Exemple complet: fichier de type « TOF »[1]

(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:

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

// Les types utilisés :

    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[ b ] de Tenreg        // un tableau d'enreg d'une capacité maximal = b
        NB : entier                         // nombre d'enreg dans tab ( ≤ b)
    Fin

// Les variables globales : F et buf
    F : Fichier de Tbloc Buffer buf  Entete (entier, entier)
    /* Description 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.
    */
```

[1] Hidouci W. K. / Opérations de haut niveau / Structures de fichiers (SFSD) / ESI – 2021
https://sites.google.com/site/hidouciwk/cours/file-structures?authuser=0

Module de recherche: (dichotomique)

en entrée la clé (*c*) à chercher.

en sortie le booleen *Trouv*, le numéro de bloc (*i*) contenant la clé et le déplacement (*j*)

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

DEBUT
        /* on suppose que le fichier est déjà ouvert */
        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)

        // boucle pour la recherche dichotomique externe (dans le fichier F)
        Trouv ← faux;   stop ← faux;   j ← 1
        TQ  ( bi ≤ bs  et  Non Trouv et Non stop )
           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 )
                 // boucle pour la recherche dichotomique interne dans le bloc i (dans 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
FIN // Recherche
```

Le coût de l'opération de recherche est logarithmique car la recherche dichotomique effectue, dans le cas le plus défavorable, $\log_2 N$ lectures de blocs pour un fichier formé de N blocs. La complexité est $O(\log N)$.

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
   Ouvrir( F,nomfich, 'A')
   // 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
         continu ← vrai                       // en décalant les enreg j, j+1, j+2, ... vers le bas
                                              // 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
   FSI

   Fermer( F )
FIN // insertion
```

L'opération d'insertion peut nécessiter dans le cas le plus défavorable, des décalages inter-blocs qui se propagent jusqu'à la fin du fichier. Comme chaque décalage coûte une lecture et une écriture de bloc (c-a-d 2 accès disques), le coût total d'une insertion, en pire cas, est au voisinage de 2N accès disques pour un fichier formé de N blocs plus le coût de la recherche ($\log_2$ N lectures). La complexité est donc *O*(N).

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
   Ouvrir( F,nomfich, 'A')
   // 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 )
         // Après Rech(...) buf contient déjà le contenu du bloc i (ce n'est donc pas la peine de le relire une 2e fois)
         buf.tab[j].effacé ← VRAI
         EcrireDir( F, i, buf )
   FSI
   Fermer( F )
FIN // suppression
```

La suppression logique coûte une seule écriture de bloc, en plus du coût de la recherche ($\log_2$ N lectures) pour un fichier de N blocs. La complexité est donc celle de la recherche *O*(log N).

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 dans ]0,1] 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
```

Le chargement initial avec *n* enregistrements, nécessite la création d'un fichier formé de *n / (b*u)* blocs (donc *n / (b*u)* écritures) avec *b* la capacité maximale d'un bloc et *u* le facteur de chargement souhaité (un réel entre 0 et 1).

La réorganisation du fichier consiste à recopier les enregistrements 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 parcourt 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 )
            SINO
                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

L'opération de fusion de 2 fichiers F_1 et F_2 de tailles respectives N_1 et N_2 blocs, nécessite le parcours complet de 2 fichiers (soit $N_1 + N_2$ lectures) et la création du fichier résultat (F_3) formé par $N_1 + N_2$ blocs (soit $N_1 + N_2$ écritures), en supposant un même facteur de chargement pour les trois fichiers.
Le coût total de la fusion est donc $2N_1 + 2N_2$ accès disques.
Si les fichiers F_1 et F_2 était de même taille (N blocs chacun), le coût total de la fusion serait alors de 4N accès disques (2N lectures et 2N écritures). La complexité de la fusion est donc en O(N).