Projet: Database
Algorithmique et structures de données
Licence 2 Informatique

Julien Bernard

Le but de ce projet est de réaliser un outil interactif en mode texte permettant d'interroger une base de données. Une base de données permet de stocker et de retrouver des données. Traditionnellement, pour interroger les données de manière optimale, on dispose de deux outils : les clefs primaires et les index. Une manière d'implémenter une clef primaire est de trier les données suivant le champ considéré. Une manière d'implémenter les index est d'utiliser une table de hachage. Ce projet consiste à implémenter ces deux types de recherche sur un exemple d'annuaire contenant potentiellement plusieurs millions d'entrées.

Il est conseillé de lire le sujet en entier et de réfléchir avant de commencer à coder. Les fonctions demandées devront respecter les prototypes donnés dans l'énoncé.

Partie 1 : Génération de l'annuaire

Dans cette partie, nous allons générer un annuaire aléatoire de manière à avoir des données à manipuler en quantité importante. Nous considérons que les entrées de l'annuaire sont composées d'un nom *(last name)* et d'un prénom *(first name)*, de taille comprise entre 3 et 10, et d'un numéro de téléphone de 8 chiffres. La structure suivante permet de stocker ces informations.

```
#define NAME_LENGTH_MIN 3
#define NAME_LENGTH_MAX 10
#define TELEPHONE_LENGTH 8

struct directory_data {
  char last_name[NAME_LENGTH_MAX + 1];
  char first_name[NAME_LENGTH_MAX + 1];
  char telephone[TELEPHONE_LENGTH + 1];
};
```

Pour générer une entrée aléatoirement, on respectera les exigences suivantes :

- les noms sont composés uniquement de majuscules ;
- les noms alternent consonnes et voyelles ;
- les noms commencent par une consonne ;
- les consonnes autorisées dans un nom sont : B, C, D, F, G, H, J, L, M, N, P, R, S, T, V ;
- les voyelles autorisées dans un nom sont : A, E, I, O, U ;

Question 1.1 Écrire une fonction qui affiche les informations d'une entrée sous la forme :

```
- NOM PRENOM: 00000000
```

```
void directory_data_print(const struct directory_data *data);
```

Question 1.2 Écrire une fonction qui remplit une entrée avec des données aléatoires. On prendra garde à ce que les champs de la structure soient bien des chaînes de caractères, c'est-à-dire qu'elles se terminent bien par `\0`.

```
void directory_data_random(struct directory_data *data);
```

—

Le répertoire sera représenté par un tableau dynamique de pointeurs sur des entrées de répertoire.

```
struct directory {
  struct directory_data **data;
  size_t size;
  size_t capacity;
};
```

Question 1.3 Écrire une fonction qui initialise un répertoire.

```
void directory_create(struct directory *self);
```

Question 1.4 Écrire une fonction qui détruit un répertoire et toutes ses entrées.

```
void directory_destroy(struct directory *self);
```

Question 1.5 Écrire une fonction qui ajoute une entrée dans le répertoire.

```
void directory_add(struct directory *self,
                                    struct directory_data *data);
```

Question 1.6 Écrire une fonction qui génère un répertoire avec n entrées au hasard.

```
void directory_random(struct directory *self, size_t n);
```

Partie 2 : Recherche suivant le nom

Dans cette partie, nous allons chercher les entrées de l'annuaire correspondant à un nom de famille.

Question 2.1 Écrire une fonction qui cherche et affiche toutes les entrées qui ont un nom donné en parcourant tout le tableau.

```
void directory_search(const struct directory *self,
                                          const char *last_name);
```

—

De manière à faire une recherche optimisée, nous allons trier le tableau en fonction du nom.

Question 2.2 Écrire une fonction qui trie le répertoire.

```
void directory_sort(struct directory *self);
```

Question 2.3 Écrire une fonction qui cherche et affiche toutes les entrées qui ont un nom donné dans le répertoire trié. On prendra garde à bien afficher toutes les entrées avec le même nom.

```
void directory_search_opt(const struct directory *self,
                                          const char *last_name);
```

Partie 3 : Création et utilisation d'index

Dans cette partie, nous allons créer un index pour le répertoire à l'aide d'une table de hachage. Plus précisément, avec la même implémentation, nous considérerons un index pour les prénoms et un index pour les numéros de téléphone.

Une table de hachage est une structure de données qui permet une implémentation d'un tableau associatif, c'est-à-dire un tableau dans lequel les éléments sont représentés par une clef. Dans notre cas, la clef sera soit le prénom, soit le numéro de téléphone.

Concrètement, une table de hachage est un tableau de k listes chaînées. Pour insérer un élément dans l'ensemble, on utilise une fonction de hachage qui prend un élément et renvoie un entier h, l'élément est alors inséré dans la liste chaînée qui se trouve à l'indice ($h \mod k$).

Si deux éléments ont le même indice, ils se retrouvent dans la même liste chaînée, on parle alors de *collision*. On essaie de choisir une fonction de hachage qui permet d'éviter les collisions. On appelle *facteur de compression* le rapport $\frac{n}{k}$, c'est-à-dire le nombre d'éléments de la table divisé par le nombre de cases du tableau.

Quand le facteur de compression dépasse une certaine valeur pour laquelle le risque de collision devient important (0.5 par exemple), alors on effectue un *rehash*, c'est-à-dire qu'on va doubler la taille du tableau et recalculer tous les indices des éléments déjà présents.

—

On s'intéresse tout d'abord à la structure de liste chaînée (qu'on appelle généralement *bucket* dans le cadre des tables de hachage).

```
struct index_bucket {
  const struct directory_data *data;
  struct index_bucket *next;
};
```

Question 3.1 Écrire une fonction qui ajoute un élément à une liste chaînée éventuellement vide et renvoie la nouvelle liste chaînée.

```
struct index_bucket *index_bucket_add(struct index_bucket *self,
                                      const struct directory_data *data);
```

Question 3.2 Écrire une fonction qui détruit une liste chaînée éventuellement vide.

```
void index_bucket_destroy(struct index_bucket *self);
```

—

De manière à avoir un index générique, on utilise un pointeur de fonction pour désigner la fonction de hachage des éléments. Nous allons créer une fonction de hachage pour chacun des deux index envisagés.

```
typedef size_t (*index_hash_func_t)(
                                      const struct directory_data *data);
```

Question 3.3 Écrire une fonction qui calcule un hash FNV [1] 32 bits ou 64 bits en fonction de votre machine et de votre système d'exploitation.

```
size_t fnv_hash(const char *key);
```

Question 3.4 Écrire une fonction de hachage sur le prénom d'une entrée de répertoire en utilisant le hash FNV.

```
size_t index_first_name_hash(const struct directory_data *data);
```

Question 3.5 Écrire une fonction de hachage sur le numéro de téléphone d'une entrée de répertoire en utilisant le hash FNV.

```
size_t index_telephone_hash(const struct directory_data *data);
```

—

La table de hachage est représentée par la structure suivante où `count` désigne le nombre d'éléments dans la table et `size` désigne la taille du tableau `buckets`.

```
struct index {
  struct index_bucket **buckets;
  size_t count;
  size_t size;
  index_hash_func_t func;
};
```

1. `http://www.isthe.com/chongo/tech/comp/fnv/index.html`

Question 3.6 Écrire une fonction qui crée un index vide et initialise la fonction de hachage.

```
void index_create(struct index *self, index_hash_func_t func);
```

Question 3.7 Écrire une fonction qui détruit un index. On détruira toutes les listes chaînées mais pas les entrées du répertoire.

```
void index_destroy(struct index *self);
```

Question 3.8 Écrire une fonction qui effectue un rehash. On prendra garde au cas où l'index est vide.

```
void index_rehash(struct index *self);
```

Question 3.9 Écrire une fonction qui ajoute une entrée dans l'index. On effectuera un rehash si nécessaire.

```
void index_add(struct index *self,
                              const struct directory_data *data);
```

Question 3.10 Écrire une fonction qui remplit un index à l'aide d'un répertoire.

```
void index_fill_with_directory(struct index *self,
                                    const struct directory *dir);
```

Question 3.11 Écrire une fonction qui cherche les entrées en fonction d'un prénom grâce à un index.

```
void index_search_by_first_name(const struct index *self,
                                       const char *first_name);
```

Question 3.12 Écrire une fonction qui cherche les entrées en fonction d'un numéro de téléphone grâce à un index.

```
void index_search_by_telephone(const struct index *self,
                                         const char *telephone);
```

Partie 4 : Application interactive

Dans cette partie, nous allons créer une application interactive permettant d'utiliser toutes les fonctions que nous venons de coder. Pour cela, nous allons utiliser la fonction `fgets(3)` qui permet de charger un fichier ligne par ligne. Cette fonction copie chaque ligne du fichier dans un buffer, il est donc nécessaire de supprimer le caractère de changement de ligne avant de traiter le buffer.

Question 4.1 Écrire une fonction qui supprime le caractère de changement de ligne final et le remplace par `\0`.

```
void clean_newline(char *buf, size_t size)
```

—

La boucle de saisie peut alors s'écrire de la manière suivante :

```
char buf[BUFSIZE];

for (;;) {
  printf("> ");
  fgets(buf, BUFSIZE, stdin);
  clean_newline(buf, BUFSIZE);
}
```

Question 4.2 Écrire un menu qui propose les choix suivants et implémenter le choix `q`.

```
What do you want to do?
        1: Search by last name (not optimised)
        2: Search by last name (optimised)
        3: Search by first name
        4: Search by telephone
        q: Quit
Your choice:
```

Question 4.3 Avant la boucle de saisie, initialiser un répertoire avec 1 000 000 d'entrées au moins. Puis trier le répertoire et créer les deux index (un pour le prénom et un pour le numéro de téléphone). Pour chaque opération, on pourra afficher l'opération puis le temps en microsecondes pour réaliser l'opération (indice : `gettimeofday(3)`).

Question 4.4 Pour chaque entrée du menu, ajouter la saisie adéquate et afficher le résultat.

Question 4.5 Après la boucle de saisie, libérer correctement la mémoire.

Considérations générales

Le projet est à faire en binôme. Le résultat, sous forme d'une archive `tar.gz` contenant l'ensemble de vos sources et nommée à l'aide des noms des deux membres du binôme, est à rendre sur MOODLE pour la date suivante (aucun retard autorisé).

vendredi 6 mars 2015 à 18h00

La note du projet tiendra compte des points suivants (liste non-exhaustive) :
— la complexité optimale des algorithmes proposés ;
— la séparation du projet en plusieurs unités de compilation ;
— la présence d'un Makefile fonctionnel ;
— les commentaires dans le code source ;
— l'absence de fuites mémoire.