Licence 2 Informatique de Sorbonne Université

LU2IN006 Structures de données

RAPPORT PROJET
TME 2-3-4

KERDOUCHE Nabil BOUYOUCEF Assirem

Groupe 1

18/03/2020

Introduction :

Il existe plusieurs façons d'organiser ou de structurer le stockage de ses données en informatique. Dans ce projet, on se concentre sur l'étude de différentes structures de données. Chaque structure a ses propriétés, donc des points forts et des points faibles. En fonction de nos besoins, nous devons analyser les différences entres ces structures pour identifier ce qui correspond le mieux. Le facteur le plus important est bien évidemment le temps.

PARTIE 1 : Description des structures de données utilisées et organisation du code (plus de détails : types des variables… voir les fichiers .h):

Globalement on stocke des morceaux de musique dans une structure appelée CellMorceau(composé d'un numéro ,un titre et un artiste) qui sont eux mêmes stockés dans une bibliothèque musicale (importées dans le fichier BiblioMusicale.txt) dans une structure appelée Biblio gérée de 4 façons différentes. Ce qui nous amène à manipuler 4 structures différentes :

1- Listes chaînées :

une liste d'éléments, chaque élément de cette liste comprend un morceau , un pointeur vers un élément suivant (voir fichier biblio_liste .h).

2- Arbres Lexicographiques :

les éléments sont stockés dans des nœuds structurés comme suivant : liste car qui correspond aux caractères possible horizontalement ,car suiv qui correspond au caractère suivant qui compose le nom de l'artiste (voir fichier biblio_arbrlex .h).

3- Tableaux dynamiques :

les éléments sont stockés dans un tableau avec une certaine capacité de stockage qui peut être modifiée a tout instant (voir fichier biblio_tabdyn .h).

4- Tables de hachage :

les éléments sont stockés dans une table de hachage à taille variable (m) (voir fichier biblio_hachage .h).

>La biblio contient toujours son nombre d'éléments (nE) .
> Toutes ces structures sont liées par un fichier biblio.c et donc son header biblio.h est importé dans le main (programme principal) afin de reconnaître toutes les signatures des fonctions utilisées qui sont les mêmes pour toutes les structures.

Les fonctions communes les plus importantes(voir fichier biblio.h) qui sont surtout utilisées pour les jeux de tests, donc pour étudier nos structures :
-charge_n_entrees: nous permet de lire un certain nombre de lignes dans un fichier texte en utilisant le parser (qui recupere des caracteres dans un fichier .txt).
-uniques : nous créer une bibliothèque qui ne contient pas de doublons
-rechercheParNum : permet de trouver un morceau par son numéro.

-rechercheParTitre : permet de trouver un morceau par son titre.
-rechercheParArtiste :permet de trouver un morceau par son artiste et son titre.
Nous calculerons donc le temps de calcul de ces fonctions pour chaque structure.

PARTIE 2 : analyse des structures :

Après études de ces structures grâce a notre fichier main, qui nous offre un menu polyvalent nous permettons de tester les fonctionnalités de nos structures et leurs optimalités.

-Aiste chaînée :

Elle nous permet un gain de mémoire grâce a ça flexibilité, vu que la mémoire d'un élément est allouée lors de son insertion cependant l'accès aux éléments est difficile, elle nous impose de parcourir la liste jusqu'au morceau souhaité et une maniabilité restreinte .

-Arbre lexicographique :

Elle nous permet une organisation des morceaux par leurs artistes idéal si l'ont veut retrouver tout les morceau d'un meme artiste cependant elle consomme énormément de mémoire vu le nombre de nœuds nécessaires créés pour organiser la structure et elle reste assez compliqué a modéliser et a utiliser.

-Tableau dynamique :

Il nous offre un accès simple a tout les éléments. Facile a manipuler , consomme moins qu'un tableau classique vu que sa taille s'adapte aux besoins mais reste très peu flexible et peu économe en mémoire par rapport a une liste chaînée .

-Table de hachage :

chaque artiste a une clé calculer a partir du code ASCII des caractères de son nom qui passe ensuite par une fonction de hachage qui renvoie son indice dans la table ce qui nous permet un accès presque direct aux morceaux de l'artiste. Gourmande en mémoire et on peu être confronté a des collisions (pas très grave) si la fonction de hachage n'effectue pas une répartition uniforme

PARTIE 3 : test de complexité, recherche d'un élément :

D'après les tests effectués avec les jeux de tests, pour 100000 morceaux lus dans la Biblio (disponible dans le menu du main « option 6 ») on peut tirer quelques conclusions cependant ils ne faut pas prendre comme vérité absolue ces tests car ils ont été effectue une seule fois sur la recherche d'un seul morceau car le facteur du hasard joue un rôle important dans cette partie. Les vrais résultats à prendre en compte sont ceux de la prochaine partie (uniques) vue que les fonctions de recherche sont exécutées plusieurs fois. Donc pour l'instant les conclusions sont provisoires .
Voici les résultats pour la recherche du morceau : (voir fichier ex5.txt)
num=1026; titre=The Purple People Eater Meets The Witch Doctor; artiste=Joe South
-Fonction rechercher par num :

liste : temps_rechercheParNum= 0.006351
tabdyn : temps_rechercheParNum= 0.000007 (s)
table de hachage : temps_rechercheParNum= 0.010799 (s)

arbre lexicographique : 0.000778

on peut conclure que la recherche par num est optimisées pour le tableau dynamique.

-Fonction rechercher par titre :

liste : temps_recherchePartitre= 0.000157 (s) (s)
tabdyn : temps_recherchePartitre= 0.000058 (s)
table de hachage : temps_recherchePartitre= 0.013511 (s)
arbre lexicographique : 0.000073

on peut conclure que pour la liste et le tableau dynamique la rechercher est rapide mais elle est plus lente sur la table de hachage.

-Fonction rechercher par artiste (extraire morceaux d'un artiste):

liste : temps_rechercheParartistre= 0.005739 (s) (s)
tabdyn : temps_rechercheParartiste= 0.007537 (s)
table de hachage : temps_rechercheParartiste = 0.001070 (s)
arbre lexicographique : 0.012364

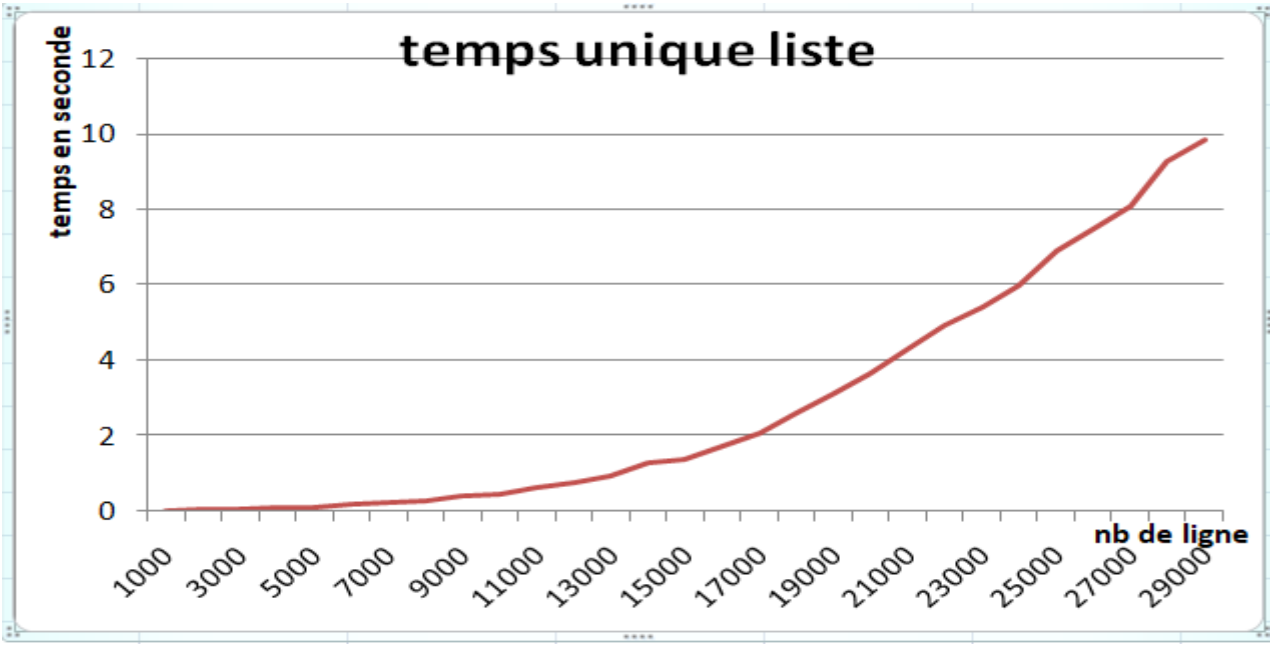
on remarque que dans la table de hachage on arrive a trouvé le morceau 5 a 7 fois plus rapidement que sur les autres structures.

PARTIE 4 : test de complexité, recherche de plusieurs éléments :

Maintenant testons pour 29000 morceaux lus dans la Biblio la fonction uniques (« option 7 » du main) qui utilise les fonctions de recherche suffisamment de fois pour considérer que les résultats sont très fiables :
pour cela on s'intéresse juste, pour un nombre de morceaux fixe=29000. Résultats :
-liste : 9,8s
-tableau dynamique : 6.25s
-table de hachage : 0,16s
-arbre lexicographique :
On remarque bien que la table de hachage est très rapide par rapport aux autres structures. C'est logique voici comment elle procède : après quelques instructions itératives dont le calcul de la clé, le hash, on a directement l'indice de la petite liste qui contient l'élément recherché dans la table de hachage. Il ne reste plus qu'a effectuer une simple recherche sur une très courte liste chainée (pourvu que la fonction de hachage est efficace*).
*fonction de hachage efficace : permet une distribution uniforme dans la table, donc évite au maximum les collisions. Dans notre cas l'idéal est d'avoir un artiste par indice du tableau.

PARTIE 4 : Efficacité sur un grand volume de données :

Utilisons l'option 7 du main pour voir le temps de calcul en fonction du volume de données correspondant au nombres de lignes lues (1 ligne = 1 Morceau)



Liste chaînée :

Algo : Fixer un élément, parcourir le reste de la liste à la recherche d'un doublon, si trouvé : on passe, si pas de doublon, on insere l'élément et on continue avec le suiv. Dans le pire cas, tous les éléments sont différents. Donc a chaque fois on parcours la liste jusqu'a la fin. Complexité = O (n*n)

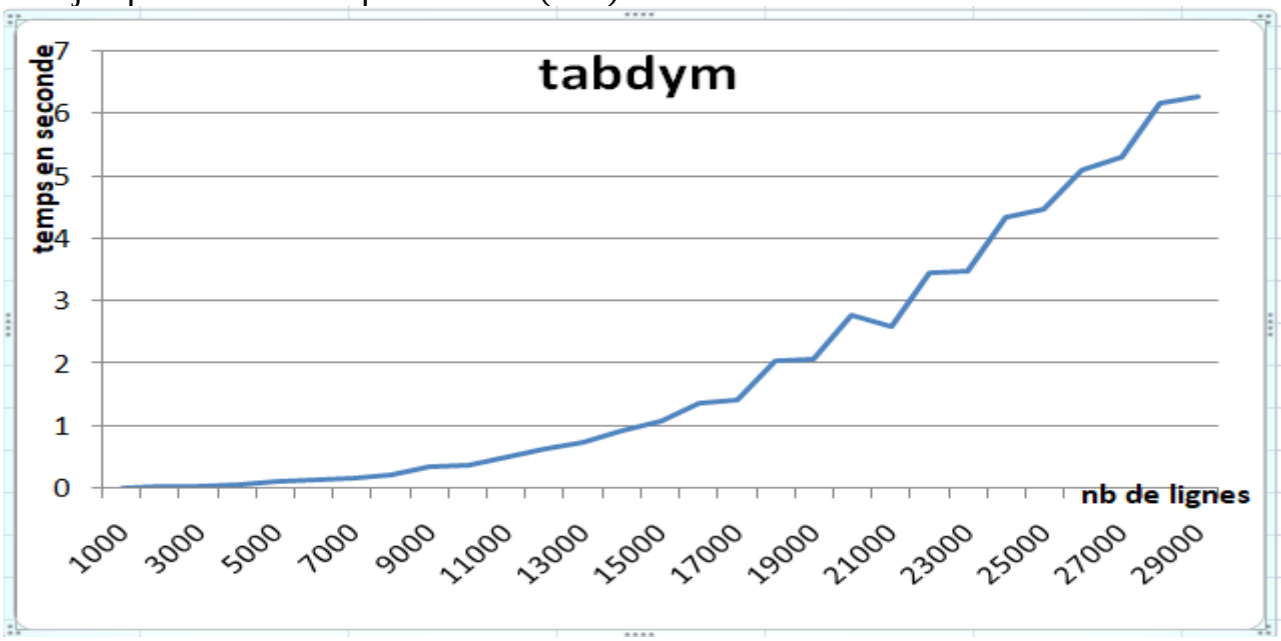


Tableau dynamique :

Algo : Fixer un élément, parcourir le reste du tableau à la recherche d'un doublon, si trouvé : on passe, si pas de doublon, on insère l'élément et on continue avec le suiv. Dans le pire cas, meme chose que la liste (si tous les éléments différents). Mais on remarque un petit gain de temps due à l'accès direct facile au cases du tableau, mais cela est payé par la mémoire consommée par les cases vides.

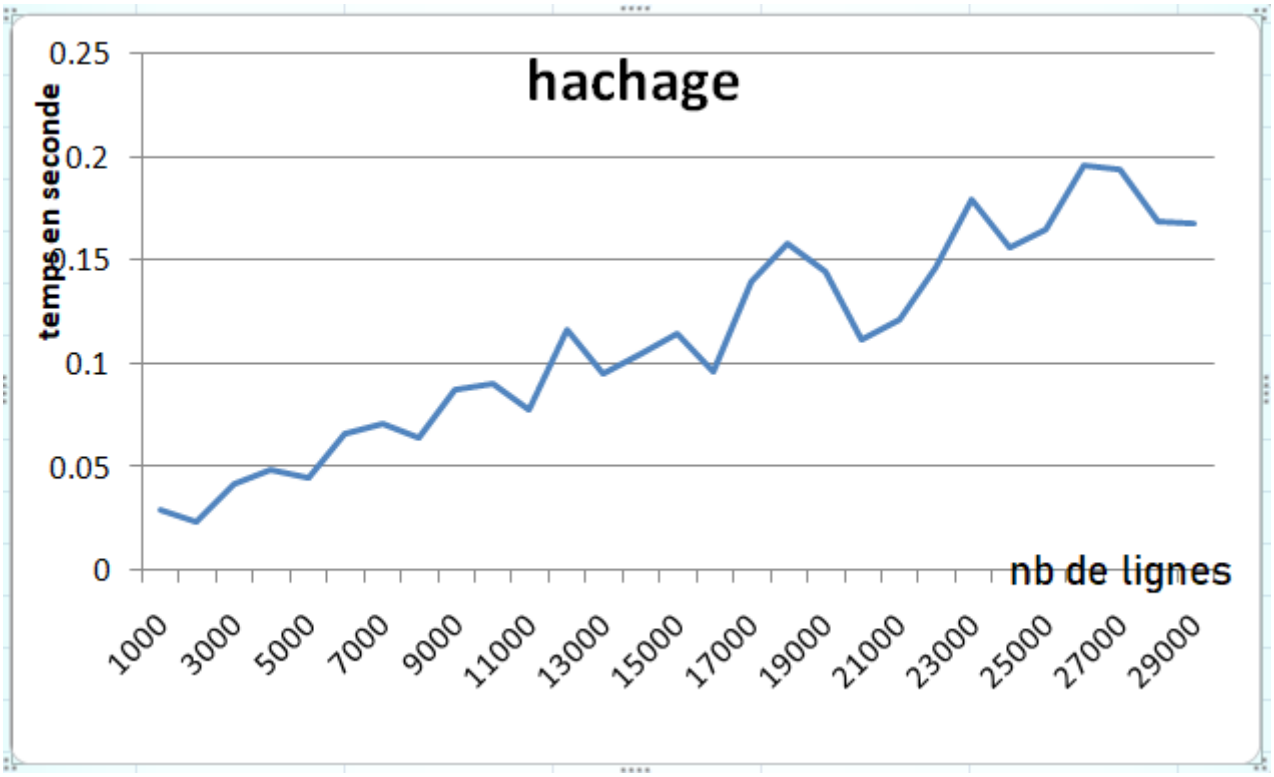


Table de hachage :
Algo : Calcul de la clé de l'artiste, de son hash, et recherche dans la liste correspondante à l'indice.
Dans le pire cas : tous les éléments sont insérés dans la même case de la table ! Ça devient donc une liste chaînée. Mais bien sure cela n'arrivera jamais avec une bonne fonction de hachage. Ici l'accès est presque direct et ne dépend pas trop du volume de données (voir courbe) tant que la taille de la table est assez grande.
>>>Concernant le volume de données, les tables de hachage sont les plus efficaces.

PARTIE 4 : Efficacité de la table de hachage par rapport à sa taille :

Taille de la table	rechercheParNum	rechercheParTitre	rechercheParArtiste
60	0.010799 s	0.013511 s	0.001070 s
6000	0.005627 s	0.006103 s	0.000933 s
600000	0.005627 s	0.006103 s	0.000821s

>>La taille de la table de hachage est bien sure proportionnelle à son efficacité. (sans prendre en compte la mémoire consommée).

Conclusion:

L'utilisation d'une structure de données dépend pleinement des besoins de son utilisateur. Si par exemple un grand volume de données doit être stocké, on favorise la table de hachage. Pour une économie en mémoire, c'est plutôt la liste chaînée. Pour équilibrer un peu les deux il y a les tableaux. Si par exemple l'ordre alphabétique est important, l'arbre lexicographique est aussi idéal. Il n'existe pas de structure parfaite !