TP4
Dictionnaire et Table

Mathieu ARQUILLIERE

Table des matières

I – Présentation

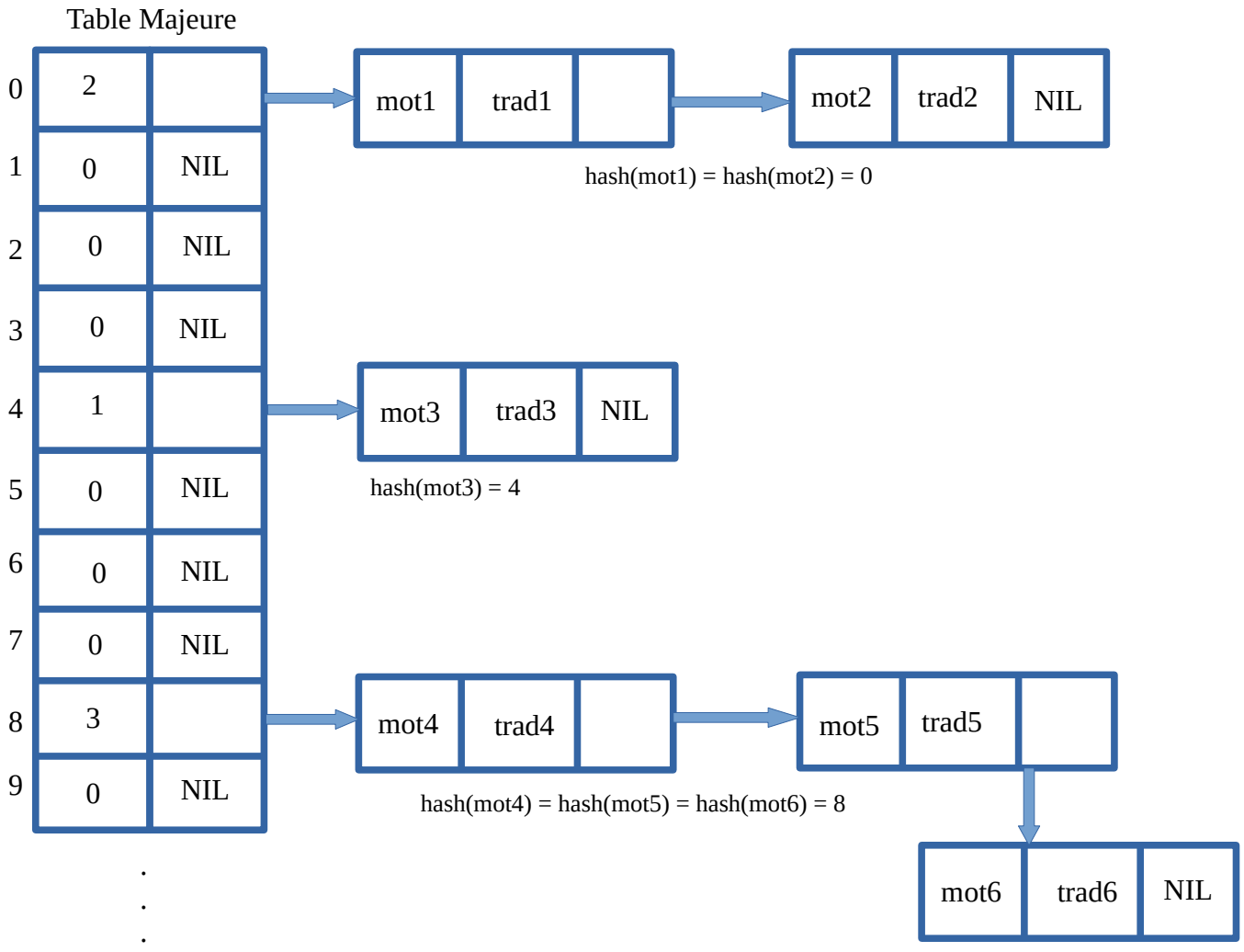
L'objectif de ce TP est de créer une bibliothèque de gestion de table pouvant notamment ici servir de dictionnaire en utilisant comme couple (clé ; valeur) le couple (mot ; traduction). Le programme devra donc utiliser des fichiers avec des mots afin de créer un dictionnaire en utilisant ce module. Il pourra également traduire des phrases avec le dictionnaire créé.

II – Description des structures de données

1) Table

La structure de donnée utilisée ici est une table de hachage indirect avec une table majeure (de taille 29) et une fonction de hachage donnée. La table majeure est un tableau contiguë de pointeurs de tête vers une liste chaînée des éléments ayant un mot avec un hachage donnant l'indice de ce pointeur dans la table. On ajoutera également un compteur d'éléments pour chaque sous-table.

Exemple de table (en prenant un fonction de hash « fictive » différente de celle donnée) :



2) Fichier de dictionnaire

Les fichiers contiennent des couples de mots (mot, traduction). Ainsi les fichiers traités doivent être de la forme :

- une ligne = un couple de mots
- un couple de mots est de la forme « mot;traduction »

Exemple de fichier (anglais-français.txt) :

```
hello;bonjour
happy;joyeux
clement;clement
thing;chose
computer;ordinateur
felix;le chat
mathis;ours
other;autre
mouse;souris
keyboard;clavier
screen;ecran
bottle;bouteille
phone;telephone
```

III – Organisation du code (module table)

Une table est représentée grâce à deux structures. L'une est une « case » de la table majeure (table_t) et l'autre est une cellule de sous-table (cell_t). La table est donc sous la forme d'un tableau de la structure table_t et les sous-table sont un chaînage des structures cell_t.

La structure cell_t contient une chaîne de caractères « mot » contenant le mot dont on a la traduction et dont le hash défini l'indice correspondant dans la table majeure. Elle contient également la chaîne de caractères « trad » contenant la traduction du mot et enfin elle contient un pointeur vers l'élément suivant de la sous-table.

```c
struct cell
{
    char* mot;
    char* trad;
    struct cell* suivant;
};
typedef struct cell cell_t;
```

Cette structure de cellule possède les fonctions de gestion de liste chaînée suivantes :

```c
int             creer_cell      (cell_t** nouv, char* mot, char* trad);
void            adj_cell        (cell_t* nouv, cell_t** prec);
void            liberer_cell    (cell_t** prec);
int             rech_cell       (cell_t* liste, char* mot, cell_t** trouve);
```

- **creer_cell** permet d'allouer et initialiser une cellule
- **adj_cell** permet de relier une cellule dans une liste grâce à un précédent
- **liberer_cell** permet de supprimer une cellule dans une liste et de libérer la mémoire associée
- **rech_cell** permet de trouver une cellule dans une liste via un mot

```c
struct table
{
    int compteur;
    cell_t* premier;
};
typedef struct table table_t;
```

La structure table_t elle contient donc un compteur qui correspond au nombre d'élément dans la sous-table et un pointeur vers le premier élément de la sous-table.

```c
table_t*        init_table      ();
void            ajouter_table   (table_t* MAJ, char* mot, char* trad);
int             recherche_table (table_t* MAJ, char* mot, cell_t** trouve);
void            debug_table     (table_t* MAJ);
void            liberer_table   (table_t* MAJ);
```

Afin de gérer la table, on a les fonctions associées suivantes :

- **init_table** permet d'allouer une table majeure (tableau de table_t)
- **ajouter_table** permet d'ajouter dans une table un mot et une traduction
- **recherche_table** permet de rechercher un mot dans une table
- **debug_table** permet d'afficher une table
- **liberer_table** permet de libérer la mémoire associée à une table

IV – Présentation du programme

1) Module table

cf. code de dictionnaire.h et dictionnaire.c

2) Programme principal

Le programme du code du fichier « test.c » contient les fonctions qui se servent du module de table et créent des dictionnaires à partir de fichier texte et traduisent des phrases grâce à ces dictionnaires.

Afin de tester l'efficacité et le fonctionnement de ces fonctions, la fonction principale « main » (qui se trouve également dans ce fichier) teste plusieurs cas possibles avec plusieurs fichiers textes. Pour plus de précision sur les tests réalisés, cf. compte-rendu d'exécution du programme de tests.

V – Compte-rendu d’exécution

1) Tests de création de dictionnaire

Liste des cas à traiter :

- fichier « classique » respectant les normes
- fichier vide
- fichier non existant
- fichier « corrompu », ne respectant pas les normes

Fichier classique :

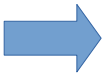
|indice table majeure|(nombre d’élément sous-table)

mot;traduction

```
charger_fichier(dicoEN_FR, "anglais-français.txt");
printf("\n=== Table dicoEN_FR ===\n");
debug_table(dicoEN_FR);
```

anglais-français.txt :

```
hello;bonjour
happy;joyeux
clement;clement
thing;chose
computer;ordinateur
felix;le chat
mathis;ours
other;autre
mouse;souris
keyboard;clavier
screen;ecran
bottle;bouteille
phone;telephone
```



```
=== Table dicoEN_FR ===
|0|(1)
 \-> mouse;souris
|1|(2)
 \-> phone;telephone
 \-> other;autre
|2|(0)
|3|(1)
 \-> thing;chose
|4|(0)
|5|(0)
|6|(1)
 \-> hello;bonjour
|7|(1)
 \-> keyboard;clavier
|8|(1)
 \-> screen;ecran
|9|(0)
|10|(0)
|11|(0)
|12|(1)
 \-> computer;ordinateur
|13|(1)
 \-> felix;le chat
|14|(1)
 \-> happy;joyeux
|15|(0)
|16|(2)
 \-> bottle;bouteille
 \-> mathis;ours
|17|(0)
|18|(0)
|19|(0)
|20|(0)
|21|(0)
|22|(1)
 \-> clement;clement
|23|(0)
|24|(0)
|25|(0)
|26|(0)
|27|(0)
|28|(0)
```

Fichier vide et fichier non existant :

```
charger_fichier(dicoVide, "vide.txt");
printf("\n=== Table dicoVide ===\n");
debug_table(dicoVide);
```

```
charger_fichier(dicoNExist, "dicoNExist.txt");
printf("\n=== Table dicoNExist ===\n");
debug_table(dicoNExist);
```

```
=== Table dicoVide ===
|0|(0)
|1|(0)
|2|(0)
|3|(0)
|4|(0)
|5|(0)
|6|(0)
|7|(0)
|8|(0)
|9|(0)
|10|(0)
|11|(0)
|12|(0)
|13|(0)
|14|(0)
|15|(0)
|16|(0)
|17|(0)
|18|(0)
|19|(0)
|20|(0)
|21|(0)
|22|(0)
|23|(0)
|24|(0)
|25|(0)
|26|(0)
|27|(0)
|28|(0)
```

```
=== Table dicoNExist ===
|0|(0)
|1|(0)
|2|(0)
|3|(0)
|4|(0)
|5|(0)
|6|(0)
|7|(0)
|8|(0)
|9|(0)
|10|(0)
|11|(0)
|12|(0)
|13|(0)
|14|(0)
|15|(0)
|16|(0)
|17|(0)
|18|(0)
|19|(0)
|20|(0)
|21|(0)
|22|(0)
|23|(0)
|24|(0)
|25|(0)
|26|(0)
|27|(0)
|28|(0)
```

Avec vide.txt un fichier vide et dicoNExist.txt un fichier qui n'existe pas

Fichier corrompu :

```
charger_fichier(dicoCorr, "dicoCorr.txt");
printf("\n=== Table dicoCorr ===\n");
debug_table(dicoCorr);
```

```
Segmentation fault (core dumped)
```

```
ce fichier est corrompu
il ne suit; pas les normes
imposées;
```

dicoCorr.txt

La fonction charger_fichier ne supporte donc pas les fichiers qui ne respectent pas la norme définie.

Dans les cas où le dictionnaire n'est pas vide, il n'y a pas de différence de comportement. En effet la fonction « charger_fichier » ne fait qu'appeler la fonction « ajouter_table ». Si la table comporte déjà des éléments alors :

- si on rajoute le même élément, seule la traduction est modifiée
- si l'élément n'existe pas dans la table, on créer une cellule et on la rajoute dans la table

C'est pour cela qu'on peut également mettre plusieurs fois le même mot dans le même fichier sans poser de problème au chargement.

2) Tests de traductions

Liste des cas à traiter :

- traduire une chaîne de caractères vide
- traduire une chaîne ayant plusieurs séparateurs (point, espace, …)
- traduire une chaîne ayant plusieurs mots à traduire
- traduire une chaîne avec un dictionnaire vide
- traduire une chaîne avec des mots à traduire et d'autres qui ne sont pas dans le dictionnaire

Chaîne vide :

```
printf("\n=== Traduction chaineVide ===\n");
traduction(dicoEN_FR, chaineVide, trad);
printf("texte: %s\n", chaineVide);
printf("trad: %s\n\n", trad);
```

```
=== Traduction chaineVide ===
texte:
trad:
```

Chaîne avec plusieurs séparateurs, plusieurs mots à traduire, plusieurs mots sans traduction :

```
printf("\n=== Traduction phrase ===\n");
traduction(dicoEN_FR, texte, trad);
printf("texte: %s\n", texte);
printf("trad: %s\n", trad);
```

```
=== Traduction phrase ===
texte: where is bryan? Bryan is behind his computer.
he move his mouse and type his name on the screen with his keyboard
trad: ou est bryan? Bryan est derriere son ordinateur.
il bouge son souris et tape son nom sur le ecran avec son clavier
```

```
hello;bonjour
happy;joyeux
clement;clement
thing;chose
computer;ordinateur
felix;le chat
mathis;ours
other;autre
mouse;souris
keyboard;clavier
screen;ecran
bottle;bouteille
phone;telephone
where;ou
is;est
behind;derriere
his;son
he;il
move;bouge
on;sur
and;et
name;nom
type;tape
with;avec
the;le
```

Dictionnaire vide :

```
printf("\n=== Traduction dicoVide ===\n");
traduction(dicoVide, texte, trad);
printf("texte: %s\n", texte);
printf("trad: %s\n", trad);
```

```
=== Traduction dicoVide ===
texte: where is bryan? Bryan is behind his computer.
he move his mouse and type his name on the screen with his keyboard
trad: where is bryan? Bryan is behind his computer.
he move his mouse and type his name on the screen with his keyboard
```

VI – Makefile

Afin de compiler les programmes, on utilise un makefile :

Ce makefile génère l'exécutable faisant les tests sur le module de table et l'utilisation des dictionnaires à partir de fichiers et les traductions..

```
OPT = -g -Wall -Wextra
LOG = @echo "\#MAKE"
TEST = test

all: $(TEST)

$(TEST): test.c dictionnaire.o
    gcc -o $(TEST) test.c dictionnaire.o $(OPT)
    $(LOG) "executable $(TEST) généré"

dictionnaire.o: dictionnaire.h dictionnaire.c
    gcc -o dictionnaire.o -c dictionnaire.c $(OPT)
    $(LOG) "lib dictionnaire généré"

clean:
    rm *.o
    rm $(TEST)
```

Il a besoin du module de table (donc dictionnaire.o) et du code source de test (test.c).

On a donc une règle pour le .o et une règle pour l'exécutable.

Commandes :

- "make" pour faire l'exécutable « test »
- "make clean" pour effacer l'exécutable et le .o
- "./test" pour lancer le programme