Corrigé

```
P1) Module Localiser( c, i, j )
bi ←1 ;  bs ← entete(F,1) ;  trouv ← FAUX ;
TQ ( bi ≤ bs & Non trouv )
        i ←  (bi + bs) div 2 ;
        LireDir( F, i, buf ) ;
        SI ( c < buf.tab[1].cle )
                bs ← i-1
        SINON
                SI ( c > buf.tab[ buf.nb ].cle )
                        bi ← i+1
                SINON
                        trouv ← VRAI ;    stop ← FAUX ;
                        inf ← 1 ;  sup ← buf.nb ;
                        TQ ( inf ≤ sup et Non stop )
                                j ← (inf + sup) div 2 ;
                                SI ( c < buf.tab[j].cle ) sup ← j-1
                                SINON
                                        SI ( c > buf.tab[j].cle )  inf ← j+1
                                        SINON  stop ← VRAI
                                        FSI
                                FSI
                        FTQ ;
                        SI ( Non trouv ) j ← inf FSI
                FSI
        FSI
FTQ ;
SI ( Non trouv ) i ← bi ; j ← 1 FSI
```

La complexité de cette opération est celle d'une recherche dichotomique sur un fichier formé de N blocs. Donc c'est O(log N).

```
P2) Module Lister( Vmin, Vmax )
Localiser( Vmin, i, j ) ;
// On commence par rechercher le 1er élément de l'intervalle ...
n ← i ;  m ← j ;
SI ( j = b & buf.lien <> -1 )
        // cas où l'élément est en zone de débordement
        trouv ← FAUX ;
        TQ ( Non trouv & n <> -1 )
                LireDir( F, n, buf ) ;    m ← 1 ;
                TQ (  m ≤ buf.nb & Non trouv )
                        SI ( buf.tab[m].cle ≥ Vmin )  trouv ← VRAI
                        SINON   m++
                        FSI
                FTQ // (  m ≤ buf.nb & Non trouv )
                SI ( Non trouv )  n ← buf.lien  FSI
        FTQ ; // ( Non trouv & n <> -1 )
        SI ( Non trouv )  n ← i ; m ← j  FSI
FSI ; // ( j = b & buf.lien <> -1 )
```

```
// Parcours séquentiel des éléments de l'intervalle …
stop ← FAUX ;
TQ ( Non stop )
        SI ( buf.tab[m].cle ≤ Vmax )
                SI ( Non buf.tab[m].eff )       Afficher_enreg( buf.tab[m] )   FSI ;
                Prochain_element( i, n, m ) ;  // passer au prochain éléments
                SI ( i > entete( F,1 )  stop ← VRAI  FSI
        SINON
                stop ← VRAI
        FSI
FTQ  // ( Non stop )

Module Prochain_element( i, n, m )
// retourne dans n et m le prochain élément
// les 3 paramètres en entrée/sortie
SI ( n = i )
        SI ( buf.nb < b )
                SI ( m < buf.nb)  m++
                SINON   i++ ;  n ← i ;  m ← 1 ;
                        SI ( i ≤ entete( F, 1 ) )  LireDir( F, i, buf )  FSI
                FSI
        SINON  // donc buf.nb = b
                SI ( m < b-1 ) m++
                SINON // donc m = b-1  ou  m = b
                        SI ( m = b-1 )
                                SI ( buf.lien = -1 )    m ← b
                                SINON   n ← buf.lein ;   m ← 1 ;   LireDir( F, n, buf )
                                FSI
                        SINON  // m = b
                                i++ ;  n ← i ;  m ← 1 ;
                                SI ( i ≤ entete( F, 1 ) )   LireDir( F, i, buf )  FSI
                        FSI
                FSI // ( m < b-1 )
        FSI  // ( buf.nb < b )

SINON   // ( n <> i )
        SI ( m < buf.nb )
                m++
        SINON
                n ← buf.lien ;   m ← 1 ;
                SI ( n <> -1 )  LireDir( F, n, buf )
                SINON
                        n ←  i ;  m ← b ;   LireDir( F, i, buf )
                FSI
        FSI
FSI // ( n = i )
```

La longueur des listes en débordement augmente les coûts pour localiser la borne inférieure de l'intervalle, par contre, elle n'influe pas sur le coût de l'opération 'next' pour passer au prochain élément de l'intervalle.

```
P3) Module Reorganiser( taux, nouvNom )
Ouvrir( F2, nouvNom, 'N' ) ;
i2 ← 1 ;  j2 ← 0 ;  cpt ← 0
Pour i = 1, entete(F,1)
        LireDir( F, i, buf ) ;
        Pour j = 1, min( b-1,  buf.nb )
                SI ( Non buf.tab[j].eff )
                        cpt++ ;
                        ins_dans_buf2( buf.tab[j], taux, i2, j2 ) ;
                FSI
        Fpour // j
        SI ( buf.nb = b )
                SI ( buf.lien = -1 )
                        SI (Non buf.tab[b].eff )
                                cpt++ ;
                                ins_dans_buf2( buf.tab[b], taux,  i2, j2 ) ;
                        FSI
                SINON
                        // buf.lien <> -1
                        n ← buf.lien ;  e ← buf.tab[b] ;
                        // parcourir la liste de débordement avant d'insérer le dernier elt tab[b] …
                        TQ ( n <> -1 )
                                LireDir(F, n, buf ) ;
                                Pour j = 1, buf.nb
                                        SI ( Non buf.tab[j].eff )
                                                cpt++ ;
                                                ins_dans_buf2( buf.tab[j], taux, i2, j2 ) ;
                                        FSI
                                Fpour // j
                                n ← buf.lien
                        FTQ // ( n <> -1 )
                        // insertion du dernier elt tab[b] …
                        SI ( Non e.eff )
                                cpt++ ;
                                ins_dans_buf2( e, taux, i2, j2 ) ;
                        FSI
                FSI //  ( buf.lien = -1 )
        FSI // ( buf.nb = b )
Fpour // i
// dernière écriture pour vider buf2 …
buf2.nb ← j2 ; buf2.lien ← -1 ;
EcrireDir( F2, i2, buf2 ) ;
Aff_entete( F2, 1, i2 ) ;  Aff_entete( F2, 2, -1 ) ;  Aff_entete( F2, 3, cpt ) ;
Fermer( F2 )
```

```
Module ins_dans_buf2( e, taux, i2, j2 )
//  insère e à la position j2 dans buf2, si le buffer contient plus de taux*b il sera vider sur disque (i2)
// i2 et j2 sont des paramètres en entrée/sortie
SI ( j2 ≤ taux*b )
        j2++ ;  buf2.tab[j2] ← e
SINON
        buf2.nb ← j2 ;
        buf2.lien ← -1 ;
        EcrireDir( F2, i2, buf2 ) ;
        i2++ ;
        j2 ← 1 ;
        buf2.tab[1] ← e
FSI
```

C1) Comparaison des structures de fichiers

- Hachage statique vs hachage dynamique :
Dans le cas du hachage statique, la fonction de hachage reste, donc si le fichier change souvent de taille (fichiers dynamiques) les performances se dégradent.
Dans le cas du hachage dynamique, les performances restes bonnes quelque soit l'évolution en taille du fichier, car la fonction de hachage change et s'adapte dynamiquement en fonction de la taille du fichier.

- B-arbres vs B+-arbres
Les méthodes sont presque les mêmes sauf que dans le cas du B+-arbres, les requêtes à intervalles sont plus performantes que pour le cas des B-arbres, car, toutes les données se retrouvent au niveau des feuilles et ces dernières sont chaînées entre elles, facilitant ainsi l'opération next.

C2) Les requêtes algébriques

```
Q1 :    R1 ← Select( Appels, dateApp = d ) ;
        R2 ← Project( R1, numAppelé ) ;
        R3 ← Project( Abonnés,  nomAb, numTel ) ;
        Resultat ← Join( R3, R2, numTel = numAppelé )

Q2 :    R1 ← Select( Appels, dateApp ≥ '28/10/2013' and dateApp ≤ '02/11/2013' ) ;
        R2 ← Project( R1, numAppelant, numAppelé ) ;
        R3 ← Project( Abonnés, numTel, nomAb ) ;
        R4 ← Select( R3, nomAb = 'Ali' ) ;
        R5 ← Join( R4, R2, numTel = numAppelant ) ;
        R6 ← Join( R5, Abonnés, numAppelé = numTel ) ;
        Resultat ← Project( R6, nomAb ) ;
```