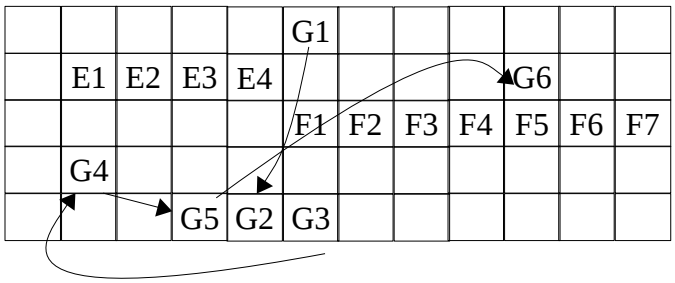
Chapitre 2 Les Structures Simples

1) Organisation globale des blocs

Dans un premier temps, on étudiera deux possibilités distinctes d'organiser les blocs au sein d'un fichier:
- soit le fichier est « vu comme un tableau » : tous les blocs qui le forment sont contigus
- soit le fichier est « vu comme une liste » : les blocs ne sont pas forcément contigus, mais sont chaînés entre eux.

Dans la figure ci-dessous, on a deux fichiers vus comme tableau (E et F) et un fichier vu comme une liste (G). Les blocs F1, F2, … F7 sont contigus, de même pour les blocs E1, E2, E3 et E4. Par contre les blocs G1, G2, … G6 ne sont pas contigus, ils sont chaînés entre eux formant une liste de blocs.



Parmi les caractéristiques nécessaires pour manipuler un fichier vu comme tableau, on pourra avoir par exemple :
- Le numéro du premier bloc,
- Le numéro du dernier bloc (ou alors le nombre de blocs utilisés).

Pour un fichier vu comme liste, il suffirait par contre de connaître le numéro du premier bloc (la tête de la liste), car dans chaque bloc, il y a le numéro du prochain bloc (comme le champ suivant dans une liste). Dans le dernier bloc, le numéro du prochain bloc pourra être mis à une valeur spéciale (par exemple -1) pour indiquer la fin de la liste.

2) Organisation interne des blocs

Les blocs sont censés contenir les enregistrements d'un fichier. Ces derniers peuvent être de longueur fixe ou variable.

Si on est intéressé par des enregistrements de longueur fixe, chaque bloc pourra alors contenir un tableau d'enregistrements de même type.

Exemple:

```
Type Tenreg = structure                    // structure d'un enregistrement du fichier
            matricule : chaine(10);
            nom : chaine(20);
            age : entier;
            ...
      fin;

Type Tbloc = structure                     // structure d'un bloc du fichier
            tab : tableau[1..b] de Tenreg; // un tableau pouvant contenir (au max) b enreg.
            NB : entier;                   // nombre d'enreg insérés dans tab (entre 0 et b).
      fin;
```

Si on opte pour des enregistrements de tailles variables, chaque enregistrement sera vu comme étant une chaîne de cararactère (de longueur variable).

Les enregistrements seront de longueur variable, car par exemple, il y a un ou plusieurs champs ayant des tailles variables, ou alors le nombre de champs varie d'un enregistrement à un autre.

Pour séparer les champs entre eux (à l'intérieur de l'enregistrement), on pourra soit utiliser un caractère spécial ('#') ne pouvant pas être utilisé comme valeur légale, ou alors préfixer le début des champs par leur taille (sur un nombre fixe de positions). Dans l'exemple ci-dessous, on utilise 3 positions pour indiquer la taille des champs.

```
...|0|2|7|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|0|1|7|b|b|b|b|b|b|b|b|b|b|b|b|b|b|0|3|7|c|c|c|.......|c|c|.....
  <-----------------------------------------------> <-------------------------------> <----------....------>
        un champ sur 27 caractères                    un champ sur 17 caractères          …..
```

Dans le cas d'enregistrements à taille variable, le bloc ne peut pas être défini comme étant un tableau d'enregistrements, car les éléments d'un tableau doivent toujours être de même taille.

La solution est de considérer le bloc comme étant (ou contenant) un grand tableau de caractères de taille fixe, renfermant les différents enregistrements (stockés caractère par caractère).

Pour séparer les enregistrements entre eux, on utilise les mêmes techniques que celles utilisées dans la séparation entre les champs d'un même enregistrement (soit avec un caractère spécial '$', soit on préfixe chaque enregistrement par sa taille).

Voici un exemple de déclaration d'un type de bloc pouvant être utilisé dans la définition d'un fichier vu comme liste avec format (taille) variable des enregistrements.

```
Type Tbloc = structure
            tab : tableau[1..b'] de caractères;     // tableau de caractères pour les enreg.
            suiv : entier;                          // numéro du bloc suivant dans la liste
      fin;
```

Remarque: Même si les enregistrements sont de longueurs variables, la taille des blocs reste toujours fixe.

Pour minimiser l'espace perdu dans les blocs (dans le cas : format variable uniquement), on peut opter pour une organisation avec chevauchement entre deux ou plusieurs blocs:

Quand on veut insérer un nouvel enregistrement dans un bloc non encore plein et où l'espace vide restant n'est pas suffisant pour contenir entièrement cet enregistrement, celui-ci sera découpé en 2 parties de telle sorte à occuper tout l'espace vide du bloc en question par la 1ere partie, alors que le reste (la 2e partie) sera insérée dans un nouveau bloc alloué au fichier. On dit alors que l'enregistrement se trouve à cheval entre 2 blocs.

3) Taxonomie des structures simples de fichiers

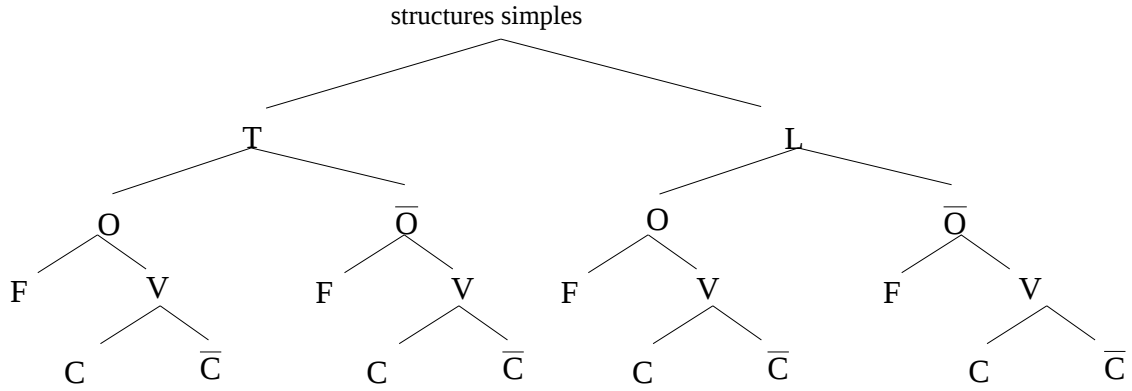
En combinant entre l'organisation globale des fichiers (tableau ou liste) et celle interne aux blocs (format fixe ou variable des enregistrements), on peut définir une classe de méthodes d'accès (dites « simples ») pour organiser des données sur disque.

Si de plus on prend en compte la possibilité de garder le fichier ordonné ou non, suivant les valeurs d'un champ clé particulier, on doublera le nombre de méthodes dans cette classe de structures simples de fichiers.

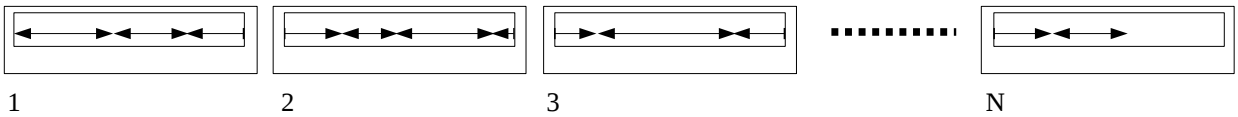
Utilisons la notation suivante:

T : pour fichier vu comme tableau, L : pour fichier vu comme liste
O : pour fichier ordonné, $\overline{O}$: pour fichier non ordonné
F : pour format fixe des enregistrements, V : pour format variable
C : avec chevauchement des enregistrements entre blocs, $\overline{C}$: sans chevauchement

Les feuilles de l'arbre suivant, représentent les 12 méthodes d'accès simples:



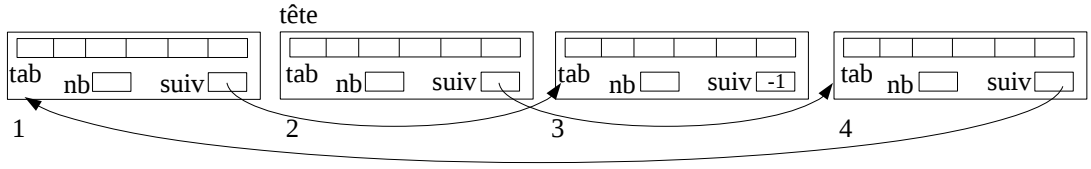
Par exemple la méthode T $\overline{O}$ VC représente l'organisation d'un fichier vu comme tableau (T), non ordonné ($\overline{O}$), avec des enregistrements de taille variables (V) et acceptant les chevauchements entre blocs (C) :



La recherche est séquentielle, l'insertion en fin de fichier et la suppression est logique.

Dans le cas d'un fichier LOF (fichier vu comme liste, ordonné avec enregistrements à taille fixe), chaque bloc pourra contenir par exemple, un tableau d'enregistrements (tab), un entier indiquant le

nombre d'enregistrements dans le tableau (nb) et un entier pour garder la trace du bloc suivant dans la liste (suiv) :



La recherche est séquentielle, l'insertion provoque des décalages intra-blocs (pour garder l'ordre des enregistrements) et la suppression peut être logique ou physique.

4) Exemple complet: fichier de type « TOF »
(fichier vu comme tableau, ordonné avec enregistrements à taille fixe)

- La recherche d'un enregistrement est dichotomique (rapide).
- L'insertion peut provoquer des décalages intra et inter-blocs (coûteuse).
- La suppression peut être réalisée par des décalages inverses (suppression physique coûteuse) ou alors juste par un indicateur booléen (suppression logique beaucoup plus rapide). Optons pour cette dernière alternative.
- L'opération du chargement initial consiste à construire un fichier ordonné avec n enregistrements initiaux, en laissant un peut de vide dans chaque bloc. Ce qui permettra de minimiser les décalages pouvant être provoqués par les futures insertions.
- Avec le temps, le facteur de chargement du fichier (nombre d'insertions / nombre de places disponibles dans le fichier) augmente à cause des insertions futures, de plus les suppressions logiques ne libèrent pas de places. Donc les performances tendent à se dégrader avec le temps. Il est alors conseillé de réorganiser le fichier en procédant à un nouveau chargement initial. C'est l'opération de réorganisation périodique.

Déclaration du fichier:

```
const
    b = 30;                 // capacité maximale des blocs (en nombre d'enregistrements)

type

    Tenreg = structure      // Structure d'un enregistrement :
        effacé : booleen;   // booléen pour la suppression logique
        cle : typeqlq;      // le champs utilisé comme clé de recherche
        champ2 : typeqlq;   // les autres champs de l'enregistrement,
        champ3 : typeqlq;   // sans importance ici.
        ...
    Fin;

    Tbloc = structure       // Structure d'un bloc :
        tab : tableau[1..b] de Tenreg;      // un tableau d'enreg d'une capacité b
        NB : entier;                        // nombre d'enreg dans tab ( <= b)
    Fin;
```

var // variables globales (F et buf)
F : Fichier de Tbloc Buffer buf Entete (entier, entier);
/* Desription de l'entête du fichier F :
L'entête contient deux caractéristiques de type entier.
- la première sert à garder la trace du nombre de bloc utilisés (ou alors le numéro logique du dernier bloc du fichier)
- la deuxième servira comme un compteur d'insertions pour pouvoir calculer rapidement le facteur de chargement, et donc voir s'il y a nécessité de réorganiser le fichier.

*/

Module de recherche: (dichotomique)
en entrée la clé (c) à chercher et le nom externe du fichier (nomfich).
en sortie le booleen Trouv, le num de bloc (i) contenant la clé et le déplacement (j)

```
Rech( c:typeqlq; nomfich:chaine; var Trouv:bool; var i,j:entier )
var
   bi, bs, inf, sup : entier;
   trouv, stop : booleen;

DEBUT
        Ouvrir( F, nomfich, 'A' );
        bs ← entete( F,1 );          // la borne sup (le num du dernier bloc de F)
        bi ← 1;                      // la borne inf  (le num du premier bloc de F)
        Trouv ← faux;   stop ← faux;   j ← 1;

        TQ  ( bi ≤ bs  et  Non Trouv et Non stop )          // recherche externe
          i ← (bi + bs) div 2;       // le bloc du milieu entre bi et bs
          LireDir( F, i, buf );
          SI ( c ≥ buf.tab[1].cle  et  c ≤ buf.tab[buf.NB].cle )
                // recherche dichotomique à l'intérieur du bloc (dans la variable buf)...
                inf ← 1;  sup ← buf.NB;
                TQ  (inf ≤ sup  et  Non Trouv)              // recherche interne
                        j ← (inf + sup) div 2;
                        SI  (c  = buf.tab[j].cle)  Trouv ← vrai
                        SINON
                                SI (c  < buf.tab[j].cle)      sup ← j-1
                                SINON                         inf ← j+1
                                FSI
                        FSI
                FTQ
                SI ( inf > sup )   j ← inf   FSI            // fin de la recherche interne.
                                                            // j : la position où devrait se trouver c dans buf.tab
                stop ← vrai
          SINON  // non ( c ≥ buf.tab[1].cle  et  c ≤ buf.tab[buf.NB].cle )
                SI ( c < buf.tab[1].cle )
                        bs ← i-1
                SINON   // donc c > buf.tab[buf.NB].cle
                        bi ← i+1
                FSI
          FSI
        FTQ
        SI ( bi > bs )   i ← bi ;    j ← 1   FSI            // fin de la recherche externe.
                                                            // i : num du bloc où devrait se trouver c
        fermer( F )
FIN
```

Module d'insertion: (avec éventuellement des décalages intra et inter blocs)

```
Inserer( e:Tenreg; nomfich:chaine )
var
   trouv : booleen;
   i,j,k : entier;
   e,x : Tenreg;

DEBUT
  // on commence par rechercher la clé e.cle avec le module précédent pour localiser l'emplacement (i,j)
  // où doit être insérer e dans le fichier.
  Rech( e.cle, nomfich, trouv, i, j );
  SI  ( Non trouv )                        // e doit être inséré dans le bloc i à la position j
      Ouvrir( F,nomfich, 'A');             // en décalant les enreg j, j+1, j+2, ... vers le bas
      continu ← vrai;
                                           // si i est plein, le dernier enreg de i doit être inséré dans i+1
      TQ ( continu  et  i ≤ entete(F,1) )  // si le bloc i+1 est aussi plein son dernier enreg sera
            LireDir( F, i, buf );          // inséré dans le bloc i+2, etc ... donc une boucle TQ.
            // avant de faire les décalages, sauvegarder le dernier enreg dans une var x ...
            x ← buf.tab[buf.NB];

            // décalage à l'intérieur de buf ...
            k ← buf.NB;
            TQ  k > j
                  buf.tab[k] ← buf.tab[k-1];
                  k ← k-1
            FTQ

            // insérer e à la pos j dans buf ...
            buf.tab[j] ← e;

            // si buf n'est pas plein, on remet x à la pos NB+1 et on s'arrête ...
            SI ( buf.NB < b )              // b est la capacité max des blocs (une constante)
                  buf.NB ← buf.NB+1;
                  buf.tab[buf.NB] ← x;
                  EcrireDir( F, i, buf );
                  continu ← faux;

            SINON // si buf est plein, x doit être inséré dans le bloc i+1 à la pos 1 ...
                  EcrireDir( F, i, buf );
                  i ← i+1;
                  j ← 1;
                  e ← x;   // cela se fera (l'insertion) à la prochaine itération du TQ
            FSI  // non ( buf.NB < b )
      FTQ

      // si on dépasse la fin de fichier, on rajoute un nouveau bloc contenant un seul enregistrement e
      SI i > entete( F, 1 )
            buf.tab[1] ← e;
            buf.NB ← 1;
            EcrireDir( F, i, buf );        // il suffit d'écrire un nouveau bloc à cet emplacement
            Aff-entete( F, 1, i );         // on sauvegarde le num du dernier bloc dans l'entete 1
      FSI

      Aff-entete( F, 2 , entete(F,2)+1 );  // on incrémente le compteur d'insertions
      Fermer( F );
  FSI

FIN
```

La suppression logique consiste à rechercher l'enregistrement et positionner le champs 'effacé' à vrai :

```
Suppression( c:typeqlq; nomfich:chaine )
var
    trouv : booleen;
    i,j : entier;
DEBUT
   // on commence par rechercher la clé c pour localiser l'emplacement (i,j) de l'enreg à supprimer
   Rech( c, nomfich, trouv, i, j );
    // ensuite on supprime logiquement l'enregistrement
   SI  ( trouv )
          Ouvrir( F,nomfich, 'A');
          LireDir( F, i, buf );          // lecture pas vraiment nécessaire à cause de l'effet de bord de Rech sur buf
          buf.tab[j].effacé ← VRAI;
          EcrireDir( F, i, buf );
          Fermer( F )
   FSI
FIN // suppression
```

Le chargement initial d'un fichier ordonné consiste à construire un nouveau fichier contenant dès le départ n enregistrements. Ceci afin de laisser un peu de vide dans chaque bloc, qui pourrait être utilisé plus tard par les nouvelles insertions tout en évitant les décalages inter-blocs (très coûteux en accès disque) :

```
Chargement_Initial( nomfich : chaine; n : entier; u : reel )
// u est un réel compris entre 0 et 1 et désigne le taux de chargement voulu au départ
var
    e : Tenreg;
    i,j,k : entier;
DEBUT
          Ouvrir( F, nomfich, 'N' );  // un nouveau fichier
          i ← 1;                              // num de bloc à remplir
          j ← 1;                              // num d'enreg dans le bloc
          ecrire( 'Donner les enregistrements en ordre croissant suivant la clé : ' );
          POUR k←1 , n
                lire( e );
                SI  ( j ≤ u*b )               // ex: si u=0.5, on remplira les bloc jusqu'à b/2 enreg
                      buf.tab[j] ← e
                      j ← j+1;
                SINON                         // j > u*b : buf doit être écrit sur disque
                      buf.NB ← j-1;
                      EcrireDir( F, i, buf );
                      buf.tab[1] ← e;   // le kème enreg sera placé dans le prochain bloc, à la position 1
                      i ← i+1;
                      j ← 2;
                FSI
          FP
          // à la fin de la boucle, il reste des enreg dans buf qui n'ont pas été sauvegardés sur disque
          buf.NB ← j-1;
          EcrireDir( F, i, buf );
          // mettre à jour l'entête (le num du dernier bloc et le compteur d'insertions)
          Aff-entete( F, 1, i );
          Aff-entete( F, 2, n );
          Fermer( F )
FIN // chargement-initial
```

La réorganisation du fichier consiste à recopier les enreg vers un nouveau fichier de telle sorte à ce que les nouveaux blocs contiennent un peu de vide (1-u). Cette opération ressemble au chargement initial sauf que les enregistrements sont lus à partir de l'ancien fichier.

Fusion de 2 fichiers ordonnés (TOF)

On parcours les 2 fichiers (F1 et F2) en parallèle avec 2 buffers (buf1 et buf2) et on rempli un 3e buffer (buf3) pour construire un 3e fichier (F3) en ordre croissant.
Les déclarations sont celles utilisées dans les fichier TOF standards.

```
Fusion (nom1,nom2, nom3: chaine)
var
   F1 : Fichier de Tbloc Buffer buf1 Entete( entier, entier);
   F2 : Fichier de Tbloc Buffer buf2 Entete( entier, entier);
   F3 : Fichier de Tbloc Buffer buf3 Entete( entier, entier);
   i1, i2, i3 :  entier;
   j1, j2, j3 :  entier;
   continu : booleen;
   e, e1, e2 : Tenreg;

   buf : Tbloc;
   i, j, indic : entier;

Debut
        ouvrir(F1, nom1, 'A' );
        ouvrir(F2, nom2, 'A' );
        ouvrir(F3, nom3, 'N' );

        i1←1; i2←1; i3 ←1;        // les num de blocs de F1, F2 et F3
        j1←1; j2←1; j3 ←1;        // les num d'enreg dans buf1, buf2 et buf3

        LireDir(F1, 1, buf1) ;
        LireDir(F2, 1, buf2) ;

        continu ← vrai ;
```

```
TQ ( continu )          // tant que non fin de fichier dans F1 et F2 faire

    SI ( j1 ≤ buf1.NB  et  j2 ≤ buf2.NB )
        // choisir le plus petit enreg, dans buf1 et buf2
        e1 ← buf1.tab[j1];
        e2 ← buf2.tab[j2] ;
        SI ( e1.cle ≤ e2.cle )
            e ← e1;   j1 ← j1 + 1;
        SINON
            e ← e2;   j2 ← j2 + 1;
        FSI

        // et le mettre dans buf3
        SI ( j3 ≤ b )
            buf3.tab[j3] ← e;   j3 ← j3 + 1
        SINON
            buf3.NB ← j3 – 1;
            EcrireDir(F3, i3, buf3 );
            i3 ← i3 + 1;
            buf3.tab[1] ← e;
            J3 ← 2;
        FSI

    SINON // c-a-d : non ( j1 ≤ buf1.NB  et  j2 ≤ buf2.NB )
        // si tous les enreg d'un des blocs (buf1 ou buf2) ont été traités, passer au prochain
        SI ( j1 > buf1.NB )
            SI ( i1 < entete(F1, 1) )
                i1 ← i1 + 1;
                LireDir( F1, i1, buf1 ) ;
                j1 ← 1
            SINON // ( donc i1 ≥ entete(F1, 1) )
                continu ← faux ;
                i ← i2;   // pour la suite du TQ
                j ← j2;
                N ← entete(F2,1) ;
                buf ← buf2 ;
                Indic ← 2
            FSI  // ( i1 < entete(F1, 1) )

        SINON // c-a-d ( j2 > buf2.NB )
            SI ( i2 < entete(F2, 1) )
                i2 ← i2 + 1;
                LireDir( F2, i2, buf2 );
                j2 ← 1
            SINON // ( donc i2 ≥ entete(F2, 1) )
                continu ← faux;
                i ← i1;   // pour la suite du TQ
                j ← j1;
                N ← entete(F1,1);
                buf ← buf1;
                Indic ← 1;
            FSI // ( i2 < entete(F2, 1) )

        FSI // ( j1 > buf1.NB )

    FSI  // ( j1 ≤ buf1.NB  et  j2 ≤ buf2.NB )

FTQ
```

```
// continuer à recopier les enregistrement d'un seul fichier (i,j,buf) dans F3
continu ← vrai;
TQ ( continu )              // tant que non fin de fichier dans F1 ou F2 faire
        SI ( j ≤ buf.NB )
                SI ( j3 ≤ b )
                        buf3.tab[j3] ← buf.tab[j];    j3 ← j3 + 1
                SINON
                        buf3.NB ← j3 – 1;
                        EcrireDir(F3, i3, buf3 );
                        i3 ← i3 + 1;
                        buf3.tab[1] ← buf.tab[j];
                        J3 ← 2;
                FSI ; // ( j3 ≤ b )
                j ← j + 1

        SINON // c-a-d non ( j ≤ buf.NB )
                SI ( i ≤ N )
                        i ← i + 1;
                        SI ( Indic = 1 )
                                LireDir( F1, i, buf )
                        SINON
                                LireDir( F2, i, buf )
                        FSI ;
                        j ← 1
                SINON
                        continu ← faux
                FSI

        FSI // ( j ≤ buf.NB )

FTQ ;

// Le dernier buffer (buf3) n'a pas encore été ecrit sur disque ...
buf3.NB ← j3 – 1 ;
EcrireDir( F3 , i3, buf3 ) ;
Aff-entete( F3, 1, i3) ;                          // le nombre de blocs dand F3
Aff-entete( F3, 2, entete(F1,1) + entete(F2,1) ) ;      // le nombre d'enregistrements dans F3

Fermer( F1 ) ;
Fermer( F2 ) ;
Fermer( F3 ) ;
```

Fin