P :entier) :entier
        I := 1 ;
        Trouv := FAUX
        TANTQUE I <= Max ET NON Trouv FAIRE
                SI T(I).Vide ALORS
                        Trouv := VRAI
                SINON
                        I := I + 1
                FSI
        FINTANTQUE
        SI NON Trouv ALORS
                Allouer := - 1
        SINON
                Allouer := i
                T(I).Vide := FAUX
        FSI
```

27

Les Listes linéaires chainées (LLC)

Cette implémentation est simple mais pas très efficace. A chaque fois qu'on alloue un maillon, la fonction 'Allouer(P)' réalise un parcours séquentiel. Cela peut être très pénalisant quand la taille du tableau est grande.

Il existe une autre implémentation, plus efficace en évitant de faire un parcours séquentiel lors de l'allocation d'un maillon. Toutes les opérations peuvent être implémentées sans utiliser de boucle, car on maintient une liste particulière de cases libres dans le tableau T.

Initialement toutes les cases sont chaînées entre elles dans cette liste de cases libres.

A chaque allocation d'un maillon, on retire de cette liste la première case (la tête de liste).

A chaque libération d'un maillon, on rajoute la case libérée au début de la liste des cases libres (insertion au début d'une liste).

Les autres opérations restent inchangées.
Le champ 'Vide' devient inutile.

28

Les Listes linéaires chainées (LLC)

Les autres opérations en bref :

Libérer(P)	T(P).Vide := VRAI
Valeur(i)	Valeur := T(i).info
Suivant(I)	Suivant := T(I).Lien
Aff_Adr(I,J)	T(I).Lien := J
Aff_Val(i, val)	T(i).info := val

29

Les Listes linéaires chainées (LLC)

7. Conclusion

Faut-il utiliser un tableau ou une liste chaînée ?

Cette question se pose au programmeur dans de nombreuses applications. Lorsque le choix est possible, on le fait en considérant les points suivants.

30

Les Listes linéaires chainées (LLC)

Avantages des tableaux par rapport aux listes :

• ***Accès direct*** : C'est la définition même des tableaux : l'accès au $i^{ème}$ élément se fait en un temps indépendant de *i*, alors que dans une liste chaînée ce temps est de la forme $k \times i$ (car il faut exécuter *i* fois l'opération $p = suivant(p)$).

• ***Pas de sur-encombrement*** *:* La relation *successeur* étant implicite (définie par la contiguïté des composantes), il n'y a pas besoin d'espace supplémentaire pour son codage. Alors que dans une liste chaînée l'encombrement de chaque maillon est augmenté de la taille du pointeur *suivant*.

Avantages des listes par rapport aux tableaux :

• ***Liberté dans la définition de la relation successeur*** *:* Cette relation étant explicite, on peut la modifier aisément (les maillons des listes chaînées peuvent être réarrangés sans avoir à déplacer les informations qu'ils portent).
Autre aspect de la même propriété, un même maillon peut faire partie de plusieurs listes.

• ***Encombrement total selon le besoin***, si on utilise l'allocation dynamique des maillons. Le nombre de maillons d'une liste correspond au nombre d'éléments effectivement présents, alors que l'encombrement d'un tableau est fixé d'avance et constant.

31

Les Listes linéaires chainées (LLC)

NOTE :

Une conséquence de l'encombrement constant d'un tableau est le risque de saturation: l'insertion d'un élément échoue parce que toutes les cases du tableau sont déjà occupées. Bien sûr, ce risque existe aussi dans le cas des listes chaînées (il se manifeste par l'échec de la fonction *malloc*), mais il correspond à la saturation de la mémoire de l'ordinateur; c'est un événement grave, mais rare. Alors que le débordement des tableaux est beaucoup plus fréquent, car il ne traduit qu'une erreur d'appréciation de la part du programmeur.

32