Corrigé - Examen Semestriel – Structures de Fichiers (SFSD) – 2CPI – ESI 2018/2019

Exercice 1 (Structures séquentielles)

a) Avantages et inconvénients des fichiers séquentiels ordonnés, formés par des blocs contigus
=> (sur 2 pts)

Les avantages :

- la possibilité de faire la **recherche dichotomique** (s'il n'y pas de chevauchement dans le cas variable)
- l'efficacité de la **requête à intervalle** (pour passer à l'enregistrement suivant)

Les inconvénients :

- l'insertion qui risque de provoquer les décalages inter-blocs
- la suppression dans le cas physique

b) Séparation des enregistrements de taille variables, lorsque la longueur n'est pas limitée
=> (sur 2 pts)

- Soit **en utilisant des caractères spéciaux** pour séparer les enregistrements entre eux (dans ce cas un traitement particulier doit être fait dans le cas où ce caractère spécial peut apparaître comme valeur à l'intérieur d'un enregistrement).
- Soit **en préfixant chaque enregistrement par sa taille** (sous forme de chaîne de longueur variable). Cette chaîne de caractères numériques est elle même de taille variable, donc elle doit être terminée par un caractère spécial quelconque (non numérique).

c) Les métriques permettant d'évaluer les performances d'une structure de fichiers
=> (sur 2 pts)

- Le **coût en opération de lecture et écriture de bloc** pour la réalisation des différentes opérations de base (recherche, insertion, suppression …)
- Le **facteur de chargement** moyen (taux de remplissage des blocs)
- L'**encombrement mémoire** (en MC et MS) : l'espace mémoire nécessaire pour les informations de contrôle (autres que les données elles mêmes), comme par exemple les index en MC, les numéros de blocs dans les champs suivant, fils, les champs additionnels pour les suppression logiques … etc

Exercice 2 (Hachage)

Fichier avec Essai-linéaire combinant suppressions logique et physique

a) Représentation d'une case vide et vérification s'il y a toujours au moins une case vide dans le fichier
=> (sur 1 pt)

La case vide est représentée par un **bloc non plein**

Pour être sûr qu'il existe toujours au moins une case vide dans le fichier, on peut utiliser (comme caractéristique) un **compteur d'insertions**, celui-ci doit toujours être **< N*b**

b) Les caractéristiques du fichier ainsi que la déclaration d'un buffer
=> (sur 1 pt)

- Les caractéristiques du fichiers peuvent inclure :
  le nombre d'insertions, le nombre de suppressions logiques, …
- Déclaration d'un buffer ;
  buf:Tbloc ;
  avec Tbloc étant une structure contenant un tableau de données 'tab' et un tableau de booléens 'eff' indiquant les effacements logiques.
  Un exemple de déclaration du type d'un buffer :

```
Type Tbloc = struct
                    NB : entier                    // nombre de données dans le bloc
                    tab : tableau[b] d'entiers     // les données du bloc
                    eff : tableau[b] de booléens   // les indicateurs d'effacement logiques
             fin.
```

c) Algorithmes de suppression et d'insertion
=> (sur 4 pts : 2 pts pour la suppression et 2 pts pour l'insertion)
- Algo de Suppression

sup(x)

```
Rech( x, trouv, i, j ) ;
SI ( trouv )
    // optionnellement lire le bloc i
    // c'est possible aussi de ne pas le lire car le module de recherche l'a déjà lu
    LireDir( F, i, buf ) ;

    SI ( buf.NB < b )
        // bloc non plein, donc suppression physique …
        buf.tab[ j ] ←  buf.tab[ buf.NB ] ; buf.NB-- ;  EcrireDir( F, i, buf ) ;
        // ajuster le nombre d'insertions ...
        Aff_Entete( F, 1, Entete(F,1) - 1 ) ;
    SINON
        // bloc plein, donc suppression logique …
        buf.eff[j] ←  VRAI ;  EcrireDir( F, i, buf )
    FSI
FSI
```

- Algo d'insertion

ins(x)

```
Rech( x, trouv, i, j ) ;
SI ( Non trouv )
    // optionnellement lire le bloc i
    // c'est possible aussi de ne pas le lire car le module de recherche l'a déjà lu
    LireDir( F, i, buf ) ;

    SI ( j ≤ buf.NB )
        // donc remplacement d'une donnée effacée logiquement
        buf.tab[ j ] ←  x ; buf.eff[ j ] ← FAUX ;  EcrireDir( F, i, buf )
    SINON
        // insertion dans une case vide (si ce n'est pas la dernière) …
        SI ( Entete(F,1) < N*b – 1 )
            buf.NB++ ;  buf.tab[  buf.NB ] ← x ; buf.eff[  buf.NB ] ← FAUX ;
            EcrireDir( F, i, buf ) ;
            // incrémenter le nombre d'insertions ...
            Aff_Entete( F, 1, Entete(F,1) + 1 ) ;
        SINON
            ecrire(« Pas de place pour insérer une nouvelle donnée »)
        FSI
    FSI
FSI // (Non Trouv)
```

Exercice 3 (Résolution d'entité)
Recherche de correspondances entre les enregistrements d'un un fichier F TOF formé de N blocs physiques.
Nous disposons en mémoire centrale de M buffers d'entrée (sous forme d'un tableau de buffers) 'bf[1..M]' pour lire les blocs de F (avec M << N) et d'un buffer de sortie 'bs' pour écrire dans le fichier S.

a) L'approche la plus efficace pour résoudre ce problème.

=> (sur 2 pts)

La recherche de correspondance entre enregistrement d'un même fichier ressemble à l'opération de jointure d'un fichier avec lui même. Donc les mêmes algorithmes peuvent être utilisés pour cette tâche.
Si la condition pour vérifier la correspondance entre 2 enregistrements est une simple égalité entre deux attributs, les 3 approches sont alors possibles et dans ce cas, la plus performante est l'approche par tri-fusion qui bénéficie du fait que le fichier F est déjà trié. N lectures seront alors nécessaires pour le parcours de type fusion et détecter les correspondances qui dans ce cas ce réduit à détection de valeur doubles après le tri. Dans l'approche par hachage il faudrait au moins 2N lectures (N pour la phase du 'Build' d'un seul fichier et N pour la phase de 'Probe' du même fichier). La méthode la moins performante est l'approche par boucles-imbriquées ayant une complexité de $O(N^2)$.
Si par contre la condition est quelconque, **comme c'est le cas dans cet exercice**, alors la seule approche possible pour résoudre ce problème est celle par '**boucles-imbriquées**'.

b) Algorithme de type 'boucles-imbriques', pour construire le fichier S (contenant toutes les correspondances) à partir du fichier F.

Nous supposerons que les fonctions suivantes sont disponibles et peuvent donc être utilisées dans l'algorithme :

- ***Match(e1, e2)*** vérifie s'il existe une correspondance entre les enregistrements e1 et e2 de F
- ***Concat(e1, e2)*** retourne un enregistrement formé par la concaténation de e1 et e2

=> (sur 4 pts)

Les M-1 premiers buffers seront utilisés pour parcourir le fichier F dans la boucle externe.
Pour chaque fragment de taille M-1 blocs en MC, le buffer buf[M] sera utilisé pour parcourir le fichier F dans la boucle interne (à partir de du fragment courant).
Le buffer 'bs' sera utilisé pour remplir le fichier de sortie S avec les résultats des correspondances trouvées.

- Algorithme de construction du fichier S des correspondances à partir de F

```
i1 ← 1 ;                    // pour la boucle externe  (fichier en entrée)
i3 ← 1 ;                    // pour le fichier de sortie
j3 ← 1 ;

Ouvrir( F, 'entrée.dat' , 'A' ) ;
Ouvrir( S, 'sortie.dat', 'N' ) ;

N ← Entete(F,1) ;
```

```
// La boucle externe (pour parcourir le fichier en entrée par fragments de M-1 blocs) ...
TQ ( i1 ≤ N )
        // lecture d'un fragment de M-1 blocs au max, dans les buffers buf[1], buf[2], … buf[M-1]
        k ← 1 ;
        TQ ( k ≤ M-1 et i1 + k – 1 ≤ N )
                LireDir( F, i1 + k – 1, buf[ k ] ) ;   k++
        FTQ ; // k-1 est le nombre de buffers formants le fragment courant (en général k=M)

        // La boucle interne (pour parcourir le fichier en entrée avec le buffer buf[M]) …
        i2 ← i1 ;
        TQ ( i2 ≤ N )
                // lecture d'un seul bloc à la fois (s'il n'est pas déjà en MC) …
                SI ( i2 > i1 + M – 2 )  LireDir( F, i2, buf[ M ] )
                SINON                   buf[ M ] ← buf[ i2 - i1 + 1 ]
                FSI
                // chercher toutes les correspondances entre les enregistrements
                // de buf[M] avec ceux du fragment buf[1..k-1]  (en évitant les tests redondants) …
                POUR j2 = 1, buf[ M ].NB
                    e2 ← buf[ M ].tab[ j2 ] ;
                    i ← 1 ;  // tester buf[M] avec le fragment courant: buf[1..k-1]
                    TQ ( i < k et i2 ≥  i1 + i - 1 )  // la 2e cnd, évite les tests redondants
                        POUR j1 = 1, buf[ i ].NB
                            e1 ← buf[ i ].tab[ j1 ] ;
                            SI ( i2 > i1 + i - 1 ou j2 > j1 ) // pour éviter les tests redondants
                                SI ( Match(e1,e2) )
                                        bs.tab[ j3 ] ← Concat(e1,e2) ;   j3++
                                        SI ( j3 > b )
                                                bs.NB ← b ;
                                                EcrireDir( S, i3, bs ) ;  i3++ ;  j3 ← 1
                                        FSI
                                FSI // Match(e1,e2)
                            FSI // évitement des tests redondants
                        FP // j1 : fin de parcours des enreg du buffer i du fragment courant
                        i++
                    FTQ // i : fin de parcours du fragment courant (des blocs i1 à i1+k-1)
                FP // j2 : fin de parcours des enreg du buffer M (le bloc i2)
                i2++
        FTQ // i2 (la boucle interne)
        i1 ← i1 + M - 1 // pour passer au prochain fragment
FTQ ; // i1 (la boucle externe)

// dernière écriture dans S …
SI ( j3 > 1 )
        bs.NB ← j3 - 1 ;
        EcrireDir( S, i3, bs ) ;
        i3++ ;
FSI
Aff_entete( S, 1, i3 - 1 ) ;
Fermer(F) ;
Fermer(S)
```

c) Le coût de l'algorithme, en opérations de lecture et sa complexité.

=> (sur 2 pts)

Posons T = M-1
Dans la boucle externe, le fichier F est lu une seule fois par fragment de taille T (soit N lectures)
Pour chaque fragment lu, la boucle interne permet de lire le fichier F bloc par bloc à partir du début de chaque fragment. Comme les blocs formants le fragment en cours sont déjà présents en MC dans les buffers buf[1], buf[2], … buf[M-1], ce n'est pas la peine de les lire (le SINON du SI (i2 > i1 + M – 2) ….). Ainsi le nombre de lectures physiques effectivement réalisées dans la boucle interne est :
N-T (lors du 1er passage) , N-2T (lors du 2e passage) , N-3T (3e passage) , …. N-kT (lors de l'avant dernier passage, avec k = N div T) et 0 lectures lors du dernier passage (car le nombre de blocs restant à ce moment là est ≤ T, donc les blocs sont déjà présents en MC).
Donc au total, le nombre de lectures est :
N + (N-T + N-2T + N-3T + … + N-kT) =
N + N(N div T) - (T + 2T +3T + … kT) =
N + N(N div T) - T * k(k+1)/2 =
N + N(N div (M-1)) - (M-1) * (N div (M-1))((N div (M-1))+1)/2 ≈
$N + N^2/(M-1) - (M-1)(N^2/(M-1)^2 + N/(M-1))/2$ =
$N/2 + N^2/(2(M-1))$ opérations de lectures
La complexité en opérations de lecture est donc **$O(N^2)$**