Rapport du TME 2-3-4

KANDEEPAN Mathuran 3800103 (DM Info/EEA)
Moutymbo Ugo 3800601 (INFO DANT)

Le but de se mini-projet était d'utiliser quatre structures de données différentes (trois linéaires et une non linéaires) afin d'apprendre à les comparer pour savoir laquelle était la plus efficace en fonction du contexte. On rappelle que dans ce TME, nous avons comparer l'efficacité de nos fonctions et de nos structures en fonction du temps et non pas en fonction de la mémoire.
Dans notre cas, nous nous étions intéressés à la gestion d'une bibliothèque musicale, composée de morceaux, eux-mêmes composés d'un numéro, d'un titre et d'un artiste. Nous allons ainsi voir les résultats de nos recherches et de nos test.

Dans un premier temps, nous allons effectuer une description globale de notre code. Premièrement, nous avons fait effectuer un *Makefile* qui nous permettait de compiler seulement une seule structure à la fois. Voici un exemple qui compilait la structure liste en tapant la commande '*make liste*' dans le terminal (nous vous avons épargné les autres structures):

```
GCC_FLAGS = -g -Wall

liste: biblio_liste.o biblio.o parser.o main.o
gcc $(GCC_FLAGS) biblio_liste.o biblio.o parser.o main.o -o main

parser.o: parser.c parser.h
gcc $(GCC_FLAGS) -c parser.c

biblio_liste.o : biblio_liste.c biblio.h
gcc $(GCC_FLAGS) -c biblio_liste.c

biblio.o : biblio.c biblio.h parser.h
gcc $(GCC_FLAGS) -c biblio.c

main.o : main.c biblio.h parser.h
gcc $(GCC_FLAGS) -c main.c
```

Ainsi, nous avions '*make liste*', '*make arbre*', '*make tab*', et '*make hachage*' qui menaient vers le même fichier : *main*. La commande pour exécuter le programme était donc toujours la même (contrairement à ce que vous aviez proposé sur le site de l'ue) : '*./main BiblioMusicale.txt nb_mor*'.

Comme indiqué dans la consigne, nous avons donc utilisé le même *main* et le même *biblio.h* pour compiler les différents fichiers des différentes structures. Nous n'avons pas eu besoin de créer un *.h* pour chaque fichier *.c* car les structures étaient déjà défini dans les *.c* (nous avons plus l'habitude de les définir dans des *.h* cependant le sujet nous imposait cela).
biblio.h comprenait toutes les fonctions que devait comprendre les fichiers *.c* des différentes structures :

```
Biblio *nouvelle_biblio(void);
void libere_biblio(Biblio *B);
void insere(Biblio *B, int num, char *titre, char *artiste);
void afficheMorceau(CellMorceau *);
void affiche(Biblio *B);  Biblio *uniques(Biblio *B);
Biblio *supprimerUnMorceau(Biblio *B, char *titre, char *artiste);
int supprimeMorceauParNum (Biblio *B, int num);
void rechercheMorceauNumero(Biblio *B, int num);
void rechercheMorceauArtiste(Biblio *B , char *artiste);
void rechercheMorceauTitre(Biblio *B , char *titre);
```

Il comprenait aussi deux fonctions qui étaient communes à toutes les structures, définies dans *biblio.c* :

```
Biblio *charge_n_entrees(const char *nom_fichier, int n);
void uniques_graphe(int borne_inf , int pas, int borne_sup, char*fichier);
```

Toutes ces fonctions sont commentées dans le *biblio.h*.
Deuxièmement, nous avons définies des fonctions visiblement non demandées soit pour faciliter le fonctionnement de notre code, soit parce que nous avions mal compris les attentes du sujet mais nous ne les avons pas supprimées.

Pour *biblio_hachage.c*, nous avons créé une fonction `void collision(CellMorceau *list, int *cpt)` qui nous permet d'afficher le nombre de collisions dans chaque case du tableau et d'afficher le nombre de collision totale dans la bibliothèque. Elle nous a permis de tester différentes fonctions clés (de nous mais aussi celles de certains camarades) pour savoir laquelle était la meilleure et aussi pour voir que la taille de la table joue sur le nombre de collision

Pour *biblio_liste.c*, nous avons gardé une fonction `void supprimerTouslesMorceaux(Biblio *B, char *titre, char *artiste)` qui supprime toutes les occurrences d'un morceau car la consigne n'était pas clair au début concernant la suppression d'un morceau.

Pour *biblio_arbre.c*, nous avons beaucoup utilisé la récursivité pour parcourir les différents nœuds. De plus, nous avons souvent fait appels à des fonctions auxiliaires qui facilite le parcours entre les nœuds ou bien au sein de la liste de morceaux, et la compréhension du code.

Pour *biblio_tabdyn.c*, nous n'avons rien de spécial à relever. Toutes les fonctions ont été définies à partir de *biblio.h*.

Pour la fonction *uniques* de chaque structure, nous avons fait appel à une fonction auxiliaire *RechercheDoublons* qui retourne 1 si le morceau en paramètre est la seule occurrence dans la bibliothèque ou sinon 0. La fonction permet de savoir si l'on doit insérer ou non le morceau dans la bibliothèque final.

Nous avons bien évidemment vérifier grâce à '*valgrind --leak-check=yes ./*' que nos codes n'avaient pas de fuites de mémoire pour les quatre structures.

Dans notre *main*, nous avons 11 options :
0) Sortie de la fonctionne
1) Affichage de la bibliothèque
2) Retourne la bibliothèque de la fonction unique
3) Affiche le morceau recherché par numéro grâce à *RechercheMorceauParNum*
4) Affiche le morceau de titre *Titre* grâce à *RechercheMorceauTitre*
5) Affiche les morceaux de l'artiste *Artiste* grâce à *RechercheMorceauArtiste*
6) 7) 8) On calcule le temps CPU de ces trois fonctions de recherche sur 6 morceaux bien répartis dans la bibliothèque et un 7ème morceaux inventés de toute pièce pour tester le pire cas.
9) Écris dans un *fichier.txt* le temps des fonctions *uniques* en fonction du nombre de morceaux
10) Jeu de Test des différentes fonctions exigées par le sujet

Vous pouvez bien entendu modifier le jeu de test à votre guise si vous jugez que nous avons oublié des cas mais il faut savoir qu'il n'est pas facile d'établir un jeu de test cohérent pour les quatre structures. Certains tests peuvent vous sembler inutiles mais ils devraient l'être pour une structure différente.

Nous arrivons maintenant dans la partie où nous allons vous montrer les résultats de nos comparaisons.

Nous avons calculer le temps que prenait la recherche de 7 morceaux (numéros : 1/ 100 / 1 000 / 10 000 / 100 000/ 300 599 / 1 000 000) pour les trois fonctions de recherches. Pour que les tests soient pertinents, ils faut bien entendu charger les 300 600 morceaux de la bibliothèque.

Pour simplifiez la suite, nous appellerons la structure tableau dynamique → Tab_dynamique et table de hachage → Hachage.

Structure	Liste	Arbre	Tab_dynamique	Hachage
Temps en seconde de *RechercheMorceauNumero*	0.036428	0.184213	0.006282	0.187570
Temps exprimée en fonction du meilleur temps (arrondi au dixième)	5,8	29,3	1	29,9

Observations : On remarque que la fonction *RechercheMorceauParNum* est la plus efficace (toujours en terme de temps) pour le tableau dynamique ensuite pour la liste.

Structure	Liste	Arbre	Tab_dynamique	Hachage
Temps en seconde de *RechercheMorceauTitre*	0.010347	0.168740	0.006568	0.180292
Temps exprimée en fonction du meilleur temps (arrondi au dixième)	1,6	26,7	1	27,5

Observations : On remarque que la fonction *RechercheMorceauTitre* est beaucoup plus efficace avec le tableau dynamique ensuite pour la liste.

Structure	Liste	Arbre	Tab_dynamique	Hachage
Temps en seconde de *RechercheMorceauArtiste*	0.052879	0.006982	0.033754	0.006616
Temps exprimée en fonction du meilleur temps (arrondi au dixième)	8,0	1,1	5,1	1

Observations : On remarque que la fonction *RechercheMorceauArtiste* est la plus efficace pour le Hachage et pour Arbre.

(Il est préférable bien sûr de faire de faire une moyenne de plus de test mais nous ne sommes pas la pour faire des études en statistique, une réponse à la louche devrait suffire)

Interprétation: On remarque que pour parcourir toute la bibliothèque (dans le pire cas), il est préférable d'utiliser Liste ou Tab_dynamique car on utilise moins de pointeurs et de calculs pour parcourir TOUTE la bibliothèque par rapport à Arbre ou bien Hachage.

Cependant, lorsqu'il est question d'artiste, les structures Hachage et Arbres sont les plus rapide car respectivement pour l'un, on utilise le nom de l'artiste pour créer la clé qui va permettre d'aller directement dans la case du tableau de hachage, et pour l'autre, on insère les morceaux en fonction

de la chaîne de caractères du nom l'artiste pour ainsi connaître directement le chemin à prendre sans parcourir des nœuds inutiles de l'arbre.

Si la clé pour Hachage dépendait du titre et non par de l'artiste , on aurait eu des temps inférieurs à Liste et Tab_dynamique. Même principe pour les arbres.

Ce qu'on peut retenir de cette partie est que la recherche d'un élément est plus ou moins efficace en fonction de comment est organiser la structure. Ici, on nous a demandé de coder en fonction de l'artiste pour Arbre et Hachage donc la fonction de recherche par artiste est la plus rapide, mais pour généraliser, cela dépend bien évidemment du contexte dans lequel on souhaite utiliser la structure.

Pour répondre à la question 5.2, la fonction *collision* (juste un affichage du nombre de collisions intégré à l'affichage) nous a permis de conclure que plus la taille du tableau est grande, moins le nombre de collisions entre artiste (la clé étant trouvé par le nom de l'artiste) est important. Vous pouvez tester par vous même en jouant sur la taille du tableau de hachage.

Enfin, nous allons répondre à la question 5.4 en affichant les graphes des fonctions uniques des différentes en fonction du nombre d'éléments de la bibliothèque (en ayant utilisé l'option 9 du main).

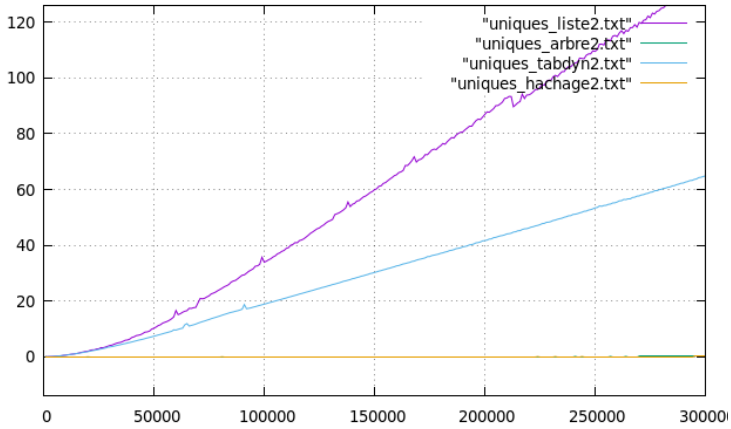

Figure 1

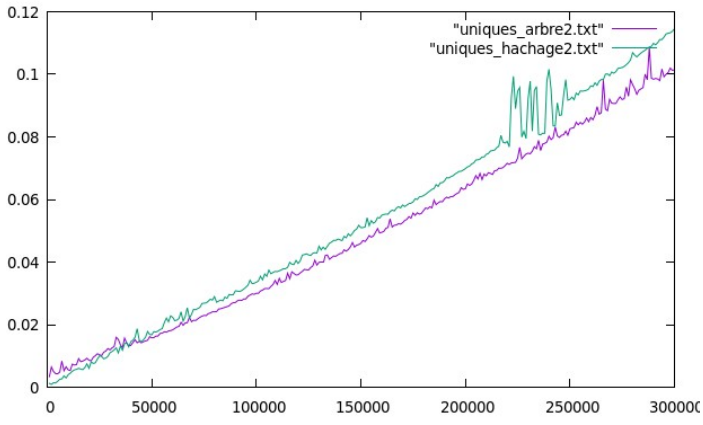

Figure 2

Sur la figure 1, nous avons affiché le temps en seconde des fonctions uniques en fonction du nombre d'éléments dans la bibliothèque. Les courbes d'Arbre et de Hachage sont superposées à cette échelle c'est pour cela que nous avons zoomé sur ces 2 courbes dans la figure 2.

Observations : Nous voyons effectivement que la structure Liste est la plus longue en terme de temps, ensuite celle de Tab_dynamique, et enfin nous pouvons mettre sur un pied d'égalité Arbre et Hachage.

Interprétation : D'après la complexité pire cas du code que nous avons écrit, pour Liste et Tab_dynamique, nous étions sur du $O(n^2)$. Cependant, le temps de Liste et de Tab_dynamique ne sont pas les mêmes car pour Tab_dynamique, on alloue de la mémoire contiguë grâce à *realloc* ce qui nous évite de passer par une succession de pointeurs comme pour Liste, d'où cette différence de temps.
Pour Arbre, nous appliquons la fonction *uniques* à chaque nœud de l'arbre. De plus, nous n'avons que besoin de comparer les titres car il n'y a que les morceaux d'un seul artiste par nœud. C'est pourquoi nous appliquons la fonction *uniques* seulement à une petite liste de morceaux donc la fonction *RechercheDoublons* s'appliquera sur une petite liste comparé à la structure Liste où on

devait parcourir à chaque fois toute la bibliothèque pour savoir si un doublon était présent. Donc finalement, il est normal que le temps parcouru soit fortement inférieur à Liste et Tab_dynamique. Nous sommes ici dans une complexité Θ(n).
C'est pratiquement la même situation pour Hachage, sauf que dans chaque case, il est possible d'avoir des collisions donc il faut quand même comparer le nom des artistes. Dans le pire des cas, les éléments de la table appartiennent à la même liste chaînée donc la fonction unique se fait en Θ(n). Cependant, vu que la mémoire alloué est contiguë, il est plus facile de se déplacer entre les différentes cases contrairement à Arbre qui est structuré par des pointeurs (Même principe que Tab_dynamique avec Liste).

Conclusion :

Ce TME permet de conclure que la structure à utiliser dépend en parti de l'utilisation prévue pour cette dernière (comme pour tout en informatique finalement). Dans notre cas, on voit bien que les structures Arbre et Hachage sont les plus efficaces pour les fonctions *RechercheMorceauArtiste* et *uniques* tandis que Liste et Tab_dynamique sont plus efficaces pour *RechercheMorceauNumero* et *RechercheMorceauTitre*. Cependant lors de la conception d'une bibliothèque comme celle-ci, il faut se poser la question de quelles fonctions seront les plus utilisées pour choisir la structure la plus adaptée à ces dernières.

On se demandait aussi pour optimiser la fonction clé de Hachage, s'il était possible d'élaborer une fonction clé qui prendrait en compte le titre et l'artiste, mais visiblement, cela rendrait les fonctions de recherches légèrement plus compliqué mais nous y réfléchirons...