TME 2-3-4 : Gestion d'une bibliothèque musicale

Version du 28/02/2020. Les ajouts sont en couleur bleue.
Ce mini-projet s'étale sur deux semaines (séances 2 et 3 de TME). Il est noté et doit être rendu sur moodle lors du TME numéro 4. A cette occasion, vous lui ferez une démonstration rapide de votre code à votre chargé de TD/TME.
Dans ce projet, nous nous intéressons à 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;
```

L'objectif de ce mini-projet est d'apprendre à comparer des structures de données. Nous utiliserons dans ce projet pour implémenter une bibliothèque :

- une liste simplement chaînée de struct et un arbre lexicographique
- un tableau dynamique de struct et une table de hachage de struct.

L'ensemble des fichiers nécessaires à ce mini-projet peuvent être récupérés sur le site web de l'UE : `https://www-licence.ufr-info-p6.jussieu.fr/lmd/licence/2019/ue/LU2IN006-2020fev/`
Il s'agit :

- d'un fichier `BiblioMusicale.txt` [1] dont le format est très simple : chaque morceau (une entrée) correspond à une ligne terminée par le caractère `'\n'` (format UNIX) ; chaque ligne comprend le numéro du morceau, son titre et enfin l'artiste qui l'interprète ; ces éléments sont séparés par des tabulations (`'\t'`) ; les titres et les noms d'artistes ne peuvent pas contenir de tabulations ou de retours à la ligne ;
- d'une bibliothèque d'aide à la lecture de fichier `parser.c` (et son `.h`) (son fonctionnement est donné en commentaires).

Dans ce mini-projet, les quatre structures de données à implémenter doivent toutes suivre la même interface et devront donc inclure le fichier `biblio.h` suivant qui prédéclare un type `Biblio` ainsi qu'un ensemble de fonctions pour manipuler ce type.

```
1   /* biblio.h */
2
3   #ifndef biblio_h
4   #define biblio_h
5
6   typedef struct Biblio Biblio;
7
8   Biblio *nouvelle_biblio(void);
9   Biblio *charge_n_entrees(const char *nom_fichier, int n);
10  void libere_biblio(Biblio *B);
11
12  void insere(Biblio *B, int num, char *titre, char *artiste);
13  void affiche(Biblio *B);
14  Biblio *uniques(Biblio *B);
15
16  /* Ajoutez toutes les fonctions communes 'a toutes les structures de donn'ees. */
17
18  #endif /* biblio_h */
```

Afin d'organiser votre code sur les 4 parties avec un seul main permettant de lancer avec les mêmes fonctions les 4 structures de données.
Un exemple complet est proposée dans l'archive `TME2-3-4-MODELE.zip` que l'on peut trouver sur le site. Cet exemple propose un fichier entête différent pour chacune des structures mais reprenant l'unique définition de `Biblio` du fichier `Biblio.h`. Grâce à la commande `typedef struct Biblio`

1. Fichier réalisé à partir de la base : `https://www.kaggle.com/miteshsingh/hollywood-music-dataset/`

`Biblio;`, le compilateur sait quelle est la structure à utiliser dans les fonctions correspondantes à un même fichier de code .c.
Notez que l'appel à la fonction `uniques(biblio);` retourne un pointeur sur `Bilbio` qui est donc du type de la structure manipulée lors de l'appel.

Ainsi, une compilation réalisée en liant avec l'un des 4 fichiers de structures `biblio_liste.o, biblio_arbrelex.h`, (sans oublier leurs entêtes), prouira un exécutable différent.
Le `Makefile` correspondant à cette idée permet de faire cela en produisant ainsi 4 exécutables à partir du même fichier `main.c`.

Exercice 1 – Gestion d'une bibliothèque avec une liste chaînée de struct

Dans ce premier exercice, nous allons coder une bibliothèque comme une liste chaînée de struct de type `CellMorceau`.
On utilisera alors les structures suivantes à déclarer dans un fichier `biblio_liste.c` :

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

struct Biblio {
  CellMorceau *L; /* Liste chainee des morceaux */
  int nE;         /* Nombre de morceaux dans la liste */
};
```

Q 1.1 Créer les fichiers correspondants et créer un MakeFile.

Q 1.2 Créer une fonction `Biblio *nouvelle_biblio(void)` qui alloue une structure `Biblio` pour qu'elle corresponde à une bibliothèque vide.

Q 1.3 Dans un fichier `main.c`, créez un main en suivant les principes suivant :
- entrée des données en ligne de commande pour le nom du fichier et le nombre de lignes à lire dans le fichier (voir ci-dessous)
- menu permettant à l'utilisateur d'utiliser les différentes fonctions du programme.

Voici un exemple possible qui pourra être utilisé pour les quatres structures de données sans être dupliqué.

```c
/* main.c */

#include <stdio.h>
#include <stdlib.h>

#include "biblio.h"

void menu() {
  printf("Menu:\n");
  printf("0-Sortie\n");
  printf("1-Affichage\n");
  printf("2-Recherche morceaux uniques\n");

  printf("Votre choix : ");
}

int main(int argc, const char *argv[]) {
  if (argc != 3){
    printf("Erreur format : %s <NomFichierBiblio.txt> <NbLigneALire>", argv[0]);
    return 1;
  }

```

```
23   const char *nomfic = argv[1];
24   int nlignes = atoi(argv[2]);
25
26   printf("Lecture :\n");
27   Biblio *biblio = charge_n_entrees(nomfic, nlignes);
28
29   int ch;
30   do {
31     menu();
32     int lus = scanf("%d", &ch);
33     if (lus == 0) {
34       ch = 0;
35     }
36
37     switch(ch) {
38       case 1:
39         printf("Affichage\n");
40         affiche(biblio);
41         break;
42       case 2:
43       {
44         Biblio *Bunique = uniques(biblio);
45         affiche(Bunique);
46         libere_biblio(Bunique);
47         break;
48       }
49       default:
50         ch = 0;
51         break;
52     }
53   } while (ch != 0);
54
55   libere_biblio(biblio);
56   printf("Au revoir\n");
57
58   return 0;
59 }
```

Q 1.4 Dans le fichier `biblio_liste.c`,
créer une fonction `void insere(Biblio *B, int num, char *titre, char *artiste)` qui insère un nouveau morceau dans la bibliothèque `B`.

Q 1.5 Dans le fichier `biblio.c`,
créer une fonction `Biblio *charge_n_entrees(const char *nom_fichier, int n)` permettant de lire n entrées de ce fichier et de renvoyer une bibliothèque les contenant. Pour l'ajout, on appellera la fonction `insere`.
Pour lire facilement les champs du fichier, vous pouvez utiliser les fichiers `parser.h` et `parser.c`.
Le module `parser` permet de parser des fichiers délimités par certains caractères en langage C. Vous pouvez lire les commentaires décrivant les fonctions dans le fichier `parser.h`.

Q 1.6 Créer les fonctions suivantes permettant, à partir de la structure de données chargées en mémoire :

- 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. Deux morceaux sont identiques s'ils ont le même artiste et le même titre. Cette fonction devra renvoyer une liste comprenant les morceaux qui n'apparaissent qu'une fois dans la bibliothèque.

Vous déclarerez les fonctions dans `biblio.h`, les implémenterez dans `biblio_liste.c` et les testerez dans le fichier `main.c`.

Exercice 2 – Gestion d'une bibliothèque avec un arbre lexicographique

Bien que les listes soient très souples (allocation et libération dynamiques de la mémoire pour l'ajout et la suppression de données), si on cherche à récupérer un élément précis de la liste, dans le pire des cas, il faudra parcourir la liste en entier jusqu'à ce qu'on le trouve.
Dans cette deuxième partie, pour accélérer l'accès à un élément de la bibliothèque, nous allons utiliser un arbre lexicographique sur les noms d'artiste. L'ensemble des titres d'un même artiste seront stockés dans une liste chaînée. La figure 1 représente un arbre lexicographique et la figure 2 une possible implémentation.

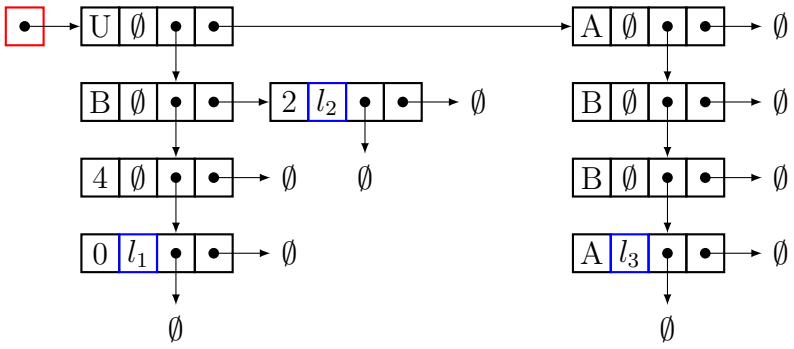

FIGURE 1 – La figure suivante représente un arbre lexicographique dans lequel sont stockés trois artistes : UB40, U2 et ABBA. La liste des morceaux de ces artistes sont respectivement les listes l_1, l_2 et l_3.

```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 */
};
```

FIGURE 2 – Pour implémenter votre arbre lexicographique, vous pourrez utiliser ces struct (par exemple). Par rapport à la figure 1, les listes l_1, l_2 et l_3 correspondent au champ `liste_morceaux` ; les liens horizontaux au champ `liste_car` ; les liens verticaux au champ `car_suiv`.

Cette structure classique est appelée "Arbre lexicographique" mais elle peut être vue comme une "double liste simplement chaînée". En effet, en suivant itérativement le pointeur `liste_car` en partant d'une lettre x, on trouve toutes les lettre alternatives à x ; et en suivant itérativement le pointeur `car_suiv`, on lit un des mots possibles à partir de x.

Notez qu'il est difficile de commencer par la fonction `insere` directement. Il est préférable de coder en premier une fonction `recherche` qui à partir d'un `char *mot` retourne soit l'emplacement du mot,

soit l'emplacement où il doit être complété.
Le code suivant correspond à l'idée d'une telle fontion recherche où les le pointeur `prec` et un entier `i` obtenu en fin de boucle donne des indications sur l'insertion.

```
// Recherche si un artiste est present
void recherche_artiste(Biblio *B, char *artiste){
  int i=0;           // Indice lisant nom
  Noeud *cour=B->A; // Pointeur courant
  Noeud *prec=cour;      // Pointeur sur la valeur precedant cour
  while ( (cour!=NULL) && (artiste[i]!='\0')){
    prec=cour;
    if (cour->car==artiste[i]){ // On a lu le bon caractere
      cour=cour->car_suiv;
      i++;
    }
    else{  // On tente un caractere alternatif
      cour=cour->liste_car;
    }
  }

  if (artiste[i]=='\0'){
    printf("La chaine %s est presente\n",artiste);
  }
  else{
    printf("La chaîne %s est presente jusqu'au caractere numero %d \n",artiste,i-1);
    if (artiste[i-1]==prec->car){
      printf("La suite de la chaîne peut être inseree a la suite de la lettre %c.\n
          ", prec->car);
    }
    else{
      printf("La suite de la chaîne peut être en alternative a la lettre %c.\n",
          prec->car);
    }
  }
}
```

Il y a de multiples façons d'implémenter ces fonctions (par exemple par récurrence), n'hésitez pas à concevoir votre propre algorithme.

Q 2.1 Dans le fichier `biblio_arbrelex.c`,
créer une fonction `voidinsere(Biblio *B, int num, char *titre, char *artiste)` qui insère un nouveau morceau dans la bibliothèque `B`.
Pour insérer un morceau dans une bibliothèque, vous devrez parcourir chaque caractère du nom de l'artiste. Pour chaque caractère, vous devez :

- le rechercher en suivant la liste chaînée `liste_car` ; si vous le trouvez, passez à l'étape suivante, sinon créez un nouveau noeud dans `liste_car` puis passez à l'étape suivante ;
- passez au caractère suivant en suivant le lien `car_suiv`.

Lorsque vous avez lu tous les caractères du nom de l'artiste, noeud qui correspond à la dernière lettre est celui dans lequel vous allez stocker le morceau parmi l'ensemble des morceaux de cet artiste éventuellement déjà présents (dans `liste_morceaux`).

Q 2.2 Dans `biblio_arbrelex.c`, implémenter toutes les autres fonctions déclarées dans `biblio.h`.

Exercice 3 – Gestion d'une bibliothèque avec un tableau dynamique

Les deux précédentes structures de données présentent, sur les machines actuelles, des difficultés d'accès à la mémoire du fait des nombreuses indirections qu'elles comportent (le prefetching du processeur est inefficace).
Dans cette troisième partie, pour accélérer le parcours de la bibliothèque, nous allons utiliser un tableau dynamique, c'est à dire une zone de mémoire contiguë qu'on réallouera si nécessaire pour qu'elle puisse stocker davantage de morceaux. Il restera néanmoins les indirections dues aux chaînes de ca-

ractères.

Pour implémenter votre tableau dynamique, vous pourrez utiliser les struct suivantes (par exemple) :

```
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 */
};
```

Q 3.1 Dans le fichier `biblio_tabdyn.c`,
créer une fonction `void insere(Biblio *B, int num, char *titre, char *artiste)` qui insère un nouveau morceau dans la bibliothèque `B`.
Pour insérer un morceau dans une bibliothèque, vous devrez déjà vérifier si la capacité du tableau `T` est atteinte. Si c'est le cas, vous devez d'abord réallouer (avec `realloc`) le tableau `T` en doublant sa capacité.
Les nouveaux morceaux seront ajoutés à la première case libre du tableau.

Q 3.2 Dans `biblio_tabdyn.c`, implémenter toutes les autres fonctions déclarées dans `biblio.h`.

Exercice 4 – Gestion d'une bibliothèque avec une table de hachage

Dans les structures de données précédentes, il est nécessaire de parcourir de nombreuses cellules avant de trouver l'élément que l'on cherche. Les tables de hachage cherchent à réaliser cette recherche en un temps constant. Dans cette dernière partie, pour accélérer l'accès à un élément de la bibliothèque, nous allons donc utiliser une table de hachage. La résolution des collisions se fera par chaînage.

Pour implémenter votre table de hachage, vous pourrez utiliser les struct suivantes (par exemple) :

```
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 */
};
```

Q 4.1 Fonction clé : La fonction `fonction_cle` de la table de hachage à implémenter doit permettre d'associer une valeur numérique au contenu présent dans la table de hachage. Le contenu à stocker est le struct de type `CellMorceau`, c'est-à-dire un titre, un nom d'artiste et un numéro de morceau. Comme fonction cle, vous pouvez utiliser les caractères contenus dans le nom de l'artiste et utiliser la somme des valeurs ASCII de chaque lettre du nom ou mieux pour éviter les collisions.
Créez la fonction `unsigned int fonction_cle(const char *artiste)` réalisant cette opération.

Q 4.2 Fonction de hachage : Il est ensuite nécessaire de transformer la clef obtenue en une valeur entière utilisable par la table de hachage (c'est-à-dire entre 0 et m non compris) et permettant d'éviter au maximum les collisions.

Vous pourrez par exemple utiliser la fonction de hachage $h(k) = \lfloor m(kA - \lfloor kA \rfloor) \rfloor$ pour toute clef k, où $A = \frac{\sqrt{5}-1}{2}$ (nombre d'or diminué de 1, proposé par Donald Knuth [2]). Vous testerez expérimentalement plusieurs valeurs de m afin de déterminer la valeur la plus appropriée. Il est également possible d'utiliser des fonctions de hachage bien différentes : vous pouvez proposer d'autres fonctions.
Créez la fonction `unsigned int fonction_hachage(unsigned int cle, int m)` réalisant cette opération.

Q 4.3 Dans le fichier `biblio_hachage.c`,
créer une fonction `void insere(Biblio *B, int num, char *titre, char *artiste)` qui insère un nouveau morceau dans la bibliothèque `B`.
Pour insérer un morceau dans une bibliothèque, vous devrez procéder en trois étapes :

- Identifier la clé correspondant au morceau (grâce à son artiste).
- Trouver dans quelle case de la table de hachage insérer le morceau (grâce à la fonction de hachage).
- Insérer le morceau dans la liste chaînée correspondant à cette case.

Q 4.4 Dans `biblio_hachage.c`, implémenter toutes les autres fonctions déclarées dans `biblio.h`.

Exercice 5 – Comparaison des quatre structures

On veut comparer les quatre structures (liste, arbre lexicographique, tableau dynamique et table de hachage) par rapport au temps nécessaire pour effectuer les fonctions de recherche.

Q 5.1 Comparer les temps de calcul entre les quatre structures pour réaliser la recherche d'un morceau par son numéro, son titre et son artiste.
Pour avoir des temps de calcul significatifs, vous pourrez procéder à plusieurs recherches consécutives. Quelle structure (liste, arbre lexicographique, tableau dynamique ou table de hachage) est la plus appropriée pour chacune de ces recherches ?

Q 5.2 Modifiez la taille de votre table de hachage. Comment évoluent vos temps de calcul en fonction de cette taille ?

Q 5.3 On veut déterminer les temps de recherche des morceaux uniques en fonction de la taille de la bibliothèque et de la structure de données utilisée. Pour cela, vous adapterez votre fonction pour qu'elle puisse relancer la fonction de lecture en lisant partiellement les n premières lignes du fichier, n prenant les valeurs de 1000 à 300 000 [3] avec un pas croissant.
Pour chaque valeur de n vous sauvegarderez les temps de calcul obtenus avec chacune des quatre structures. Vous visualiserez ensuite par des courbes les séries de nombres obtenues.

Q 5.4 Justifiez les courbes obtenues en fonction de la complexité pire-cas attendue.

N'oubliez pas de rendre votre code ET votre rapport au début de la séance 5, en respectant les consignes du site du module !

2. Né le 10 janvier 1938 dans l'état du Wisconsin aux États-Unis, Donald Knuth est un informaticien et mathématicien américain de renom et professeur émérite en informatique à l'université Stanford (États-Unis). Il est un des pionniers de l'algorithmique et a fait de nombreuses contributions dans plusieurs branches de l'informatique théorique. Il est également l'artiste de l'éditeur de texte LaTeX.

3. Si sur votre machine le temps de calcul devient trop long pour 300 000 entrées (c'est-à-dire supérieur à 5 minutes), vous pouvez diminuer le nombre maximal d'entrées.