Les tables de hachage en C

par Nicolas Joseph

Date de publication : 04/05/2005

Dernière mise à jour : 04/05/2005

Les structures de données en C. Cinquième partie : les tables de hachage.

I - Introduction

Après avoir vue les listes et avant d'attaquer les arbres, je vai vous présenter les tables de hachage en guise d'interlude. Qu'est ce qu'une table de hachage? C'est tout simplement un tableau; à une différence près : les indices ne sont pas des entier mais des chaînes de caractères (on parle désormais de clés). Elles sont présentes de façon natives dans certains langages tel que Perl.

II - Principe

Les seules fonctions dont nous avons besoin sont les mêmes que pour un tableau classique :

- Création
- Ajout et modification d'un élément
- Retrouver un élément
- Destruction

III - La pratique

III-A - Structure d'une table de hachage

La table est composée de cellule qui doivent contenir la chaîne de caractère servant de clés pour retrouver une donnée elle aussi stockée dans la strcuture. Voici donc la structure qui va servir d'élément de base à notre table :

```
typedef struct
{
   char *indice;
   void *data;
} cell_s;
```

Ensuite, nous avons besoin d'un endroit où ranger ces cellules. Dans un premier temps, j'ai pensé utiliser une liste chaînée pour bénéficer de la simplicité d'ajout d'un élément mais cela impliquait d'ajouter le code vue lors du second tutoriel. Cela me semblai inutile surtout qu'il n'est pas nécessaire de supprimer un élément, j'ai donc décidé d'utiliser un tableau. Comme le tableau va être manipulé à l'aide d'un pointeur (pour pouvoir modifier sa taille de façon dynamique), il faut garder en mémoire sa taille, voici donc la structure retenue pour stocker les cellules de notre table :

```
typedef struct hach
{
   cell_s *p_cell;
   int size;
} hach_s;
```

III-B - Initialisation

Lors de l'initialisation, il nous faut allouer une structure qui servira de table et initialiser ses membres :

```
hach_s *hach_new (void)
{
   hach_s *p_table = NULL;

   p_table = malloc (sizeof (*p_table));
   if (p_table != NULL)
   {
      p_table->p_cell = NULL;
      p_table->size = 0;
   }
   else
   {
      fprintf (stderr, "Memoire insufisante\n");
      exit (EXIT_FAILURE);
   }
   return (p_table);
}
```

III-B - Ajout et modification d'une clés

L'ajout et la suppression d'une clés va se faire par l'intermédiaire d'une même fonction. Pourquoi? Tout simplement pour coller le plus possible au modèle du tableau, qui n'utilise qu'un seul opérateur pour la création (première affectation) et la modication d'un élément (à savoir l'opérateur =). Comme la fonction est assez conséquente, je vai vous la présenter en différentes parties.

Lorsque l'on passe la valeur NULL comme premier paramètre à la fonction realloc, elle se comporte comme malloc. J'utilise cette propriété par deux fois dans cette fonction.

```
void hach_add (hach_s * p_table, const char *key, void *data)
{
   if (p_table != NULL && key != NULL)
   {
      int i = 0;
      cell_s *p_cell = NULL;
      char *tmp_key = NULL;
...
```

Le premier paramètre de la fonction est la table de hachage et ensuite, il s'agit de la clés à créer ou modifier ainsi que la nouvelle valeur à lui assigner. Avant toute chose, on écarte les possibilités de déréférencement d'un pointeur NULL. La variable *i* va servir de compteur pour une boucle, *p_cell* va être la cellule crée ou modifiée et *tmp* est une variable temporaire pour la réallocation.

```
...
while (i < p_table->size)
{
   if (strcmp (p_table->p_cell[i].key, key) == 0)
   {
      p_cell = &p_table->p_cell[i];
      break;
   }
   i++;
}
...
```

Ensuite, on détermine si la clés existe déjà ou pas. Si elle existe, on fait pointer *p_cell* dessus.

```
...
if (p_cell == NULL)
{
...
```

Si la clés n'existe pas...

```
...
p_table->size++;
tmp = realloc (p_table->p_cell, sizeof (*p_table->p_cell) * p_table->size);
if (tmp != NULL)
{
   p_table->p_cell = tmp;
   p_cell = &p_table->p_cell[p_table->size - 1];
   p_cell->key = NULL;
}
else
{
   fprintf (stderr, "Memoire insufisante\n");
   exit (EXIT_FAILURE);
}
...
```

...on la crée, au passage on augmente d'un la taille du tableau, et on fait pointer *p_cell* dessus, ce qui signifie qu'à partir de maintenant *p_cell* pointe sur une cellule valide du tableau, et par conséquent le reste de la fonction est commun à la création et à la modification.

```
...
tmp = realloc (p_cell->key, sizeof (*p_cell->key) * (strlen (key) + 1));
if (tmp != NULL)
{
   p_cell->key = tmp;
   strcpy (p_cell->key, key);
   p_cell->data = data;
}
else
{
   fprintf (stderr, "Memoire insufisante\n");
   exit (EXIT_FAILURE);
```

```
   }
   ...
```

On (re)dimentionne la taille de la clés en fonction du paramètre de la fonction et on le copie et enfin, on copie le pointeur sur la donnée utilisateur.

III-D - Recherche d'une clés

Cette fonction est simple comparée à la précédente, elle se contente de parcourir le tableau de l'indice 0 à *p_table->size* et de s'arréter si le paramètre *key* est identique au champs *key* de l'une des cellule et retourne le pointeur sur le champs *data* de cette même cellule. Par contre, si aucune correspondance n'est trouvée, la fonction retourne NULL :

```
void *hach_search (hach_s * p_table, const char *key)
{
   void *data = NULL;

   if (p_table != NULL && key != NULL)
   {
      int i = 0;

      while (i < p_table->size)
      {
         if (strcmp (p_table->p_cell[i].key, key) == 0)
         {
            data = p_table->p_cell[i].data;
            break;
         }
         i++;
      }
   }
   return (data);
}
```

Dans un premier temps, j'ai voulue utiliser cette fonction pour tester si une clés existait ou non (dans la fonction hach_add), en supposant que si la clés n'existait pas la fonction retournait NULL. Cependant comme la fonction hach_search se contente de retourner un pointeur sur une donnée utilisateur, rien n'empêche l'utilisateur de mettre ce pointeur à NULL. Dans ce cas, la fonction hach_search retourne NULL alors que la clés existe.

III-E - Destruction

Pour finir, voici la fonction qui permet de libérer la mémoire allouée dans le cadre de notre table de hachage :

```
void hach_delete (hach_s ** p_table)
{
   int i;

   for (i = 0; i < (*p_table)->size; i++)
      free ((*p_table)->p_cell[i].key);
   free ((*p_table)->p_cell);
   free (*p_table);
   *p_table = NULL;
   return;
}
```

IV - Conclusion

Les tables de hachage sont interessantes pour réaliser des annuaires (faire correspondre un nom avec des données). Comme pour les autres structures de données déjà abordées, la glib propose une implémentation bien plus complète que celle vue ici. Cependant, si vous souhaiter faire une utilisation intensive de ce type de table, je vous conseille de vous tourner vers un langage qui les implémente nativement, tel que Perl.

VI - Remerciements

VII - Code source complet

Téléchargez l'archive zippée.

tablehachage.h

```
#ifndef H_TABLEHACHAGE
#define H_TABLEHACHAGE

typedef struct hach hach_s;

hach_s *hach_new (void);
void hach_add (hach_s *, const char *, void *);
void *hach_search (hach_s *, const char *);
void hach_delete (hach_s **);

#endif /* not H_TABLEHACHAGE */
```

tablehachage.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "tablehachage.h"

typedef struct
{
   char *key;
   void *data;
} cell_s;

struct hach
{
   cell_s *p_cell;
   int size;
};

hach_s *hach_new (void)
{
   hach_s *p_table = NULL;

   p_table = malloc (sizeof (*p_table));
   if (p_table != NULL)
   {
      p_table->p_cell = NULL;
      p_table->size = 0;
   }
   else
   {
      fprintf (stderr, "Memoire insufisante\n");
      exit (EXIT_FAILURE);
   }
   return (p_table);
}

void hach_add (hach_s * p_table, const char *key, void *data)
{
   if (p_table != NULL && key != NULL)
   {
      int i = 0;
      cell_s *p_cell = NULL;
      void *tmp;

      while (i < p_table->size)
      {
         if (strcmp (p_table->p_cell[i].key, key) == 0)
         {
            p_cell = &p_table->p_cell[i];
            break;
         }
         i++;
      }
      if (p_cell == NULL)
      {
         p_table->size++;
         tmp =
```

tablehachage.c

```
                realloc (p_table->p_cell,
                         sizeof (*p_table->p_cell) * p_table->size);
            if (tmp != NULL)
            {
               p_table->p_cell = tmp;
               p_cell = &p_table->p_cell[p_table->size - 1];
               p_cell->key = NULL;
            }
            else
            {
               fprintf (stderr, "Memoire insufisante\n");
               exit (EXIT_FAILURE);
            }
         }
         tmp =
            realloc (p_cell->key,
                     sizeof (*p_cell->key) * (strlen (key) + 1));
         if (tmp != NULL)
         {
            p_cell->key = tmp;
            strcpy (p_cell->key, key);
            p_cell->data = data;
         }
         else
         {
            fprintf (stderr, "Memoire insufisante\n");
            exit (EXIT_FAILURE);
         }
      }
      else
      {
         fprintf (stderr, "Memoire insufisante\n");
         exit (EXIT_FAILURE);
      }
      return;
   }

   void *hach_search (hach_s * p_table, const char *key)
   {
      void *data = NULL;

      if (p_table != NULL && key != NULL)
      {
         int i = 0;

         while (i < p_table->size)
         {
            if (strcmp (p_table->p_cell[i].key, key) == 0)
            {
               data = p_table->p_cell[i].data;
               break;
            }
            i++;
         }
      }
      return (data);
   }

   void hach_delete (hach_s ** p_table)
   {
      int i;

      for (i = 0; i < (*p_table)->size; i++)
         free ((*p_table)->p_cell[i].key);
      free ((*p_table)->p_cell);
      free (*p_table);
      *p_table = NULL;
      return;
   }
```