Polytech Nice Sophia
SI3 / 2021-2022
Erick Gallesio

Programmation Procédurale

Feuille 8

Programmation modulaire

Gestion d'une table

> *Les exercices de cette feuille de TD mettent l'accent sur la construction de programmes modulaires. Le programme que vous devez écrire devra être formé de deux fichiers et la compilation devra se faire avec la commande* ***make****.*

Pour réaliser ce TD, vous devez récupérer l'archive tables.tgz

1 La structure de données *Table*

Nous allons définir une table triée par ordre croissant d'*identificateurs* (des mots) dont l'implémentation est masquée pour l'utilisateur. A chaque identificateur est associé le nombre d'occurrences de celui-ci (nombre de fois ou il a été ajouté dans la table).

Pour ce faire, nous allons définir:

- le type *Table* comme un type pointeur vers une structure C non définie ainsi qu'un ensemble de fonctions permettant la manipulation de cette table.

- le type *t_fonction* comme un pointeur vers une fonction ayant le prototype suivant:

```
void (*fonction) (char *elt, int nb);
```

Ces deux définitions devront être ajoutées au début du fichier `table.h` .

Contenu du fichier `table.h` :

```c
#ifndef TABLE_H
#define TABLE_H /* Protection contre les inclusions multiples */

/* Définition du type de fonction s'appliquant aux éléments d'une table */
#error A FAIRE

/* Type encapsulé pour la table: une table est un pointeur vers une
 * structure C dont on ne connaît pas le détail */
#error A FAIRE AUSSI


//
// Fonctions elementaires de manipulation de la table
//

/* Creation d'une table vide */
Table creer_table(void);


/* Insertion d'un élément dans la table triée. Si l'élément est déjà
 * présent dans la table, le compteur d'occurrences est incrémenté.
 * La fonction renvoie le nombre actuel d'occurrences de elt */
int ajouter_table(Table *table, char *elt);

/* Impression triée des éléments de la table */
void imprimer_table(Table table);

/* Appel d'une fonction sur chacun des éléments de la table */
void appliquer_table(Table table, t_fonction fonction);

/* Recherche du nombre d'occurrences d'un élément */
int rechercher_table(Table table, char *elt);

/* Destruction d'une table */
void detruire_table(Table *table);

#endif /* TABLE_H */
```

2 Implementation à l'aide d'une liste chaînée

Nous allons implémenter maintenant la table sous la forme d'une liste ordonnée simplement chaînée. Cette implémentation se fera dans le fichier `table_liste.c` (ne pas oublier d'inclure `table.h`).

Définir la structure de données équivalente (chaque cellule de la liste contiendra une chaîne de caractères, un entier et un pointeur vers la cellule suivante).

Implémenter les fonctions de manipulation de la table définies dans le fichier `table.h` donné précédemment.

3 Implémentation à l'aide d'un tableau

Nous allons implémenter la table définie précédemment sous la forme d'un tableau trié alloué dynamiquement. Cette implémentation se fera dans le fichier `table_tableau.c` . On pourra commencer par une version qui utilise un tableau de taille fixe, puis par une version où le tableau est rallongé si nécessaire (comme avec la fonction `readline` vue dans la feuille précédente).

4 Implémentation à l'aide d'un arbre (facultatif)

Dans cette troisième implémentation, nous allons utiliser un arbre binaire. L'implémentation se fera cette fois dans le fichier `table_arbre.c` . On rappelle qu'en général, on définit un arbre binaire par une structure (récursive) de donnée de la forme:

```
struct node {
    char* key;
    ....
    struct node *left;
    struct node *right;
};
```

Pour un nœud donné de l'arbre, les éléments dans le sous arbre de gauche sont plus petits que la clé (*a contrario* ceux dans le sous arbre de droite sont plus grands que la clé).

Les fonction de construction ou de parcours d'arbre sont en général plus faciles à écrire de façon récursive.