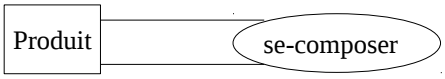
Corrigé de l'examen de « Structures de Fichiers » (SFSD) – 2CPI – 2015/2016

Exercice 1

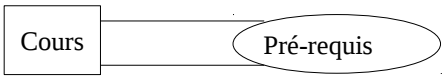
1/ Remplir le tableau suivant avec uniquement des opérateurs dérivés de l'algèbre relationnelle :

Opérateur dérivé	**BUT**
INTERSECTION $R \cap S = T$	- Permet d'obtenir l'ensemble des tuples appartenant aux deux relations
JOINTURE $R \bowtie S = T$	- Permet d'établir le lien sémantique entre les relations
DIVISION $R \div S = T$	- Répondre aux requêtes de type « tous les » - Un tuple t est dans T si et seulement si pour tout tuple s de S, le tuple <t,s> est dans R

2/ Définir une relation réflexive dans le modèle entité/association ? Donner un exemple ?
La relation réflexive (récursive ou circulaire) c'est une relation qui associe la même entité.
Exemples :
- un produit est composé d'un autre produit



- un cours nécessite des pré-requis (d'autres cours)



3/ Soit les tables relationnelles suivantes représentant le suivi des cours des étudiants :
ETUDIANT (Matricule, Nom, Adress)
COURS (NumC, Désignation, Crédit)
SUIVRE (Matricule, NumC, Note)
Exprimez les requêtes suivantes en langage algébrique :

3.1/ Quels sont les noms des étudiants qui suivent tous les cours ? Proposer deux solutions.

Solution 1 :

$$R1 = \Pi_{Matricule,\ NumC}(Suivre)$$

$$R2 = \Pi_{NumC}(Cours)$$

$$Résultat = R1 \div R2$$

Solution 2 :

$$Résultat = R1 - R2 \text{ avec}$$

$$R_1 = \pi_{A_1,\ldots,A_m}(R) \ et \ R_2 = \pi_{A_1,\ldots,A_m}((R_1 \times S) - R)$$

3.2/ Quels sont les homonymes parmi les étudiants ?

$$R1 = Etudiant \bowtie_{(Nom = Nom)} Etudiant$$

$$R2 = Etudiant \bowtie_{(Matricule = Matricule)} Etudiant$$

$$Résultat = R1 - R2$$

Exercice 2

Soit un b-arbre d'ordre m.

1/ a quoi correspond m et quelle est sa valeur min
L'ordre d'un b-arbre représente le nombre maximum de fils par nœud.
Sa valeur minimale est 3 (par exemple les arbres 2-3)

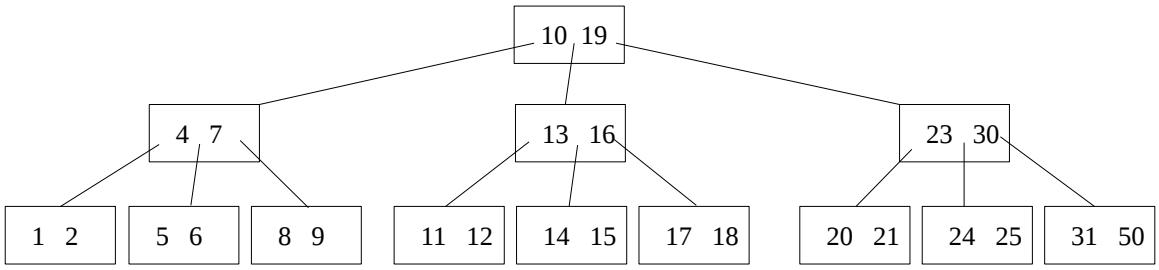
2/ combien de clés au minimum dans un nœud autre que la racine
Le nombre minimal de clés par nœud (autre que la racine) est $\lceil m/2 \rceil - 1$

3/ comment est choisit m
En général, pour la gestion des fichiers, on choisit m de telle sorte à remplir au maximum un bloc physique

4/ combien de clés peut on stocker dans b-arbre de hauteur h et d'ordre m
Si Nb_Noeuds représente le nombre total de nœuds dans l'arbre et tous les nœuds sont remplis à 100 %,
le nombre total de clés = Nb_Noeuds * (m-1)
De plus, si tous les nœuds sont remplis à 100 %, on aura dans chaque niveau i, m^i nœuds.
Dans ce cas Nb_Noeuds = $1 + m + m^2 + m^3 + ... m^h = (m^{h+1} - 1) / (m - 1)$
Et donc le nombre de clés = $m^{h+1} - 1$

5/ même question avec m=5 et h=2
Le nombre de clé = $5^3 - 1 = 124$

6/ Déterminer l'arbre obtenu en ajoutant la clé 21



Exercice 3

Pour réaliser l'opération d'intersection de deux ensembles d'entiers représentés par deux fichiers F et G de type TOF (fichiers contigus, ordonnés et à format d'enregistrement fixe), nous disposons en mémoire centrale de 3 buffers.

1/ Algorithme de l'intersection des fichiers F et G. Le résultat dans un nouveau fichier H

```
Début   // (Intersection)
   OUVRIR( F, nom1, 'A' ) ;
   OUVRIR( G, nom2, 'A' ) ;
   OUVRIR( H, nom3, 'N' ) ;
   i1 ← 1 ; i2 ← 1 ; i3 ← 0 ;    j1 ← 1 ; j2 ← 1 ; j3 ← 0 ;
   SI ( Entete(F,1) <> -1 )  LireDir(F, 1, buf1 ) FSI ;
   SI ( Entete(G,1) <> -1 )  LireDir(G, 1, buf2 ) FSI ;

   TQ ( i1 ≤ Entete(F,1)  && i2 ≤ Entete(G,1) )
        SI ( buf1.tab[j1] < buf2.tab[j2] )
                j1 ← j1 + 1 ;
                SI ( j1 > buf1.nb )
                        i1 ← i1 + 1 ;
                        SI ( i1 ≤ Entete(F,1) ) LireDir( F, i1, buf1 ) ; j1 ← 1  FSI
                FSI
        SINON
           SI ( buf1.tab[j1] > buf2.tab[j2] )
                j2 ← j2 + 1 ;
                SI ( j2 > buf2.nb )
                        i2 ← i2 + 1 ;
                        SI ( i2 ≤ Entete(G,1) ) LireDir( G, i2, buf2 ) ; j2 ← 1  FSI
                FSI
           SINON  //  donc ( buf1.tab[j1] = buf2.tab[j2] )
                j3 ← j3 + 1 ;
                SI ( j3 > b )
                        buf3.nb ←j3 - 1 ;   // ou b
                        i3 ← i3 + 1 ;
                        EcrireDir( H, i3, buf3 ) ;
                        j3 ← 1 ;
                FSI ;
                buf3.tab[j3] ← buf1.tab[j1] ;
                j1 ← j1 + 1 ;
                SI ( j1 > buf1.nb )
                        i1 ← i1 + 1 ;
                        SI ( i1 ≤ Entete(F,1) ) LireDir( F, i1, buf1 ) ; j1 ← 1  FSI
                FSI ;
                j2 ← j2 + 1 ;
                SI ( j2 > buf2.nb )
                        i2 ← i2 + 1 ;
                        SI ( i2 ≤ Entete(G,1) ) LireDir( G, i2, buf2 ) ; j2 ← 1  FSI
                FSI
           FSI   // ( buf1.tab[j1] > buf2.tab[j2] )
        FSI   //( buf1.tab[j1] < buf2.tab[j2] )
   FTQ ;  // ( i1 ≤ Entete(F,1)  && i2 ≤ Entete(G,1) )

   // Dernière écriture de buf3 …
   SI ( j3 > 0 )    // si l'intersection n'est pas vide ...
        buf3.nb ← j3 ;
        EcrireDir( H, i3, buf3 ) ;
        Aff-Entete( H, 1, i3 ) ;  // le numéro du dernier bloc utilisé dans H
   SINON         // sinon le fichier H restera vide ...
        Aff-Entete( H, 1, -1 ) ;  // valeur spéciale pour dire que le fichier est vide
   FSI ;
```

```
    Fermer( F ) ;
    Fermer( G ) ;
    Fermer( H ) ;
```

Fin // (Intersection)

2/ Coût de l'algorithme

Cet algorithme tire avantage que les 2 fichiers en entrée (F et G) sont déjà triés.
Donc pour réaliser l'intersection de F et G il suffit de parcourir, au plus, une seule fois les blocs de F et une seule fois les blocs de G.
Posons N1 le nombre de blocs du fichier F et posons N2 le nombre de blocs du fichier G.
Le nombre de lectures physiques totale dans le pire cas, est donc : N1 + N2.
Pour le nombre d'écritures physiques, le pire cas est atteint lorsque tous les éléments de l'un des fichiers en entrée se retrouvent dans l'autre (c-a-d lorsque $F \subseteq G$ ou $G \subseteq F$). Dans ce cas le fichier résultats sera formé par le même nombre de blocs que le plus petit des fichiers en entrée.
Le nombre d'écritures physiques est donc : min(N1,N2).
Le coût total de l'opération d'intersection est donc = **N1 + N2 + min(N1,N2)** E/S physiques
Si F et G sont de même taille (N blocs chacun), la complexité sera alors O(N).

Exercice 4

Dans la méthode du hachage dynamique « Hachage Linéaire », l'éclatement de la case n provoque le « rehachage » de toutes les clés du bloc n et éventuellement de sa liste de débordement, en utilisant la fonction $h_{i+1}(x) = x \bmod 2^{i+1}$. Deux fichiers (F et S) sont utilisés pour sa mise en œuvre.

- La zone principale est un fichier (F) de type TOF. Ses caractéristiques sont :
  - i : le niveau du fichier.
  - n : le numéro de la prochaine case à éclater.
  - Nenr : le nombre d'enregistrements total dans F et S.
- La zone de débordement est un autre fichier (S) renfermant les différentes listes de débordements (LOF). Ses caractéristiques sont :
  - Nbl : le nombre de blocs utilisés dans S.
  - Tête : la tête de liste des blocs vides, pour la récupération des blocs libérés.

4.1/ Donnez la définition des structures

Les 2 fichiers F et S partagent le même type de blocs

```
Type TBloc = Structure
        tab : tableau[b] de Tenreg ;        // tableau d'enregistrements qlq…
        nb : Entier ;                       // nombre d'enregistrements dans tab
        lien : Entier ;                     // numéro du bloc suivant dans S
fin // TBloc
```

Déclaration globale des fichiers :

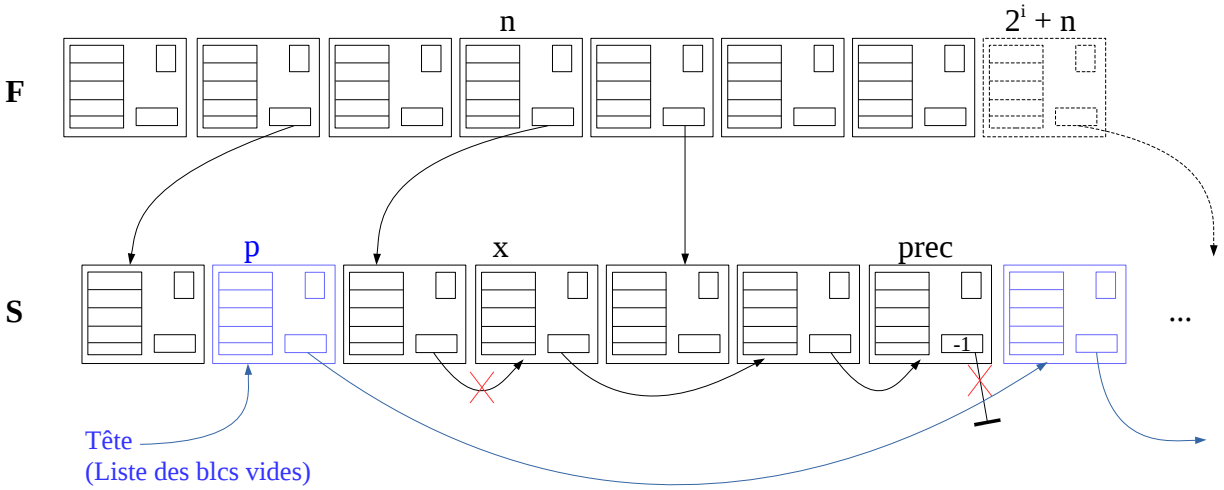
```
var
        F:Fichier de TBloc Entete (entier /* le niveau i */, entier /* n */, entier /* Nenr */) ;
        S:Fichier de TBloc Entete (entier /* Nbl */, entier /* Tête */) ;
        buf0, buf1, buf2 : TBloc ;
```

4.2/ Algorithme d'éclatement

```
Début  // (Eclatement)

    Soit L une file d'entiers ;
    // On l'utilisera pour sauvegarder (en vue de les réutiliser) les numéros de blocs de la liste de
    // débordement associée à n
    i ← Entete( F, 1) ;   n ← Entete( F, 2 ) ;
    CreerFile(  L ) ;
    i0 ← n ; prec ← -1 ;
    i1 ← n ; j1 ← 0 ; buf1.nb ← 0 ;
    i2 ← 2^i+n ; j2 ← 0 ; buf2.nb ← 0 ;
    x ← -1 ;

    TQ ( i0 <> -1 )
        SI ( i0 = n )  LireDir( F, i0, buf0 )  SINON  LireDir( S, i0, buf0 )  FSI ;
        suiv ← buf0.lien ;
        Enfiler( L, suiv ) ;
        POUR  j=1, buf0.nb
                e ← buf0.tab[ j ] ;
                SI ( h_{i+1}(e.cle) = n )
                        j1 ← j1 + 1 ;
                        SI ( j1 > b )
                                Defiler( L, x ) ;    // réutiliser un ancien bloc de la liste de débordement
                                buf1.lien ← x ;
                                buf1.nb ← b ;      // ou (j1 – 1)
                                SI ( i1 = n ) EcrireDir( F, i1, buf1 )  SINON  EcrireDir( S, i1, buf1 )  FSI ;
                                i1 ← x ;
                                j1 ← 1 ;
                        FSI ;
                        buf1.tab[ j1 ] ← e ;
```

```
                SINON   // c-a-d ( h_{i+1}(e.cle) = 2^i+n )
                        j2 ← j2 + 1 ;
                        SI ( j2 > b )
                                // Allouer un nouveau ou réutiliser un bloc depuis la liste des blocs vides de S…
                                SI ( Entete( S, 2 ) = -1 )    // si (Tête = -1) :
                                        x ← AllocBloc( S ) ;        // allouer un nouveau bloc.
                                SINON                         // si (Tête <> -1) :
                                        x ← Entete( S, 2 ) ;        // récup. 1er elt de la liste des blocs vides.
                                FSI ;

                                buf2.lien ← x ;
                                buf2.nb ← b ;     // ou (j2 – 1)
                                SI ( i2 = 2^i+n ) EcrireDir( F, i2, buf2)  SINON  EcrireDir( S, i2, buf2 )  FSI ;

                                // MAJ de la Tête de liste des blocs vides...
                                LireDir( S, x, buf2 ) ;                 // en récupérant le champ 'lien'
                                Aff-Entete( S, 2, buf2.lien ) ;         // de l'ancien 1er elt de la liste.

                                i2 ← x ;
                                j2 ← 1 ;
                        FSI ;
                        buf2.tab[ j2 ] ← e
                FSI //  ( h_{i+1}(e.cle) = n )
        FP ;
        prec ← i0 ;
        i0 ← suiv
    FTQ ;  // ( i0 <> -1 )

    // Ecriture des derniers blocs …
    x ← buf1.lien ;
    buf1.lien ← -1 ;  buf1.nb ← j1 ;
    buf2.lien ← -1 ;  buf2.nb ← j2 ;
    SI ( i1 = n )  EcrireDir( F, i1, buf1 )  SINON   EcrireDir( S, i1, buf1  FSI ;
    SI ( i2 = 2^i+n )  EcrireDir( F, i2, buf2 )  SINON   EcrireDir( S, i2, buf2  FSI ;

    // MAJ de la liste des blocs vides de S en y rajoutant les blocs de la liste de débordement  de 'n', non réutilisés durant
    // l'éclatement …
    SI ( x <> -1 )
        buf0.lien ← Entete( S, 2 ) ;   // l'ancienne Tête de la liste des blocs vides
        EcrireDir( S, i0, buf0 ) ;
        Aff-Entete( S, 2, x ) ;          // la nouvelle Tête devient x (voir figure ci-dessous)
    FSI

Fin // (Eclatement)
```

L'espace mémoire utilisé = 3 buffers (buf0, buf1 et buf2) + une file d'entier de longueur égale au nombre de blocs de la liste de débordement associée au bloc principal 'n' (généralement $\ll$ 10, donc négligeable).



A la fin de l'opération d'éclatement, les blocs de la sous-liste commençant par 'x' et se terminant par 'prec' seront insérés en début de la liste des blocs vides. C-a-d la nouvelle 'tête' sera 'x' et le suivant de 'prec' sera l'ancien premier bloc de la liste ('p').