Polytech Nice Sophia
SI3 / 2021-2022
Erick Gallesio

Programmation Procédurale

Corrigé TD 9

Table de Hachage

1 Une version de la table de taille fixe

1.1 Question 1: structures de données

Dans cette version, la table est allouée à l'initialisation et n'est jamais retaillée, au risque de voir les performances chuter lorsque le taux de remplissage croît.

La première chose à faire est de définir la structure de la table. Pour chaque éléments de la table, nous avons besoin

- d'une clé (un `char *`),
- d'une valeur (ici un `void *`) et
- d'un pointeur sur un éventuel élément ayant la même valeur de *hash*.

Par ailleurs, pour représenter la table elle-même, nous allons utiliser une structure contenant:

- un pointeur sur un tableau d'éléments tels que définis plus haut
- la taille de la table
- le nombre d'élément effectivement présents dans la table. Cette donnée n'est pas strictement nécessaire ici; elle sera par contre indispensable pour la version de la table qui se retaille.

Comme dans la feuille de TD précédente, ces structures de données sont définie dans le fichier C chargé de l'implémentation de la table. Il **ne doivent pas être** définis dans le fichier `hash.h`, afin de ne pas révéler aux utilisateurs l'organisation interne choisie. Ainsi, ceux-ci ne pourront pas introduire de dépendances à notre implémentation dans leur code; nous serons alors donc libres de changer l'organisation de notre code de façon indépendante, sans impacter le reste du code qui utilise notre module de tables de hachage.

Nos structures de données sont donc:

```
typedef struct hash_element *HashElement; // Un élément de la table

struct hash_element {                     // c'est en fait un maillon de liste
  char *key;
  void *value;
  struct hash_element *next;
};

struct hash_table {
  HashElement *table;
  int size;
  int items;
};
```

1.2 Question 2: Création de la table

La fonction de création de la table:

```
HashTable hash_new(int size) {
  HashTable new = my_malloc(sizeof(struct hash_table));

  new->table = my_malloc(size * sizeof(struct hash_element));
  new->size  = size;
  new->items = 0;

  // Initialisation du tableau de pointeurs
  for (int i = 0; i < size; i++) // inutile si on utilisait calloc plus haut
    new->table[i] = NULL;

  return new;
}
```

Ici nous utilisons la fonction interne `my_malloc` qui alloue de la mémoire et arrête le programme si la fonction `malloc` renvoie `NULL`. Cette fonction est bien évidement déclarée `static`, puisqu'elle ne fait pas partie des fonctions publiques du module:

```
static void *my_malloc(size_t size) {
  void *res = malloc(size);

  if (!res) {
    fprintf(stderr, "my_malloc: impossible d'allouer %ld caractères\n", size);
    exit(1);
  }
  return res;
}
```

1.3 Question 3: Fonction de hachage

La fonction de *hash* demandée est simple à écrire (mais ce n'est pas une bonne fonction de *hash*, **celle du cours est bien meilleure**). Là encore, la fonction doit être déclarée comme privée (i.e utilisation du mot-clé `static`).

```
static unsigned int hash(const char *key) {
  unsigned int H = 0;

  for (int i = 0; key[i]; i++) {
    H += (unsigned int) key[i] * (i+1);
  }
  return H;
}
```

1.4 Question 4: Recherches dans la table.

Nous avons deux fonction de recherche dans la table:

- la fonction `hash_find` qui renvoie la valeur rangée dans la table (si elle est présente bien sûr)
- la fonction `hash_find_reference` qui renvoie un pointeur sur valeur rangée dans la table. Cette fonction permet à l'utilisateur de modifier la valeur dans la table (sans avoir à sortir l'élément pour le ré-entrer modifié par la suite).

La fonction `hash_find_reference` renvoie un pointeur sur la valeur qui est dans la table (qui est de type `void *`). Par conséquent, c'est donc une fonction qui renvoie un résultat de type `void **` :

```
void **hash_find_reference(const HashTable ht, const char *key) {
  int pos = hash(key) % ht->size;

  for (HashElement el = ht->table[pos]; el; el = el->next) {
    if (strcmp(el-> key, key) == 0) return &(el->value);
  }
  return NULL;
}
```

Nous pouvons maintenant écrire la fonction `hash_find` en utilisant la fonction précédente (il suffit de renvoyer la valeur qui pointé par le résultat de `hash_find_reference` , en faisiant attention de ne pas utiliser `*` sur la valeur `NULL`):

```
void *hash_find(const HashTable ht, const char *key) {
  void **res = hash_find_reference(ht, key);

  return res? *res: NULL;
}
```

1.5 Question 5: Ajout dans la table.

Pour ajouter un élément dans la table nous devons:

- vérifier qu'il n'est pas déjà présent dans la table (utilisation de la fonction `hash_find` précédente)
- dans le cas contraire
  - calculer son indice dans la table
  - l'ajouter à la liste des éléments dont la valeur de *hash* modulo la taille de la table donne le même indice. Nous insérerons ici en tête de liste, puisque nous avons vu que c'était de loin la façon la plus simple d'ajouter un élément dans une liste chaînée (de toutes les façons cette liste est en principe courte; il n'est donc pas nécessaire d'avoir une stratégie sioux ici).

```
void hash_add(HashTable ht, const char *key, const void *value) {
  if (hash_find(ht, key)) {
    fprintf(stderr, "hash_add: %s déja présent dans la table\n", key);
  } else {
    int pos = hash(key) % ht->size;
    HashElement el  = my_malloc(sizeof(struct hash_element));

    el->key        = strdup(key);
    el->value      = (void *) value;
    el->next       = ht->table[pos];
    ht->table[pos] = el;
    ht->items     += 1;
  }
}
```

1.6 Question 6: Impression de la table

Pas de difficulté ici: on parcourt la table et pour chacune de ses cases, on imprime la liste qui lui est associée).

```
void hash_print(const HashTable ht) {
  printf("Hash table de taille %d (%d occupés)\n", ht->size, ht->items);
  for (int i= 0; i < ht->size; i++) {
    HashElement el = ht->table[i];

    if (el) {
      printf("%6d:", i);
      while (el) {
        printf(" '%s'", el->key);
        el = el->next;
      }
      printf("\n");
    }
  }
}
```

1.7 Question 7: hash_apply

Là encore, pas de difficulté particulière:

```
void hash_apply(const HashTable ht, void (*func)(const char *key,
                                                  const void *value)) {
  for (int i = 0; i < ht->size; i++)
    for (HashElement p = ht->table[i]; p ; p = p->next)
      func(p->key, p->value);
}
```

1.8 Question 8: Libération mémoire.

Ici, il faut libérer tout ce que nous avons alloué (ne pas oublier que `strdup` est une fonction qui alloue et que par conséquent les clés de la table doivent aussi être libérées):

```
void hash_free(HashTable *ht){

  for (int i = 0; i < (*ht)->size; i++) {
    HashElement tmp, p = (*ht)->table[i];

    while (p) {
      tmp = p; p = p->next;
      free(tmp->key);  // Libérer les clés (que nous avons allouées par strdup)
                       // Ne pas libérer les valeurs (que nous n'avons pas allouées)
      free(tmp);
    }
  }
  free((*ht)->table);
}
```

Code complet de la table de hachage de taille fixe. 💾

2 Une version de la table de taille dynamique

L'idée ici est d'avoir une table dont la taille est déterminée à la création, mais qui peut croître si on s'aperçoit qu'elle commence à être trop remplie.

Une seule fonction doit être modifiée ici: `hash_add`. En effet, quand on veut ajouter un élément dans la table, on regarde au préalable si celle-ci commence à être trop remplie. Si tel est le cas, la table est retaillée et l'élément est entré dans la nouvelle (grande) table.

Le code de la fonction `hash_add` devient:

```
void hash_add(HashTable ht, const char *key, const void *value) {
  // Voir si on doit agrandir la table.
  float ratio = ((float) ht->items) / ht->size;
  if (ratio > 0.75) resize_table(ht);

  if (hash_find(ht, key)) {
    fprintf(stderr, "hash_add: %s déja présent dans la table\n", key);
  } else {
    int pos = hash(key) % ht->size;
    HashElement el  = my_malloc(sizeof(struct hash_element));

    el->key        = strdup(key);
    el->value      = (void *) value;
    el->next       = ht->table[pos];
    ht->table[pos] = el;
    ht->items     += 1;
  }
}
```

La fonction interne `hash_new` est donnée ci-dessous:

```c
static void resize_table(HashTable ht) {     // Cette fonction est bien-sûr static
// Pour retailler la table on a deux solutions:
//    - soit créer une nouvelle table de la bonne taille et appeler
//      hash_add sur tous ses éléments. C'est le plus facile, mais cela
//      implique que l'on va réallouer des HashElements pour chaque clé
//      (alors que c'est déjà fait)
//    - soit juste allouer un nouveau tableau et transférer ensuite tous les
//      élément de l'ancienne table dans ce dernier. C'est ce qu'on fait ici

  int new_size           = ht->size * 3 / 2;
  HashElement *new_table = calloc(new_size, sizeof(HashElement *));

  if (!new_table) {
    // Ici on a pas pu allouer une nouvelle table. Pas grave, on continue avec
    // celle que l'on a, au risque de ne pas être efficace.
    return;
  }
  // printf("*** Before Resize\n"); hash_print(ht);

  // On transfère tous les éléments dans la nouvelle table
  for (int i= 0; i < ht->size; i++) {
    HashElement current,next;
    for (current=ht->table[i]; current; current = next) {
      next = current->next;

      // Calculer la nouvelle place de l'élément courant dans la nouvelle table
      int new_idx = hash(current->key) % new_size;

      // Placer l'élément courant dans la table
      current->next = new_table[new_idx];
      new_table[new_idx] = current;
    }
  }

  // Libérer l'ancienne table qui ne sert plus
  free(ht->table);

  // Mettre à jour la structure de la table avec les nouvelle valeurs
  ht->table = new_table;
  ht->size  = new_size;

  // printf("*** After Resize\n");  hash_print(ht);
}
```

Code complet de la table de hachage de taille dynamque

Notes:

- Vous pouvez décommenter la fin du fichier `main.c` distribué pour entrer un grand nombre de mots dans la table et étudier son comportement
- Cette nouvelle implémentation de la table utilise la fonction de *hash* vue en cours qui est bien plus efficace.