Devoir surveillé

22 octobre 2019 — Durée 1h15

Document autorisé : **Mémento C** vierge de toute annotation manuscrite

Introduction

Une table clef/valeur est une structure de donnée dans laquelle on peut (i) stocker des couples clefs/valeurs ; (ii) rechercher la valeur associée à une clef ; ou encore (iii) supprimer une clef.

Il s'agit d'une structure de donnée très utilisée car, bien implémentée et dans de bonnes conditions, les trois opérations peuvent se faire en temps quasiment constant.

Dans ce devoir, les clefs et les valeurs seront toujours des entiers (type `int`).

Définition du type `TableCV`

Le paquetage `type_table_clef_valeur` (en Annexe A) définit le type `TableCV` utilisé dans tout ce devoir. Il implémente aussi la fonction `initialiser_table` qui doit être appellée avant la première utilisation d'une table clef/valeur.

Une `TableCV` est un tableau de `MAX_ELEM` cellules. Chaque cellule contient une structure `Paire`. Cette structure contient trois champs : la `clef`, la `valeur` associée, ainsi qu'un champ ayant le type `Bit_Validite` qui peut prendre deux valeurs : `PRESENT` ou `ABSENT` et indique si la cellule est actuellement utilisée.

Finalement, le type `ResultatRecherche` est défini. Il s'agit d'une structure utilisée comme valeur de retour pour les opérations de recherche de clef. Cette structure contient un champ `present` du type `Bit_Validite` qui indique si la clef est presente ou absente. Le second champ `valeur` contient la valeur associée à la clef recherchée, le cas échéant (et n'est pas utilisé en cas d'absence).

Élements de spécification des opérations

Ajout d'un couple $\langle c, v \rangle$ **:** Modifie la table de sorte que la valeur v soit associée à la clef c ;

Suppression d'une clef c **:** Modifie la table de sorte qu'il n'y ait pas de valeur associée à la clef c ;

Chercher une clef c **:** Renvoie un `ResultatRecherche` dont le champ `valeur` vaut la valeur v associée à la clef c dans la table (qui reste inchangée). Si la clef est absente, le champ `present` est `ABSENT`.

Le fichier de spécification (fichier d'en-tête `.h`) est fourni en Annexe B.

Organisation du devoir

Le but de ce devoir est :

— De comprendre comment fonctionnent les opérations d'une table clef/valeur. Pour ce faire, on va écrire un programme qui lit une séquence d'opérations à executer puis execute ces opérations. Dans cette partie, on utilisera une implémentation naïve d'une table clef/valeur. *(Première partie)*

— De comprendre quelles sont les propriétés qui doivent être garanties par une implémentation. On cherchera à construire un jeu de tests pour s'assurer que la version naïve y répond bien. *(Seconde partie)*

— On étudiera enfin une deuxième implémentation optimisée. Pour s'assurer que la version optimisée est correcte, on implémentera un oracle qui se basera sur la version naïve. *(Troisième partie)*

— Finalement, on verra comment passer d'une implémentation à l'autre et les modifications que ça implique. *(Dernière partie)*

Première partie

Exercice 1. [4pts]

Fichier de commandes et paquetages associés

On se donne un format de fichier de commandes d'accès à une table clef/valeur. Le format est le suivant :

- La première ligne contient un unique entier n.
- Le reste du fichier contient n lignes de commande d'accès.
- Une ligne de commande d'accès peut prendre trois formes :
  - Un caractère `A` suivi de deux entiers.
  - Un caractère `C` suivi d'un entier.
  - Un caractère `S` suivi d'un entier.

Un exemple d'un fichier de commandes d'accès est fourni en Annexe C. Le paquetage `commande_es` implémente des fonctions pour lire un tel fichier.

Le paquetage `commande_es` utilise les types `Commande` et `Commandes` (notez le pluriel) définis dans le paquetage `commande` :

- Le type `Commande` est une structure qui contient trois champs : (i) un type de commande (qui peut prendre trois valeurs : (a) `AJOUTER`; (b) `SUPPRIMER`; et (c) `CHERCHER`); (ii) une première opérande, qui contient toujours la clef sur laquelle l'opération agit et (iii) une seconde opérande, utilisée seulement pour l'opération `AJOUTER` qui contient la valeur à associer à la clef fournie en première opérande.
- Le type `Commandes` est une structure qui contient deux champs : (i) le champ `nb_cmds` de type `int` qui contient le nombre de commandes; et (ii) le champ `cmds` qui contient un pointeur de type `Commande*` qui pointe vers un tableau de `nb_cmds` commandes.

Le fichier de spécification du paquetage est fourni en Annexe D.

Question

En utilisant les paquetages `commande_es`, `commande` et `operation_table_clef_valeur_naive`, écrire un programme de test `test_commandes` qui lit une séquence d'accès depuis un fichier, puis execute cette séquence.

Un exemple d'execution du programme `test_commandes` utiliant le fichier en Annexe C est donné en Annexe F. Les commandes `A` et `S` sont silencieuses (n'affichent rien) et seule la commande `C` affiche le résultat sur la sortie standard.

```
1  #include "operation_table_clef_valeur_naive.h"
2  #include "type_table_clef_valeur.h"
3  #include "commande.h"
4  #include "commande_es.h"
5
6  int main(int argc, char* argv[]) {
7      if(argc != 2) {
8          fprintf(stderr, "Usage: %s <Fichier commandes>\n", ↩
               ↪ argv[0]);
9          return -1;
10     }
11
12     FILE* fcmd = fopen(argv[1], "r");
13     if (fcmd == NULL) {
14         fprintf(stderr, "Impossible d'ouvrir le fichier %s"↩
               ↪ , argv[1]);
15         return -1;
16     }
17     Commandes cmds = lire_commandes(fcmd);
18     TableCV table;
19     initialiser_table(&table);
20     for (int i = 0; i < cmds.nb_cmds; i++) {
21         int op1 = cmds.cmds[i].operande1;
```

```
22          ResultatRecherche res_chercher;
23          switch (cmds.cmds[i].tc) {
24              case AJOUTER:
25                  ajouter(op1, cmds.cmds[i].operande2, &table)↩
                        ↪ ;
26                  break;
27              case SUPPRIMER:
28                  supprimer(op1, &table);
29                  break;
30              case CHERCHER:
31                  res_chercher = chercher(op1, table);
32                  if (res_chercher.present == ABSENT) {
33                      printf("La clef %d n'est pas présente\n",↩
                            ↪  op1);
34                  } else {
35                      printf("La clef %d est associée àla ↩
                            ↪ valeur %d\n",
36                          op1, res_chercher.valeur);
37                  }
38                  break;
39          }
40      }
41  }
```

Seconde partie

Exercice 2. [3pts]

Quelles propriétés doivent être garanties par implémentation d'une table clef/valeur (quel est le comportement attendu de chaque fonction) ?

Indice : on peut utiliser la notion d'*être dans la table* pour un couple et expliquer ce que chaque opération fait en fonction de cette notion d'inclusion.

Exemple : "Propriété 1 : Après l'opération O avec les opérandes o_1, o_2, les élements qui sont dans la table sont les élements e_1, e_i, etc.."

On étudie les propriétés pour chaque opération :

Fonction `initialiser_table(TableCV *t)`, *après un appel à* `initialiser_table(t)` *où* `t` *est un pointeur valide (pas demandé) :*

— *P0 : la table* `t` *ne contient aucun couple.*

Fonction `ajouter(int clef, int valeur, TableCV* t)`, *après un appel à* `ajouter(c, v, t)` *où* `t` *est un pointeur valide :*

— *P1 : Tous les couples $\langle c_1, v_1\rangle$ tels que (i) $\langle c_1, v_1\rangle \in t$ avant l'appel ; et (ii) $c1 \neq c$ sont dans t après l'appel ;*

— *P2 : Le couple $\langle c, v\rangle \in t$ après l'appel ;*

— *P3 : t ne contient pas d'autres couples.*

Fonction `supprimer(int clef, TableCV* t)`, *après un appel à* `supprimer(c, t)` *où* `t` *est un pointeur valide :*

— *P4 : Tous les couples $\langle c_1, v_1\rangle$ tels que (i) $\langle c_1, v_1\rangle \in t$ avant l'appel ; et (ii) $c1 \neq c$ sont dans t après l'appel ;*

— *P5 : t ne contient pas d'autres couples.*

Fonction `chercher(int clef, TableCV t)`, *après un appel à* `chercher(c, t)` *:*

— *P6 : Si $\exists v, \langle c, v\rangle \in t$, alors l'appel renvoie $\langle$`PRESENT`$, $`v`$\rangle$.*

— *P7 : Si $\neg\exists v, \langle c, v\rangle \in t$, alors l'appel renvoie $\langle$`ABSENT`$, \_\rangle$.*

Un pointeur est valide *s'il pointe vers une zone allouée de taille correcte.*

Note : dans ce corrigé, on a fait le choix, si on ajoute un couple $\langle c, v\rangle$ et que la clef c est déjà associée à v_0, de remplacer v_0 par v (propriété P1). Il est aussi tout à fait correct de refuser l'ajout d'une clef déjà existante. Les propriétés P1 et P2 sont alors remplacées par :

— *P1′ : Tous les couples $\langle c_1, v_1\rangle \in t$ avant l'appel sont dans t après l'appel ;*

— *P2′ : Si, $\forall\langle c_1, v_1\rangle \in t$ avant l'appel $c_1 \neq c$, alors $\langle c, v\rangle \in t$ après l'appel.*

Exercice 3. [5pts]

Les fichiers de commandes d'accès peuvent servir de tests fonctionnels. En partant de votre réponse à la question précédente, décrire comment construire un jeu de tels fichiers pour tester la correction d'une implémentation d'une table clef/valeur.

Remarque : Les tests doivent être des tests fonctionnels.

On veut tester toutes les propriétés qui ont été décrites à la question précédente :

P1 & P6 : *On peut enchainer l'ajout de deux couples $\langle c_1, v_1\rangle$ et $\langle c_2, v_2\rangle$ (avec $c_1 \neq c_2$), puis chercher la clef c_1 qui doit renvoyer $\langle \texttt{PRESENT}, v_1\rangle$ (l'ajout du second couple ne supprime pas le premier).*

P2 & P6 : *On peut ensuite chercher c_2, qui doit renvoyer $\langle \texttt{PRESENT}, v_2\rangle$ (le second couple ajouté est bien dans la table).*

P3 & P7 : *On peut chercher une clef $c_3 \neq c_1$, $c_3 \neq c_2$, qui doit renvoyer $\langle \texttt{ABSENT}, \_\rangle$.*

P5 & P7 : *On peut supprimer c_1, puis chercher c_1 et c_3, ce qui doit renvoyer $\langle \texttt{ABSENT}, \_\rangle$ à chaque fois.*

P4 & P6 : *On peut ensuite rechercher c_2, ce qui doit renvoyer $\langle \texttt{PRESENT}, v_2\rangle$.*

On oubliera pas de faire cette procédure pour plusieurs valeurs de c_1, c_2 et c_3, ainsi qu'en entremelant plusieurs valeurs selon la même procédure.

Note : on adaptera les tests des propriétés P1 et P2 si on a choisi la version sans remplacement (voir P1' et P2' dans l'exercice 1).

Troisième partie

Exercice 4. [4pts]

Dans cet exercice, on introduit une version optimisée des opérations sur les tables clefs valeurs. Le but de l'exercice est d'écrire un oracle qui vérifie que la version optimisée est correcte en la comparant à la version naïve.

Dans cet exercice, on admet que l'implémentation naïve est correcte.

Le fichier d'en-tête (fichier `.h`) est donné en Annexe G. Les operations portent le même nom que leur équivalent naïf, avec le suffixe `_opt`.

Remarque : L'implémentation optimisée ne change que l'implémentation des opérations, et pas le type `TableCV`, qui reste commun aux deux implémentations.

Question

Écrire un oracle `oracle` qui vérifie la correction de l'implémentation optimisée en se basant sur l'implémentation naïve.

Indice : on pourra s'inspirer du programme `test_commandes`.

```
1  #include "type_table_clef_valeur.h"
2  #include "operation_table_clef_valeur_naive.h"
3  #include "operation_table_clef_valeur_optimisee.h"
4  #include "commande.h"
5  #include "commande_es.h"
6
7  int main(int argc, char* argv[]) {
8      if(argc != 2) {
9          fprintf(stderr, "Usage: %s <Fichier commandes>\n", ↩
               ↪ argv[0]);
10         return -1;
11     }
12
```

```
13      FILE* fcmd = fopen(argv[1], "r");
14      if (fcmd == NULL) {
15          fprintf(stderr, "Impossible d'ouvrir le fichier %s"↩
                ↪ , argv[1]);
16          return -1;
17      }
18      Commandes cmds = lire_commandes(fcmd);
19      TableCV table_naive;
20      TableCV table_optimisee;
21      initialiser_table(&table_optimisee);
22      initialiser_table(&table_naive);
23
24      for (int i = 0; i < cmds.nb_cmds; i++) {
25          int op1 = cmds.cmds[i].operande1;
26          ResultatRecherche res_chercher_naive;
27          ResultatRecherche res_chercher_optimisee;
28          switch (cmds.cmds[i].tc) {
29              case AJOUTER:
30                  ajouter(op1, cmds.cmds[i].operande2, &↩
                        ↪ table_naive);
31                  ajouter_opt(op1, cmds.cmds[i].operande2, &↩
                        ↪ table_optimisee);
32                  break;
33              case SUPPRIMER:
34                  supprimer(op1, &table_naive);
35                  supprimer_opt(op1, &table_optimisee);
36                  break;
37              case CHERCHER:
38                  res_chercher_naive = chercher(op1, ↩
                        ↪ table_naive);
39                  res_chercher_optimisee = chercher_opt(op1, ↩
                        ↪ table_optimisee);
40                  if (res_chercher_naive.present != ↩
                        ↪ res_chercher_optimisee.present) {
41                      fprintf(stderr, "À la ligne %d, l'implé↩
                            ↪ mentation naive renvoie %s et l'↩
                            ↪ implémentation optimisée renvoie %↩
                            ↪ s.\n",
42                              i+2,
43                              res_chercher_naive.present == ↩
                                    ↪ PRESENT ? "\"PRESENT\"" : "↩
                                    ↪ \"ABSENT\"",
44                              res_chercher_optimisee.present == ↩
                                    ↪ PRESENT ? "\"PRESENT\"" : "↩
                                    ↪ \"ABSENT\"");
45                      return -1;
46                  }
47                  if (res_chercher_naive.present == PRESENT
48                          && res_chercher_naive.valeur != ↩
                                ↪ res_chercher_optimisee.valeur) ↩
                                ↪ {
49                      fprintf(stderr, "À la ligne %d, l'implé↩
                            ↪ mentation naive renvoie %d et l'↩
                            ↪ implémentation optimisée renvoie %↩
                            ↪ d.\n",
50                              i+2, res_chercher_naive.valeur, ↩
                                    ↪ res_chercher_optimisee.↩
                                    ↪ valeur);
51                      return -1;
52
53                  }
54                  break;
55          }
56      }
57
```

```
58      fprintf(stderr, "Pas d'erreur détectée.\n");
59  }
```

Dernière partie

Exercice 5. [4pts]

Dans cette question, on admettra que le nom des fonctions optimisées est le même que le nom des fonctions naïves. De fait, un programmeur n'a pas besoin de changer le nom de ses appels pour passer d'une implémentation à l'autre.

Décrire les dépendances entre les paquetages pour compiler le programme `test_commandes` de l'exercice 1 (on pourra par exemple dessiner l'arbre de dépendances).

> **`test_commandes` :** *dépend de* `operation_table_clef_valeur_naive`, `type_table_clef_valeur`, `commande` *et* `commande_es`;
>
> **`commande_es` :** *dépend de* `commande`;
>
> **`commande` :** *ne dépend d'aucun paquetage;*
>
> **`operation_table_clef_valeur_naive` :** *dépend de* `type_table_clef_valeur`;
>
> **`type_table_clef_valeur` :** *ne dépend d'aucun paquetage.*

Quelles dépendances changent si l'on veut passer de l'implémentation naïve à l'implémentation optimisée ?

> `test_commandes` *ne dépend plus de* `operation_table_clef_valeur_naive` *mais de* `operation_table_clef_valeur_optimisee`*. Les dépendances du paquetage* `operation_table_clef_valeur_optimisee` *sont les mêmes que celles du paquetage* `operation_table_clef_valeur_naive`*.*

Indiquez les modifications que cela implique sur le Makefile (en Annexe H). Répondez sur votre copie, décrivez uniquement les lignes qui changent dans le Makefile, par exemple : *La ligne 10 devient ....*

> *À la ligne 10,* `operation_table_clef_valeur_naive.o` *est changé en* `operation_table_clef_valeur_optimisee.o`*.*
>
> *La règle pour construire* `operation_table_clef_valeur_naive.o` *est renommée pour construire* `operation_table_clef_valeur_optimisee.o`*.*

Annexes

A. Paquetage type_table_clef_valeur

Le fichier type_table_clef_valeur.c n'est pas nécessaire pour ce devoir.

Fichier type_table_clef_valeur.h :

```
1  #ifndef __TYPE_TABLE_CLEF_VALEUR__
2  #define __TYPE_TABLE_CLEF_VALEUR__
3
4  #define MAX_ELEM 20
5
6  typedef enum {PRESENT, ABSENT} Bit_Validite;
7
8  typedef struct {
9     Bit_Validite b;
10    int clef;
11    int valeur;
12 } Paire;
13
14 typedef struct {
15    Paire associations[MAX_ELEM];
16 } TableCV;
17
18 typedef struct {
19    Bit_Validite present;
20    int valeur;
21 } ResultatRecherche;
22
23 void initialiser_table(TableCV* t);
24
25 #endif
```

B. Paquetage operation_table_clef_valeur_naive

Le fichier operation_table_clef_valeur_naive.c n'est pas nécessaire pour ce devoir.

Fichier operation_table_clef_valeur_naive.h :

```
1  #ifndef __OPERATION_TABLE_CLEF_VALEUR_NAIVE__
2  #define __OPERATION_TABLE_CLEF_VALEUR_NAIVE__
3
4  #include "type_table_clef_valeur.h"
5
6  void ajouter(int clef, int valeur, TableCV* t);
7  void supprimer(int clef, TableCV* t);
8  ResultatRecherche chercher(int clef, TableCV t);
9
10 #endif
```

C. Exemple de fichier de commandes

```
1  3
2  A 12 3
3  C 12
4  S 12
```

D. Paquetage commande_es.h

Fichier commande_es.h :

```c
#include <stdio.h>
#include "commande.h"

// Renvoie un tableau de commandes
Commandes lire_commandes(FILE* f);
```

E. Paquetage commande

Fichier commande.h :

```c
#ifndef _COMMANDE_H
#define _COMMANDE_H

typedef enum {AJOUTER, SUPPRIMER, CHERCHER} Type_Commande;

typedef struct {
   Type_Commande tc;
   int operande1;
   int operande2;
} Commande;

/* Le champs cmds est un pointeur vers un tableau de 'Commande's.
 * Le champs nb_cmds est un entier qui indique la taille du tableau. */
typedef struct {
   int nb_cmds;
   Commande* cmds;
} Commandes;

#endif
```

F. Exemple d'execution de test_commandes

```
moi@ordinateur> ./test_commandes exemple_fichier_acces
La clef 12 est associée àla valeur 3
```

G. Paquetage operation_table_clef_valeur_optimisee

Le fichier operation_table_clef_valeur_optimisee.c n'est pas nécessaire pour ce devoir.

Fichier operation_table_clef_valeur_optimisee.h :

```c
#ifndef __OPERATION_TABLE_CLEF_VALEUR_OPTIMISEE__
#define __OPERATION_TABLE_CLEF_VALEUR_OPTIMISEE__

#include "type_table_clef_valeur.h"

void ajouter_opt(int clef, int valeur, TableCV* t);
void supprimer_opt(int clef, TableCV* t);
ResultatRecherche chercher_opt(int clef, TableCV t);

#endif
```

H. Makefile

Fichier Makefile utilisé dans le devoir :

```
C_FLAGS=-g
CC=clang
```

```
3
4  all: test_commandes oracle
5
6  commande_es.o: commande_es.c
7
8  test_commandes.o: test_commandes.c operation_table_clef_valeur_naive↩
       ↪ .h commande.h commande_es.h
9
10 test_commandes: test_commandes.o operation_table_clef_valeur_naive.o↩
       ↪  commande_es.o type_table_clef_valeur.o
11     $(CC) $^ -o $@
12
13 oracle: oracle.o type_table_clef_valeur.o ↩
       ↪ operation_table_clef_valeur_naive.o ↩
       ↪ operation_table_clef_valeur_optimisee.o commande_es.o
14     $(CC) $^ -o $@
15
16 clean:
17     -rm *.o test_commandes test_es oracle
18
19 %.o:%.c
20     $(CC) $(C_FLAGS) -c $^
```

Remarque : Les flèches rouges (↩ et ↪) ne sont pas présentes dans le Makefile et indiquent que les sauts de ligne ont été rajoutés en raison de la mise-en-page et n'étaient pas présents initialement dans le Makefile.