L3-MI / PROG
TP#8 : Tables de Hachage

Jérôme David

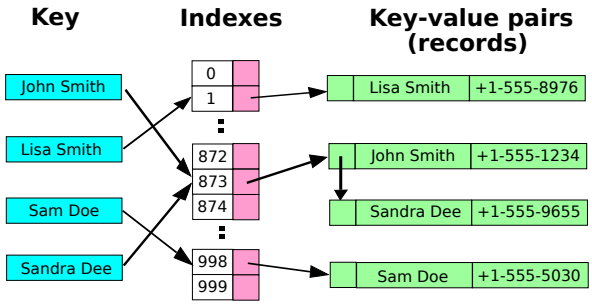
2024-11-19

Présentation

Le but de ce TP est d'implémenter une petite librairie pour représenter un dictionnaire/tableau associatif sous forme de table de hachage. Pour rappel les objectifs d'un tableau associatif est d'indexer des valeurs en fonction d'une clé et de pouvoir retrouver rapidement la valeur associée à une clé.

Dans une table de hachage, l'emplacement d'une paire clé-valeur est déterminé par l'application d'une fonction de hachage sur la clé (modulo la taille du tableau utilisé pour stocker les éléments). Cependant les fonctions de hachage ne sont pas en général pas injectives, et lorsque deux clés ont la même valeur de hachage on a une collision. Il existe différentes façons de gérer des collisions et dans ce TP nous utiliserons une résolution des collision par chaînage.

Le principe est illustré par la figure suivante (issue de wikipedia) :



Pour représenter, la table de hachage nous utiliserons les structures suivantes :

```
typedef struct Entry {
        struct Entry* next; // pointeur vers l'entrée suivante ou NULL
        void* value; // pointeur vers la valeur
        uint64_t hashcode; // valeur de hachage de la clé
        char key[]; // contenu de la clé
} Entry;

typedef struct HashMap {
        Entry** array; // tableau des entrées
        size_t capacity; // taille detu tableau 'array'
        size_t size;  // nombre d'entrées stockées dans la table de hachage
        uint64_t (*hash)(const char[static 1]); // pointeur vers la fonction
↪  utilisée pour le hachage
} HashMap;
```

La structure de table de hachage que nous proposons d'implémenter repose sur les principes suivants :

— Les clés seront des chaînes de caractères (recopiées dans la structure) et les valeurs des pointeurs génériques (de type `void*`).
— La position est donnée par la valeur de hachage de la clé modulo la capacité de la table.
— Chaque objet de type `struct HashMap` possède sa fonction de hachage.
— Lorsque que le facteur de charge dépasse 0.75 alors la taille du tableau des entrées (`array`) doit être doublée et les entrées re-indexées en conséquence.
— La taille (`array`) doit toujours être une puissance de 2.

Q.1 : Implémentez les fonctions suivantes de création, de destruction et d'information d'une table de hachage :

1. `HashMap* hm_create(size_t minCap, uint64_t (*hash_func)(const char[static 1]));` qui alloue (sur le tas) et initialise une table de hachage (et retourne un pointeur vers celle-ci). `minCap` donne la capacité minimale de table de hachage et `*hash_func` est un pointeur vers la fonction de hachage qui sera utilisée. Si `minCap` n'est pas une puissance de deux alors la taille du tableau sera la plus petite puissance de 2 supérieure à `minCap` tout en étant inférieure à la plus grande puissance de deux représentable par `size_t`.
2. `void hm_clear(HashMap* restrict self)` qui libère toute les entrées de la table de hachage. Les références vers les valeurs indexées ne seront pas libérées. A l'issue de cette fonction la taille de la table de hachage `self->size` doit être égal à 0. Le tableau `self->array` garde sa taille.
3. `void hm_destroy(HashMap* restrict self)` qui libère toute la mémoire utilisée par la table de hachage (toujours sans libérer la mémoire pointée par les valeurs).
4. `void hm_size(HashMap* restrict self)` qui retourne le nombre d'entrées de la table.
5. `void hm_capacity(HashMap* restrict self)` qui retourne la capacité actuelle.

Testez vos fonctions, et vérifiez avec `valgrind` que vous n'avez pas de fuite mémoire ou d'accès mémoire illégaux[1].

Q.2 : Implémentez les fonctions suivantes d'ajout, de recherche, et suppression d'éléments (il est conseillé d'introduire une fonction permettant de factoriser le code de recherche) :

1. `void* hm_put(HashMap* self, const char key[static 1], void* value)` qui permet d'associer la paire (`key`,`value`) dans la table de hachage passée en paramètre. Si la clé n'est pas déjà présente alors une entrée sera ajoutée (et `self->size` sera incrémenté), sinon la valeur actuellement associée sera remplacée par celle passée en paramètre. La méthode retourne un pointeur vers l'ancienne valeur ou NULL si la clé n'était pas présente. On suppose qu'il y a toujours assez de mémoire. Pour l'instant, on ne s'occupe pas du redimensionnement du tableau en cas de dépassement du seuil de facteur de charge. Pour réaliser cette fonction il est conseillé de définir également une fonction interne à la librairie de signature :
   `static Entry* createEntry(const char key[static 1], uint64_t hashcode).`
2. `void* hm_get(HashMap* restrict self, const char key[static 1])`, qui recherche et retourne un pointeur vers la valeur associée à `key`. Si la clé n'est pas présente alors `NULL` est retourné.
3. `void* hm_delete(HashMap* restrict self, const char key[static 1])` qui supprime (et désalloue) l'entrée associé à à `key`. Comme pour la fonction précédente on retourne un pointeur vers la valeur associée à la clé et `NULL` sinon.

Testez vos fonctions, et vérifiez avec `valgrind` que vous n'avez pas de fuite mémoire ou d'accès mémoire illégaux.

1. Pour cela, lancez votre programme `prog` (optionnellement compilé avec une option `-g`) avec la commande `$> valgrind --leak-check=full ./prog`.

Q.3 : Implémentez les fonctions suivantes :

1. `static bool hm_resizeAndRehash(HashMap* self)` qui permet de doubler la taille actuelle du tableau `self->array` et de replacer les entrées dans le nouveau tableau. Cette fonction retourne `true` ssi le redimensionnement à été effectué. Faites en sorte que cette méthode soit appelée avant qu'une insertion provoque un dépassement du facteur de charge.
2. `void* hm_getOrPutIfAbsent(HashMap* restrict self, const char key[static 1], void* (*generate_value)(void))` qui permet retourner la valeur associée à la clé données en paramètre. Si la clé n'est pas présente alors une entrée sera crée et la valeur sera donnée par la fonction `generate_value`. Vérifiez si vous ne pouvez pas factoriser du code avec la fonction `hm_put`.

Testez vos fonctions, et vérifiez avec `valgrind` que vous n'avez pas de fuite mémoire ou d'accès mémoire illégaux.

Q.4 : Implémentez la fonction de parcours `void hm_visit(HashMap* restrict self, void (*visit_entry)(const size_t, const char[], void*))`. Cette fonction permet de "visiter" la table de hachage `self` et prend comme paramètre un fonction qui sera appelée sur chaque entrée de la table avec comme paramètres l'indice dans le tableau `self->array`, la clé et le pointeur vers la valeur.

Testez votre fonction et proposez deux fonctions pour (1) afficher la liste des éléments d'une table de hachage au format { `k=v, ...` } où `k` sera la valeur de la clé et `v` la valeur formaté grâce une fonction qui sera passée en paramètre. ; (2) afficher la structure de la table au format { `[0]={ k=v, ... }, ..., [n]={ ... }` } . Pour cela, vous aurez sans doute besoin de déclarer une fonction dans une fonction [2].

Q.5 : Dans cette question, il est demandé d'associer chaque mot d'un fichier texte à son nombre d'occurrences (i.e. apparition dans le fichier). En vous servant de votre table de hachage et des fonctions (fournies) pour lire et indexer un fichier texte, proposez des fonctions pour :

— Afficher les k mots les plus fréquents en ordre croissants de fréquence (en effectuant si possible qu'un seul parcours de la table de hachage).

— Vérifier si la distribution du nombre d'entrées par position du tableau suit une loi de Poisson de paramètre $\lambda = n/m$ (facteur de charge courant) où n est le nombre d'entrées et m la capacité du tableau.

2. https ://gcc.gnu.org/onlinedocs/gcc/Nested-Functions.html