UNIVERSITE DES SCIENCES ET DE LA TECHNOLOGIE HOUARI BOUMEDIENE

FACULTE D'INFORMATIQUE

Ingénieur Informatique 2éme année

Module : Structures de fichier et Structures de Données

(SFSD)

Projet TP : Simulateur Simplifié d'un Système de Gestion de Fichiers (SGF)

Réalisé par :

NOM	PRENOM	MATRICULE	GROUPE
MILOUDI	Yacine	232331406303	2
BAATOUT	Mohamed Amine	232331396111	2
HATTAB	Hamza Riadh	232331394516	1
HAMROUN	Sami Abdelmalek	232331406007	2

PLAN DU RAPPORT

Chapitre 1 : Conception de notre Projet.

Chapitre 2 : Implémentation et Exécution.

Chapitre 3 : Conclusion.

- **Lien GitHub de Projet :** dilatcha/Projet-TP-Simulateur-SGF: Projet TP : Simulateur Simplifié d'un Système de Gestion de Fichiers (SGF)
- **Ou cliquez sur ce lien :** https://github.com/dilatcha/Projet-TP-Simulateur-SGF
- **Ou Télécharger le projet a partir de lien google drive:** https://drive.google.com/file/d/1EsXtnqZQ_UKM4Ow1GX3iJ8-ypSNP6o2m/view?usp=sharing

REMARQUE IMPORTANTE :

- Si vous voulez **compilez** le code source suivre les étapes de fichier **README.TXT.**
- Si vous voulez lancer le simulateur exécuter le programme **Simulateur Gestion de Fichiers.exe.**

Chapitre 1 : Conception de notre Projet

Dans le cadre de ce projet, nous avons pour objectif de concevoir un simulateur simplifié d'un système de gestion de fichiers (SGF), une composante essentielle des systèmes d'exploitation modernes. Ce simulateur vise à modéliser les principes fondamentaux liés à la gestion des fichiers, notamment l'organisation de la mémoire secondaire, la manipulation des fichiers de données, et la gestion des métadonnées. La conception s'appuie sur des structures adaptées pour représenter la mémoire secondaire, les fichiers, et leurs enregistrements, tout en intégrant des mécanismes efficaces pour gérer des opérations comme la création, la recherche, et la suppression. Ce chapitre explore les bases conceptuelles et les structures utilisées pour mettre en œuvre les fonctionnalités clés du simulateur, en fournissant une vue d'ensemble de notre approche méthodologique et de la logique sous-jacente.

1. La Mémoire Secondaire :

Toutes les opérations de notre système sont effectuées dans **la mémoire secondaire**. Par conséquent, la conception et la structuration de cette mémoire dans notre simulateur sont cruciales et revêtent une grande importance.

La mémoire secondaire est représentée comme un **fichier binaire** nommé *mémoire_secondaire.bin*, contenant des blocs numérotés de Bloc 0 à Bloc n, où n représente la taille totale en blocs. Cette représentation reflète fidèlement la réalité de la mémoire secondaire, qui se compose d'un ensemble de secteurs ou de blocs séparés par des inter-blocs ou inter-secteurs.

Alors, les opérations d'accès à la mémoire secondaire dans notre simulateur sont équivalentes à l'accès à ce fichier binaire. Voici la *Figure 1* qui illustre bien ce que nous avons dit.

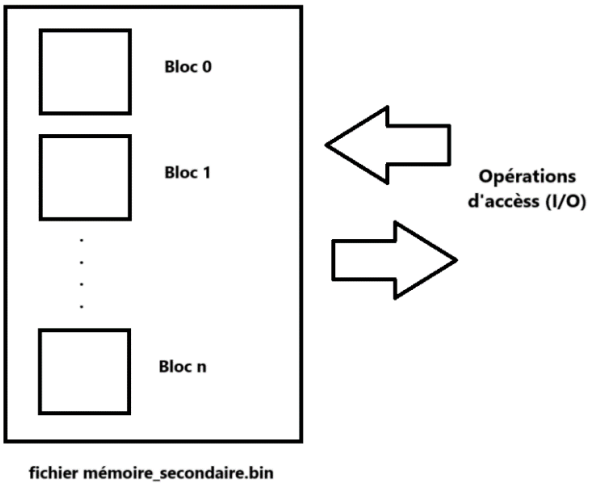


Figure 1 : Représentation de la mémoire secondaire.

- **La structure de Bloc :**

Comme on a vu dans le cours, chaque Bloc est représenté come un tableau d'enregistrements avec d'autres variables. **(Voir la structure de Bloc dans Chapitre 2 partie implémentation).**

La taille de ce tableau est définie par le Facteur de blocage FB qui indique le nombre d'enregistrement dans un seule Bloc.

- **La structure d'enregistrement**

Puisque dans le concept de base, le caractère est la plus petite unité d'information quand peut lire et écrire, on a choisi de représenter un enregistrement come un tableau de caractères. **(Voir la structure de Bloc dans Chapitre 2 partie implémentation).**

Cette approche nous offre la flexibilité et la possibilité de traiter plusieurs types de fichiers, quel que soit le type de variable qu'ils contiennent (entier, float, double, etc.), car toutes ces variables peuvent être stockées dans le tableau de caractères en utilisant l'instruction memcpy().

2. La Table d'Allocation :

La table d'allocation est cruciale dans notre simulateur. Elle permet de suivre l'état de chaque bloc de la mémoire secondaire, et de nombreuses fonctions en dépendent, telles que l'allocation d'un fichier, le compactage, le décalage inter-blocs, etc.

Étant donné que la table d'allocation doit être sauvegardée dans le Bloc 0, et que ce dernier contient des enregistrements constitués de tableaux de caractères de taille fixe, nous avons constaté qu'un seul enregistrement ne peut pas suivre l'état de tous les blocs de la mémoire secondaire. Cela nous a conduits à adopter l'implémentation suivante :

L'état d'un **Bloc i** est représenté par le caractère **1 OU 0** tell que **1** indique le bloc et occupé et **0** indique qu'il est libre

On sauvegarde l'état de Bloc i dans Bloc 0 **->** enregistrement[j] **->**data[k] tell que :

- **j = i DIV Taille(data)**
- **k = i MOD Taille(data)**

Avec cette technique-là, pour savoir si le Bloc est occupé ou non, il faut juste avoir l'adresse de ce Bloc la (qui est i dans notre exemple) et faire le calcule(voir la **Figure 2**).

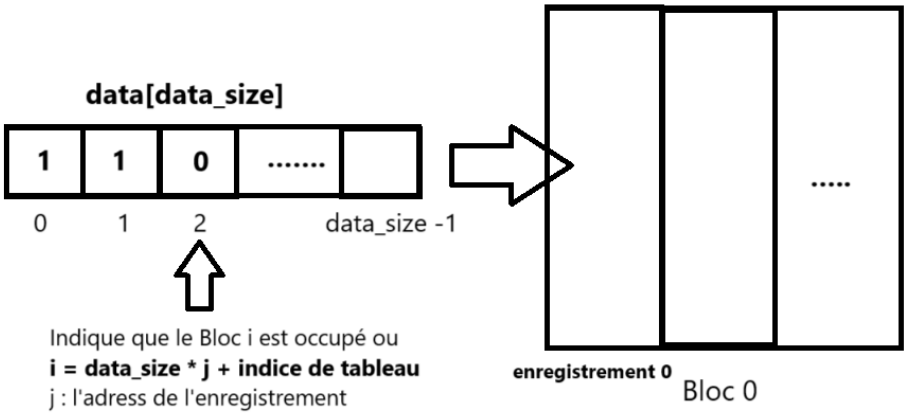


Figure 2. Représentation de la Table d'Allocation

3. Le Fichier de Données :

Le fichier de données est représenté comme un ensemble de blocs dans le fichier binaire avec une certaine organisation (tableau ou non-tableau (chaînée), ordonné ou non, taille fixe), il doit aux minimums occupe 1 seule bloc dans le fichier binaire (MS), ça veut dire il contient au moins 1 seule enregistrement.

4. Le Ficher de Métadonnées :

Tout d'abord, on veut parler comment on sauvegarde les métadonnées d'un fichier de données quelconque dans le fichier de MS.

On a vu que métadonnée d'un seul fichier contient :

- Nom du fichier.
- Taille du fichier en blocs.
- Taille du fichier en enregistrements.
- Adresse du premier bloc.
- Modes d'organisation globale.
- Modes d'organisation interne.

Ces variables, dans l'organisation à taille fixe, occupent toujours le même espace. Par exemple, « l'adresse du premier bloc » (représentée comme un entier) occupe toujours 4 octets, ce qui équivaut à occuper un espace de 4 caractères dans le

tableau de caractères pour chaque enregistrement. La même chose pour les autres variables.

Ainsi, si l'on choisit la bonne taille de tableau de caractères pour stocker les métadonnées d'un seul fichier, on peut stocker les métadonnées d'un fichier dans un seul enregistrement dans un bloc. Cela implique que l'on peut stocker les métadonnées de n fichiers, où n = FB dans un seule Bloc.

Cette implémentation nous aide à éliminer le gaspillage de l'espace mémoire, plutôt que de réserver le bloc entièrement pour les métadonnées d'un seul fichier, qui n'occupe même pas un seul enregistrement dans le bloc.

De plus, on a vu que on peut dans un seul enregistrement d'un Bloc stocker 2 fichiers de métadonnées de deux fichiers, ça implique que dans un seule Bloc on peut sauvegarder 2*FB fichiers de métadonnées. (Voir Figure 3).

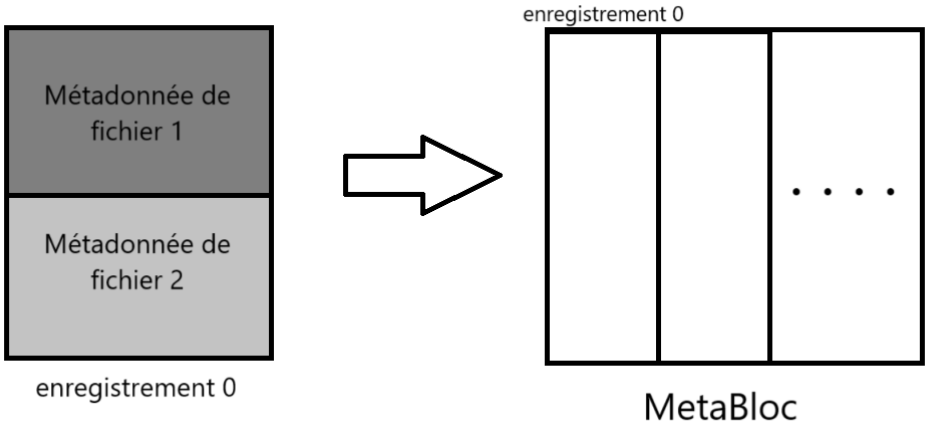


Figure 3: Représentation de Métadonnée dans un Bloc.

Avec ce concept, lors de l'initialisation du disque (fichier de mémoire secondaire), le Bloc 0 est réservé pour la table d'allocation, et **m** blocs sont utilisés comme Méta-Blocs, où **m** est calculé comme suit :

$$m = (NbrBlocs - 1) / (FB * nbrEMeta)$$

- **NbrBlocs** : Nombre total de blocs dans la mémoire secondaire
- **FB** : Facteur de blocage
- **nbrEMeta** : Nombre d'enregistrements de métadonnées pouvant être sauvegardés dans un seul bloc

REMARQUE : On choisir **nbrEMeta** aux dépend de la taille de tableau de caractères d'un enregistrement.

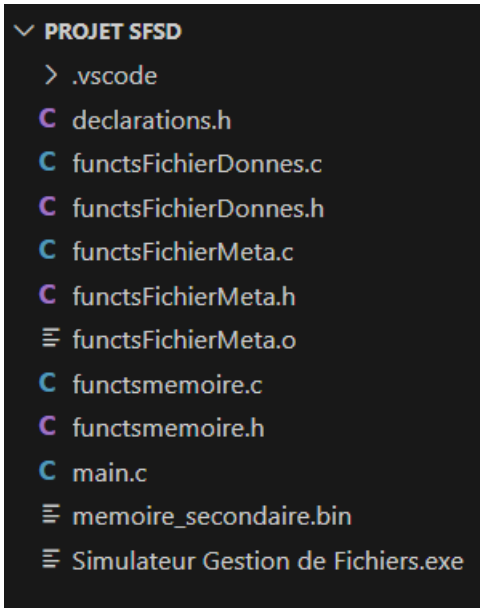
Exemple : Si la mémoire secondaire contient 256 blocs, avec le Bloc 0 réservé pour la table d'allocation, il reste 255 blocs libres. En supposant que FB = 6 et nbrEMeta = 2, on calcule 255 / 12 ≈ 21 Ainsi, 21 Méta-Blocs sont réservés dans la mémoire secondaire. Au total, 22 blocs (1 pour la table d'allocation et 21 pour les Méta-Blocs) sont réservés, ce qui laisse **234 blocs** disponibles pour les fichiers de données.

Chapitre 2 : Implémentation et Exécution

I. Partie 1 : Implémentation

❖ Organisation de code source du projet :

Avant de parler des structures et des fonctions du projet, **l'organisation du code source** est cruciale pour une meilleure lisibilité et une meilleure compréhension du code. Ainsi, nous avons **divisé le code source** en différentes parties (fichiers d'en-tête (header) .h et fichiers source .c), de sorte que chaque fichier d'en-tête et son fichier source associé contiennent les fonctions et structures liées à une partie spécifique de notre projet (voir Figure 4).



- **declaration.h :** contient les structures élémentaire de simulateur come la structure de la mémoire secondaire, l'enregistrement, le bloc, …etc.
- **functsFichierDonnes.c :** contient toutes les fonctions qui ont une relation avec fichier de donnée come AllouerFichier(), SupprimerFichier …etc .
- **functsFichierMeta.c :** contient toutes les fonctions qui ont une relation avec fichier de méta donnée come CreeMeta(),SupprimerMeta(), …etc.
- **functsmemoire.c :** contient les fonctions qui ont une relation directe avec la mémoire secondaire come initialiserDisk(),ViderMS() , …etc.

Figure 4. Organisation de code source

Et maintenant on parle sur l'implémentation des structures du projet :

1. Mémoire secondaire :

```
2.
3. typedef struct Memoire_Secondaire{
4.
5.     FILE *memoireS;
6.
7.     int nbrBlocs; //cette variable indique le nombre de Blocs maximale
   de la memoire secrondaire.
8.
9.     int bloc_size; //nombre d'enregistrements maximale dans un bloc
10.
11.    double TailleMemoire;
12.
13.}MS;
14.
```

La structure **Memoire_Secondaire** sert à simuler une mémoire secondaire. Elle comprend :

- **memoireS** : un pointeur vers le fichier binaire **« memoire_secondaire.bin »** représentant la mémoire.
- **nbrBlocs** : le nombre maximal de blocs.
- **bloc_size** : le nombre maximal d'enregistrements par bloc.
- **TailleMemoire** : la taille totale de la mémoire secondaire

Cette structure facilite la gestion de la mémoire dans la simulation.

Chaque fonction qui nécessite l'accès a la mémoire secondaire elle passe MS MemoireS comme un paramètre puis elle accès le fichier binaire MemoireS.memoireS.

- Les fonctions implémenter pour la Mémoire Secondaire :

➢ **void InitialiseDisk(MS* MemoireS)** : cette fonction est utilisé si on veut initialiser la mémoire secondaire à nouveau , elle calcule les nombres de MetaBloc pour les réserve , elle initialise la table d'allocation et initialiser toutes les Blocs.

➢ **void vider_MS(MS* ms)** : cette fonction est utilisé pour vider la mémoire secondaire et elle utilise aussi la fonction InitialiseDisk.

➢ **bool GES_Creation(MetaD *fichierinfo,Bloc *Bloc0)** : cette fonction vérifier avant chaque allocation d'un fichier si il existe un espace libre , si oui elle modifié la Métadonnée de fichier précisément adresse de premier bloc.

- **bool GES_insertion(MS MemoireS,enregistrement Enreg,char nom[50])** : la meme chose que GES_Creation, mais pour insérer un enregistrement dans un fichier de donné, au plus elle insère l'enregistrement automatiquement si il existe un espace libre.

- **void AfficherEtatMemoire(MS* MemoireS) :** La fonction AfficherEtatMemoire affiche l'état de la mémoire secondaire en parcourant tous les blocs à partir du bloc 22. Pour chaque bloc, elle indique s'il est occupé (avec le nombre d'enregistrements) ou libre, en utilisant des couleurs pour la mise en forme. Après un certain nombre de blocs, elle ajoute une nouvelle ligne pour faciliter la lecture. Les blocs sont affichés avec un nom incrémenté pour chaque bloc occupé.

2. Fichier de Donné :

Au début on a utilisé ces définitions :

```
//déclaration des constants nécessaire:

#define FB 6 //facteur de blocage

#define NbrBlocs 256 //nombre de blocs maximale pour la memoire secrondaire.

#define data_size 140 /*chaque enregistrement contient un tableau de
caractéres pour stocker des infromations multiples,
    le data_size représente la taille maximale de ce tableau*/

#define nbrMeta_enreg 2 /*nombre de fichiers on peut représenter ses
métadonnées dans un seule enregistrement.
ça veut dire on peut représenter des métadonnées de 2 fichiers dans un seule
enregistrement de bloc réservé pour les métadonnées.*/
```

Puis on a défini les structures d'Enregistrement, Bloc et Métadonnée qui sont les structures de base pour un fichier de données :

```
typedef struct enregistrement{

    int id;

    char data[data_size];

    bool isDeleted; //ce variable est utilisé pour la suppression logique des
enregistrements.
}enregistrement;
```

Le id est très essentiale dans l'organisation ordonné d'un fichier, et le variable booléen isDeleted est utilisé pour la suppression logique d'un enregistrement.

```
typedef struct Bloc{

    enregistrement enregistrements[FB];

    int nbrE; //le nombre d'enregistrements actuellement occupé dans un Bloc

    int nextBloc; /*un entier qui conserve l'index de prochaine Bloc dans le
cas
    d'organisation chainée*/

}Bloc, Buffer; //le buffer contient la même structure que un Bloc.

typedef struct Meta_Donnes{

    char nom[50];

    int taille_Blocs; //il prend 4 charachtéres dans data[data_size] d'un
l'enregistrement.

    int taille_Enrgs; //+4

    int adresse_PremierBloc; //+4

    int OrganisationE; //on représent L'Organisation Contigue par 1 et Chainée
par 0.

    int OrganisationI; //on représente L'Organisation Ordonné par 1 et
NonOrdonné par 0
}MetaD;
```

Voici que la métadonnée de l'importe quel fichier de donnée réserve exactement 70 Bytes, ce qui équivalent à 70 caractères dans le tableau de caractères data[data_size] d'un enregistrement.

Et puisque le data_size est défini a 140 et donc on peut stocker deux fichiers de métadonnée de cette structure-là dans un seul enregistrement.

```
typedef struct produit{ //exemplaire a utilisé

    char nom[27];

    double prix;

}produit;
```

La structure produit est un exemplaire d'un fichier de produit utilisé dans notre simulateur, on a utilisé juste cet exemplaire, mais notre simulateur est flexible pour plusieurs structures de fichier tant que la taille de la structure inférieure ou égale à le data_size (la taille de tableau de caractères).

- Les fonctions implémenter pour les fichiers de données :

- **MetaD CreeFichier()** :Cette fonction est utilisée pour créer un fichier de données. Elle retourne un objet de type MetaD, qui représente les métadonnées du fichier. Cette fonction permet de définir les informations de base du fichier, comme son nom, son emplacement, et les informations nécessaires à sa gestion dans la mémoire secondaire.

- **void AllouerFichier(MS MemoireS, MetaD fichierinfo, bool nouveau, Buffer* fichier)** : Cette fonction alloue de l'espace pour un fichier dans la mémoire secondaire. Elle prend en compte les métadonnées du fichier (MetaD fichierinfo) et détermine si le fichier est nouveau (nouveau). Si c'est le cas, elle alloue les blocs nécessaires dans la mémoire secondaire. Le paramètre Buffer* fichier permet de gérer l'espace mémoire alloué.

- **void SupprimerFichier(MS MemoireS, char nom[50])** : Cette fonction permet de supprimer un fichier de données en utilisant son nom (nom). Elle supprime le fichier de la mémoire secondaire en libérant l'espace qu'il occupait. Elle effectue également les ajustements nécessaires dans la table d'allocation et les métadonnées.

- **void LireFichier(MS MemoireS, char nom[50])** : Cette fonction permet de lire un fichier de données en utilisant son nom. Elle est utilisée pour lire un fichier de produits, par exemple. Elle permet d'accéder aux informations contenues

dans le fichier de données et de les afficher ou les manipuler, selon les besoins. **Remarque importante :** Cette fonction est spécifiquement utilisée pour lire un fichier de produits.

- **void RenommerFichier(MS MemoireS, char nom[50], char nouveauNom[50])** Cette fonction permet de renommer un fichier de données dans la mémoire secondaire. Elle prend le nom actuel du fichier (nom) et son nouveau nom (nouveauNom), puis modifie les métadonnées et la table d'allocation en conséquence.

- **void SuppressionLogique(MS MemoireS, char nom[50], int id) :** Cette fonction effectue une suppression logique d'un fichier de données, ce qui signifie qu'elle ne supprime pas réellement les blocs de données mais les marque comme inactifs ou supprimés dans la table d'allocation. Elle prend en compte le nom du fichier (nom) et un identifiant (id) pour localiser et marquer les données correspondantes pour la suppression logique.

- **void insererEnregistrement_Ordonne(Bloc fichierblocs[],int size,enregistrement enreg) :** La fonction insère un enregistrement dans un tableau de blocs de manière ordonnée en fonction de son ID. Elle recherche d'abord l'emplacement approprié en parcourant les blocs et les enregistrements. Si l'insertion se fait au début ou au milieu, les enregistrements suivants sont décalés, et les blocs sont ajustés si nécessaire. Si aucun emplacement intermédiaire n'est trouvé, l'enregistrement est ajouté à la fin. Enfin, le compteur d'enregistrements est mis à jour, et un message confirme le succès de l'insertion.

- **bool Recherche_Enregistrement(MS MemoireS,char nom[50],int id,int* adressBloc,int*adressEnreg) :** La fonction Recherche_Enregistrement recherche un enregistrement dans une mémoire secondaire (structure MS) en fonction de son ID et de son nom de fichier. Si l'enregistrement est trouvé, elle affiche ses détails (ID, nom du produit, et prix) et retourne true. De plus, elle fournit les adresses du bloc et de l'enregistrement dans les paramètres adressBloc et adressEnreg, si ces derniers ne sont pas NULL.

- **void SuppressionLogique(MS MemoireS,char nom[50],int id) :** La fonction marque logiquement un enregistrement comme supprimé dans la mémoire secondaire sans le retirer physiquement des blocs, elle le recherche par id.

- **void SuppressionPhysqiue(MS MemoireS,char nom[50],int id) :** La fonction SuppressionPhysique supprime un enregistrement de la mémoire secondaire en réorganisant les blocs pour combler l'espace vide, puis met à jour les structures et libère l'espace occupé.

3. Fichier de métadonnée :

Comme on a dit précédemment, notre implémentation donne la possibilité de sauvegarder deux fichiers de métadonnée dans un seule enregistrement d'un MetaBloc, cette implémentation conserve la structure d'un Bloc et évite le gaspillage d'espace mémoire aux maximums.

Mais, pour gérer cette implémentation on a fait des fonctions qui passe come paramètre le **index**, **niveau** et **adresse** et d'autres qui donne ces variables-là, car ils sont cruciaux dans les opérations sur les métadonnées d'un fichier de donnée, voici la *Figure 5* qui explique bien cette approche-là.

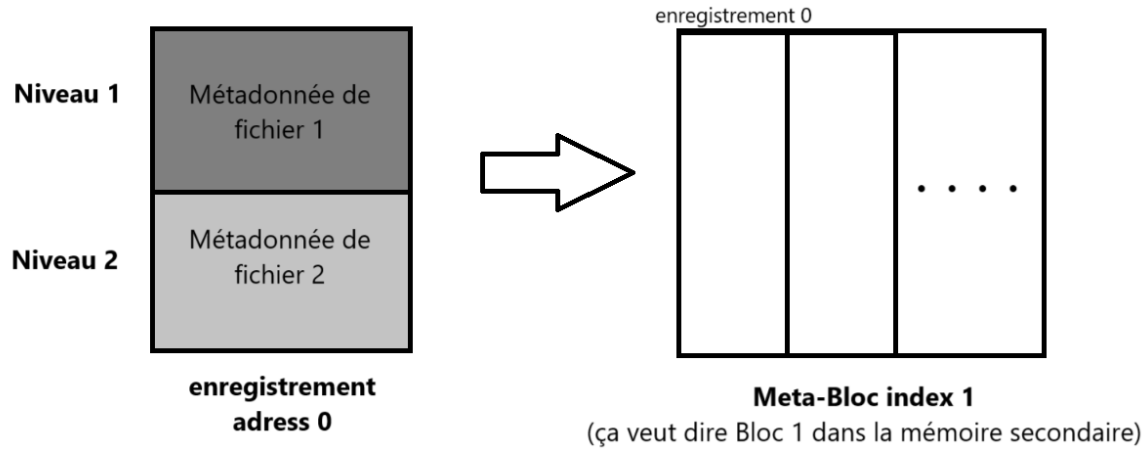


Figure 5 : Implémentation de métadonnée sur MétaBloc

Et alors, pour accéder par exemple à métadonnée de fichier 1 il faut aller a le Bloc d'index 1 (Bloc 1), enregistrement [0] et niveau 2 de data[data_size], en réalité le niveau 2 de data[data_size] est data[data_size] + 70.

- Les fonctions implémenter pour les fichiers de métadonnée :

- **void EcrireMetaD_Enreg(Bloc MetaBloc, MetaD fichierinfo, int niveau, int adress) :**
  Cette fonction permet d'écrire les métadonnées d'un fichier dans un bloc de métadonnées (MetaBloc). Elle prend en compte le niveau et l'adresse pour localiser l'endroit où les métadonnées doivent être enregistrées. Cela permet de sauvegarder les informations associées à un fichier à des emplacements précis dans la mémoire secondaire.

- **MetaD LireMetaD_Enreg(Buffer MetaBloc, int niveau, int adress) :** Cette fonction permet de lire les métadonnées d'un bloc de métadonnées (MetaBloc) en utilisant le niveau et l'adresse spécifiés. Elle récupère les

informations du fichier à partir du bloc et renvoie un objet MetaD qui contient les métadonnées associées.

- **void Sauvegarder_MetaD(MS MemoireS, MetaD fichierinfo, Buffer MetaBloc) :**
  Cette fonction permet de sauvegarder les métadonnées d'un fichier dans la mémoire secondaire. Elle prend en entrée les informations de métadonnées (MetaD fichierinfo) et le bloc de mémoire (Buffer MetaBloc), puis elle écrit les données dans la mémoire secondaire, garantissant leur persistance.

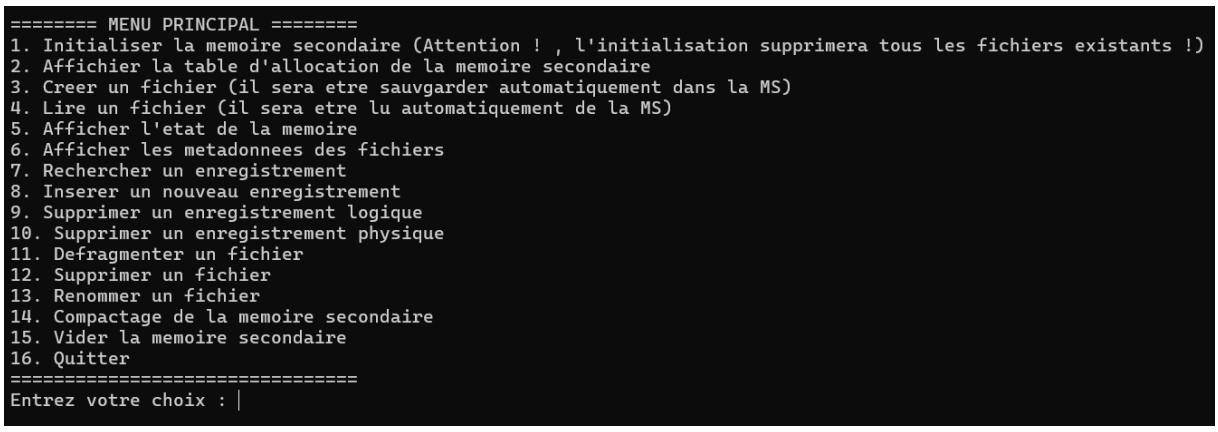
- **void SupprimerMetaD(Bloc* MetaBloc, int niveau, int adress) :** Cette fonction permet de supprimer les métadonnées d'un fichier dans un bloc de métadonnées spécifique. Elle prend le niveau et l'adresse pour localiser la métadonnée à supprimer, puis elle efface les données associées à cet emplacement, libérant ainsi l'espace dans le bloc de mémoire.

- **MetaD RechercheMetaD(MS MemoireS, char nom[50], bool* trouver, int* index, int* adress, int* niveau)** : Cette fonction permet de rechercher les métadonnées d'un fichier en utilisant son nom. Elle renvoie un objet MetaD contenant les informations du fichier si celui-ci est trouvé. En plus de cela, elle met à jour les variables trouver, index, adress et niveau pour indiquer où les métadonnées sont situées dans la mémoire secondaire.

II. Partie 2 : Exécution

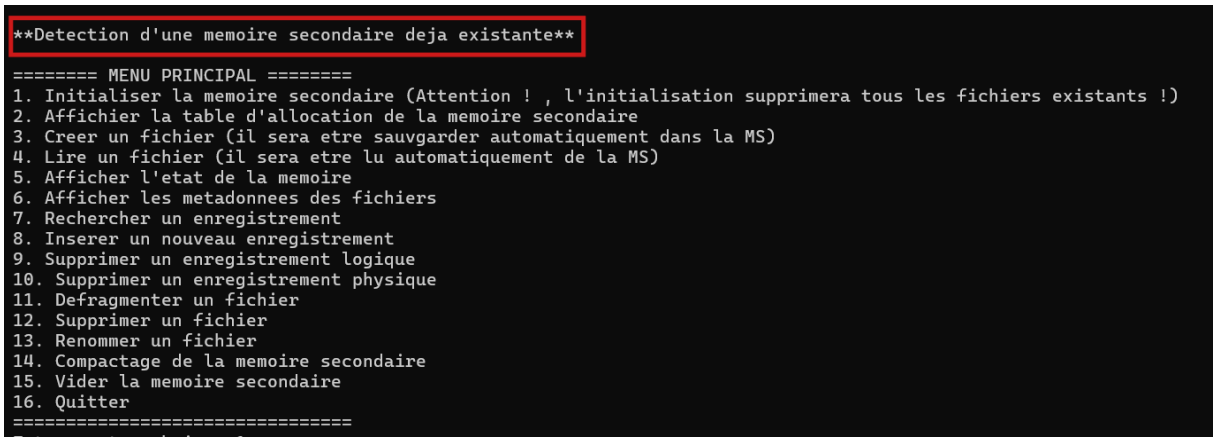
❖ Affichage du Menu Principal :

Lors de l'exécution du programme, le menu principal s'affiche pour permettre à l'utilisateur de sélectionner l'une des fonctionnalités disponibles. Voici un exemple de l'interface textuelle du menu principal :



Fonctionnalité principale : Ce menu sert de point d'entrée pour l'utilisateur. En saisissant un numéro correspondant à une option, le programme exécute la fonctionnalité associée.

REMARQUE : Notre Simulateur détecte si la mémoire secondaire **« memoire_secondaire.bin »** déjà existe, il vérifie si le fichier binaire a été déjà créé dans le répertoire principale du programme. S'il le trouve il va afficher un message que une mémoire déjà existe dans le simulateur :



Option 1. Initialiser la mémoire secondaire :

Cette fonction est utilisée pour réinitialiser la mémoire secondaire. Après avoir effectué les opérations nécessaires, elle sauvegarde l'état et les données de la mémoire secondaire dans un fichier binaire.
Si un fichier binaire contenant la mémoire secondaire existe déjà, cette fonction n'est pas appelée ; le programme lit simplement le fichier binaire et alloue l'espace mémoire correspondant à la mémoire secondaire sauvegardée.

Fonctionnalité principale :
Réinitialiser la mémoire secondaire, en supprimant toutes les données existantes, et sauvegarder l'état initial dans un fichier binaire.

```
======== MENU PRINCIPAL ========
1. Initialiser la memoire secondaire (Attention ! , l'initialisation supprimera tous les fichiers existants !)
2. Affichier la table d'allocation de la memoire secondaire
3. Creer un fichier (il sera etre sauvgarder automatiquement dans la MS)
4. Lire un fichier (il sera etre lu automatiquement de la MS)
5. Afficher l'etat de la memoire
6. Afficher les metadonnees des fichiers
7. Rechercher un enregistrement
8. Inserer un nouveau enregistrement
9. Supprimer un enregistrement logique
10. Supprimer un enregistrement physique
11. Defragmenter un fichier
12. Supprimer un fichier
13. Renommer un fichier
14. Compactage de la memoire secondaire
15. Vider la memoire secondaire
16. Quitter
================================
Entrez votre choix : 1

Detection d'une memoire secondaire deja existante ! Etes-vous sur de vouloir l'initialiser a nouveau ?
Saisissez n'importe quel caractere pour continuer, sinon fermez le programme !
d

Initialisation du disk avec succes.
Saisissez n'importe quel caractere pour continuer...
d
```

Option 2. Affichier la Table d'Allocation :

Cette fonction est utilisée pour affichier l'état de la table d'allocation qui se trouve dans le Bloc 0 de la mémoire secondaire.

```
 ** NOTE ** : Bloc etat: 1 --> Occupe \ Bloc etat:0 --> Libre

Bloc 0 etat: 1            Bloc 1 etat: 1            Bloc 2 etat: 1
Bloc 3 etat: 1            Bloc 4 etat: 1            Bloc 5 etat: 1
Bloc 6 etat: 1            Bloc 7 etat: 1            Bloc 8 etat: 1
Bloc 9 etat: 1            Bloc 10 etat: 1            Bloc 11 etat: 1
Bloc 12 etat: 1            Bloc 13 etat: 1            Bloc 14 etat: 1
Bloc 15 etat: 1            Bloc 16 etat: 1            Bloc 17 etat: 1
Bloc 18 etat: 1            Bloc 19 etat: 1            Bloc 20 etat: 1
Bloc 21 etat: 1            Bloc 22 etat: 0            Bloc 23 etat: 0
Bloc 24 etat: 0            Bloc 25 etat: 0            Bloc 26 etat: 0
Bloc 27 etat: 0            Bloc 28 etat: 0            Bloc 29 etat: 0
Bloc 30 etat: 0            Bloc 31 etat: 0            Bloc 32 etat: 0
Bloc 33 etat: 0            Bloc 34 etat: 0            Bloc 35 etat: 0
Bloc 36 etat: 0            Bloc 37 etat: 0            Bloc 38 etat: 0
Bloc 39 etat: 0            Bloc 40 etat: 0            Bloc 41 etat: 0
```

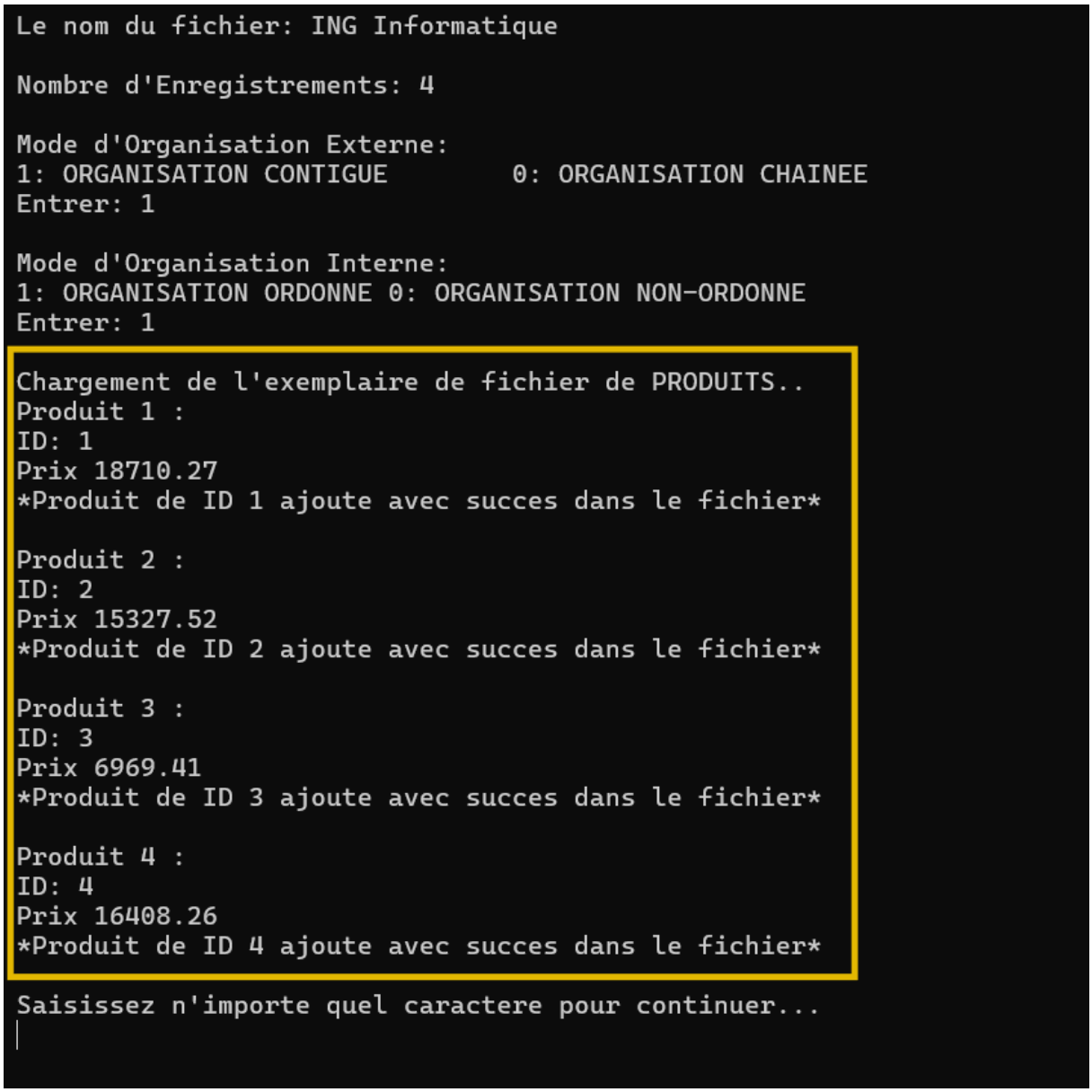
Option 3. Création d'un Fichier :

Cette fonctionnalité permet de créer un fichier qui sera automatiquement sauvegardé dans la mémoire secondaire. Lors de la création du fichier, l'utilisateur doit entrer des informations telles que le nom du fichier, le nombre d'enregistrements, ainsi que le mode d'organisation externe et interne.

Étapes de l'Exécution : (+exemplaires) :

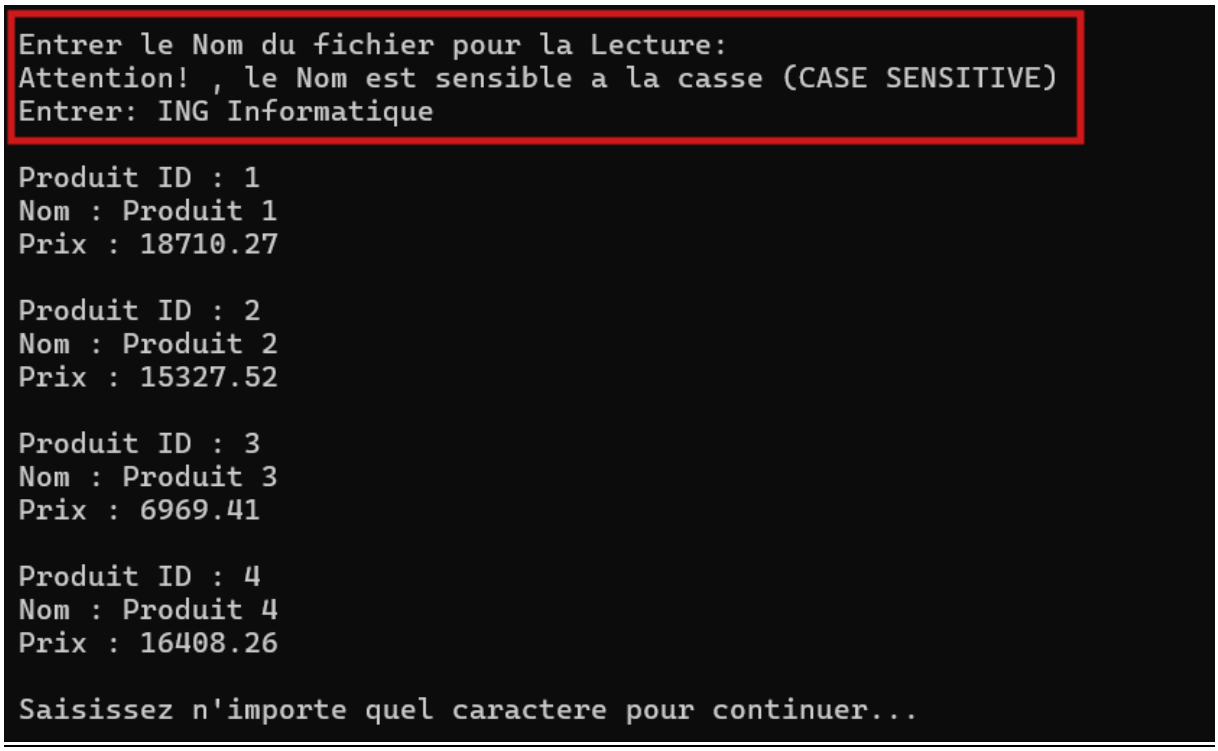
```
==============================
Entrez votre choix : 3

Le nom du fichier: ING Informatique

Nombre d'Enregistrements: 4
```

- **Saisie du nom du fichier**
- **Nombre d'enregistrements**
- **Mode d'organisation externe 1 → Contigue , 0 → Chainée**
- **Mode d'organisation interne 1→ Ordonnée , 0 → Non Ordonnée**
- **Chargement du fichier**
- **Insertion des produits (chargement d'exemplaire des produits automatiquement fait par le simulateur)**
- **Confirmation et retour au menu principal**
  Après l'insertion automatique des produits, l'utilisateur peut appuyer sur une touche pour continuer.


Le nom du fichier: ING Informatique
Nombre d'Enregistrements: 4
Mode d'Organisation Externe:
1: ORGANISATION CONTIGUE 0: ORGANISATION CHAINEE
Entrer: 1
Mode d'Organisation Interne:
1: ORGANISATION ORDONNE 0: ORGANISATION NON-ORDONNE
Entrer: 1
Chargement de l'exemplaire de fichier de PRODUITS..
Produit 1 :
ID: 1
Prix 18710.27
Produit de ID 1 ajoute avec succes dans le fichier
Produit 2 :
ID: 2
Prix 15327.52
Produit de ID 2 ajoute avec succes dans le fichier
Produit 3 :
ID: 3
Prix 6969.41
Produit de ID 3 ajoute avec succes dans le fichier
Produit 4 :
ID: 4
Prix 16408.26
Produit de ID 4 ajoute avec succes dans le fichier
Saisissez n'importe quel caractere pour continuer...

Option 4. Lecture d'un Fichier :

Pour lire un fichier, il faut d'abord saisie le nom du fichier (le nom est sensible a la casse), puis le simulateur va afficher les données (des produits) de ce fichier la.



Option 5. Affichage de l'État de la Mémoire Secondaire ;

Cette fonctionnalité affiche l'état actuel de la mémoire secondaire. Elle présente une vue de la mémoire où les blocs libres sont affichés en **vert** et les blocs occupés (non libres) en **rouge**, avec le nom du fichier et le nombre d'enregistrements qu'il

contient. Dans notre exemple, le fichier fichier1 a déjà été créé et il occupe actuellement 2 enregistrements

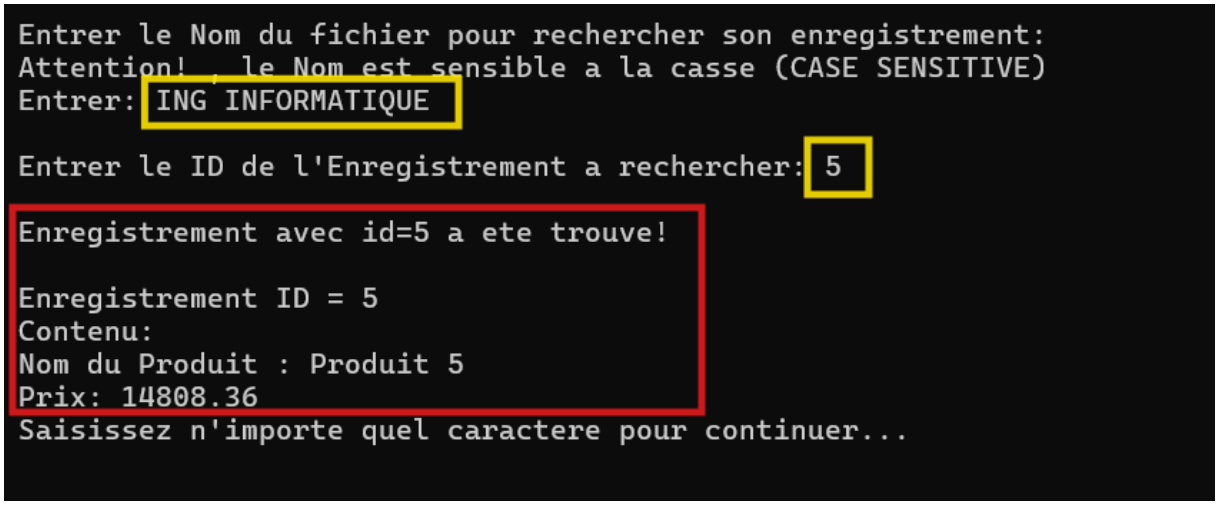
Option 6. Affichage des Métadonnées des Fichiers :

Dans cette étape, le programme affiche les métadonnées de tous les fichiers existants dans la mémoire secondaire (MS), telles que le nom du fichier, l'emplacement des blocs en mémoire, ainsi que les modes d'organisation externes et internes. Voici un exemple d'exécution :

```
Nom du fichier: ING INFORMATIQUE

Adress de Premier Bloc: 22
Nombres d'enregistrement occupe: 30
Nombres de Blocs occupe: 5
Mode d'Organisation Externe : Contigue
Mode d'Organisation Interne : Ordonne

Nom du fichier: Etudiants 2024

Adress de Premier Bloc: 27
Nombres d'enregistrement occupe: 25
Nombres de Blocs occupe: 5
Mode d'Organisation Externe : Chainee
Mode d'Organisation Interne : Non Ordonne

Nom du fichier: Fichier de Produits 2025

Adress de Premier Bloc: 32
Nombres d'enregistrement occupe: 15
Nombres de Blocs occupe: 3
Mode d'Organisation Externe : Contigue
Mode d'Organisation Interne : Non Ordonne

Saisissez n'importe quel caractere pour continuer...
```

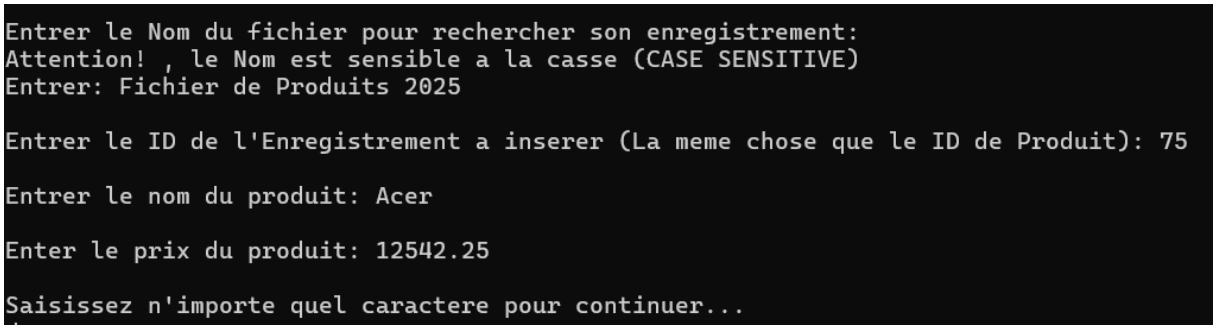
Option 7. Recherche d'un Enregistrement :

Dans cette étape, le programme permet à l'utilisateur de rechercher un enregistrement spécifique dans un fichier en entrant son nom (sensible à la casse) ainsi que l'ID de l'enregistrement. Si l'enregistrement existe, ses informations sont affichées. Voici un exemple d'exécution :



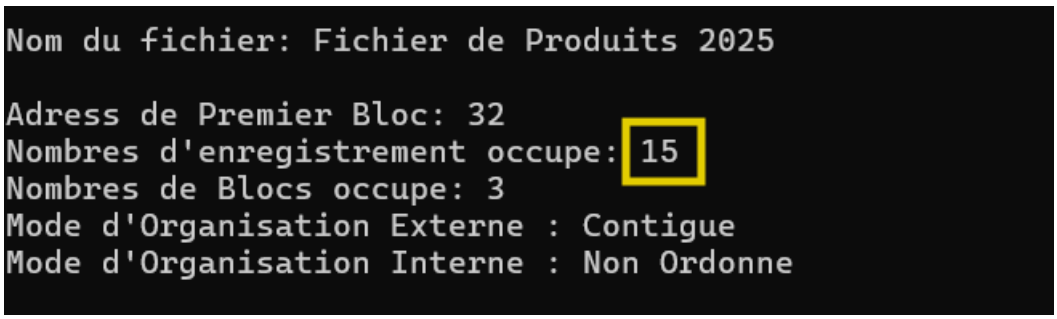
Option 8. Insertion d'un Nouveau Enregistrement

Cette étape permet à l'utilisateur d'ajouter un nouvel enregistrement à un fichier spécifique. L'utilisateur saisie le nom du fichier, l'ID de l'enregistrement à insérer, le nom du produit et son prix. Voici un exemple d'exécution :

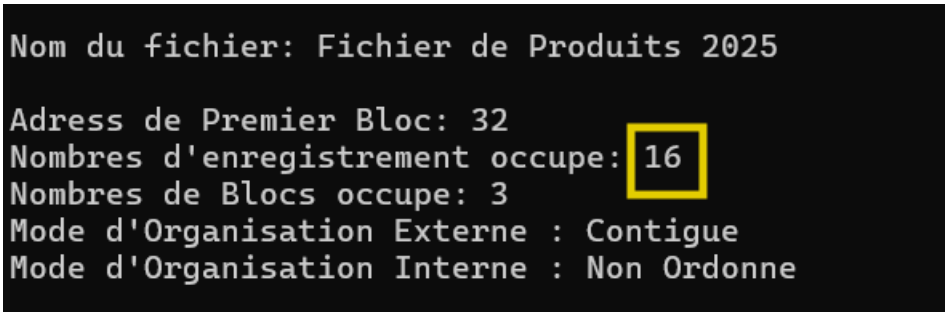


La métadonnée du fichier sera être mettre a jour :

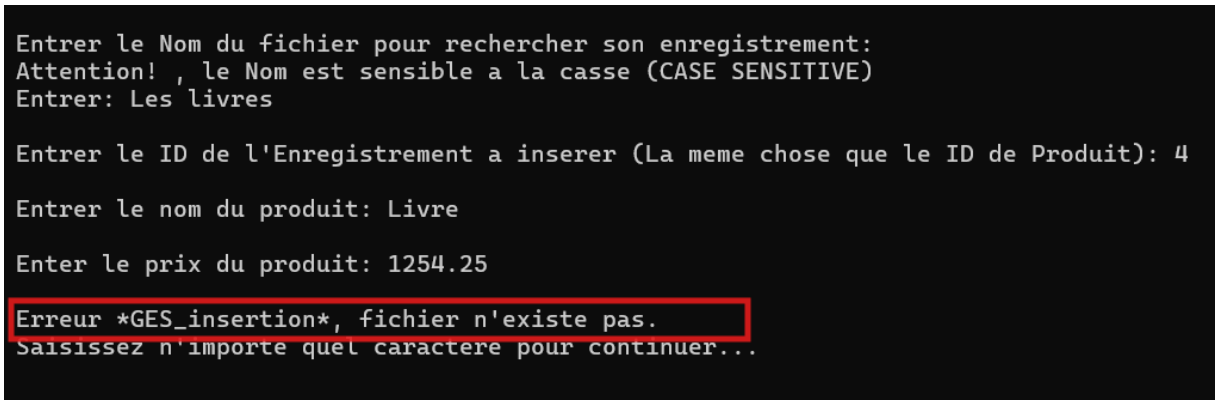
- Avant l'insertion :



- Après l’insertion :



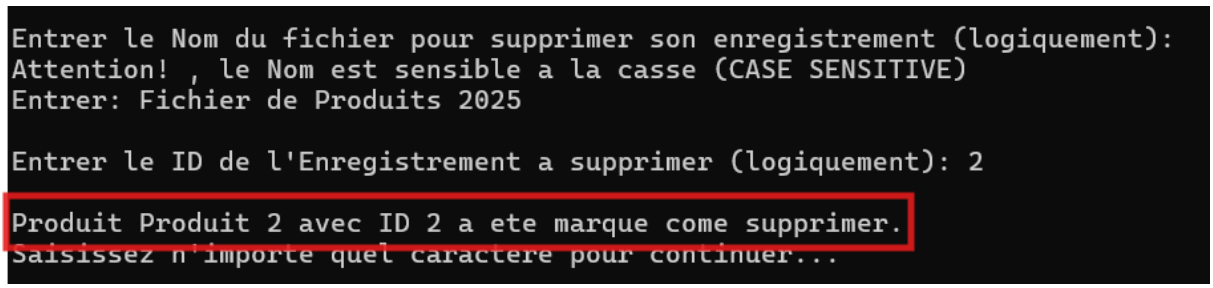
- Le cas le fichier n’existe pas :



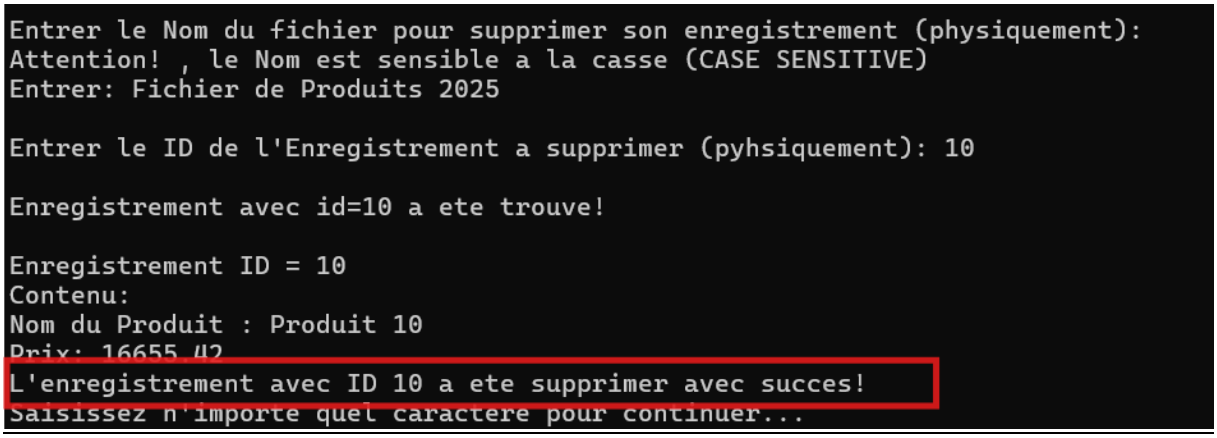
Option 9. Suppression logique d'un enregistrement

L'option 9 permet de supprimer un enregistrement de manière logique. Cela signifie que l'enregistrement est marqué comme supprimé sans être réellement retiré de la mémoire, ce qui permet de le récupérer ou de l'utiliser à nouveau plus tard.

Exemple d'exécution :



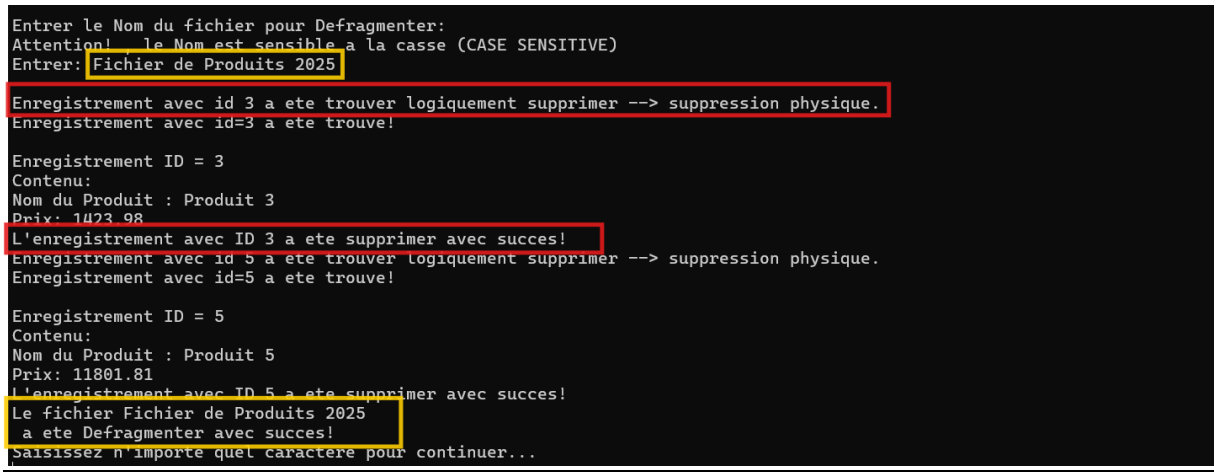
Option 10. Suppression physique d'un enregistrement :



Le Nombre d'enregistrement dans la métadonnée du fichier seront être mise a jour après cette opération.

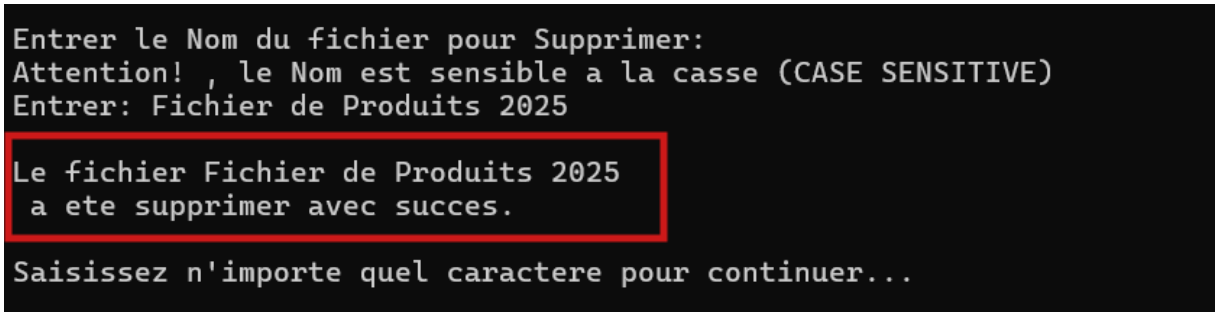
Option 11. Défragmentation d'un fichier :

La défragmentation sert a supprimer physiquement les enregistrements marqué come supprimer (logiquement) dans un fichier quel quand, elle utilise la fonction SuppressionPhysique(), voici un exemple dans le cas de **Fichier de Produits 2025,** on a marqué les enregistrement avec ID = 3 et ID = 5 come logiquement supprimer, après on a exécuter l'option 11 Défragmentation :



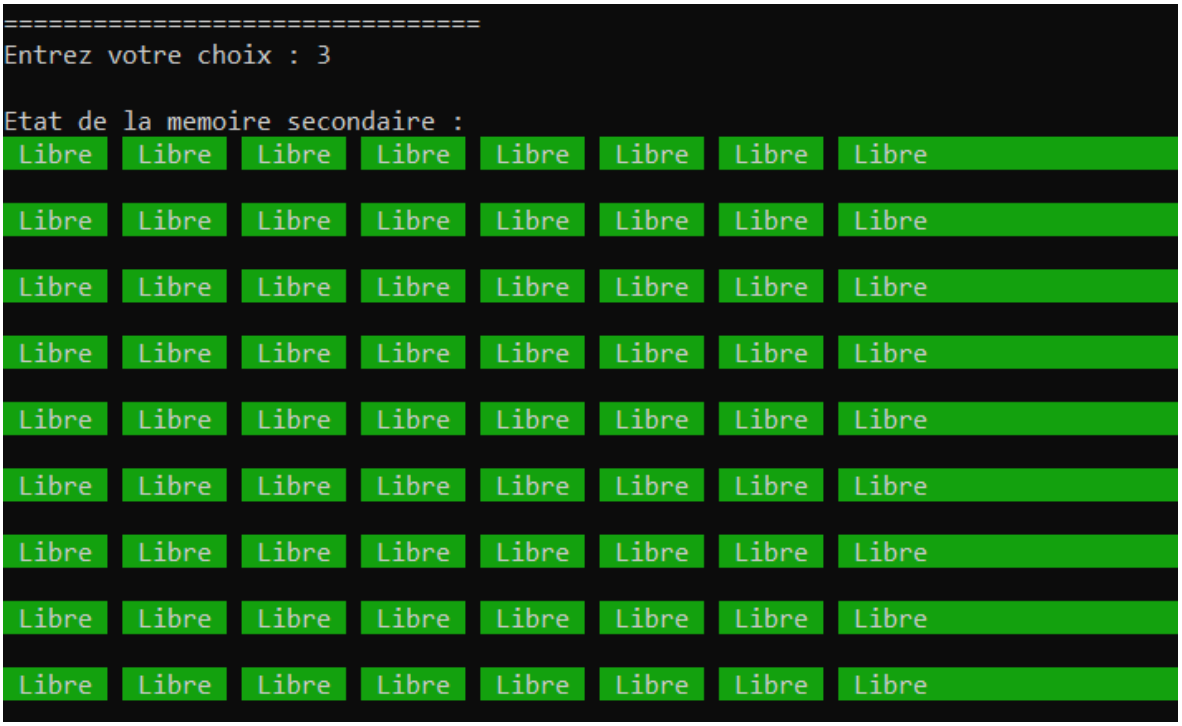
Option 12. Supprimer un fichier

Dans cette étape, nous avons utilisé l'option 12 pour supprimer un fichier existant dans la mémoire secondaire. Après avoir sélectionné l'option, le programme demande d'entrer le nom du fichier à supprimer.



Après la suppression, le fichier de métadonnées du fichier de données sera supprimé définitivement et l'espace qu'il occupait sera libéré.

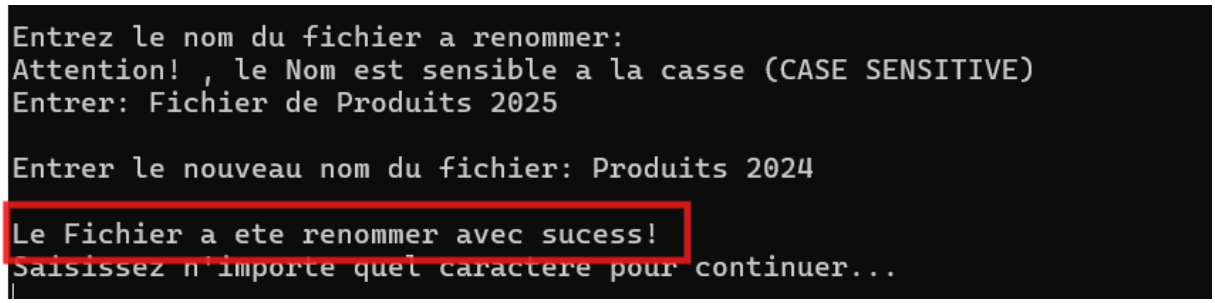
On remarque aussi que le fichier a été supprimer (tous les blocs sont vides)



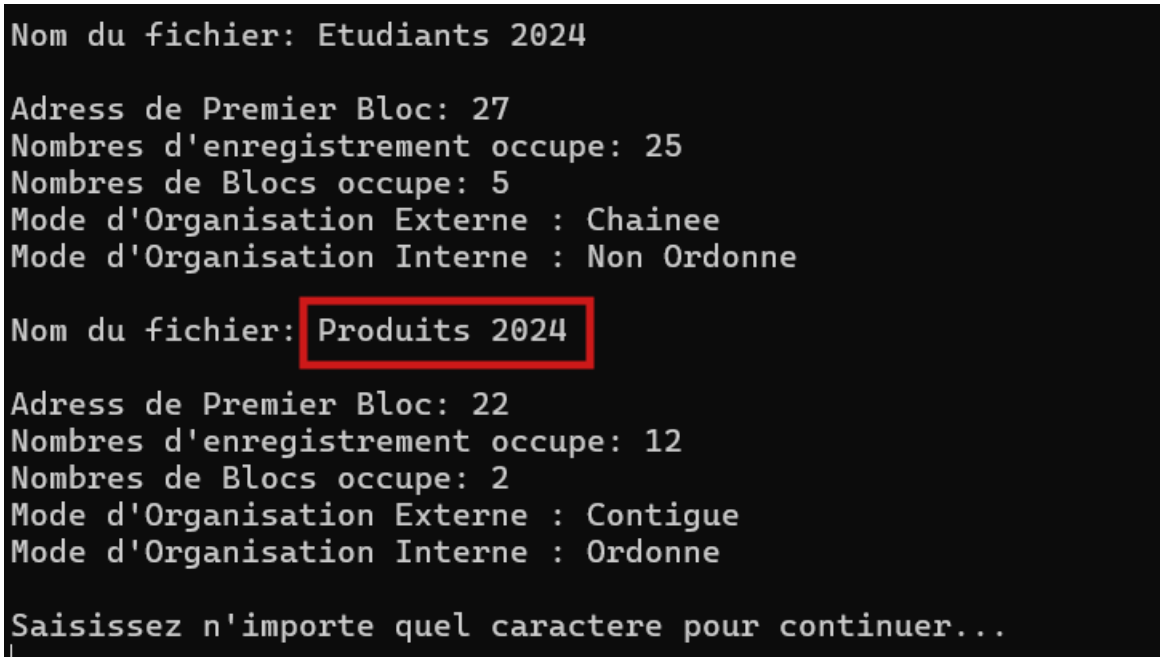
Option 13. Renommer un fichier

Dans cette étape, nous avons utilisé l'option 13 pour renommer un fichier existant dans la mémoire secondaire. Le programme demande d'abord d'entrer le nom du fichier à renommer, qui est sensible à la casse. Dans cet exemple, nous avons entré

"Fichier de Produits 2025", puis le programme nous demande d'indiquer le nouveau nom du fichier. Nous avons choisi "Produits 2024" comme nouveau nom.



Après avoir renommé le fichier, nous avons vérifié les métadonnées. Le fichier "fichierX" apparaît maintenant avec son nouveau nom.

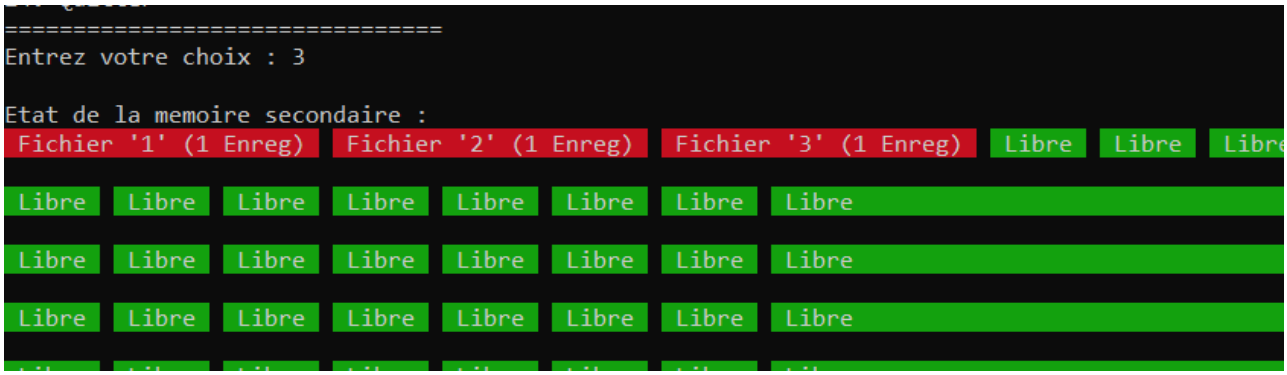


Option 13. Compactage de la mémoire secondaire

Le **compactage de la mémoire secondaire** consiste à réorganiser les blocs de mémoire afin de déplacer les fichiers et les enregistrements pour libérer de l'espace contigu.

Example

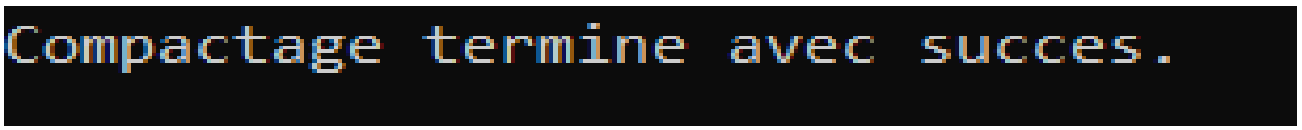
Suppose qu'on avait 3 fichiers (fichier 1, fichier 2, fichier3) de façon contigue :

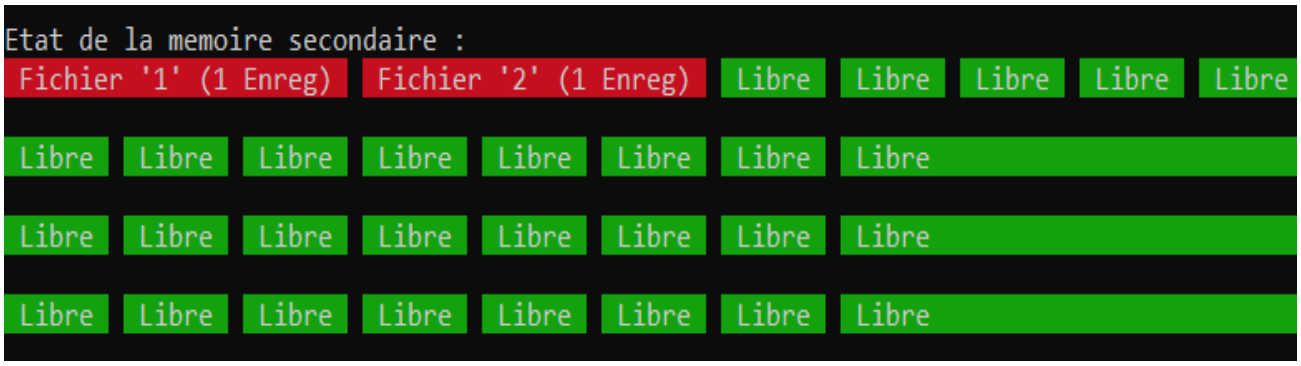


On va supprimer le fichier2 utilisant l'option 12 (donc on obtient un bloc vide entre fichier1 et fichier3)

On utilise l'option 13 pour faire compactage entre les blocs

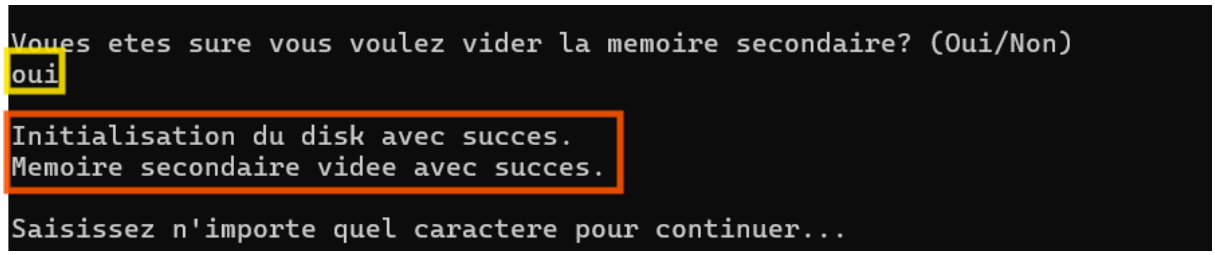
Donc on obtient ça après le compactage :



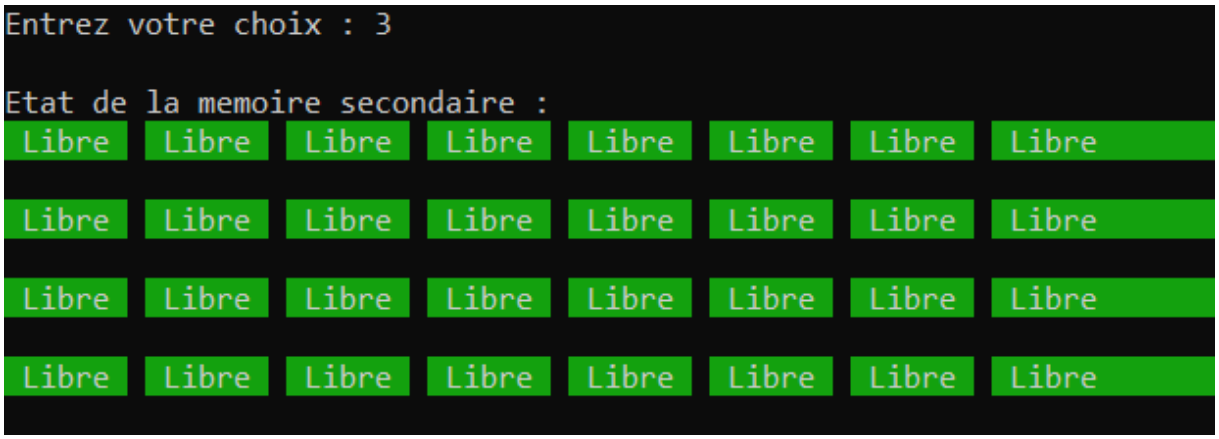


Option 15. Vider la mémoire secondaire :

L'option **15** permet de vider complètement la mémoire secondaire. Lorsque l'utilisateur choisit cette option, il est d'abord invité à confirmer l'action en répondant "Oui" ou "Non". Si la réponse est "Oui", la mémoire secondaire est réinitialisée, ce qui signifie que tous les fichiers et données stockés dans la mémoire sont supprimés.



Confirmer utilisant l'affichage d'état (l'option 5) :



Chapitre 3 : Conclusion

En conclusion, le projet de "Simulateur de Gestion de Fichiers" a permis de simuler la gestion des fichiers et de la mémoire secondaire à travers l'utilisation de structures simples, telles que des blocs et des enregistrements. Dans notre conception, chaque fichier est représenté par une série de blocs, et la mémoire secondaire est gérée à l'aide d'une table d'allocation qui suit l'état de chaque bloc (occupé ou libre). Cette table est essentielle pour l'allocation et la suppression des fichiers, garantissant ainsi une gestion efficace de la mémoire.

De plus, nous avons intégré un fichier de métadonnées pour chaque fichier, contenant des informations cruciales sur la structure du fichier, telles que la taille des blocs, le nombre d'enregistrements et l'adresse du premier bloc. Ce fichier permet d'organiser et de gérer les données de manière plus structurée, facilitant l'accès et la manipulation des fichiers dans la mémoire secondaire.

Cependant, plusieurs fonctionnalités pourraient être ajoutées pour améliorer ce simulateur dans le futur :

1. **Minimiser l'accès à la mémoire secondaire :**
   L'indexation des données à travers des structures telles que les arbres B permettrait d'optimiser les performances du simulateur en réduisant le nombre d'accès aux blocs de mémoire.
2. **Amélioration de l'interface utilisateur :**
   Actuellement en mode texte, une interface graphique moderne offrirait une expérience plus intuitive et accessible, facilitant l'interaction avec le simulateur, en particulier pour les utilisateurs non techniques.
3. **Optimisation de la gestion de la mémoire :**
   Il serait possible de mettre en place un mécanisme plus avancé pour le compactage des blocs ou la gestion dynamique des ressources mémoire afin de mieux gérer les fichiers de grande taille.

Ces améliorations contribueraient à rendre le simulateur plus performant et plus convivial, tout en le rendant plus adapté à des environnements de gestion de fichiers plus complexes.