Notes de cours – ALGO2 – INI/2008

Listes Linéaires Chaînées (LLC)

Plan

1. Introduction
2. Allocation Statique et Dynamique
3. Définition d'une LLC
4. Quelques opérations sur les LLC
5. Listes particulières
6. Implémentation en contigu

1) Introduction

L'espace mémoire occupé par une structure de données dynamique est variable. C'est intéressant pour représenter des ensembles à tailles variables.

On peut donc agrandir ou rétrécir la taille de l'ensemble durant l'exécution du programme.

Certains problèmes nécessitent la gestion d'un ensemble dynamique.

Ex: trouver tous les nombres premiers <= n (avec n un paramètre donné) et les stocker en mémoire. Le problème ici est la taille de la structure utilisée pour sauvegarder les nombres premiers. Un tableau ne conviendrait pas car on a aucune idée sur la taille à réserver au départ.

2) Notion d'allocation mémoire

- Variables

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

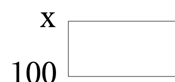
Une variable est une zone contigüe en MC (une case ou un ensemble de cases qui se suivent). Sa taille (en nombre de cases) dépend du type de la variable (ex: un entier occupe 4 cases, un réel occupe 8 cases, ...etc). L'adresse d'une variable est le numéro de sa première case.

Ex:
en Pascal on peut déclarer une variable entière comme suit:

```
var
   x : integer;
```

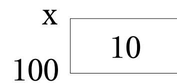
en C on aura:

```
  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 ex 10) à x. la case numéro 100 contiendra cette même valeur:
en Pascal : x := 10; ou alors en C : x = 10;
On dit alors que le contenu de l'adresse 100 est 10



- Pointeurs

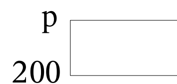
Un pointeur est une variable qui peut contenir des adresses de variables

Ex:
en Pascal on peut déclarer un pointeur vers un entier comme suit:

```
var
    p  :  ^integer;
```

ou bien en C :

```
    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. En Pascal standard, il n'existe pas un moyen de récupérer l'adresse d'une variable, à part l'utilisation de l'allocation dynamique (voir plus loin). Cependant certains compilateur Pascal (non standards) offrent un tel mécanisme (Addr(v) en turbo-pascal retourne l'adresse de la var v)

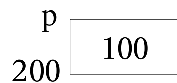
donc si on écrit par exemple, en turbo-pascal :

```
    p := Addr(x);
```

ou bien en C :

```
    p = &x;
```

on aura dans p l'adresse de x :



on dit alors que p pointe x.

Maintenant, on peut par exemple modifier la valeur de x indirectement (sans faire référence à x).

En Pascal l'expression p^ fait référence à la variable pointée par p
il en est de même en C, avec l'expression *p

si on écrit en Pascal :

```
    p^ := 20;
```

ou bien 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 Pascal et en C : l'allocation dynamique de variables.

- 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 var du programme principales d'un programme pascal ou les variables globales d'un programme C sont automatiquement créées au début de l'exécution et détruites à la fin de l'exécution.
- Les var 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 existent des fonctions prédéfinies en Pascal ou en C pour créer de nouvelles variables durant l'exécution d'un programme et pour les détruire aussi, c'est l'allocation dynamique :

En Pascal, la procédure new(p) alloue une nouvelle variable et affecte son adresse à la variable p. Le type de la nouvelle variable est celui spécifié dans la déclaration de p. La procédure dispose(p) détruit la variable pointée par p.

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 Pascal:

```
var
   p : ^char;          { allocation statique d'une var ( p ) de type pointeur }

begin
      new(p);          { allocation dynamique d'une var caractère }
      p^ := 'A';       { utilisation indirecte de la var dynamique }
      dispose(p);      { destruction de la var dynamique }
end.
```

Le même 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;
}
```

- Remarques

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

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

ex :

```
var
   p : ^integer;          { allocation (statique) de p }

begin
{ le contenu de p est pour le moment indéterminé : une adresse quelconque de la mémoire }
        p^ := 10;
{ erreur à l'exécution; affectation d'un entier (10) dans une zone indéterminée }
{ p ne pointe aucune variable de type entier }
end.
```

***** De même il est interdit de référencer une variable dynamique après l'avoir détruite :

```
var
   p : ^integer;          { allocation (statique) de p }

begin
        new( p );
{ le contenu de p est l'adresse d'une variable dynamique nouvellement créée }
        p^ := 10;
{ l'affectation est correcte, la zone concernée contient une variable de type entier }
        dispose( p );
{ destruction de la var dynamique, maintenant p pointe une zone quelconque }
        p^ := 20;
{ erreur à l'exécution; affectation d'un entier (20) dans une zone indéterminée }
end.
```

***** Les variables dynamiques peuvent être de n'importe quel type, simple ou complexe. Voici un exemple en Pascal montrant l'allocation dynamique de tableaux :

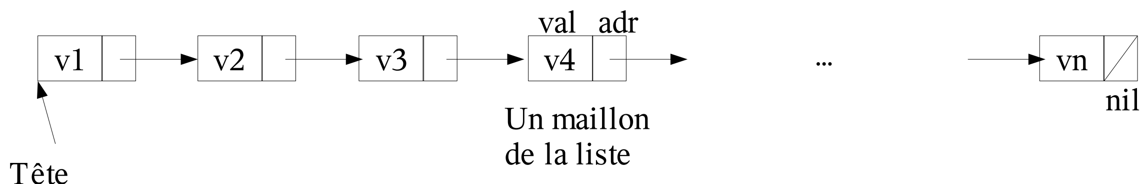
```
type
        T_tab = array[1..100] of integer;      { def d'un type tableau de 100 entiers }
                              { il n'y a aucune allocation ici, T_tab n'est pas une variable }
```

```
var
        p : ^T_tab;      { allocation statique de la var p (un pointeur) }
        i  : integer;    { allocation statique de la var i (un entier) }
begin
        { pour le moment il n'y a aucun tableau alloué; on peut en allouer avec new }
        new( p );
        { un tableau de 100 entiers vient d'être alloué dynamiquement, son adr est dans p }
        { on manipule le tableau à l'aide de l'indirection p^ }
        p^[1] := 10;
        p^[2] := 3;
        p^[3] := p^[1] * 2 + 5;
        { on peut détruire le tableau quand on en a plus besoin }
        dispose( p );
        { maintenant le tableau n'existe plus, on a plus le droit de manipuler p^ }
        { mais on peut créer un autre tableau on utilisant le même pointeur }
        new( p );
        for i:=1 to 100 do
                p^[i] := 5432;
        ...
end.
```

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

3) Définition d'une Liste Linéaire Chaînée (LLC)

Une Liste Linéaire Chaînée 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 var dynamique)
Un maillon est une structure à 2 champs :

< val : de type quelconque ,   adr : pointeur vers le prochain maillon >

Dans le dernier maillon de la liste, le champs adr contient le pointeur NIL (indiquant par convention la fin de la liste).

L'adresse du 1er maillon (la tête de la liste) est importante. Elle doit toujours être sauvegardée dans une variable pour pouvoir manipuler la liste.

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

- Le modèle des LLC

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 (M_{LLC}) :

* Allouer(p) : procédure qui alloue (dynamiquement) un nouveau maillon est 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.

* Aff-val(p,v) - procédure qui affecte v dans le champs val du maillon pointé par p.

* Aff-adr(p,q) - procédure qui affecte q dans le champs adr du maillon pointé par p.

* Valeur(p) : typeqlq - fonction qui retourne le contenu du champs val du maillon pointé par p.

* Suivant(p) : pointeur - fonction qui retourne le contenu du champs adr du maillon pointé par p.

Exemple d'utilisation : voici une procédure qui insère une valeur 'v' dans une liste ordonnée 'L'

```
procedure ins_ord( v : typeqlq;  var L : ptr )
/* insère v dans L, le résultat (éventuellement la nouvelle tête) est dans L */
var
        p, q, n : ptr;
        trouv : bool;
debut
        /* on commence par rechercher l'endroit où doit être insérée v pour que la liste
           reste ordonnée (parcours séquentiel jusqu'à trouver une valeur >= v) */
        trouv := FAUX;          /* booléen indiquant la fin du parcours */
        p := L;                 /* pointeur pour parcourir les maillons de la liste L */
        q := NIL;               /* pointeur pour garder le précédent de p */
        TQ  (p <> NIL) ET (Non trouv)
                SI  ( v <= Valeur(p) )
                        trouv := VRAI
                SINON
                        q := p;                 /* on sauvegarde dans q l'adr du maillon, */
                        p := Suivant(p);        /* avant de passer au prochain avec p. */
                FSI
        FTQ;

        /* quand on sort de cette boucle TQ, on pourra insérer v entre les maillons pointés
           par q et p, avec les cas particuliers suivants :
                        si on sort avec q=NIL, alors v sera insérée au début de la liste
                        si on sort avec  p=NIL, alors v sera insérée à la fin de la liste */

        Allouer( n );           /* n pointe maintenant un nouveau maillon */
        Aff-val( n, v );        /* on y affecte la valeur v */
        Aff-adr( n, p );        /* le suivant de n est maintenant p (même si c'est NIL) */
        SI (q <> NIL)           /* s'il existe un maillon qui précède p :
                Aff-adr( q, n )         /* le suivant de q devient alors n */
        SINON                   /* sinon (q=NIL) :
                L := n                  /* n devient la nouvelle tête de la liste */
        FSI
fin
```

4) Quelques opérations sur les LLC

- 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)

- 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^{eme} 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.

- trier une liste

Les même algo de tri utilisable pour les tableaux, en prenant en compte que l'accès par position coûte beaucoup plus cher que dans un tableau.

- fusion (ou interclassement) de 2 listes ordonnées

A 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

A partir d'une liste L et d'un critère (un prédicat) donnés, on construit 2 listes, l'une contenant toutes les valeurs de L vérifiant le critère, l'autre, celles qui ne le vérifient pas.

- construction d'une liste à partir de n valeurs données

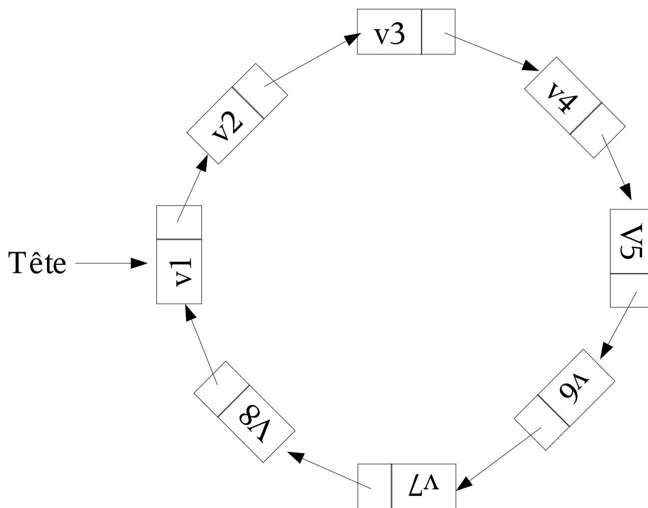
- destruction de tous les maillons d'une liste

5) Listes particulières

- 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.

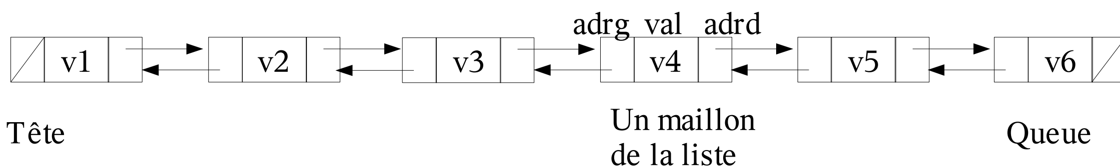
Le même modèle des LLC est utilisé pour écrire des algorithmes sur ce type de listes.



- 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 : <val : typeqlq, adrg, adrd : ptr >

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

Le modèle des listes bidiectionnelles 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 'Precedent(p)' est rajoutée pour retourner le contenu du champs 'adrg' alors que la fonction 'Suivant(p)' retourne le champs 'adrd'.

6) Implémentation des listes en contigu

Il s'agit d'implémenter le modèle des LLC sans utiliser d'allocation dynamique.

C'est donc l'utilisation d'un tableau (un espace contigu) pour simuler la mémoire réservée à l'allocation dynamique. Chaque élément du tableau peut être utilisé pour représenter un maillon alloué. Les indices du tableau représentent alors les adresses de maillons. Une valeur spéciale (par exemple 0 ou -1) sera utilisée pour indiquer la constante NIL.

Voici un implémentation simple du modèle des LLC avec un tel tableau :

	Val	Vide	adr
N		V	
		V	
		V	
5	v2	F	2
4		V	
3	v1	F	5
2	v3	F	0
1		V	

```
type
    ptr = entier;
   Tmaillon = struct
         val : Tqlq;
         Vide :  Booleen;        // booléen utilisé pour différencier entre  case libre et allouée
         adr : entier
   fin;

var
   T : tableau[N] de Tmaillon ;         // l'espace contigu sous forme de tableau. N est une constante

Init( )                          // procédure d'initialisation : toutes les cases sont au départ libres
  POUR i:=1,N
         T[i].vide := VRAI
  FP

Allouer( var p:ptr )             // On recherche la première case libre et on l'alloue.
  p := 1;  trouv := Non T[p].vide;
  TQ p<N et Non trouv                    // parcours séquentiel !!!
         p := p+1;
         trouv := Non T[p].vide
  FTQ
  SI Non trouv    p := 0   SINON    T[p].vide := FAUX  FSI
```

```
Liberer( p:ptr )              // Repositionne une case  à l'état libre
   T[p].vide := VRAI

Aff-val( p:ptr; v:Tqlq )      // MAJ du champs val d'un maillon occupé
   T[p].val := v

Aff-adr( p:ptr; q:ptr )       // MAJ du champs adr d'un maillon occupé
   T[p].adr := q

Valeur( p:ptr ) : Tqlq        // Consultation du champs val d'un maillon occupé
   Valeur := T[p].val

Suivant( p:ptr ) : ptr        // Consultation du champs adr d'un maillon occupé
   Suivant := T[p].adr
```

Cette implémentation est simple mais pas très efficace. A chaque fois qu'on alloue un maillon, la procédure 'Allouer(...)' 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 champs 'Vide' devient inutile.