Projet : Structure de données

(TME 2-3-4)

Felix Kassel & Gabour Smaïl

3702385 - 3700149

INTRODUCTION

Nous nous intéressons à la gestion d'une bibliothèque musicale, que nous modifierons grâce à quatre structures de données différentes les unes des autres, que nous allons implémenter dans notre code :

- Tableau dynamique
- Liste chaînée
- Arbre lexicographique
- Table de hachage

L'objectif est de comparer la complexité et l'efficacité de ces 4 structures par le biais d'une fonction qui calculera le temps d'exécution pour chaque structure.
Ces 4 structure seront implémenter par des algorithmes, à savoir la recherche d'un élément ou encore l'insertion d'un élément dans notre bibliothèque musicale.
Pour chaque structure nous présenterons les fonctions associés et discuterons sur leurs efficacité et leurs complexité.

À SAVOIR

Chaque structure se voit implémenter une bibliothèque de fonctions :

- la recherche d'un morceau par son numéro
- la recherche d'un morceau par son titre
- la recherche de tous les morceaux d'un même artiste
- l'insertion d'un nouveau morceau - la suppression d'un morceau
- la recherche des morceaux qui n'ont pas de doublon.

Le *main* gère le fonctionnement du programme, ainsi il est possible d'utiliser chaque fonction cité ci-dessus via la ligne de commande suivant :

./bibli_<liste, arbrelex, tabdyn, hachage> BiblioMusicale.txt <nombre de morceaux charges>

1 Les structures

1.1 Bibliothèque musicale

Dans ce projet, nous nous intéressons `a la gestion d'une bibliothèque musicale composée de morceaux. Un morceau est repéré par son titre, le nom de l'artiste qui l'interprète et un numéro d'enregistrement. Plus précisément, un morceau est représenté par les données suivantes :

```
1  int num;
2  char *titre;
3  char *artiste;
```

1.2 Liste Chaînée

Pour la première structure, nous implémenterons la bibliothèque musicale sous forme de liste simplement chaînée. Pour cela, on séparera la bibliothèque en deux structures distinctes, l'une sous forme de cellule qui pointe sur le morceau suivant et l'autre qui contient une liste de morceaux :

```
1  typedef struct CellMorceau {
2    struct CellMorceau *suiv;
3    int num;
4    char *titre;
5    char *artiste;
6  } CellMorceau;
7
8  struct Biblio {
9    CellMorceau *L; /* Liste chainee des morceaux */
10   int nE;         /* Nombre de morceaux dans la liste */
11 };
```

L'avantage de la liste chaînée et d'apporter de la souplesse pour l'allocation et la libération de la mémoire dynamique lors de l'ajout ou de la suppression d'un élément.
Néanmoins, pour la recherche d'un élément, dans la pire des situations, il faut parcourir entièrement la liste.

1.3 Arbre lexicographique

Cette structure permet de simplifier la recherche d'un élément.
On créer un arbre lexicographique sur les noms d'artistes qui stockera dans des listes chaînées

les morceaux d'un même artiste.

```c
typedef struct CellMorceau {
  struct CellMorceau *suiv;
  int num;
  char *titre;
  char *artiste;
} CellMorceau;

/* Cellule de l'arbre lexicographique */
typedef struct Noeud {
  struct Noeud *liste_car;      /* liste des choix possibles de caract'eres */
  struct Noeud *car_suiv;       /* caract'ere suivant dans la cha^ine */
  CellMorceau *liste_morceaux; /* liste des morceaux ayant le m^eme interpr'ete */
  char car;
} Noeud;

struct Biblio {
  int nE;     /* nombre d''elements contenus dans l'arbre */
  Noeud *A;   /* arbre lexicographique */
};
```

1.4 Tableau dynamique

Pour éviter les nombreuses indirections des allocations par pointeur présente dans les deux structures précédentes, on allouera cette fois qu'une seule zone de mémoire.

```c
typedef struct {
  char *titre;
  char *artiste;
  int num;
} Morceau;

struct Biblio {
  int nE;        /* Nombre de morceaux dans le tableau */
  int capacite; /* Capacit'e du tableau */
  Morceau *T;    /* Tableau de morceaux */
};
```

1.5 Table de hachage

Toutes les structures citées ci-dessus, la recherche d'un ou plusieurs éléments présentait un parcours important. Ainsi, nous allons modéliser la bibliothèque sous forme de table de hachage.

```c
typedef struct CellMorceau {
  struct CellMorceau *suiv;
  unsigned int cle;
  int num;
  char *titre;
  char *artiste;
} CellMorceau;


struct Biblio {
  int nE;            /* nombre d''elements contenus dans la table de hachage */
  int m;             /* taille de la table de hachage */
  CellMorceau **T;  /* table avec resolution des collisions par chainage */
};
```

On utilisera deux fonctions spécifiques à la structure, fonction_cle et fonction_hachage qui permettent respectivement 'attribuer une clé à un morceau selon, son artiste, plus précisément de la somme des valeurs ASCII de ses caractères et de le placer dans la table selon la fonction de Knuth :

$$H(k) = [m\ (kA - [kA])] \text{ pour toute clef k, où } A = (\sqrt{5}-1) / 2.$$

2 Les fonctions

2.1 Insertion

L'insertion est assez efficace pour la liste chaînée et le tableau dynamique.
Elle se fera respectivement en tête de liste et à la première case vide.
Cependant l'arbre lexicographique stocke les morceaux par artiste, l'insertion nécessite alors d'un appel de fonction qui parcourra et allouera si besoin les nœuds correspondant.
Il en est de même pour le tableau de hachage, où les clefs sont attribuées en fonction des noms d'artiste. Néanmoins elle reste plus efficace puisqu'il n'est pas nécessaire de parcourir une liste pour insérer un élément, et elle se fait en tête de liste.

2.2 Uniques

On créer dans un premier temps une bibliothèques vide, ensuite on va parcourir une bibliothèque *non-vide* ou nous allons insérer chaque élément de cette bibliothèque dans l'autre bibliothèque si ces éléments ni sont pas déjà. On utilisera alors à la fois l'insertion d'élément, et leurs recherches dans la bibliothèque retournée.
Là encore il est adapté à chaque structure, avec un accès plus rapide dans un arbre lexicographique et dans un tableau de hachage.

2.3 Recherche

Recherche par titre et par numéro

La recherche par titre ou par numéro est identique pour toutes les structures.
La bibliothèque et parcourue en entier dans le pire des cas, ou jusqu'à trouver une occurrence d'un morceau correspondant à la recherche.
Pour les arbres on utilisera une seconde fonction récursive qui parcourra les nœuds de façon à couvrir l'arbre à la fois verticalement et horizontalement comme on nous l'a indiqué en TD

```
CellMorceau* rechercheNoeudNum(Noeud* N,int num){
      if(N){
            CellMorceau* M=N->liste_morceaux;
            while(M){
                  if(M->num==num){
                        return M;
                  }
                  M=M->suiv;
            }
            M=rechercheNoeudNum(N->liste_car,num);
            if(M) return M;
            return rechercheNoeudNum(N->car_suiv,num);
      }
      return NULL;
}


CellMorceau * rechercheParNum(Biblio *B, int num)
{
      return rechercheNoeudNum(B->A,num);

}
```

Recherche par artiste

Nous voulons obtenir une bibliothèque ne contenant des morceaux que d'un seul artiste. L'organisation des structures va jouer beaucoup plus sur l'efficacité de l'algorithme.
Nous l'avons vu, l'arbre lexicographique et la table de hachage organisent l'insertion des morceaux selon les noms d'artistes. Ainsi ces deux structures permettent donc d'éviter un parcours complet.

3 Test des performances

Mise en place

Nous testerons séparément chaque fonction de recherche, puis la création d'une bibliothèque de morceaux uniques, et comparerons les résultats pour chaque structure selon le protocole suivant :
On récupèrera les cent mille premiers titres de notre bibliothèque BiblioMusicale, et non pas les trois cent mille car le temps d'exécutions des fonctions biblio_liste serait trop grand.
Pour chaque structure et chaque fonction, on prendra le même numéro, le même artiste et le même titre.
Nous allons représenter toutes ces données sous forme de graphe, qui seront graduées en secondes sur l'axe des ordonnés et en nombre d'élément chargée sur l'axe des abscisse.

La complexité « pire-cas » attendue pour la recherche par artiste est en Θ(n) pour la liste chaînée et le tableau dynamique, où n est le nombre de morceaux, mais en Θ(n') pour l'arbre lexicographique et la table de hachage – sans compter les collisions, où n' est le nombre maximal de morceau d'un même artiste. De plus, les complexités pire-cas attendues pour les recherches par titre et par numéro sont en Θ(n), bien qu'il faille rajouter la récursivité pour l'arbre et la combinaison de structures pour la table de hachage. Dit autrement, on s'attend `a ce que la recherche par artiste, et l'algorithme de morceaux uniques soient plus efficaces pour l'arbre lexicographique et la table de hachage. Et à l'opposé, que la recherche par titre et par numéro, qui nécessitent un parcours complet, soit plus efficace pour la liste chaînée et le tableau dynamique.

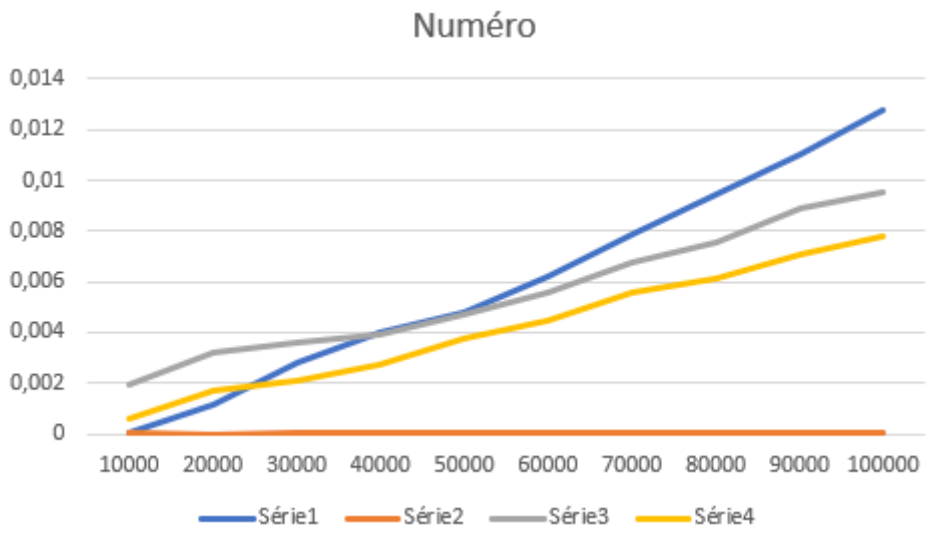
3 Résultats

Pour l'ensemble des graphes, nous utiliserons la légende suivante :

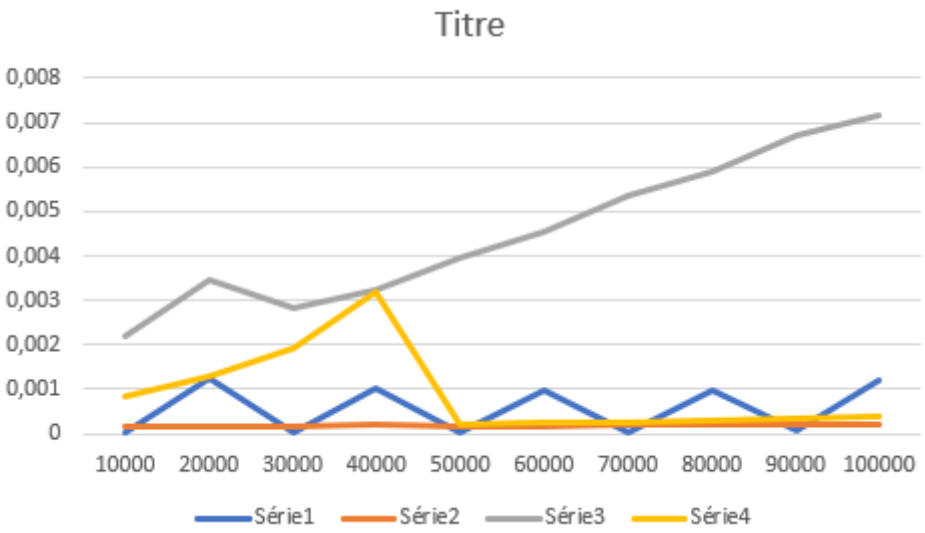
: biblio_liste
: biblio_tabdyn
: biblio_arbrelex
: biblio_hachage

**rappel* Abscisses (en nombre d'éléments chargées)*
Ordonnées (en secondes)

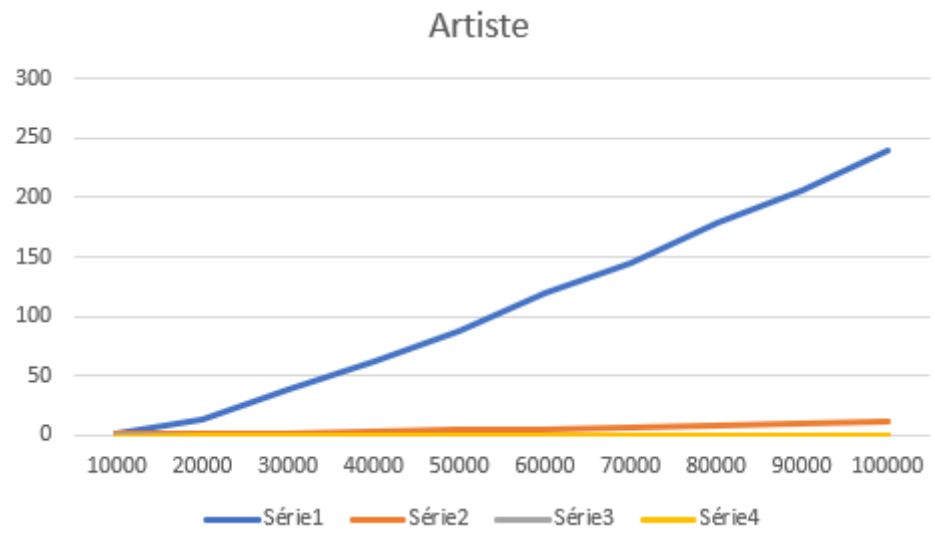
3.1 Recherche par numéro



3.2 Recherche par titre



3.3 Recherche par artiste



3.4 Recherche par uniques