Chapitre 2 : Représentation des Données en Mémoire

Dans ce chapitre, nous introduisons les structures contiguës et chainées utilisées de façon très intensive en programmation. Leur but est de gérer un ensemble fini d'éléments. Cette gestion des ensembles doit, pour être efficace, répondre au mieux à deux critères parfois contradictoires : un minimum de place mémoire utilisée et un minimum d'instructions élémentaires pour réaliser une opération.

II.1 Définition

Une *structure de données* (SDD) est une organisation des données pour en faciliter l'accès et la modification vis-à-vis d'un problème donné. Elle est définie par le type de base des composants ainsi que les opérations que l'on peut leur appliquer.

Autre définition : une *structure de données* est une structure logique destinée à contenir des données, afin de leur donner une organisation permettant de simplifier leur traitement.

Soit un ensemble E = {e_1, …, e_n}. Il s'agit d'organiser E en une structure de données afin de pouvoir appliquer les opérations de manipulation (accès/modification) de façon efficace.

SDD ≡ $\{E, op\}$ (E : les données ; op : les opérations)

- Les opérations sur un composant e_i ou sur l'ensemble E sont liées au type de base.
- L'accès à la SDD :
  - Comment rechercher un composant ?
  - Comment ajouter, supprimer et modifier un composant dans la structure ?

Quelques structures de données :

- Les structures séquentielles : les **listes**, les **piles** et les **files**.
- Les structures arborescentes : les **arbres** (binaires ou généraux).
- Les structures relationnelles : les **graphes**.
- Les structures à accès par clé : les **tables**.

II.2 Choix de la Structure de Données

La manière de représenter les données en mémoire doit respecter deux contraintes :

1) utiliser un minimum d'espace mémoire;
2) exécuter un nombre minimal d'instructions pour réaliser une opération.

Le choix d'un algorithme peut déterminer le type de la structure de données, et parfois l'adaptation d'une structure de données particulière peut déterminer quel algorithme utiliser.
Avant de décider quelle structure de données choisir, il est nécessaire de comprendre la manière dont les données vont être utilisées. Il est utile d'analyser le problème considéré à partir de plusieurs angles. Ainsi, nous aurions besoin de savoir :

1) la longueur des champs (par exemple, le champ nom)
2) la plage des valeurs possibles des données numériques
3) la quantité des données à traiter
4) les différentes opérations à effectuer sur les données.
5) etc …

La réponse à ce genre de questions aide à mieux comprendre et à mieux choisir les structures de données supportant la résolution du problème en question.

Le choix des bonnes structures de données et des bons algorithmes peut faire la différence entre un programme s'exécutant en quelques secondes ou en quelques jours.

II.3 Représentation Contiguë

Un **tableau** (*array* en anglais) est une structure de données qui consiste en un ensemble d'éléments ordonnés accessibles par leur indice (ou index). C'est une structure de données de base que l'on retrouve dans chaque langage de programmation.
Tous les éléments d'un tableau doivent être du même type[1]. L'accès à un élément par son indice est **direct**, cela s'explique par le fait que les éléments d'un tableau sont contigus dans l'espace mémoire. Ainsi, il est possible de calculer l'adresse mémoire de l'élément auquel on veut accéder, à partir de l'adresse de base du tableau et de l'indice de l'élément. L'accès est direct, comme il le serait pour une variable simple.

[1]Dans certains langages (comme Python, APL, etc.), cette restriction n'existe plus.

Exemple : Soit E={3, 21, 6, 13, 68, 4, 29} un ensemble d'éléments de type entier qui peut être représenté par un tableau T schématisé comme suit :

	a_1	a_2	a_3	a_4	a_5	a_6	a_7
T :	*3*	*21*	*6*	*13*	*68*	*4*	*29*
	1	*2*	*3*	*4*	*5*	*6*	*7*

tel que : 1, 2, …,7 représentent les indices

a_1 *représente l'adresse de T[1]*
…
a_n *représente l'adresse de T[n]*

En général, a_i *représente l'adresse de T[i], en calculant l'adresse en mémoire du* $i^{ème}$ *élément, on accède directement à l'élément T[i] ainsi :*

$a_i = a_{i-1}$ + <taille du type> ou encore $a_i = a_1 + (i-1)$*<taille du type>

Remarque : <taille du type > en C est sizeof (int) si les éléments sont de type entier.

Les limites de la structure tableau sont :

- Vous devez connaître au moment de l'écriture de l'algorithme le nombre de données que devra recevoir le tableau, c'est-à-dire son nombre de cases.
- Un tableau étant représenté en mémoire sous la forme de cellules contiguës, les opérations d'insertion et de suppression d'éléments nécessitent de faire des décalages.
- L'insertion peut être impossible si le tableau est plein.

Cela fait donc beaucoup d'opérations et oblige certains langages fournissant de telles possibilités à implémenter leurs données, non pas sous la forme traditionnelle (cellules adjacentes), mais en utilisant une représentation chainée (liste chainée).

II.4 Représentation Chaînée

Une liste chaînée est une structure de donnée qui permet de stocker une suite d'éléments de même type en mémoire et de taille arbitraire. Chaque élément est stocké dans une cellule représentée par un enregistrement qui contient, en plus de l'élément, l'adresse de la cellule suivante (successeur), ce qui permet de chainer les cellules entre elles.

Avec les structures de données chaînées, il n'y a nul besoin de savoir combien d'éléments, va contenir la structure. Agrandir la liste revient à créer un nouvel élément en mémoire et à le lier à la liste chaînée par une référence (adresse) vers son emplacement mémoire.
De plus, et c'est là sûrement son plus gros avantage, vous pouvez insérer ou supprimer une valeur n'importe où dans la liste, sans algorithme de décalage et d'une manière très aisée. Cependant l'accès à une valeur ne sera pas possible directement comme avec les tableaux, il faudra pour cela un algorithme de parcours. L'accès aux éléments d'une liste se fait de manière séquentielle.

Par convention, l'accès à une liste chaînée ou à une cellule est simplement un ***pointeur*** vers le premier élément de cette liste ou vers n'importe quelle cellule de la liste.

II.5. Les Variables Statiques, Dynamiques et Les Pointeurs

Toute variable manipulée dans un programme est stockée quelque part en mémoire centrale. Cette mémoire est constituée d'octets qui sont identifiés de manière univoque par un numéro qu'on appelle *adresse*.
Pour retrouver une variable, il suffit donc de connaître l'adresse de l'octet où elle est stockée (ou, s'il s'agit d'une variable, qui recouvre plusieurs octets contigus, l'adresse du premier de ces octets).
Pour des raisons évidentes de lisibilité, on désigne souvent les variables par des identificateurs, et non par leur adresse. C'est le compilateur qui fait alors le lien entre l'identificateur d'une variable et son adresse en mémoire.

a. Les Variables Statiques :

Ce qui a été vu?

- variables de type simple (entier, réels, caractère, etc.)
- variables de type structuré tableau, enregistrement

Chaque variable dans un programme a besoin d'un certain nombre d'octets en mémoire. Jusqu'ici, la réservation de la mémoire s'est déroulée automatiquement par l'emploi des déclarations des variables. Dans tous ces cas, le nombre d'octets à réserver était déjà connu pendant la compilation. Nous parlons alors d'***allocation statique*** des variables (l'espace mémoire est réservé automatiquement). Donc, l'espace associé à une variable statique est alloué à la compilation et son existence demeure le long de l'exécution du programme.

Exemples :
Variable c : caractère;
Variable tab : tableau [20] entier;
Variable Etudiant : enregistement
nom, prénom : chaine;
datenaiss : enregistrement
jour, mois, année : entier;
finenreg;
finenreg;

b. Les Variables Dynamiques :

Souvent, nous devons travailler avec des données dont nous ne pouvons pas prévoir le nombre et la grandeur lors de la programmation. Ce serait alors un gaspillage de réserver toujours l'espace maximal prévisible.
Il nous faut donc un moyen de gérer manuellement la mémoire lors de l'exécution du programme, c'est-à-dire pour réserver des espaces en mémoire, désigner ces espaces et les rendre lorsqu'ils ne sont plus utilisés. On parle alors d'allocation dynamique (contrôlée). Donc une variable dynamique est allouée pendant l'exécution et au moment où il y en besoin. On accède à une variable dynamique grâce à un pointeur.

c. Les Variables Pointeur :

Une adresse est une valeur que l'on peut stocker dans une variable. Les pointeurs sont justement des variables qui contiennent l'adresse d'autres variables d'un type donné. Il faut noter qu'un pointeur est une variable statique spéciale

Si un pointeur P contient l'adresse d'une variable A, on dit que ***'P pointe sur A'***.

Les opérations de base :

La manipulation des pointeurs nécessite deux opérateurs :

- d'un opérateur *'adresse de'* : @ pour obtenir l'adresse d'une variable.
- d'un opérateur *'contenu de'* : ^ pour accéder au contenu d'une adresse.

Nous avons besoin aussi de la fonction TailleDe (sizeof en C) qui retourne la taille d'un bloc mémoire.

Déclaration :

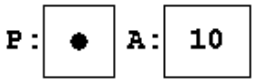
<Nom du Pointeur> : ^<TypeObjetPointé>;

Exemple : *p : ^entier ;* « *p* est une variable de type pointeur vers un entier
*// en C on déclare int *p ;*

p est une ***variable statique*** de type pointeur vers un objet de type *entier*.

Représentation Schématique

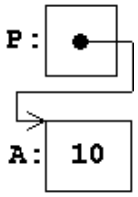
Soit **P** un pointeur sur un entier non initialisé et **A** une variable (de type entier) contenant la valeur 10 :



L'opérateur 'adresse de' @

<nomPointeur>← @ (variable statique)

L'instruction **P←@A;** affecte l'adresse de la variable A à la variable P. Dans la représentation schématique, nous pouvons illustrer le fait que 'P pointe sur A' par une flèche:



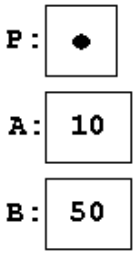
// En langage C, l'opérateur 'adresse de' utilisé est le symbole &. Alors on écrit : P=&A

L'opérateur 'contenu de' ^

^<NomPointeur> : désigne le contenu de l'adresse référencée (pointée) par le pointeur <NomPointeur>

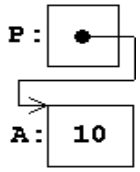
//En langage C l'opérateur 'contenu de' utilisé est le symbole * alors on écrit : *p=10

Exemple : Soit A une variable contenant la valeur 10, B une variable contenant la valeur 50 et P un pointeur:

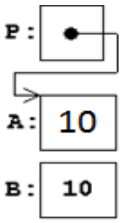


Après les instructions :

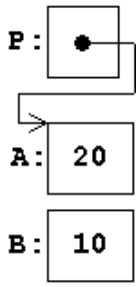
P ← @A; P pointe sur A



B ← ^P; le contenu de A (référencé par ^P) est affecté à B



^P ← 20; le contenu de A (référencé par ^P) est mis à 20.



Allocation Dynamique de la Mémoire:

*a. Allocation d'une variable dynamique de **type simple** :*

<nom-pointeur>←Allouer (TailleDe(<TypeObjet>))

« Réservation d'une zone mémoire correspondant à une variable de type <typeObjet> et affecte au pointeur l'adresse de cette zone.»

Exemple:

```
p : ^entier ;
p ← Allouer (TailleDe(entier)) ;
// en langage C on écrit p =(int*) malloc(sizeof(int)) ;
```

*b. Allocation d'une variable dynamique de **type structuré :***

- *Enregistrement*

<nom-pointeur>← Allouer (TailleDe(<nom_enregistrement>))

Exemple:

```
Type date = enregistrement
        jour, mois, année : entier;
    finenreg;
```

d :^date ;
d←Allouer(TailleDe(date)) // taille d'un type structuré

L'accès à un champ de l'enregistrement :
^nom-pointeur.nom-champ
Exemple : si (^d.mois=2) alors …

- Tableaux

*<nom-pointeur>← Allouer (<nombre d'éléments>*TailleDe(<élément>)*

Exemples :

- *Tableaux à une dimension*

```
t :^entier ; n : entier ;
lire(n) ; // nombre d'éléments du tableau
t←Allouer(n*TailleDe(entier)) ;
```

- *Tableaux à deux dimensions*

```
A :^^entier ; n, m :entier ;
lire(n) ; / / nombre de lignes du tableau
lire(m) ; / / nombre d'éléments par ligne (colonnes)
A←Allouer(n*TailleDe(^entier)) ;
Pour i←1 à n faire
        A[i]←Allouer(m*TailleDe(entier)) ;
    fait ;
```

Libérer la Variable Dynamique (pointée)

Libérer (<nom du pointeur>)

« Rendre disponible l'espace mémoire occupé par la variable dynamique créée par allouer »
Exemple:

```
Variable p : ^entier ;
p←Allouer (TailleDe(entier));
Libérer (p) ;      //en langage C : free(p);
```

Opérations Courantes sur les Pointeurs

- Affectation : on peut affecter un pointeur à un autre.
  Exemple : p1, p2 :^entier ; A : entier ;

```
p1←@A ;
p2←p1 ;
```

- Addition et soustraction :
  Si P pointe sur l'élément A[i] d'un tableau, alors
  P+n pointe A[i+n]
  P-n pointe sur A[i-n]
  Exemple : *A : tableau[10] entier ;*
  p←A ; //ou bien p=&A[1]
  p←p+9 //p pointe sur A[10]
  p←p-1 //p pointe sur A[9]

Constante Nil

Le type pointeur admet une seule constante prédéfinie nil (NULL en C).

« p←nil ; signifie que p ne pointe vers rien »

Exemple : p: ^entier ; p←nil ;

Exemple récapitulatif :

Soient les instructions suivantes :

i : entier ; p :^entier ;
i←3; ecrire("valeur de p avant allocation=", p) ;
p←Allouer(TailleDe(entier)) ; ecrire("valeur de p après allocation=", p) ;
^p← i ; ecrire("valeur de ^p", ^p) ;

objet	*adresse*	*valeur*	
i	1245060	3	
p	1245064	0	Avant allocation
i	1245060	3	
p	1245064	8004260	
^p	8004260	? (int)	Après allocation
i	1245060	3	
p	1245064	8004260	
^p	8004260	3	

i←3 ;
p← @i ;

objet	***adresse***	***valeur***	
i	1245060	3	
p	1245064	1245060	i et ^p sont identiques (même adresse)
^p	1245060	3	

II.6 Listes Chainées :

Une liste chainée est une structure dynamique qui s'agrandit au fur et à mesure de l'ajout des données. Une liste est constituée de ***cellules chaînées***.

1. Une cellule est composée de deux champs :
   a) élément, contenant un élément de la liste
   b) suivant, contenant un pointeur sur une autre cellule.
2. les cellules ne sont pas rangées séquentiellement en mémoire. D'une exécution à une autre leur localisation peut changer.
3. une position (adresse) est un pointeur sur une cellule.
4. rajouter ou supprimer un élément ne nécessite pas de décalage.
5. l'accès à une liste se fait par un pointeur sur la cellule qui contient le premier élément de la liste que l'on appelle point d'entrée (ou *tête de liste*).
6. la liste vide est le pointeur ***nil***.

II.6.1 Listes Simplement Chaînées

a) Déclaration

```
Type Liste=enregistrement
                info : <TypeElement> ;
                suivant :^Liste ;
            finenreg ;
variable L :^Liste ;
```

ou bien

```
Type Liste=^Cellule ;
Type Cellule=enregistrement
                    info : < TypeElement > ;
                    suivant : Liste ;
                finenreg ;
variable L : Liste ;
```

b) Accès à une valeur d'une liste

```
L : liste ; a :<typeElement> ;
a←(^L). info ;     // en C :  a = (*L).info ;
```

c) Création d'un nœud

```
Fonction creer_noeud ( ) : Liste
Variable  L : Liste ;
 Début
     L←Allouer(TailleDe(Cellule)) ;
     Retourner(L) ;
 Fin ;
```

d) Ajout d'un élément en tête de liste

```
Procédure ajout_tete (E/S tete : Liste ; E/ X :<typeElement>)
Variable nouv : Liste ;
Début
   nouv←creer_noeud ( ) ;
   si (nouv≠nil) alors
                    (^nouv).info←X ;
                    (^nouv).suivant←tete ;
                     tete←nouv ;
  finsi ;
Fin ;
```

e) Ajout d'un élément après une adresse donnée

```
Procédure ajout_après (E/ prd : Liste ; E/ X : <typeElement>)
Variable nouv : Liste ;
Début
   nouv←creer_noeud ( ) ;
   si (nouv≠nil) alors
                     (^nouv).info←X ;
                     (^nouv).suivant←(^prd).suivant ;
                     (^prd).suivant←nouv ;
   finsi ;
 Fin ;
```

f) Suppression d'un élément en tête de liste

```
Procédure supprim_tete (E/S tete : Liste)
Variable temp : Liste ;
Début
    temp←tete ;
    tete←(^tete).suivant ;
    libérer (temp) ;
Fin ;
```

g) Suppression de l'élément après une adresse donnée

```
Procédure supprim (E/ prd : Liste)
Variable p : Liste ;
Début
   p←(^prd).suivant ;
   (^prd).suivant←(^p).suivant ;
   libérer(p) ;
Fin ;
```

h) Parcours d'une liste

Exemple : Afficher les éléments d'une liste d'entiers, dont le point d'entrée est **tete**.

```
Procédure Affiche_liste (E/ tete : Liste)
Début
Tantque  (tete ≠ nil) faire
                              ecrire ((^tete).info);
                              tete←(^tete).suivant;
                         fait ;
Fin ;
```

En langage C on peut écrire une boucle pour comme suit :

```
void Affiche_liste (Liste tete)
{   for ( ; tete!=NULL; tete=tete→suivant)
        printf("%d", tete→info);
}
```

i) Recherche d'une valeur

Recherche d'une valeur donnée, dans une liste **L** d'entiers ***quelconques*** et retourne son adresse, si elle existe sinon retourne **nil**

```
Fonction recherche (E/ L : Liste ; E/ val : typeElem) : Liste
Variable B: booleen ;
Début
   B←faux ;
   Tantque ((L ≠ nil) et (B=Faux)) faire
        Si ((^L).info≠Val) alors L←(^L).suivant;
                              Sinon B←Vrai ; Fsi ;
                                   fait ;
   retourner L;
Fin;
```

Recherche d'une valeur donnée, dans une liste **L** d'entiers ***triés*** par ordre croissant et retourne l'adresse de l'élément précédent pour une éventuelle insertion (si la valeur n'existe pas) ou suppression (si la valeur existe).

```
Fonction RechercheTrie (E/ L : Liste ; E/ val : typeElem) : Liste
Variable prd : Liste ; B: booleen ;
Début
  prd←L; B←faux ;
  Tantque ((L ≠nil) et (B=Faux))
    faire
       Si ((^L).info<Val) alors debut
                                        prd←L;
                                        L←(^L).suivant;
                                    Fin ;
                                Sinon B←Vrai; Fsi;
    fait ;
  retourner prd;
Fin;
```

j) Création d'une liste

La création d'une liste revient à insérer les éléments de la liste un par un. Les insertions se font soit à la tête (LIFO : Last In First Out) soit à la queue (FIFO : First In First Out).

Exemple : Ecrire une Procédure qui permet de lire n nombres et de les ranger dans une liste chaînée.

- **Création LIFO :**

```
Procédure CreerListe_LIFO (E/S Tete : Liste ; E/ n : entier )
Variable nouv : Liste ; nb : entier ;
Début
   pour  i←1 à n
    faire
        lire (nb) ;
        nouv←creer_noeud ( ) ;
        si (nouv≠nil) alors
                          (^nouv).info←nb;
                          (^nouv).suivant←Tete ;
                          Tete←nouv ;
        finsi ;
    fait
 Fin ;
```

- **Création FIFO :**

```
Procédure CreerListe_FIFO (E/S Tete : Liste ; E/ n : entier )
Variable nouv, Queue : Liste ; nb : entier ;
Début
   pour  i←1 à n
    faire
        lire (nb) ;
        nouv←creer_noeud ( ) ;
        si (nouv≠nil) alors
                              (^nouv).info←nb;
                              si (i=1) alors Tete←nouv ;
                                       sinon (^Queue).suivant←nouv ;
                              fsi
                              Queue ←nouv ;
        fsi ;
     fait
 (^Queue).suivant←nil ;
Fin
```

II.6.2 Listes Bidirectionnelles :

Dans une liste bidirectionnelle, chaque cellule contient deux pointeurs : l'un sur l'élément suivant (comme pour les listes simplement chainées) et l'autre sur l'élément précédent dans la liste.

a) Déclaration

```
Type ListeBid=^Cellule ;
Type Cellule=enregistrement
                    info : <typeElement> ;
                    suivant : ListeBid ;
                    precedant : ListeBid ;
                finenreg ;
variable L : ListeBid ;
```

b) Création d'un nœud

```
Fonction creer_noeud ( ) : ListeBid
Variable  L : ListeBid ;
Début
    L←Allouer(TailleDe(Cellule)) ;
    Retourner(L) ;
Fin ;
```

c) Ajouter un élément en tete de liste

```
Procédure ajout_tete (E/S Tete : ListeBid ; E/ X :<typeElement>)
Variable nouv : ListeBid ;
Début
nouv←creer_noeud ( ) ;
 si (nouv≠nil) alors
                    (^nouv).info←X ;
                    (^nouv).suivant←tete ;
                   (^nouv).precedant←nil ;
                    (^tete).precedant←nouv ;
                    tete←nouv ;
  finsi ;
Fin ;
```

d) Ajouter un élément après une adresse donnée

```
Procédure ajout_après (E/ prd : ListeBid ; E/ X : <typeElement>)
Variable nouv : ListeBid ;
Début
   nouv←creer_noeud ( ) ;
   si (nouv≠nil) alors
                    (^nouv).info←X ;
                    (^nouv).suivant←(^prd).suivant ;
                    (^nouv).precedant←prd ;
                    (^prd).suivant←nouv ;
                    ^((^prd).suivant).precedant←nouv ;
   finsi ;
 Fin ;
```

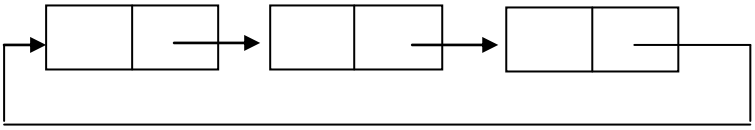
e) Mise à jour d'une liste

A titre d'exemple, écrire les procédures qui permettent d'insérer une valeur Val donnée dans une liste bidirectionnelle L d'entiers triées dans l'ordre croissant et de supprimer une valeur **Val** donnée dans une liste bidirectionnelle **L** d'entiers quelconques.

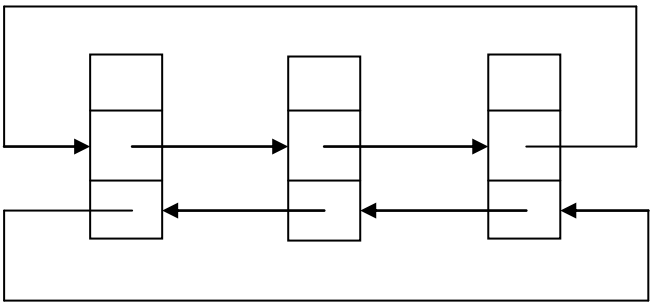
II.6.3 Listes Circulaires

Les listes circulaires sont des listes chaînées avec la particularité que le dernier élément pointe le premier élément de la liste. Les listes circulaires peuvent être simplement chaînées ou doublement chaînées.

a) Listes circulaires simplement chaînées (unidirectionnelles) :



b) Listes circulaires doublement chaînées (bidirectionnelles) :



```
Procédure CreerListeCirculaire (E/S Tete : liste ; E/ n : entier )
Variable nouv, Queue : Liste ; nb : entier ;
Début
   pour  i←1 à n
    faire
        lire (nb) ;
        nouv←creer_noeud ( ) ;
        si (nouv≠nil) alors
                              (^nouv).info←nb;
                              si (i=1) alors Tete←nouv ;
                                       sinon (^Queue).suivant←nouv ;
                              fsi
                              Queue ←nouv ;
        fsi ;
     fait
 (^Queue).suivant←Tete;
Fin
```

Exemple : Vérifier si un mot donné représenté dans une liste chaînée bidirectionnelle circulaire de caractères est un palindrome.