Projet de programation avancée
S6

NEUS Maxence

Contents

1 Introduction

Dans le cadre du module de programmation avancée, nous avons eu à plusierus reprises l'occasion de travailler avec la notion de structures de donnée en C.
Lors de ce projet, nous avons eu à mettre à profit les connaissances acquisent à ce sujet au cours du module pour traiter le plus efficacement possible un enseble de données concernant des vols entre un grand nombre d'aéroports aux Etats-Unis.
Il nous est proposé de réaliser un certain nombre de fonctions pour obtenir des informations intéressantes sur les vols présentés.
La mise en oeuvre de ce projet peut être divisée en trois parties:

- Lecture des données et gestion des structures.
- Algorithmes permettant d'implémenter les différtentes fonctions proposées dans le sujet.
- Mise en place d'une interface en ligne de commande pour acceder aux données.

2 Lecture des données

2.1 Structures de données

Tout d'abord, nous allons nous pencher sur les choix qui ont été fait concernant le stockage des données.

2.1.1 Aéroports

Pour stocker les informations sur les différents aéroports, l'approche evidente est d'utiliser une structure de type Table de Hashage.
En effet la présence des `IATA_code` ,nous offre un choix simple de clé de hashage car ils sont unique à chaque aéroport.
Cette approche nous permet d'acceder directement à toutes les informations relatives à un aéroport en aillant simplement son `IATA_code`.
Le choix de la fonction de hashage est détaillé avec celui des companies en 2.1.3.

2.1.2 Companies

Pour les mêmes raisons que pour les aéroports, la présence de `IATA_code` nous invite à utiliser ici une table de hashage pour stocker les données relatives aux companies. Le choix de la fonction de hashage est ici aussi détaillé en 2.1.3.

2.1.3 Fonctions de hashage

Se pose alors la question de la fonction de hashage, pour résoudre ce problème, quelques recherches ont menés à la découverte d'un projet appelé `perfect-hash` (projet github par ilanschnell). Cet outil consiste en un script python qui prends un fichier de type csv et génere à partir d'une colone une fonction de hashage "parfaite" (c'est à dire qui est parfaitement bijective). Ce script a été ici utilisé

pour générer les fonctions de hashage pour les aéroports et pour les companies.

Une limite de ce choix de fonction de hashage est que le code ne pourra traiter que les aéroports et companies qui sont mises à disposition ici. Mais les ajout de companies ou la construction de nouveaux aéroports étants aprioris peu fréquent, ce choix semble raisonnable. Un problème majeur de cette aproche reste lors d'une éventuelle extension du système à l'exterieur des USA, il faudrait regénerer de nouvelles fonctions de hashage.

2.1.4 Vols

Un certain nombre de fonctions utilisent la date des vols comme argument, il semble donc utile de rendre l'acces aux vols à un jour précis le plus rapide possible. Pour ce faire, le choix a été fait ici d'utiliser encore une fois une table de hashage dont la clé d'acces est cette fois-ci la date du vol. Avec cette approche, la structure consistera en n listes de vols qui ont eu lieu le même jour. Le nombre de vols par jour etant variable, et le parcours de chaque liste fréquent, nous choisirons ici de représenter ces listes de vols par des listes contigües de vols.
La structure globale de stockage des vols sera donc une Table de hashage de ces listes contigües où l'indice dans la table correspond à la date à laquelle les vols de la liste ont eu lieu.
Pour déterminer cet indice, la fonction de hashage suivante :

$$f(m, j) = (1 - m) * 31 + j - 1 \tag{1}$$

permets d'obtenir des indices entre 0 pour le 1^{er} janvier et 371 pour le 31 decembre. soit 372 valeurs ce qui correspond à seulement 6 jours de plus qu'une année bissextile, La consommation mémoire de cette solution est donc plutôt optimale tant que la longueure maximale des listes contigües reste proche du nombre réel de vols dans la journée.

2.2 Lecture

Cette partie est réalisée dans le code dans le fichier `reader.c`.
Pour ce faire, on récupère la ligne suivante du fichier via la fonction `getline` qui nous permet de recuperer le contenu de la ligne dans un `char*`, on vient ensuite grâce à `strchr` trouver la position du `'\n'` à la fin du string pour le remplacer par un `'\0'` pour terminer proprement le string.
Par la suite, nous devons encore récuperer les différents champs correspondants aux caractèristiques des vols, aéroports ert companies, pour se faire, nous utilisons encore une fois la fonction `strchr` qui va nous permettre de trouver la position du prochain séparateur et donc par `strncpy` de récuperer le contenu de la ligne entre les deux séparateur par arithmétique de pointeurs (Le dernier argument est simplement `strcpy`).
Pour les arguments de types `int` ou `float`, nous utilisons les fonction `atoi` et `atof` respectivement pour convertir le `char*` récuperé précedement vers le type qui convient.
On prendra ici le temps de terminer proprement chaque argument récupéré par `'\0'` pour éviter des problèmes avec les pointeurs qui sont utilisés à plusieurs reprises.

3 Fonctions

3.1 show-airports <airline_id>

Cette fonction est l'une des plus simples, comme nous devons afficher tout les aéroports où la companie opère des vols, nous devons parcourir l'ensemble des vols et lorsque l'on rencontre la companie `<airline_id>` dans le champs `flight.airline`, nous devons verifier que l'aéroport correspondant n'as pas déjà été affiché et si c'est bien le cas il ne reste qu'à l'afficher en accedant à la Table des aéroports vie son `IATA_code` et à l'ajouter à la liste de ceux qui ont déjà été affichés.

3.2 show-airlines <port_id>

On utilise le même raisonement que pour `show-airports` en s'arretant sur les vols où le champs `flight.org_air` correspond à `port_id`.
On doit toujours verifier que la companie n'as pas déjà été affichée de la même manière que précedement.

3.3 show-flights <port_id> <date> [time] [limit=xx]

Ici le principe de recherche est le même que pour les deux fonctions précedentes mais cette fois, il n'est pas necessaire de verifier que le vol n'a pas été affiché étant donné qu'ils sont tous uniques, et on ne recherche pas dans tout les vols mais simplement dans la liste des vols qui ont eu lieu à la date `date`.
Par ailleur, les arguments optionnels doivent être pris en compte avant le début de la recherche:
Les valeurs par defaut de ces arguments sont initialisées à 0 pour `time` et au nombre maximum de vols par jour pour `limit`. Ces deux arguments sont passés à la fonction par un `char*` qui peut être vide et qui dont on extrait les arguments de la même manière que pour la lecture.
On dit qu'un vol est valide si il part de `port_id`, que `flight.sched_dep > time` et que le nombre de vols affichés ne dépasse pas `limit`.

3.4 most-delayed-flights

Ici on doit garder en tout temps les 5 vols qui ont eu le plus de retards jusque là, pour cela on a une liste de 5 vols initialisés avec des vols "nuls" (c'est à dire dont toute les valeurs sont nulles) et on garde l'indice du vol avec le retard le plus bas de la liste (initialisé à 0).
Par la suite, à chaque vol parcouru dont le retard est supérieur au retard minimum présent dans la liste, on remplace le vol à l'indice du minimum avec le vol trouvé et on recalcule l'indice du minimum pour la suite.
Quand la fonction arrive à la fin de la table des vols, la liste contient les 5 vols avec les plus grands retards.

3.5 most-delayed-airlines

Similaire à la fonction précedente, nous devons trouver les companies avec le plus grand retard moyen, pour ce faire, nous calculons d'abord les moyennes de retards de chaque companie en bouclant sur l'ensemble des vols, puis on affiche la companie dont le retard moyen est maximum avant de mettre ce retard moyen à 0 pour eviter de l'afficher encore par la suite, on répete cette dernière opération 5 fois pour avoir les 5 plus grands retards moyens.

3.6 delayed-airline <airline_id>

Ici on réalise la même procédure que le début de la fonction précedente avec la condition que l'on ne prends en compte que les vols par la companie `airline_id` opère le vol. On peut ensuite directement afficher la moyenne de retards obtenus.

3.7 most-delayed-airlines-at-airport <port_id>

On réalise ici la même fonction que `most-delayed-airlines` avec la condition suplémentaire que le vol concidéré parte de l'aéroport `port_id`.

3.8 changed-flights <date>

Il nous suffit ici de parcourir la liste des vols qui ont eu lieu à la date `date` et d'afficher le vol si `flight.diverted == true` ou `flight.canceled == true`.

3.9 avg-flight-duration <port_id1> <port_id2>

Nous devons ici calculer la moyenne de tout les vols dont l'aéroport de départ est `port_id1` ou `port_id2` et dont l'aéroport d'arrivée est `port_id1` ou `port_id2`.

3.10 find-itinerary

L'approche utilisée ici est de séparer la recherche par nombre d'escales acceptés. En effet trouver les vols sans escales pour aller de l'aéroport de départ à l'aéroport d'arrivée est trés simple, mais pour ce qui est des itinéraires à escales, la recherche requiert de boucler plusieurs fois sur les vols de la journée afin de trouver les aéroports intermédiaires qui pourraient permettre la jonction.
Ces boucles imbriqués sont ici codées à la main et donc la fonction peut faire 0 ou 1 escale, cette limitation pourrait être levée en utilisant une méthode récursive, mais dans mes recherches je n'ai pas réussi à trouver de condition d'arrêt satisfaisante.
Par ailleur les arguments optionnels sont ici récupérés de la même mainère que pour `show-flights`.

3.11 find-multicity-itinerary

Cette fonction se dérive assez naturelement de la précedente en réalisant une recherche d'itinéraire entre chaque escale demandée. Pour gérer le nombre indéfini d'escales demandées, nous allons ici utiliser une méthode récursive pour appeler `find-itinerary` le nombre adapté de fois.
Pour passer les différentes escales d'un appel à l'autre, nous utilisons un ensemble de files d'attentes dans lesquelleq on ajoute en queue les arguments correspondants qui seront récupérés en tête lors de la récursion pour conserver l'ordre des escales.
Ces arguments sont ensuite utilisés pour appeler `find-itinerary`.

4 Interface utilisateur

Pour faire la distinction entre les différentes fonctions, nous allons commencer par récuperer le premier élément avant un `' '` dans l'entrée standard, (ce qui correspond à la fonction demandée) et nous le passons par une fonction de hashage (la somme des caractères est suffisante ici) affin d'utiliser une structure `switch () case:` où les cas correspondent aux résultats de la fonction de hashage pour les différentes fonction développées plus haut (ces valeurs ont été calculées manuellement à par avance). Chaque `case:` fait donc correspondre une fonction qui récupère les arguments dans la fin de l'entrée stantard, si il y en a, et appelle la fonction correspondante.
Des valeurs de retours sont en place pour prendre en compte les cas non définis comme une fonction non reconue ou des arguments non valides, ainsi que le signal de sortie du programme qui est aussi exectué si le programme arrive à `EOF` (dans le cas ou on a utilisé le programme avec un fichier de requetes).

5 Conclusions

Dans l'ensemble, la réalisation de ce projet a permit de développer la compréhension des conceptes de programmation en C, notament l'utilisation des pointeurs et les nombreux problèmes qu'ils peuvent causer en condition réelles et pas seulement dans le cadre restreint des TPs.
Le choix de réaliser ce projet seul s'est avéré ne pas être la decision la plus judicieuse de cette année je dois l'admettre mais je pense avoir fourni un code fonctionnelle bien que sa propreté et sa clareté laissent à désirer au vue du manque de temps.
Le choix des `switch () case:` pour l'interface n'est pas la manière la plus propre pour faire ce genre de chose mais c'etait la solution la plus rapide à implémenter sans risquer de casser l'interface tout entière à la moindre erreure.
Pour les mêmes raisons de contraintes de temps, je n'ai pas pu regler toutes les erreures de fuites mémoires de valgrind.