Tutorat de Programmation Avancée

Tx TateX | Exploration de la collection du Tate Modern

Programme réalisé par Antoine GOSSE & Antoine DUQUENOY dans le cadre d'un projet de Programmation Avancée en IMA 3 à Polytech Lille.

Les données utilisées dans ce programme sont sous licence Creative Commons Public Domain CC0 et sont diffusées librement par le Tate Modern sur leur git.

Table des matières

Introduction

L'ouverture des données, plus communément appelé *Open Data*, est un mouvement, une philosophie, qui consiste à rendre accessible à tous des données numériques. Cette démarche de libre accès n'est pas récente dans les administrations mais l'ouverture de portails dédiés sur Internet remonte à 2009 aux Etats-Unis et au Royaume-Uni.

La célèbre organisation *Tate* regroupant plusieurs musées s'est joint à cette pratique et a diffusé en 2013 les métadonnées de quelques 70.000 œuvres associées à environ 3.500 artistes. Une telle quantité de données se doit d'être organisée le plus efficacement possible afin d'être exploitable en un minimum de temps. Ces contraintes résument très bien le module de programmation avancée consacré essentiellement aux structures de données que nous avons suivi durant ce semestre.

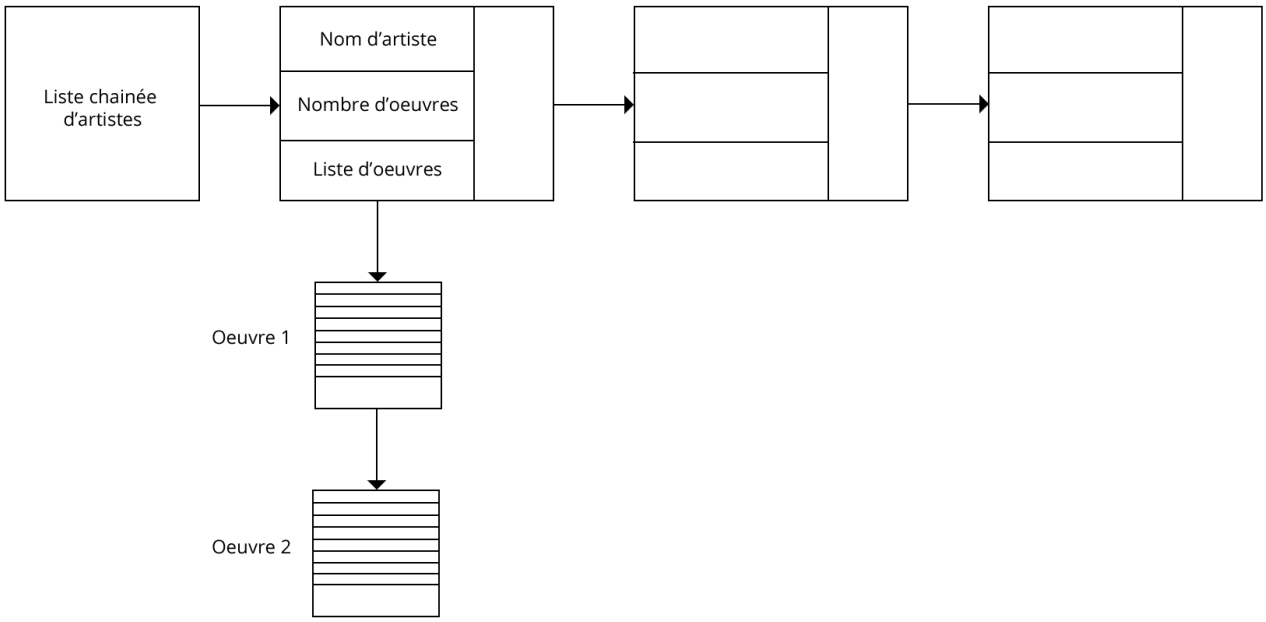
TateX, pour *Tate Exploration*, est le nom donné à notre programme. Son objectif est d'analyser la collection d'œuvres décrite ci-dessus et de proposer à l'utilisateur différentes actions lui permettant d'interagir avec. Pour mener à bien ce projet, il nous a paru nécessaire de diviser la charge de travail en trois parties, parties qui constitueront également le plan de ce rapport. En premier lieu, nous verrons comment nous avons trouvé judicieux d'organiser cette masse d'informations. Ensuite, nous détaillerons le processus qui nous a permis de récupérer les données du fichier CSV pour les stocker. Puis enfin, la dernière partie sera consacrée aux fonctions d'exploitation de la structure de données imposées dans le cahier des charges.

Partie 1 : Création de la structure de données

1. Les prémisses de la structure

Dès la première séance de tutorat, nous avons commencé a travaillé sur une ébauche de notre structure de données (SD). Nous verrons par la suite qu'elle a connu un seul gros changement et quelques modifications mineures d'optimisation.

La première version consistait en l'utilisation exclusive des listes chainées. Il était question d'une liste chainée d'artistes contenant son ID et sa liste (chainée) d'œuvres. Nous aurions, pour optimiser le chargement, classé les artistes par ordre décroissant. En effet, nous avions remarqué que les œuvres du fichiers CSV avaient plutôt tendance à être rangées par ordre croissant d'ID artiste (mais pas toujours). L'ajout d'un nouvel artiste et de son œuvre (ou seulement d'une œuvre) aurait été pour le meilleur des cas un ajout en tête. Nous avions ensuite eu l'idée de classer les œuvres de la liste par ordre croissant d'année de création afin d'optimiser la recherche de la plus vieille œuvre. Nous avions ensuite trouvé judicieux d'incrémenter à chaque ajout d'œuvre le compteur d'œuvres d'un artiste. Cette amélioration aurait permis de diminuer le temps d'exécution des commandes 4 et 5 en évitant le parcours de liste chainées.



2. Nouvelle structure

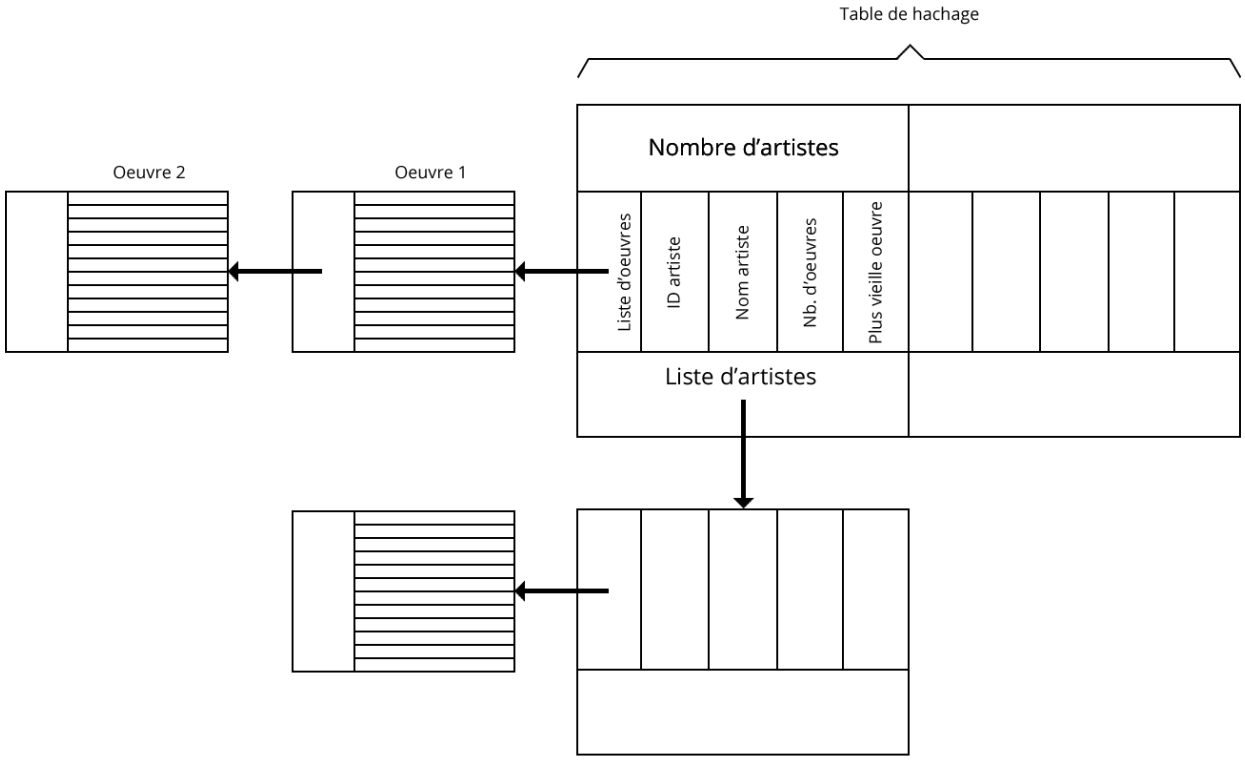
Cette nouvelle version de notre SD est en quelque sorte un mélange entre la première et une table de hachage. On le remarque sur la *Figure 2* présente ci-dessous par l'intégration d'une liste chainée d'artistes accompagnée de sa liste chainée d'œuvres au sein d'une cellule de la table de hachage. Ce choix s'est imposé après que nous ayons appris sur le Git du *Tate* la présence d'environ 3.500 artistes. En supposant que nous aurions opéré avec une recherche de type dichotomique, la complexité aurait été de 8.16.

$$T(3335) = O(log_2(3335)) \approx 8.11$$

Or, nous savons qu'avec une table de hachage, la complexité peut être de 1 à condition d'avoir une fonction de hachage provoquant le moins de collisions possibles. Nous verrons dans le paragraphe suivant que notre fonction génère une moyenne de 2.60 collisions par cellule. Pour si peu de collisions, nous n'avons pas trouvé nécessaire d'opter pour une recherche dichotomique qui en plus, aurait nécessitée le tri des artistes en collision. Au final, la recherche d'un artiste dans notre SD s'effectue avec une complexité moyenne de 2.30.

$$C = 1 + \frac{2.60}{2} = 2.30$$

En plus de cette modification majeure, nous avons implémenté d'autres ajouts pour optimiser l'ensemble des commandes du cahier des charges. Nous pouvons voir que chaque cellule de la table de hachage ne contient pas seulement une liste d'artistes. Il s'agit d'une structure à deux éléments contenant le nombre d'artistes (autrement dit le nombre de collisions) ainsi que la liste chainée d'artistes. Ensuite, nous avons rajouté des champs à la structure nommée Artiste (contenu de la liste chainée d'artistes) comme le nom de l'artiste ou encore un pointeur vers sa plus vieille œuvre. Ces différents choix seront détaillés au cours de la partie 3 sur le traitement des commandes.



3. Fonction de hachage et statistiques

Nous l'avons vu dans le paragraphe précédent, l'intérêt d'une table de hachage est de réduire le temps d'accès aux cellules. Cependant, il est important de réaliser une fonction de hachage produisant le moins de collisions possibles afin de réduire le temps de parcours de cette liste chainée de collisions.

Tout d'abord, nous avons choisi notre clé de hachage. Ce choix s'est fait naturellement par la lecture du cahier des charges qui impose des commandes exclusivement par ID d'artiste. Cette étape de détermination n'a que très peu d'importance sur l'optimisation de la table (à condition que la clé soit unique) contrairement aux deux suivantes. Il nous reste donc deux éléments clés pour terminer cette structure : une taille de tableau et une fonction de hachage.

Nous avons eu l'idée de créer un programme annexe de statistiques nous permettant de faire varier certaines constantes et de recueillir des données chiffrées sur l'optimisation. Le premier paramètre est la taille de la table définie par `TABLE_SIZE`.

Le second paramètre est la variable `c` présent dans le pseudo code ci-dessous.

```
Fonction hash(ID)

        D : ID - Chaîne de caractères (ID d'artiste)
        R : position

        i = 0
        h = 0
        c = 11
        Tant que (ID[i] != '\0')
                h = h + (ID[i]*c^i)
                i = i + 1
        Fin Tant que
        position = h % TABLE_SIZE
        Retourner position

Fin Fonction
```

Si nous devions hacher l'id '38', cette fonction retournerait la position 667.

$$Position = (ID[0].c^0 + ID[1].c^1) \bmod TABLE\_SIZE = (51 * 11^0 + 56 * 11^1) \bmod 1284 = 667$$

Ce calcul a été réalisé avec les paramètres `c = 11` et `TABLE_SIZE = 1284`. Ces nombres pourraient s'apparenter à des *magic numbers* s'ils ne découlaient pas d'une analyse. Nous ne pouvons certifier qu'il s'agit des paramètres les plus optimales mais les résultats nous semblaient convaincants :

- **Nombre de cellules sans artiste :** 11 sur 1284 (< 1%)
- **Maximum d'artistes par cellule :** 6
- **Moyenne d'artistes par cellule :** 2.5974
- **Ecart-type :** 1.0150

Pour garantir une structure optimale, nous aurions pu réaliser un script qui aurait testé à notre place des centaines de milliers de combinaisons.

Partie 2 : Découpage du fichier CSV

1. Récupération des données

> **Remarque :** dans une optique de modularité, nous avons trouvé intéressant de garder en mémoire l'ensemble des informations du fichier CSV (20 colonnes), au détriment de la durée d'exécution. Nous verrons par la suite qu'elles ne sont pas utilisées mais elles restent tout de même accessibles.

Cette partie a pu être traitée en parallèle avec la première du fait de leur quasi indépendance. En effet, si on exclut l'étape de remplissage de la SD, il est tout à fait possible de découper le fichier et d'afficher par exemple le résultat au lieu de le ranger. Avant de nous lancer dans une fonction de découpage, nous avons exploré le fichier d'œuvres afin de déterminer les cas particuliers. Assez rapidement, nous nous sommes

aperçu que certains champs pouvaient être vide mais qu'aucun espace n'était ajouté entre les virgules de séparation : `texte1,texte2,,,texte3,,texte4`. Cette première constatation représenté un frein à l'utilisation de la fonction `strtok()` de la librairie `string.h` qui ne prend pas en charge les virgules consécutives. Deuxièmement, des virgules peuvent se glisser à l'intérieur d'un champ si celui-ci est délimité par des guillemets. A nouveau, cette particularité n'est pas compatible avec `strtok()`. Son utilisation dans notre programme n'est donc pas une option.

A défaut de rebâtir cette fonction de *parsing*, nous avons trouvé judicieux de s'appuyer sur le fonctionnement de la fonction `strtok()` et de la modifier à notre souhait. N'ayant pas su trouver le code de cette fonction, nous avons dû analyser son fonctionnement un peu particulier. En effet, le premier appel de la fonction s'effectue de la sorte `strtok(ligne, separateur)` et ensuite il suffit de faire `strtok(NULL, separateur)`. Cette utilisation nous a semblé "magique" avant que l'on découvre l'existence des variables statiques. Au sein d'une fonction, une variable statique n'est définie qu'au premier appel de la fonction, ensuite, cette variable n'est plus initialisée et garde son contenu comme le ferait une variable globale.

Pour mieux illustrer son fonctionnement, voici les différentes étapes de découpage d'une ligne d'essai.

```
ligne : mot1,"mot2, mot3",,mot4

* mot1 ne commence pas par un guillemet donc on cherche la première virgule
* on remplace cette virgule par une fin de chaîne "\0"
* on place le pointeur statique après la première virgule
* on retourne un pointeur sur mot1

* "mot2, mot3" commence par un guillemet donc on change le séparateur en '",'
* on décale d'un le pointeur statique précédent pour ne pas prendre le guillemet
* on remplace le " du séparateur par un "\0"
* on place le pointeur statique après la virgule
* on retourne un pointeur sur mot2, mot3

* mot1 ne commence pas par un guillemet donc on cherche la première virgule
* on remplace cette virgule par une fin de chaîne "\0"
* on place le pointeur statique après la première virgule
* on retourne un pointeur sur NULL

* mot4 ne commence pas par un guillemet donc on cherche la première virgule
* aucune virgule trouvée donc on remplace le "\n" par un "\0"
* pointeur statique NULL
* on retourne un pointeur sur mot4

Au prochaine appel, la fonction retournera NULL pour désigner la fin de la ligne
```

Ainsi, tant qu'il ne s'agit pas de la fin de la ligne, nous utilisons la fonction `strdup()` pour copier le champ en allouant exactement la taille qu'il lui faut. L'adresse des vingt chaînes de caractères est stockée dans une liste temporaire puis quand la ligne est terminée, nous créons une structure œuvre avec les informations récupérées avant de l'ajouter à notre SD.

2. Stockage des données

Le meilleur moyen d'expliquer cette partie est de dérouler le processus d'ajout. Une fois l'œuvre créée, nous appliquons la fonction de hachage sur l'ID de l'artiste. Si le nombre d'artistes est nulle, on crée un artiste par ajout en tête, on incrémente le nombre d'artistes puis on ajoute son œuvre également en tête. Au passage, on initialise sa plus vieille œuvre avec l'adresse de sa première œuvre. Si au contraire le nombre d'artistes est non nulle, on parcourt la liste chainée dans l'espoir de trouver l'artiste et de lui ajouter une nouvelle œuvre (en modifiant sa plus vieille œuvre au besoin). Si on ne le trouve pas, c'est qu'il s'agit d'une nouvelle collision, on ajoute donc un nouvel artiste à cette liste.

Partie 3 : Traitement des commandes

Au final, bien qu'il s'agisse de la priorité du programme, les fonctions d'exploration du cahier des charges ont été la partie la plus rapide à réaliser.

- **Liste de toutes les œuvres**

`void afficher_oeuvres(Hashtable)`

Cette fonction parcourt l'ensemble de la table de hachage et affiche pour chaque artiste sa liste complète d'œuvres. Pour un affichage plus structuré, nous avons séparé les œuvres en ajoutant une ligne correspondant au nom de l'artiste. Voici un exemple réduit d'affichage :

```
Artiste : Dadd, Richard
* (130) Dadd, Richard : Wandering Musicians | 1878
* (130) Dadd, Richard : Portrait of a Young Man | 1853
```

- **Recherche d'un artiste identifiée par son ID**

`int recherche_artiste(Hashtable ht, char* id)`

On commence par hacher l'ID renseigné par l'utilisateur avant de se rendre à la cellule correspondante. Si le nombre d'artistes est nul, l'artiste n'existe pas et on retourne 0. Si par contre il est non nul, on parcourt la liste chainée d'artistes et on compare les IDs. Si deux IDs correspondent, on retourne 1.

- **Liste de toutes les œuvres d'un artiste donné par son ID**

`void liste_artiste(Hashtable ht, char* ida)`

Même démarche que la fonction précédente sauf que si l'artiste est présent, on affiche sa liste d'œuvres.

- **Compte le nombre d'œuvres pour un artiste donné par son ID**

`int nb_oeuvres_artiste(Hashtable ht, char* id)`

Le comptage étant effectué en amont, cette fonction cherche l'artiste en hachant son ID et retourne son nombre d'œuvres ou -1 s'il n'est pas présent.

- **Afficher le nombre d'œuvres par artiste**

`void nb_oeuvres_all(Hashtable ht)`

Similaire à la précédente mis à part qu'on parcourt toute la table hachage.

- **Affiche l'œuvre la plus vieille**

`void afficher_old(Hashtable ht)`

La commande du cahier des charges fait mention de "la plus vieille", cependant, il s'avère que deux œuvres peuvent prétendre à ce titre. C'est pourquoi nous avons fait le choix d'afficher les deux. Concernant l'algorithme, il parcourt toute la table de hachage et compare les plus vieilles œuvres de chaque artiste. Cette fonction était différente auparavant car la SD stockée directement la plus vieille œuvre de l'ensemble (accès direct). Le nombre de comparaisons effectuées lors du remplissage était identique mais nous avons préféré jouer la carte de la flexibilité et perdre à nouveau légèrement en temps d'exécution. Néanmoins, si une fonction donnant la plus vieille œuvre d'un artiste devait être incorporée, notre structure serait déjà prête.

Conclusion

Bien plus qu'un projet de programmation, ce tutorat nous a permis de mettre en pratique nos connaissances sur le gestionnaire de version Git et de mener ainsi un projet organisé et structuré. Pour ne rien vous cacher, nous étions septiques à l'idée d'utiliser ce moyen de partage. Habitué à travailler sur des plateformes telles que *Google Drive*, nous avions du mal à trouver les bénéfices de Git. Finalement, nous avons joué le jeu et nous avons su faire passer cela au-dessus de la simple contrainte. L'utilisation des *issues* de GitLab s'est montrée par exemple plutôt efficace dans la répartition du travail et le suivi de bugs.

Bien évidemment, ce projet a contribué avec les TPs à renforcer nos connaissances en matière de structure de données. Mais nous avons trouvé particulièrement intéressant le coté optimisation du programme, que ce soit en termes de mémoire ou de temps d'exécution. Ce point est très motivant dans la réalisation d'un projet car il nous pousse sans cesse à nous confronter dans nos idées d'amélioration. Néanmoins, lorsqu'il a été nécessaire de choisir entre rapidité et flexibilité, notre choix s'est tourné vers la seconde option. En effet, à vouloir sans cesse être le plus rapide, on en arrive à faire des concessions parfois néfastes pour la modularité. Il en est de même pour la mémoire, l'argument du programme le plus léger est-il suffisant pour exclure des données de la structure ? Les avis au sein de notre classe semblent diverger. Dans notre cas, il nous a paru raisonnable d'intégrer toutes les données, non sans négliger l'optimisation.

Toujours dans l'optique de la souplesse, nous aurions souhaité créer une librairie consacrée aux fichiers tabulaires afin d'offrir une grande variété d'options pour le découpage (choix du séparateur, nombre de colonnes variables, gestion des guillemets, choix de colonnes à conserver ...). Nous aurions également voulu rendre la structure plus flexible en faisant varier automatiquement les paramètres de la table de hachage selon le nombre d'artistes du fichier. Pour terminer avec les améliorations, nous aurions pu envisager des recherches par nom d'artiste. Mais l'inconvénient de cette idée est qu'elle peut engendrer des problèmes d'ordre syntaxique (majuscules des noms, accents ...) pouvant être résolu en exécutant des algorithmes de reconnaissance qui peuvent à eux seuls faire l'objet d'un projet.

> Nous remercions les encadrants de ce tutorat pour la rallonge de temps accordée.