**Rapport TME2-3-4**

**Binôme : Nahidath GASSA et Clara CIOCAN**

Le TME2-3-4 est un mini-projet consistant à manipuler une bibliothèque musicale contenant plusieurs morceaux. Un morceau est repéré par son titre, le nom de l’artiste qui l’interprète et un numéro d’enregistrement. Les fichiers nécessaires que nous avions utilisés pour ce mini-projet sont BiblioMusicale.txt, un fichier texte contenant tous les morceaux avec les artistes qui les interprètent, et parser.c (et son .h), une bibliothèque permettant de lire un fichier selon un format avec un caractère de délimitation.

Le but de ce mini-projet est de nous apprendre à implémenter des structures de données telles qu’une liste simplement chaînée, un arbre lexicographique, etc…

Notre code s’organise en 4 parties dans lesquelles on retrouve différents fichiers .c, .h, un Makefile, des fichiers test et un main composé d’un menu qui nous permet de manipuler la bibliothèque.

1. ***FICHIERS UTILISÉS :***
2. **main.c** : ce fichier inclut le main du programme, ainsi que le menu à l’aide duquel nous pouvons avoir accès a la bibliothèque. Nous avons créé un menu avec 10 choix possibles (Sortir du programme, Afficher la liste, Le calcul de la bibliothèque unique, Les trois fonctions de recherche, Supprimer un morceau, ainsi que le jeu de test qui nous permet d’afficher et comparer le temps de recherche pour chaque structure). Le main se finit en libérant la bibliothèque pour ne pas avoir de fuite de mémoire.
3. **Biblio.c** : ce fichier contient la seule fonction qui peut être utilisée indépendamment du type de structure, ayant le même corps et la même définition- **Biblio \*charge\_n\_entrees (const char \*nomfichier, int n)**. Elle permet de lire les champs (morceaux) du fichier BiblioMusicale.txt et de les stocker dans une nouvelle bibliothèque **B**.
4. **Biblio.h** : ce fichier contient les prototypes de toutes les fonctions utilisées dans le main pour la manipulation de la bibliothèque musicale. Biblio.h permet de faciliter la construction et de mieux organiser les différentes parties du projet. Cette bibliothèque inclue, donc, les fonctions :
   1. **Biblio \*nouvelle\_biblio()**  fonction qui gère la création d’une nouvelle bibliothèque ;
   2. **Biblio \*charge\_n\_entrees (const char \*nomfichier, int n)**permet de récupérer les morceaux écrits dans un fichier.
   3. **CellMorceau \*rechercheParNum(Biblio \*B, int num)** permet de rechercher un morceau à partir de son numéro.
   4. **CellMorceau \*rechercheParTitre(Biblio \*B, char\* titre)** permet de rechercher un morceau à partir de son titre.
   5. **Biblio \*extraireMorceauxDe(Biblio\* B, char \*artiste)** permet de rechercher tous les morceaux d’un artiste en les stockant dans une nouvelle bibliothèque.
   6. **void supprime\_morceau(Biblio \*B, int num)** permet de supprimer un morceau de la bibliothèque à partir de son numéro.
   7. **void libere\_biblio(Biblio \*B)** permet de liberer la mémoire occupe par une bibliotheque .
   8. **void affiche(Biblio \*B)** permet de afficher tous les morceaux d’une bibliotheque.
   9. **void unique\_graphe(int borne\_inf, int pas, int borne\_sup)** permet de generer un fichier avec le temps nécessaire pour la fonction unique, en fonction du nombre de morceaux enregistres dans la bibliothèque. Le but de cette fonction est de pouvoir obtenir des graphes pour mieux visualiser le fonctionnement et l’efficacité de chaque type de structure.
5. **Biblio\_liste.c /Biblio\_arbrelex.c/Biblio\_tabdyn.c/Biblio\_hachage.c:** ces fichiers contiennent les définitions des fonctions décrites ci-dessous, fonctions qui servent à pouvoir manipuler une bibliothèque stockée dans une liste chainée. Elles sont différemment définies en fonction de la structure de donnée utilisée pour manipuler la bibliothèque. Nous avons détaillé ces fonctions dans la partie suivante du rapport.
6. **Biblio\_liste.h /Biblio\_arbrelex.h/Biblio\_tabdyn.h/Biblio\_hachage.h:** ces fichier .h contiennent les déclarations des structures CellMorceau et Biblio, ainsi que différentes fonctions utilisées dans le fichier biblio\_liste.c .
7. **Makefile :** le makefile que nous avons créé permet de générer des exécutables distincts pour chaque structure de données utilisée. Ainsi, nous avons 4 fichiers exécutables : Biblio liste, Biblio arbre, Biblio\_tabdyn et Biblio\_hachage. Nous avons aussi ajoute la règle « all » qui permet de regrouper l’ensemble des exécutables à produire, et aussi la règle « clean » pour pouvoir supprimer tous les fichiers intermédiaires –o et les exécutables.
8. ***LES STRUCTURES DE DONNÉES :***
9. ***LISTE CHAÎNÉE***

La première partie de ce mini-projet a commencé par le codage d’une bibliothèque comme liste chaînée. Les structures utilisées sont **Biblio**, la structure qui contient le nombre de morceaux et un pointer vers la liste chainée, et **CellMorceau**, qui regroupe le num, artiste et titre d’un morceau, mais aussi, spécifique a une liste chainée, un pointer vers l’élément suivant de la liste.

|  |
| --- |
| CellMorceau \*suivi |
| char \*titre |
| char \*artiste |
| int num |

|  |
| --- |
| CellMorceau \*suiv |
| char \*titre |
| char \*artiste |
| int num |

|  |
| --- |
| CellMorceau \*suiv |
| char \*titre |
| char \*artiste |
| int num |

LES FONCTIONS :

* **Biblio \*nouvelle\_biblio(void)** qui permet de créer une bibliothèque vide que nous utiliserons après. La fonction alloue l’espace mémoire nécessaire pour enregistrer la bibliothèque et initialise les valeurs du premier élément de la liste a NUL et le nombre de morceaux a 0.
* **void insere(Biblio \*B, int num, char \*titre, char \*artiste)** permet d’insérer un morceau dans notre bibliothèque. Elle est base sur la fonction d’insertion en tête de la liste chainée. En plus d’insérer le nouveau morceau dans la liste, la fonction incrémente aussi le nombre de morceaux existants dans la bibliothèque.

Pour ne pas avoir de fuite de mémoire, nous sommes obliges à libérer l’espace allouée par la création de la bibliothèque et de nouveau CellMorceau.

* **Void** **libere\_CellMorceau (CellMorceau \*c)** permet de libérer la mémoire occupée par la liste chainée.
* **void libere\_biblio(Biblio \*B)** utilise la fonction d’avant pour des allouer le reste de la mémoire occupée par la bibliothèque.
* **void affiche(Biblio \*B)** parcourt la liste case par cas et affiche le num, le titre et l’artiste de chaque morceau.
* **CellMorceau \*rechercheParNum (Bbiblio \*B, int num)** gère la recherche d’un morceau par son numéro. Cette fonction parcourt la liste case par case jusqu’au moment où la case court a le même numéro que le numéro cherche. Si c’est le cas, on affiche le morceau en entier, en renvoyant en même temps un pointer vers la case correspondante. En cas contraire, on affiche un message d’erreur, la fonction retournant NUL. En considérant que la fonction parcourt la liste une fois, sa complexité est : O(n), ou n est la longueur de la liste.
* **CellMorceau \*rechercheParNum (Biblio \*B, int num)** est basée sur le même principe que la fonction de recherche par numéro. En considérant qu’un titre peut apparaitre plusieurs fois dans la liste, on renvoie seulement la sa première occurrence. Comme la fonction de recherche par numéro, cela a aussi une complexité O(n).
* **Biblio \*extraireMorceauxDe (Biblio\* B, char \*artiste)** est la fonction qui gère la recherche de tous les morceaux d’un artiste. Ainsi, la fonction crée une nouvelle bibliothèque ou on insere chaque morceau qui corresponde à l’artiste recherchée. Si, après avoir parcouru tous les morceaux, la liste de la nouvelle bibliothèque est vide, cela signifie que l’artiste ne se trouve pas dans la bibliothèque et, par conséquence, on affiche un message d’erreur. Étant basée sur les mêmes principes, cette fonction aussi parcourt la liste une fois, et a donc la même complexité que les deux autres.
* **Void supprime morceau (Biblio \*B, int num) gère** la suppression d’un morceau par son numéro. Nous avons trouvé deux cas possibles : soit le morceau recherche se trouve dans la tête de la liste, et alors la suppression consiste d’une simple affectation sur le pointer B->L ; soit le morceau se trouve à l’intérieur de la liste. On avance dans la liste jusqu'au moment où on trouve le morceau avec le numéro num ou jusqu'à la fin de la liste. Si on a parcouru toute la liste, cela signifie que le numéro num ne se trouve pas dans la bibliothèque et on affiche un message d’erreur. À la fin on libère la mémoire utilisée par le morceau supprimé.
* **Biblio \*uniques (Biblio \*B)** renvoie une bibliothèque sans doublons, ou chaque morceau se trouve une seule fois. Cette fonction est basée sur l’algorithme qui élimine les doublons d’une liste chainée. Ainsi, on parcourt la liste avec deux pointers : curr et next. Le pointer curr est le pointer qui parcourt la liste case par case jusqu’à la fin. Next parcourt la sous-liste qui commence à curr jusqu’à la fin, et on avance dans la liste en comparant les morceaux. S’ils sont différents, on avance avec le pointer next. Si, à la fin de la deuxième boucle, next est NULL, cela signifie qu’on est arrivé à la fin de la liste sans avoir trouvé de doublon, et donc on peut insérer le CellMorceau de curr dans la nouvelle bibliothèque. Cette fonction parcourt la liste avec deux boucle white et a une complexité de O (n log(n)).

1. ***ARBRE LEXICOGRAPHIQUE***

|  |
| --- |
| Char \*c |
| Nœud\*liste car |
| Nœud\*car\_suiv |
| CellMorceau \*liste\_morceau |

La deuxième partie consiste à manipuler la bibliothèque musicale comme un arbre lexicographique. Dans notre fichier **biblio\_arbrelex.h**, nous avons déclaré, en plus de la liste chaînée **CellMorceau**, la structure specifique **Nœud** qui est la cellule de l’arbre lexicographique et une structure **Biblio** qui contient l’arbre lexicographique, aussi que le nombre de morceaux.

|  |
| --- |
| Char \*c |
| Noeud\*liste\_car |
| Noeud \*car\_suiv |
| CellMorceau \*liste\_morceau |

|  |
| --- |
| Char \*c |
| Noeud\*liste\_car |
| Noeud\*car\_suiv |
| CellMorceau \*liste\_morceau |

|  |
| --- |
| Char \*c |
| CellMorceau \*liste\_car |
| CellMorceau \*car\_suiv |
| CellMorceau \*liste\_morceau |

|  |
| --- |
| CellMorceau \*suiv |
| char \*titre |
| char \*artiste |
| int num |

Prenant en considération que la structure est un arbre, l’implémentation des fonctions dans le fichier biblio.h, a signifié aussi la création des fonctions récursives pour pouvoir gérer le parcours nœud par nœud de l’arbre. Ainsi, chaque fonction de base du projet utilise des fonctions spécifiques a cette structure :

* **Void insere\_morceau (Noeud \*\*noeud, int num, char \*titre, char \*artiste) :** la fonction corresponde à une fonction d’insertion en tête d’une liste gère par le pointer liste morceau défini dans la structure Nœud. La fonction insere un nouveau morceau dans la liste associée au dernier caractère du nom de l'artiste
* **Void insere (Biblio \*B, int num, char \*titre, char \*artiste)** insère un morceau dans la bibliothèque en parcourant chaque caractère du nom de l’artiste. La fonction essaie de parcourir premièrement l’arbre en profondeur, jusqu’au moment où il n’y a pas de caractères dans l’artiste, ou jusqu’au moment ou un parcours en profondeur n’est plus possible. Dans le premier cas, la fonction va faire appel à la fonction **insere\_morceau** qui, lors du parcours en profondeur, insère le morceau dans la liste gère par le pointer liste morceau. Si le caractertere de l’artiste ne correspond pas au nœud de car\_suiv on essaie le parcours en largeur avec le pointer liste\_car.

L’intérêt de décomposer la partie **insertion** en deux fonctions est de gérer les deux structures différentes qui composent l’arbre : le nom de l’artiste, gère par les nœuds et donc la fonction **insere** et les morceaux de l’artiste c’est-à-dire la liste chainée gérée par la fonction **insere\_morceau**.

* **Void libere\_biblio (Biblio \*B)** utilise les fonctions **void libere\_CellMorceau (CellMorceau \*c)** (libère l’espace alloue par une liste chainée de CellMorceau) et **void libere\_Noeud (Nœud \*nœud)** (fonction récursive qui parcourt l’arbre en profondeur et largeur et qui utilise libere\_CellMorceau) pour pouvoir libérer l’espace mémoire utilisée par la bibliothèque.
* Les fonctions **CellMorceau \*rechercheParNum (Biblio \*B),** **CellMorceau \*rechercheParTitre (Biblio \*B)** et **Biblio \*uniques (Biblio \*B)** se décomposent en 3 fonctions : une fonction qui recherche le morceau au niveau de la liste chaîné pour les deux premières, respectivement une fonction qui s’occupe de l’unicité dans une liste chainée (comme dans la partie 1. Liste Chainée), une fonction récursive qui parcourt l’arbre en profondeur et en largeur et une fonction principale de recherche reprenant les fonctions précédentes.
* La recherche de tous les morceaux de l’artiste se fait par la fonction **Biblio \*extraireMorceauxDe(Biblio \*B, char \*artiste)** qui revoie une bibliothèque les contenant. À partir de chaque caractère du nom de l’artiste, la fonction parcourt l’arbre premièrement en profondeur, et si cela n’est plus possible, en largeur, jusqu’à la fin du nom de l’artiste. Le dernier pas est d’insérer les morceaux de la liste liste\_morceau dans la nouvelle bibliothèque.

1. ***TABLEAU DYNAMIQUE***

La troisième partie consiste à gérer une bibliothèque avec un tableau dynamique. L’utilisation d’un tableau dynamique a une bonne complexité en terme de mémoire, en nous permettant de modifier la taille de tableau, en réallouant d’espace mémoire pour stocker plusieurs morceaux.

Notre fichier **biblio\_tabdyn.h** contient la structure **CellMorceau** et la structure **Biblio** qui contient elle-même la taille max du tableau, le nombre de morceau dans le tableau et le tableau de morceau.

*LES FONCTIONS :*

Dans notre fichier **biblio\_tabdyn.c**, nous avons implémenté les fonctions suivantes :

* **void insere(Biblio \*B, int num, char \*titre, char \*artiste)** permet d’insérer un morceau dans notre bibliothèque. Vu qu’il s’agit d’un tableau dynamique, on se retrouve avec plusieurs cas : s’il y a encore de place dans le tableau, la fonction fait une simple insertion dans la première case nulle (disponible). Dans le cas où le tableau est déjà rempli et arrive à sa capacité maximale, on double sa capacité et on fait un appel récursif a la fonction même pour insérer le morceau dans la nouvelle case crées.

Pour ne pas avoir de fuite de mémoire, nous sommes obligés à libérer l’espace allouée par la création de la bibliothèque et de nouveau CellMorceau.

* **Void** **libere\_CellMorceau (CellMorceau \*c)** permet de libérer la mémoire occupée par un CellMorceau
* **Void libere\_biblio (Biblio \*B)** parcourt le tableau en desallouant la mémoire à l’aide de la fonction définie précédemment et se finit par la desallocations de la bibliothèque.
* **void affiche(Biblio \*B)** parcourt la liste case par case, vérifie s’il y a de morceaux dedans et affiche le num, le titre et l’artiste de chaque morceau.
* **CellMorceau \*rechercheParNum (Biblio \*B, int num),** **CellMorceau \*rechercheParNum (Biblio \*B, char \*Tite)** et **Biblio \*extraireMorceauxDe (Biblio\* B, char \*artiste) gèrent** la recherche d’un morceau par son numéro, par son titre, respectivement par son artiste. Ces fonctions ont le même principe de fonctionnement ; elles parcourent la liste case par case jusqu’au moment où tous les morceaux ont été vérifiés ou jusqu’à la fin du tableau. Les fonctions vérifient pour chaque case non vide si le num/titre/artiste corresponde à ce qu’on cherche. Si c’est le cas, on affiche le morceau en entier, en renvoyant en même temps un pointer vers la case correspondante, ou la bibliothèque contentant les morceaux de l’artiste pour le cas de extraireMorceauDe. En cas contraire, on affiche un message d’erreur, la fonction retournant NULL. En considérant que la fonction parcourt la liste une fois, sa complexité est : O(n).
* **Void supprime morceau (Biblio \*B, int num) gère** la suppression d’un morceau par son numéro. Comme pour les fonctions d’avant, celle-ci aussi parcourt le tableau liste case par case jusqu’au moment où tous les morceaux ont été vérifiés ou jusqu’à la fin du tableau. Au contraire aux autres fonction de suppression, celle-ci ne libère pas l’espace mémoire lorsqu’on supprime un morceau.
* **Biblio \*uniques (Biblio \*B)** renvoie une bibliothèque sans doublons, ou chaque morceau se trouve une seule fois. Cette fonction est basée sur l’algorithme qui élimine les doublons d’tableau. Ainsi, on parcourt la liste avec deux boucles white qui vérifient à chaque fois si les cases sont vides ou pas. En comparant le titre et l’artiste de l’index courant avec tous les autres morceaux jusqu’à la fin de la liste on vérifie si le morceau va apparaitre encore une fois dans le tableau. Si c’est le cas, on l’insere pas dans la bibliothèque finale.

**4.TABLE DE HACHAGE**

La quatrième partie consiste à utiliser une table de hachage pour gérer une bibliothèque. Les tables de hachage permettent de trouver l’élément que l’on cherche en un temps constant.

Dans notre fichier **biblio\_hachage.h**, nous avons implémenté les mêmes structures que dans la troisième partie mais en rajoutant une clé dans **CellMorceau**. Cette clé va nous permettre, à l’aide une fonction d’hachage, d’associer une valeur numérique à chaque morceau dans la table de hachage.

Ainsi, pour pouvoir vérifie l’efficacité de la fonction clé, nous avons aussi créé une fonction qui compte le nombre de collisions dans un tableau de hachage.

|  |
| --- |
| CellMorceau \*suiv |
| int clé |
| char \*titre |
| char \*artiste |
| int num |

|  |
| --- |
| CellMorceau \*suiv |
| int clé |
| char \*titre |
| char \*artiste |
| int num |

|  |
| --- |
|  |
|  |
|  |
|  |
|  |
|  |

|  |
| --- |
| CellMorceau \*suiv |
| int clé |
| char \*titre |
| char \*artiste |
| int num |

*LES FONCTIONS*

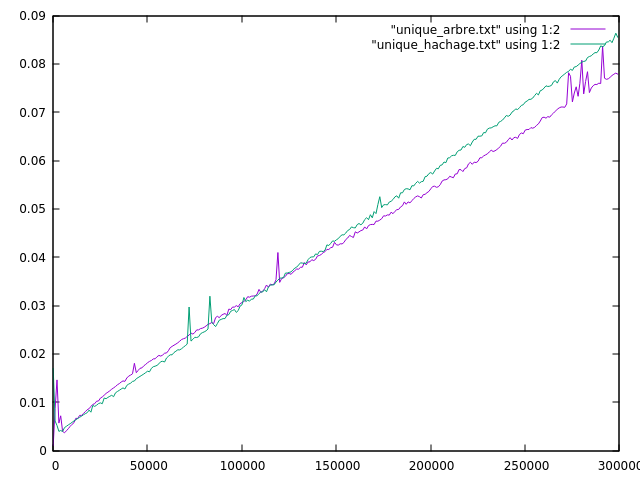
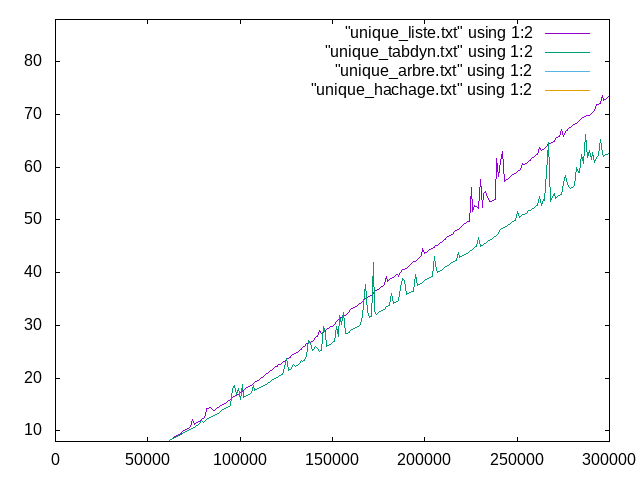
* **Int fonctionc\_cle (char \*c) :** cette fonction permet d’associer un numéro a chaque cellMorceau introduit dans la table de hachage. La fonction est très importante, car elle est un facteur qui détermine le nombre de collisions dans le tableau. Ainsi, après avoir implémente la première fonction clé donne dans l’annonce du projet, nous avons décidé de trouver des fonctions plus efficaces : un première essaie était une fonction qui prend en compte aussi l’ordre des caractères dans le nom de l’artiste : artiste[i]\*(i+1). Une deuxième tentative nous a conduit à la fonction djb2 qui réduit encore plus le nombre de collisions.
* **Int fonction hachage (insigne long clé, int m) est** la fonction qui transforme la clef en une valeur entière entre 0 et m.
* **Void insere (Biblio \*B, int num, char \*titre, char \*artiste)** gère l’insertion d’un morceau dans la bibliothèque. Avec la fonction de hachage et la fonction clé on trouve pour chaque morceau l’index ou il faut l’insérer dans le tableau. Ainsi, si la case du tableau est NUL on crée on morceau et on l’insere directement. Sinon, on Isère le nouveau morceau en tête d’une liste chainée qui gère les collisions.
* **CellMorceau \*rechercheParNum (Biblio \*B, int num),** **CellMorceau \*rechercheParNum (Biblio \*B, char \*titre) gèrent** la recherche d’un morceau par son numéro et par son titre. Ces fonctions ont le même principe de fonctionnement ; elles font aussi appel aux fonctions rechercheNumCellMorceau et rechercheTitreCellMorceau qui font la recherche dans une liste chainée. Les fonctions parcourent la table de hachage case par case jusqu’au moment où tous les morceaux ont été vérifiés ou jusqu’à la fin du tableau. Les fonctions vérifient pour chaque case non vide si le la fonction hachage corresponde à la fonction hachage du morceau recherché c’est le cas, on affiche le morceau en entier, en renvoyant en même temps un pointer vers la case correspondante, ou la bibliothèque contentant les morceaux de l’artiste pour le cas de extraireMorceauDe. En cas contraire, on affiche un message d’erreur, la fonction retournant NULL. En considérant que la fonction parcourt la liste une fois, sa complexité est : Thêta(m).
* **Biblio \*extraireMorceauxDe (Biblio\* B, char \*artiste)** renvoie la bibliothèque contenant les morceaux d’un certain artiste. Ayant déjà le nom de l’artiste, on trouve facilement l’index de la case à parcourir dans la table de hachage. On vérifie aussi, pour chaque valeur de la liste chainée qui gère les collisions si le nom correspond et si c’est le cas on insere le morceau dans la bibliothèque. Cette fonction a une complexité de Thêta(1+n/m).
* **Biblio \*uniques (Biblio \*B)** renvoie une bibliothèque sans doublons, ou chaque morceau se trouve une seule fois. Cette fonction parcourt le tableau case par case et pour chaque liste chainée, en cas de collision, elle fait les mêmes étapes que la fonction unique de la liste chainée. Ainsi, dans le pire cas, la complexité de cette fonction est O(n).

1. **EFFICACITÉ, TEMPS DE CALCULS ET CONCLUSIONS**

Les quatre structures que nous avons manipulées (liste chaînée, arbre lexicographique, tableau dynamique et table de hachage) vont nous permettre de comparer le temps nécessaire pour effectuer les fonctions de recherche. Pour cela nous avons testé le temps de recherche de chaque structure et nous avons trouvé les valeurs suivantes :

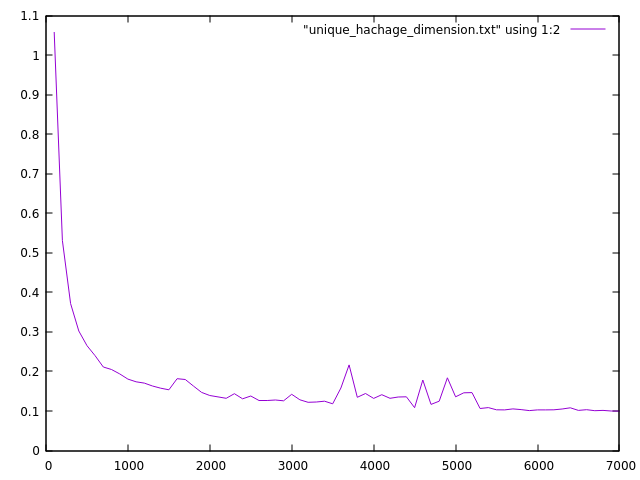
|  |  |  |  |
| --- | --- | --- | --- |
|  | Recherche Num | Rehcerche Titre | Recherche Artiste |
| Biblio\_liste | 0.032 | 0.018 | 0.050 |
| Biblio\_arbre | 0.010 | 0.008 | 0.013 |
| Biblio\_tabdyn | 0.16 | 0.08 | 0.083 |
| Biblio\_hachage | 0.183 | 0.14 | 0.010 |

Nous avons aussi modélisé les temps de recherche des morceaux uniques en fonction de taille de la bibliothèque et de la structure de données utilisée par des graphes. Ces temps de recherche sont sauvegardés dans les fichiers **unique\_arbre.txt, unique\_hachage.txt, unique\_liste.txt et unique\_tabdyn.txt** grâce à la fonction **void unique graphe(int borne\_inf, int pas, int borne\_sup)** qui nous a permis de récupérer toutes les valeurs de temps. Ensuite, avec les valeurs récupérées nous avons tracé les courbes de chaque structure :



-les 4 structures, avec arbre et hachage ~0- -arbre et hachage-

On peut aussi observer que la efficacite de la table de hachage depend aussi de sa taille :



Les complexités pire-cas de chaque structure pour la fonction unique sont :

* Structure liste chaînée :
* Structure tableau dynamique :
* Structure arbre lexicographique :
* Structure table de hachage :