

---

# BASE DE DONNÉES

## *Épreuve orale d'informatique du CCMP*

---

### Partie 1 – Manipulation de fichiers C

**Fichier de tables.** Dans cet exercice, nous allons implémenter diverses fonctions pour travailler sur des fichiers textes qui stockent des tableaux bidimensionnels où chaque case est une chaîne de caractères. Pour encoder un tableau bidimensionnel au format texte, on représente chaque ligne du tableau par une ligne de texte (deux lignes consécutives étant séparées par un caractère '**\n**') tandis que les colonnes sont séparées par des '**\t**'. Le caractère '**\t**' correspond à une tabulation qui est affichée dans le texte suivant par une espace large :

```
21191      Toy Story      1995      1481      30000000      361958736
21192      Toy Story 2    1999      1481      90000000      485015179
21193      Toy Story 3    2010      46062     200000000     \N
```

Ce texte correspond à la table suivante :

21191	Toy Story	1995	1481	30000000	361958736
21192	Toy Story 2	1999	1481	90000000	485015179
21193	Toy Story 3	2010	46062	200000000	NULL

Le format présenté correspond à une simplification du format CSV où le séparateur est un '**\t**' (aucune connaissance du format CSV n'est demandée). On fera les hypothèses simplificatrices suivantes sur le format des fichiers :

- aucune des chaînes de caractères dans les cellules ne contiendront de caractères '**\t**'
- toutes les lignes du fichier définiront le même nombre de colonnes et donc contiendront le même nombre de caractères '**\t**'
- aucune cellule ne contiendra de chaîne de caractères très longue (1000 caractères ou plus),
- les NULL sont représentés par la chaîne de caractères "**\N**".

**Lecture des fichiers de tables.** Pour vous aider dans l'implémentation on vous fournit le code suivant :

```
char donnees[1000]; // tableau pour stocker ce qui est lu

char lit_cellule (FILE * fichier) {
    char separateur ;
    if(fscanf(fichier, "%999[^\t\n]%c", donnees, &separateur) == EOF)
        separateur = '\0';
    return separateur ;
}
```

Cette fonction prend en paramètre un descripteur de fichier dans la variable `fichier`. Cette fonction lit une chaîne de caractères et s'arrête au premier `'\t'` ou `'\n'` qu'elle rencontre. Si la fonction ne trouve pas l'un de ces caractères dans les 1000 premiers caractères, elle s'arrête aussi.

La chaîne lue est stockée dans le tableau `donnees` et le caractère après la chaîne lue est renvoyé. Le caractère renvoyé est `'\0'` si le fichier a déjà été lu entièrement, `'\n'` si l'on vient de finir de lire la ligne et `'\t'` si on a rencontré un caractère de séparation entre deux colonnes. Si l'on essaie de lire une chaîne qui contient plus de 1000 caractères sans `'\t'` ou `'\n'`, alors `separateur` renverrait le 1000ème caractère qui pourra donc être une valeur quelconque mais on suppose que cela n'arrivera pas.

*Remarque :* la dernière ligne du fichier termine par un `'\n'`, le caractère `'\0'` ne sera renvoyé qu'au moment d'essayer de lire la ligne d'après.

**Fichiers fournis.** En plus de ce fichier PDF, on vous fournit quatre fichiers :

- `movie.csv` qui est un fichier au format CSV décrivant des films dont trois des lignes correspondent à la table plus haut. Ce fichier décrit une table contenant les attributs :
  - **id**, un champ identifiant chaque film,
  - **title** le titre du film,
  - **yr**, l'année de sortie du film,
  - **director** un entier qui est une clef pour le réalisateur du film,
  - **budget**, un entier, le budget du film,
  - **gross**, un entier représentant les revenus salles du film ;
- `actor.csv` qui est un fichier au format CSV décrivant des acteurs. Ce fichier décrit une table contenant les attributs :
  - **id**, un champ identifiant chaque acteur,
  - **name**, le nom de l'acteur ;
- `casting.csv` qui est un fichier au format CSV décrivant quels acteurs apparaissent dans quels films. Ce fichier décrit une table contenant les attributs :
  - **movieid**, un champ identifiant un film,
  - **actorid**, un champ identifiant un acteur,
  - **ord**, un entier décrivant l'ordre d'apparition au générique de l'acteur identifié par `actorid` dans le film identifié par `movieid`
- enfin `movie.db` est un fichier qui stocke une base de données contenant trois tables (`movie`, `actor`, et `casting`), correspondant aux trois tables décrites dans les fichiers CSV mais au format SQLite3. On pourra interroger cette base de données en lançant la commande `"sqlite3 movie.db"` puis en tapant des requêtes SQL.

**Question 1.** En utilisant la fonction `lit_cellule` donnée plus haut, écrire une fonction qui prend un chemin vers un fichier et calcule le nombre de colonnes de ce fichier (on supposera qu'il y a toujours au moins une ligne dans le fichier). Vérifiez vos résultats en comparant au nombre attendu pour les trois fichiers CSV. On pourra consulter la documentation de `fopen` pour ouvrir un fichier en lecture.

**Question 2.** Proposer un type pour stocker le contenu d'une ligne. Comme toutes les lignes d'une même table ont le même nombre de colonnes, on pourra supposer que toute fonction qui manipule un type ligne connaît le nombre de colonnes et on évitera donc de stocker le nombre de colonnes dans le type qui stocke une ligne.

**Question 3.** Proposer une fonction `lit_ligne(FILE * fichier, int nb_colonnes)` qui prend en paramètre un descripteur de fichier et un nombre de colonnes et renvoie une ligne en utilisant le type proposé à la question d'avant. Votre fonction peut supposer que le nombre de colonnes passé en paramètre est correct mais devra renvoyer une valeur spéciale dans le cas où une ligne n'a pas pu être lue car la lecture a atteint la fin du fichier (c'est-à-dire quand `lit_cellule` renvoie `'\0'`).

**Attention :** la chaîne de caractères passée en paramètre de `lit_cellule` doit pouvoir contenir 1000 chars mais votre fonction `lit_ligne` ne devra pas allouer 1000 char pour chaque chaîne lue. Dans cet objectif, on pourra utiliser `strdup (donnees)` (dans l'entête `string.h`) qui renvoie une copie de la chaîne `donnees` passée en paramètre en allouant la taille strictement nécessaire.

## Partie 2 – Vecteur en C

**Structure de vecteur.** Un *vecteur* (parfois aussi appelé *tableau dynamique*), est une structure de données qui stocke une collection d'éléments et supporte les opérations suivantes :

- allocation d'un nouveau vecteur, initialement vide,
- insertion (en dernière position) d'un élément,
- lecture ou modification du  $i$ -ème élément.

Les vecteurs sont donc assez similaires aux tableaux disponibles en C mais, contrairement à ceux-ci, leur taille n'est pas connue à l'initialisation et peut changer au fil de l'exécution (via les opérations d'insertion). Un exemple connu d'implémentation des vecteurs est celui de la `list` en python où l'on peut faire `l.append(42)` pour rajouter l'entier 42 ou `l[i]` pour accéder (lecture ou écriture) au  $i$ -ème élément.

**Implémentation naïve.** Une manière d'implémenter des vecteurs en C, c'est d'avoir une structure avec deux champs : un pointeur vers le contenu du vecteur et un entier qui décrit la taille (c'est-à-dire le nombre d'éléments du vecteur). Pour accéder à un élément du vecteur, il suffit d'utiliser le pointeur. Pour insérer, on commence par copier le contenu actuel dans un nouvel espace mémoire plus grand d'un élément puis on insère le nouvel élément.

Cette solution n'est pas très efficace car chaque insertion nécessite une copie du tableau de données (et donc coûte  $O(n)$  après  $n$  insertions). Nous allons donc voir une structure plus efficace.

**Implémentation efficace.** Au lieu d'avoir toujours un bloc mémoire dont l'espace alloué est exactement la taille courante du vecteur, on va allouer un bloc mémoire d'une taille supérieure à la taille courante du vecteur. L'insertion est très rapide quand il reste de la place disponible dans le bloc mémoire et sinon on augmente la mémoire comme dans le cas de l'implémentation naïve.

Pour implémenter en pratique cette structure, il y a besoin de trois champs : un pointeur vers le contenu du vecteur, un entier donnant la taille actuelle du vecteur et un entier donnant la taille de la mémoire allouée pour le tableau que l'on appellera sa *capacité*. Pour insérer dans un tel vecteur, il faut d'abord s'assurer que la capacité est strictement supérieure à la taille courante. En effet, quand la capacité est supérieure à la taille, il suffit de mettre la valeur dans le tableau et d'incrémenter la taille courante. Dans le cas où la capacité est égale à la taille courante, on va allouer une zone mémoire capable de stocker le double de la capacité actuelle (ou au moins de capacité au moins 1 si la capacité actuelle est 0), y copier les données puis faire l'insertion comme dans le premier cas.

**Question 4.** En partant d'un vecteur dont la capacité est 0, estimer la complexité totale de faire  $n$  insertions consécutives puis la complexité pire cas d'une insertion sur un vecteur de taille  $n$ .

**Question 5.** Proposer une structure `vecteur` pour représenter des vecteurs d'entiers en C. Pour cette question, le type stocké dans les tableaux sera `int`.

**Question 6.** Implémenter les fonctions `vecteur nouveau_vecteur()` et `ajoute(vecteur * t, int valeur)` qui créent un nouveau vecteur et ajoutent un entier `valeur` à un vecteur passé en paramètre par pointeur.

**Tables dynamiques.** Dans ce sujet, les vecteurs vont servir à stocker des tables telles que vues dans l'exercice précédant. On rappelle qu'une table est définie par un nombre de colonnes, un nombre de lignes et une chaîne de caractères pour chaque paire de colonne et ligne.

Les diverses tables que nous manipulerons n'auront pas forcément toutes le même nombre de colonnes mais pour une table donnée, le nombre de colonnes sera fixe. Pour simplifier les algorithmes qui manipulent les tables, nous aurons besoin de tables dynamiques, dans lesquelles l'on peut facilement insérer des lignes à la fin. En effet, cela nous servira dans l'implémentation des différentes opérations que nous allons voir.

**Question 7.** Proposer une structure `table` pour représenter des tables dynamiques. Implémenter les fonctions et les fonctions `nouvelle_table(int nb_colonnes)` et `ajoute_enregistrement(table * t, char ** ligne)`. Pour cette deuxième fonction on supposera que le pointeur `ligne` a des dimensions compatibles avec la table, c'est-à-dire que `ligne[i]` est une chaîne de caractère pour `i` un entier entre 0 et le nombre de colonnes de la table pointée par `t`.

### Partie 3 – Manipulation de tables C

**Question 8.** Écrire une fonction `lit_table(const char * chemin)` qui prend en argument un chemin vers un fichier au format décrit plus haut et retourne une structure `table` qui contient le contenu du fichier.

**Question 9.** Dans la table `movie`, le champ qui peut être le plus long est celui du titre. Quels sont les titres les plus longs en nombre de caractères parmi les films présents dans la base de données ? On ne recopiera pas sur le compte-rendu les titres en intégralité et on ne cherchera pas à comparer avec le contenu du fichier CSV.

*Indication : en SQLite, `length(s)` renvoie la longueur de la chaîne `s`.*

**Attention :** cette question est là pour vérifier que la contrainte des chaînes de taille 1000 est suffisante mais on ne cherchera pas allouer la taille des chaînes en fonction du résultat obtenu car les chaînes de caractères peuvent contenir des accents qui prennent plusieurs chars pour être encodés.

**Question 10.** Écrire une fonction `ecrit_table(table * t, FILE * sortie)` qui prend en argument un pointeur vers une table et un descripteur de fichier et écrit dans le fichier le contenu de la table au format textuel, c'est-à-dire que les différentes lignes sont séparées par des `'\n'` et que les différentes colonnes sont séparées par des `'\t'`. Pour écrire cette sortie, on rappelle les commandes suivantes :

- `fprintf(sortie, "\t")` écrit `'\t'` dans le fichier `sortie`,
- `fprintf(sortie, "\n")` écrit `'\n'`, le caractère de retour à la ligne,
- `fprintf(sortie, "%s", maChaine)` écrit le contenu de la chaîne de caractères `maChaine`.

Enfin on rappelle l'existence du descripteur de fichier `stdout` défini dans l'entête `stdio.h` qui correspond à la sortie standard, c'est-à-dire le terminal. Ainsi on peut tester la fonction en appelant `ecrit_table(maTable, stdout)` et cela devrait afficher le contenu de la table `maTable` sur le terminal.

**Question 11.** Écrire une fonction

`table filtre_valeur(table * t, int col, const char * v)` qui prend en paramètre une table, un numéro de colonne, une chaîne de caractère et renvoie une copie de la table `t` en ne gardant que les lignes qui ont une chaîne de caractères égale à `v` dans la colonne d'indice `col`. On pourra utiliser la fonction `strcmp` pour comparer deux chaînes de caractères.

**Question 12.** Écrire une fonction `supprime(table * t, int col)` qui prend en paramètre un pointeur vers une table et un numéro de colonne et renvoie une copie de la table pointée par `t` où la colonne `col` a été supprimée.

**Question 13.** En utilisant les fonctions ci-haut, quels sont les `movieId` et année de sortie du film enregistré sous le titre 'Rent' ? Quels sont les titres des films sortis en 1900 ? Vérifiez avec SQLite votre résultat.

**Question 14.** Écrire une fonction `filtre_egal(table * t, int col1, int col2)` qui prend en paramètre un pointeur vers une table, deux numéros de colonne et renvoie une copie de la table pointée par `t` en ne gardant que les lignes qui ont des chaînes de caractères égales dans les colonnes d'indice `col1` et `col2`.

**Question 15.** Écrire une fonction `echange(table * t, int col1, int col2)` qui prend en paramètre un pointeur vers une table, deux numéros de colonne et renvoie une copie de la table `t` où le contenu des colonnes d'indice `col1` et `col2` sont échangés.

**Question 16.** Écrire une fonction `table * produit(table * t1, table * t2)` qui prend en paramètre deux tables et renvoie une table correspondant au produit cartésien des tables `t1` et `t2`. Cela correspondrait, en SQL, au résultat de la requête `SELECT * FROM t1, t2`. Plus précisément, pour chaque tuple `a` de `t1` et chaque tuple `b` de `t2` alors la table renvoyée contient un tuple qui est la concaténation de `a` et `b`.

**Question 17.** Quels sont les noms des acteurs ayant joué dans le film `Rent` ? Calculer la réponse avec SQLite3 et avec les fonctions programmées ci-haut. Attention à la taille des tables intermédiaires que vous créez !

## Partie 4 – Manipulation avancée de tables

**Question 18.** Écrire une fonction `tri(table * t, int col)` qui prend en paramètre une table et un indice de colonne et tri la table par ordre lexicographique de la colonne d'indice `col`. Pour trier on pourra utiliser l'algorithme du tri rapide dont le pseudo-code est donné ci-dessous. On rappelle qu'un nombre aléatoire entre 0 et `RAND_MAX` (tous deux inclus) peut s'obtenir avec un appel à `rand()` disponible dans `stdlib.h`.

**Question 19.** Expliquer comment on peut faire des jointures internes efficacement dans notre langage de requêtes, par exemple une requête comme celle ci-dessous en SQLite :

```
SELECT * FROM t1 JOIN t2 ON t1.a = t2.b
```

**Question 20.** Écrire une fonction `jointure(table * tA, table * tB, int colA, int colB)` qui prend en paramètre deux tables et deux indices de colonnes et renvoie la jointure interne des deux tables avec la condition que les enregistrements de la table `tA` ne sont joint qu'avec ceux de la table `tB` pour lesquels la valeur dans la colonne `colA` de la première table est égale à la valeur dans la colonne `colB` de la seconde table.

**Algorithm 1: Tri rapide**


---

```

def tri_interval (table,colonne,debut,fin):
    if debut+1 < fin then
        pivot ← nombre aléatoire entre debut inclus et fin exclue
        v_pivot ← table[pivot][colonne]
        n_debut ← debut
        n_fin ← fin-1
        n_eq ← 0
        while n_deb ≤ n_fin do
            if table[n_deb][colonne] ≤ v_pivot then
                if table[n_deb][colonne] = v_pivot then
                    n_eq ← n_eq + 1
                    n_deb ← n_deb + 1
                else
                    échanger table[n_deb] et table[n_fin]
                    n_fin ← n_fin - 1
            if nb_eq < fin-debut then
                tri_interval (table,colonne,debut,n_deb)
                tri_interval (table,colonne,n_deb,fin)
def tri (table,colonne):
    tri_interval (table,colonne,0, table.taille)

```

---

**Partie 5 – Indexation par hachage**

Quand ils exécutent une requête comme **SELECT** \* **FROM** maTable **WHERE** col=42 les moteurs de base de données n'ont pas toujours besoin de parcourir l'intégralité de la table pour récupérer les lignes correspondantes. En effet, si la colonne testée a été indexée, les moteurs de base de données peuvent rapidement récupérer les lignes pertinentes. Nous allons maintenant étudier un index qui s'appuie sur une fonction de hachage  $h$ .

Si l'on veut indexer un fichier selon sa colonne d'indice 0, l'idée de notre index c'est de découper le fichier en  $n$  plus petits fichiers, chaque ligne étant stockée dans le fichier  $h(v_0)$  où  $v_0$  est la valeur de la ligne dans sa colonne  $c$ .

La fonction de hachage à utiliser est donnée ci-dessous. Il n'y a pas besoin de savoir comment elle marche, elle prend une chaîne de caractères  $l$  et renvoie un nombre positif  $h(l)$  en 0 et 255 avec les propriétés suivantes (que l'on admet) :

- $h$  représente une fonction mathématique, si on lui donne deux fois la même chaîne de caractères, elle renverra le même haché ;
- $h$  mélange assez bien les résultats ; pour les calculs de complexité on pourra estimer que la probabilité que deux chaînes différentes aient le même haché est de  $1/256$ .

```

unsigned int hash(const char * l ) {
    unsigned int r = 0 ;
    while (*l!='\0') {
        r = (*l)+27*r ;
        l++;
    }
    return r%256;
}

```

**Question 21.** Écrire une fonction qui prend en paramètre une table, un nom d'index (par exemple, pour le fichier `movie.csv` on pourra nommer l'index `movie.csv.idx`) et calcule puis stocke l'index de cette table selon sa colonne d'indice 0. Chacune des 256 parties de l'index sera à stocker dans un fichier unique. On recommande que le nom de ce fichier soit le nom de l'index concaténé avec l'indice du haché.

**Question 22.** Écrire une fonction qui cherche dans un index toutes les lignes dont la première colonne est égale à une valeur donnée en paramètre.

## Partie 6 – recherche rapide dans une table triée

Trier une table permet de facilement rechercher un élément par dichotomie. Une deuxième méthode pour chercher dans une table consiste à stocker une version triée de la table et à chercher par dichotomie dans ce fichier les lignes. Pour se déplacer dans un fichier, on donne les commandes suivantes :

```
fseek(fichier, 0L, SEEK_END); // se déplace à la fin du fichier
ftell(fichier) ; // donne la position courante en nombre de caractères,
                  // 0 si au début, taille du fichier si à la fin.
// Remarque : la combinaison des deux commandes plus haut permet
// d'obtenir la taille du fichier

fseek(fichier, pos, SEEK_SET) ; // se déplace à la position 'pos' du fichier
// en nombre de caractères depuis le début du fichier
```

**Attention** : toute lecture déplace la position courante dans le fichier !

**Question 23.** Mettre en place une recherche rapide par dichotomie dans un fichier. L'algorithme ne devra pas lire la table entière puis faire la dichotomie mais bien faire une dichotomie sur la lecture du fichier.