Projet court: hashtable

Julien Bernard

Le but de ce projet court est d'implémenter une *table de hachage*. 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 clé. Dans ce projet, les clés seront des chaînes de caractères et les éléments seront des «valeurs» [1] qui pourront représenter des objets de différents types. Les clés sont uniques dans la table de hachage, c'est-à-dire qu'on ne peut pas avoir deux valeurs avec la même clé.

Exercice 1 : Les valeurs

Dans cette première partie, on implémente une structure `struct value`. Cette structure est une union tagguée avec un champ `kind` qui indique quel est le type de la valeur, et un champ `as` qui est une union entre différents types de données.

```
enum value_kind {
  VALUE_NIL,
  VALUE_BOOLEAN,
  VALUE_INTEGER,
  VALUE_REAL,
  VALUE_CUSTOM,
};

struct value {
  enum value_kind kind;
  union {
    bool boolean;
    int64_t integer;
    double real;
    void *custom;
  } as;
};
```

C'est une structure qu'on rencontre parfois dans l'implémentation de langage de script, comme en Lua par exemple. Elle permet de n'avoir qu'un seul type du point de vue du langage d'implémentation (souvent le C), tout en ayant des types différents du point de vue du langage implémenté.

`VALUE_NIL` permet de représenter l'équivalent de `null` ou `nil` (suivant les langages), il n'y a pas de donnée associée. Le champ `custom` sert à stocker

1. L'important, c'est les valeurs.

des objets non-primitifs, des chaînes de caractères par exemple, de manière générique. Dans ce cas, c'est l'utilisateur qui s'assure de la gestion mémoire de l'objet lui-même.

Question 1.1 Écrire un ensemble de fonctions pour connaître le type de la valeur.

```
enum value_kind value_get_kind(const struct value *self);

bool value_is_nil(const struct value *self);
bool value_is_boolean(const struct value *self);
bool value_is_integer(const struct value *self);
bool value_is_real(const struct value *self);
bool value_is_custom(const struct value *self);
```

Question 1.2 Écrire un ensemble de fonctions pour définir une valeur.

```
void value_set_nil(struct value *self);
void value_set_boolean(struct value *self, bool val);
void value_set_integer(struct value *self, int64_t val);
void value_set_real(struct value *self, double val);
void value_set_custom(struct value *self, void *val);
```

Question 1.3 Écrire un ensemble de fonctions pour récupérer une valeur. On supposera pour chacune des fonctions que la valeur est du bon type. On pourra éventuellement mettre un `assert` pour le vérifier.

```
bool value_get_boolean(const struct value *self);
int64_t value_get_integer(const struct value *self);
double value_get_real(const struct value *self);
void *value_get_custom(const struct value *self);
```

Question 1.4 Écrire un ensemble de fonctions pour créer une valeur.

```
struct value value_make_nil();
struct value value_make_boolean(bool val);
struct value value_make_integer(int64_t val);
struct value value_make_real(double val);
struct value value_make_custom(void *val);
```

Exercice 2 : La table de hachage

Dans cette partie, on implémente la table de hachage proprement dite.

Concrètement, une table de hachage est un tableau de k listes chaînées [2]. 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 $(h\%k)$.

2. Ce n'est pas la seule manière d'implémenter une table de hachage mais c'est une des plus simples et c'est celle que je vous propose d'implémenter

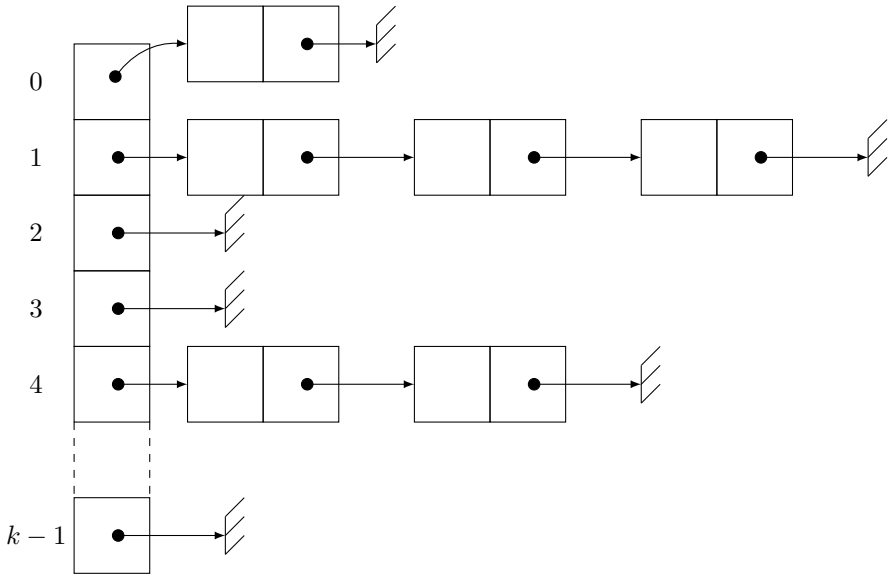

FIGURE 1 – Schéma d'une table de hachage

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 n divisé par le nombre de cases du tableau k.

Quand le facteur de compression dépasse une certaine valeur pour laquelle le risque de collision devient important ($\frac{1}{2}$ 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.

La figure 1 montre un schéma d'une table de hachage où des collisions ont eu lieu pour les indices 1 et 4 mais pas pour l'indice 0.

La structure est définie de la manière suivante en C.

```
struct bucket {
  char *key;
  struct value value;
  struct bucket *next;
};

#define HASHTABLE_INITIAL_SIZE 4

struct hashtable {
  struct bucket **buckets;
  size_t count; // number of elements in the table
  size_t size;  // size of the buckets array
};
```

La structure `struct bucket` représente un maillon de la liste chaînée. Il contient une clé (sous forme de chaîne de caractères), une valeur à laquelle est

associée la clé, et un pointeur vers l'élément suivant.

La structure `struct hashtable` représente la table de hachage, c'est un tableau dynamique de listes chaînées. `count` est le nombre d'éléments dans la table, soit n. `size` représente la taille du tableau, soit k.

La constante `HASHTABLE_INITIAL_SIZE` sert à fixer la taille initiale de la table de hachage vide. Vous devez respecter cette constante et vous ne devez pas changer sa valeur.

Pour la fonction de hachage, on utilisera un hash FNV-1a en 64 bits. On pourra utiliser le type `size_t` qui est généralement de 64 bits pour stocker la valeur du hash.

Question 2.1 Écrire les fonctions de création et destruction d'une table de hachage.

```
void hashtable_create(struct hashtable *self);
void hashtable_destroy(struct hashtable *self);
```

Question 2.2 Écrire les accesseurs pour le nombre d'éléments et la taille de la table de hachage.

```
size_t hashtable_get_count(const struct hashtable *self);
size_t hashtable_get_size(const struct hashtable *self);
```

Question 2.3 Écrire une fonction qui effectue un rehash. On doublera la taille du tableau. Attention, cette fonction étant publique, elle peut être appelé même si un rehash n'est pas nécessaire.

```
void hashtable_rehash(struct hashtable *self);
```

Question 2.4 Écrire une fonction qui insère un couple clé-valeur dans la table de hachage. La chaîne de caractères de la clé devra être copiée et gérée en interne. On effectuera un rehash si nécessaire, c'est-à-dire si le facteur de compression est plus grand que $\frac{1}{2}$. La fonction renvoie `true` si la clé n'existait pas (et alors, un nouveau nœud a été ajouté) ou `false` dans le cas contraire, c'est-à-dire si la clé existait déjà (auquel cas la valeur a été remplacée).

```
bool hashtable_insert(struct hashtable *self, const char *key,
                                                struct value val);
```

Question 2.5 Écrire une fonction qui supprime un élément de la table de hachage. La fonction renvoie `true` si un élément a bien été supprimé et `false` dans le cas inverse.

```
bool hashtable_remove(struct hashtable *self, const char *key);
```

Question 2.6 Écrire une fonction qui dit si une clé est présente dans la table de hachage.

```
bool hashtable_contains(const struct hashtable *self,
                                                 const char *key);
```

Question 2.7 Écrire un ensemble de fonctions pour insérer des valeurs typées.

```
void hashtable_set_nil(struct hashtable *self, const char *key);
void hashtable_set_bool(struct hashtable *self,
                                    const char *key, bool val);
void hashtable_set_integer(struct hashtable *self,
                                  const char *key, int64_t val);
void hashtable_set_real(struct hashtable *self,
                                   const char *key, double val);
void hashtable_set_custom(struct hashtable *self,
                                    const char *key, void *val);
```

Question 2.8 Écrire une fonction qui permet de récupérer une valeur à partir d'une clé. Si la clé n'existe pas, on renverra la valeur `nil`.

```
struct value hashtable_get(struct hashtable *self,
                                                  const char *key);
```