Corrigé de l'examen de Structures de Fichiers (SFSD) / 2CP / ESI 2020/2021

— Enoncé —

Soit **F** un fichier vu comme Liste de blocs, ordonné et contenant des enregistrements à taille fixe. La capacité maximale d'un bloc est **b** enregistrements. L'entête est formée de 3 caractéristiques :
(1) le numéro du 1er bloc (la tête de la liste) , *(2)* le nombre total d'insertions et *(3)* le nombre total de suppressions

Nous disposons en **MC** d'un seul buffer (**buf**) pour manipuler le fichier **F** et d'une table d'index (**Ind**), non dense formée par **M** entrées (on suppose que **M** est toujours plus petit que le nombre de blocs formant le fichier **F**).
Chaque entrée de la table d'index contient un couple < *cléMin* , *numBloc* >, avec *cléMin* représentant la plus petite clé d'un groupe de blocs chaînés du fichier **F**. *numBloc* est le numéro du 1er bloc de ce groupe. Comme F est ordonné, *cléMin* représente aussi la clé du 1er enregistrement du bloc numéro *numBloc*.

Lors de l'insertion d'un enregistrement **e**, si le bloc cible est déjà plein, il y aura une opération d'éclatement avec partage équitable de la séquence ordonnée (en 2 moitiés à peu près égales). Le groupe où a lieu l'insertion, s'allonge alors d'un bloc supplémentaire. De ce fait, après un certain nombre d'insertions, les différents groupes du fichier **F**, ne seront pas forcément de même taille (en nombre de blocs).
Les suppressions sont logiques.

Q1 Donnez les déclarations de la table d'index **Ind** et du fichier **F** (incluant le buffer et l'entête)
Q2 Donnez l'algorithme de la recherche (efficace) d'un enregistrement de clé **c** donnée
Q3 Donnez l'algorithme d'insertion d'un enregistrement **e** donné

Dans le but de rééquilibrer la taille des groupes formant le fichier **F**, on effectue périodiquement une opération de réorganisation. Cette dernière consiste à recopier dans un nouveau fichier, uniquement les enregistrements non supprimés logiquement. Le facteur de chargement du nouveau fichier sera **u %** (**u** donné). La nouvelle table d'index (**Ind**) est reconstruite avec **M** couples <*cléMin* , *numBloc* > (**M** donné) de manière à générer des groupes de tailles similaires (en nombre de blocs). La différence de taille ne doit pas dépasser un seul bloc.

Q4 Donnez la formule pour calculer le nombre total de blocs (*nbBlocs*) du nouveau fichier.
Q5 Comment calculer les tailles exactes (en nombre de blocs) de chaque groupe du nouveau fichier.
Q6 Donnez l'algorithme '**reorganiser(M,u)**' permettant de réorganiser le fichier **F**. On utilisera pour cela 2 buffers en **MC** (**buf** et **buf2**).

— Réponses —

Q1 Donnez les déclarations de la table d'index **Ind** et du fichier **F** (incluant le buffer et l'entête)

```
typeCouple =  structure
        cleMin : typeqlq
        numBloc : entier
fin
Ind : Tableau[ MaxInd ] de typeCouple        // la table d'index
M : entier
```

1 pt

```
type Tbloc = structure                       // structure d'un bloc et des buffers aussi
        tab : Tableau[ b ] de typeEnreg
        eff : tableau[ b ] de booléen        // pour les effacements logiques
        nb : entier
        lien : entier
fin
F : Fichier de Tbloc Buffer buf Entete( entier , entier , entier )
                                             // (  tete ,  nbIns et nbSup)
```

1 pt

Q2 Donnez l'algorithme de la recherche (efficace) d'un enregistrement de clé **c** donnée
La recherche commence dans la table d'index Ind (recherche dichotomique en MC) et se poursuit en séquentielle dans la liste de blocs du groupe sélectionné (la recherche interne dans buf est aussi dichotomique) :

```
// Recherche de la clé c. Résultats : trouv:booléen et l'adresse de l'enreg : i , j
Recherche( c : typeqlq, nomfichier:chaine, sorties trouv:booléen , i , j : entier )
        SI (nomfichier <> '' )   Ouvrir( F , nomfichier , 'A' )   FSI
        rech_Index( c , k )        // recherche en MC dans l'index Ind
        i ← Ind[ k ].numBloc ; j ← 1
        trouv ← faux ; stop ← faux
        // recherche séquentielle dans le groupe Ind[ k ].numBloc ...
        TQ ( non trouv && non stop )                                          3 pts
                LireDir( F , i , buf )
                SI ( c ≤ buf.tab[ buf.nb ] )
                        stop ← vrai
                        rech_buf( c , trouv , j )
                SINON
                        SI ( buf.lien <> -1 )   i ← buf.lien   SINON  j ← buf.nb+1 ; stop ← vrai  FSI
                FSI
        FTQ
        SI ( trouv )
                SI (buf.eff[ j ] == vrai )   trouv ← faux FSI       // en cas d'effacement logique
        FSI
        SI (nomfichier <> '' )   Fermer( F )   FSI
```

```
// recherche de la clé c dans l'index Ind …
rech_Index( c:typeqlq , sortie  k : entier )                                  3 pts
        bi ← 1 ;  bs ←  M ;  trouv ← faux
        TQ ( bi ≤ bs && Non trouv )
            k ← (bi + bs) / 2
            SI ( c == Ind[ k ].cléMin ) trouv ← vrai
            SINON
                SI ( c < Ind[ k ].cléMin )   bs ← k – 1   SINON   bi ← k + 1  FSI
            FSI
        FTQ
        SI ( Non trouv ) k ← bs ;   SI ( k < 1 )  k ← 1  FSI    FSI
```

```
// recherche de la clé c dans le buffer buf …
rech_buf( c:typeqlq , sortie  trouv:booléen, j : entier )
        bi ← 1 ;  bs ←  buf.nb ;  trouv ← faux
        TQ ( bi ≤ bs && Non trouv )
            j ← (bi + bs) / 2
            SI ( c == buf.tab[ j ].clé ) trouv ← vrai
            SINON
                SI ( c < buf.tab[ j ].clé )   bs ← j – 1   SINON   bi ← j + 1  FSI
            FSI
        FTQ
        SI ( Non trouv )   j ← bi  FSI
```

Q3 Donnez l'algorithme d'insertion d'un enregistrement **e** donné

Le principe de l'insertion d'un enregistrement est de localiser le bloc i où doit se faire l'insertion (en utilisant par exemple, le module de recherche précédent), ensuite il y a deux cas :

a) soit le bloc i n'est pas plein et dans ce cas on rajoute l'enreg e au bloc i et on s'arrête **2 pts**

b) soit le bloc i est déjà plein et dans ce cas il y aura un éclatement de bloc : **4 pts**

Allocation d'un nouveau bloc i'
Partage du contenu du bloc i (incluant e) en 2 moitiés (l'une reste dans i et l'autre dans i')
Mettre à jour le chaînage pour que i' soit le suivant de i dans la liste des blocs de F

La table d'index Ind n'est mise à jour que si e.clé représente la plus petite valeur du 1er groupe (donc de F)

```
Ins( e:typeEnreg , nomfichier:chaine )
  SI ( nomfichier <> '' ) Ouvrir( F, nomfichier, 'A' ) FSI
  Recherche( e.clé , '' , trouv , i , j )     // localiser le bloc i et la position j où doit se faire l'insertion de e
  // le bloc i est déjà en MC (dans buf)
  SI ( buf.tab[ j ].clé == e.clé && buf.eff[ j ] == vrai )  // cas particulier où la clé était effacée logiquement
        buf.tab[ j ] ← e ; buf.eff[ j ] ← faux              // on réutilise alors son emplacement …
        Aff_Entete( F , 3 , Entete(F,3) – 1 )               // le nombre total de suppressions
        EcrireDir( F , i , buf )
  SINON  SI ( buf.nb < b )      // si le bloc i n'est pas plein, on insère e dans i à la position j
           k ← buf.nb
           TQ (k ≥ j )  buf.tab[ k+1] ← buf.tab[ k  ] ; buf.eff[ k+1] ← buf.eff[ k  ] ; k--  FTQ
           buf.tab[ j ] ← e ;  buf.eff[ j ] ← faux ;  buf.nb++
           EcrireDir( F , i , buf )
           SI ( j == 1 )      // cas particulier où e.clé est la plus petite valeur de F …
                    Ind[ 1 ].cléMin ← e.clé
           FSI
         SINON                // si le bloc i est déjà plein alors faire un éclatement…
           i' ← AllocBloc( F )
           // préparer la séquence ordonné dans buf en sauvegardant le dernier élément dans sauv
           SI ( e.clé < buf.tab[ b ].clé )
              sauv ← buf.tab[ b ] ; sauv_eff ← buf.eff[ b ] ; stop ← faux ; k ← b-1
             TQ ( non stop && k ≥ 1 )     // insertion de e par décalages dans buf…
                  SI ( e.clé < buf.tab[ k ].clé )
                           buf.tab[ k+1 ] ← buf.tab[ k ] ;  buf.eff[ k+1 ] ← buf.eff[ k ] ; k--
                  SINON
                           buf.tab[ k+1 ] ← e ;  buf.eff[ k+1 ] ← faux ; stop ← vrai
                  FSI
             FTQ
             SI ( k == 0 ) // cas particulier où e.clé est la plus petite valeur de F …
                  buf.tab[ 1 ] ← e ;  buf.eff[ 1 ] ← faux ; Ind[ 1 ].cléMin ← e.clé  // m-a-j de Ind
             FSI
           SINON  sauv ← e ;  sauv_eff ← faux
           FSI
           // écrire la 1ere moitié dans le bloc i…
           sauv_lien ← buf.lien ;  buf.lien ← i' ;  buf.nb ← (b div 2) + 1
           EcrireDir( F , i , buf )
           // écrire la 2e moitié dans le bloc i'…
           k ← buf.nb+1
           TQ ( k ≤ b )              // décalages de la 2e moitié vers le début de buf…
                  buf.tab[ k – buf.nb ] ← buf.tab[ k ] ; buf.eff[ k – buf.nb ] ← buf.eff[ k ] ; k++
           FTQ
           buf.tab[ k – buf.nb ] ← sauv ;  buf.eff[ k – buf.nb ] ← sauv_eff
           buf.nb ←  k – buf.nb ;  buf.lien ← sauv_lien
           EcrireDir( F , i' , buf )
         FSI
         Aff_Entete( F , 2 , Entete(F,2) + 1 )   // m-a-j de l'entête 2 (nombre total d'insertions)
  FSI
  SI ( nomfichier <> '' ) Fermer( F ) FSI
```

Q4 Donnez la formule pour calculer le nombre total de blocs (*nbBlocs*) du nouveau fichier.

Après la réorganisation, le nombre de blocs formants le nouveau fichier dépendra du nombre d'enregistrements non supprimés logiquement et du facteur de chargement u.
Le nombre d'enregistrements non supprimés logiquement est égal à nbInsertions – nbSuppressions
c-a-d (Entete(F,2) – Entete(F3)) où F représente l'ancien fichier (avant la réorganisation)
Les blocs du nouveau fichier seront remplis à u %
Donc **nbBlocs = [(Entete(F,2) – Entete(F3)) / (u*b)]** **1 pt**
avec [x] représentant le plus petit entier ≥ x

Q5 Comment calculer les tailles exactes (en nombre de blocs) de chaque groupe du nouveau fichier.

Le nombre groupes dans le nouveau fichier est M
Pour avoir des groupes équilibrés (de même taille ou presque) il faut diviser le nombre de blocs (nbBlocs) par M
Si nbBlocs est un multiple de M, on aura des groupes exactement de même taille = nbBlocs div M
Si nbBlocs n'est pas un multiple de M, on aura quelques groupes avec une taille = nbBlocs div M et les autres avec une taille = (nbBlocs div M) + 1 :

Les **(nbBlocs mod M) premiers groupes** auront une taille = (**nbBlocs div M) + 1** **2 pts**
et les **M - (nbBlocs mod M)** groupes restant auront une taille = **(nbBlocs div M)**

Q6 Donnez l'algorithme '**reorganiser(M,u)**' permettant de réorganiser le fichier **F**. On utilisera pour cela 2 buffers en **MC** (**buf** et **buf2**).

Le principe général est d'utiliser un des buffers (par exemple buf) pour parcourir l'ancien fichier F séquentiellement et récupérer les enregistrements non supprimés logiquement. Ces derniers sont utilisés pour construire, à l'aide du 2e buffer (par exemple buf2), le nouveau fichier en remplissant les blocs à u % de leur capacité maximale.
Durant ce même parcours, la table d'index Ind est construite avec les 1ere clés de chaque groupe de blocs dans le nouveau fichier. Il y aura en tout M groupes (chacun ayant une entrée dans la table Ind)
Les (nbBlocs mod M) premiers groupes seront formés de (nbBlocs div M)+1 blocs chacun
Le reste des groupes seront formés de (nbBlocs div M) blocs chacun. **3 pts**

```
// en entrée on donne M : le nombre de groupes à former et u (un nombre réel entre 0 et 1) : représentant le
// pourcentage de remplissage des nouveaux blocs. F, G, buf et buf2 sont des variables globales
Réorganiser( M:entier , u:réel )
    Ouvrir( F , 'ancienNom' , 'A' )
    Ouvrir( G , 'nouveauNom' , 'N' )

    nbEnreg ← Entete(F,2) – Entete(F,3)    // nombre d'enreg non supprimés logiquement
    nbBlocs ← nbEnreg div (u*b)            // nombre de blocs à allouer pour G
    SI ( nbEnreg mod (u*b) > 0 ) nbBlocs++ FSI
    tailleGrp ← nbBlocs div M              // taille d'un groupe : tailleGrp ou tailleGrp+1

    i1 ← Entete(F,1)
    LireDir( F , i1 , buf )                // lire le 1er bloc de F
    j1 ← 1
    i2 ← AllocBloc( G )                    // allouer le 1er bloc du nouveau fichier G
    Aff_Entete( G , 1 , i2 )               // tête de liste pour G
    Aff_Entete( G , 2 , nbEnreg )          // nombre d'enreg dans G
    Aff_Entete( G , 3 , 0 )                // nombre d'enreg supprimés dans G
```

```
POUR  numGrp = 1 , M                    // construire les M groupes ...
        SI ( numGrp ≤ (nbBlocs mod M) )
                // les (nbBlocs mod M) premiers groupes seront formés de tailleGrp+1 blocs
                ConstruireGrp( numGrp, M, tailleGrp+1, i1, j1, i2 )
        SINON
                // les autres groupes seront formés de tailleGrp blocs
                ConstruireGrp( numGrp, M, tailleGrp , i1, j1, i2 )
        FSI
FP
Fermer( F )
Fermer( G )


// le module ConstruireGrp construit un nouveau groupe formé de taille blocs dans G ( à partir du bloc i2)
// en récupérant les enreg de F à partir de (i1, j1)
// il retourne comme sorties, les nouvelles valeurs de i1, j1 et i2 (selon l'avancement dans F et dans G)
ConstruireGrp( numGrp : entier , M:entier , taille:entier , sorties i1 : entier, j1 : entier, i2 : entier )
        cpt ← 1 ; stop ← faux
        TQ ( cpt ≤ taille && non stop )          // construire un groupe de G formé de taille blocs
                j2 ← 1
                TQ ( j2 ≤ u*b && non stop )      // remplir un bloc de G à u %
                        SI ( non buf.eff[ j1 ] )
                            SI ( cpt == 1 && j2 == 1 )
                                    // c'est le 1er bloc du groupe et le 1er enreg du bloc, donc : m-a-j de Ind
                                    Ind[ numGrp ] ← < buf.tab[ j1 ].clé , i2 >      // ajouter entrée dans Ind
                            FSI
                            buf2.tab[ j2 ] ← buf.tab[ j1 ] ;  buf2.eff[ j2 ] ← faux
                            j2 ++
                        FSI
                        j1 ++
                        SI ( j1 > buf.nb )
                            // passer au bloc suivant dans F
                            i1 ← buf.lien
                            SI ( i1 <> -1 )
                                    LireDir( F , i1 , buf )
                                    j1 ← 1
                            SINON
                                    stop ← vrai
                            FSI
                        FSI
                FTQ  // ( j2 ≤ u*b && non stop )      - FinTQ remplir un bloc de G -

                buf2.nb ← j2 - 1
                SI ( cpt == taille && numGrp == M )
                        // si on est dans le dernier bloc du dernier groupe
                        buf2.lien ← -1
                SINON
                        buf2.lien ← AllocBloc( G )
                FSI
                EcrireDir( G , i2 , buf2 )
                i2 ← buf2.lien
                cpt++

        FTQ   // ( cpt ≤ taille && non stop )          - FinTQ construire un groupe de G -
```