Université de Corse - Pasquale PAOLI	
Diplôme : Licence SPI 3ème année, parcours Informatique	**2023-2024**

UE : Paradigme de programmation

Enseignants : Marie-Laure Nivet & Sakhi Shokouh & Diego Grante

TD/TP table de hashage

Pour tous les exercices **il vous est demandé** :

- d'organiser votre code en librairies (Par exemple hashtable.h hashtable.c maintest.c).
- d'utiliser dans les codes le type bool via un #include <stdbool.h>
- d'assurer la lisibilité de vos codes et pour cela de les commenter, les indenter, de choisir avec attention les noms de vos variables et de vos fonctions.
- d'être TRES attentif à la gestion de la mémoire.
- de soigner vos tests.
- de poster vos codes sur un dépôt de code dont vous nous donnerez l'adresse.

Par ailleurs vous **DEVEZ RESPECTER VOS MANIFESTES.**

N'oubliez pas que Copilot et ChatGPT ne sont pas inscrits en L3 informatique, en revanche vous, oui ! C'est donc vous qui devez coder !!!!

Objectifs de ce TD/TP

L'objectif principal de ce TP est d'implémenter le type abstrait tableau associatif dont nous avons parlé en cours et d'utiliser pour cela une table de hachage. Ce TP vous permettra d'aguerrir vos compétences et vos connaissances sur la gestion et l'implémentation des tables de hachage. La table de hachage que nous allons implémenter permettra de stocker des couples clés-valeurs de type chaines de caractères. Si vous souhaitez implémenter des tables de hachages permettant de stocker d'autres types de données, les changements à effectuer seront mineurs.
Il existe plusieurs façons d'implémenter les tables de hachage, avec un tableau de taille fixe, de taille variable, une liste (peu efficace) ou un arbre binaire équilibré. Nous avons ici choisi l'implémentation la plus simple, à savoir celle basée sur un tableau de taille fixe, de paires clés valeurs. Nous vous encourageons à aller voir – et pourquoi pas, à programmer – les autres possibilités d'implémentation.

Partie 1 : Interface hashtable.h

Nous allons dans un premier temps nous occuper de l'interface *hashtable.h*, c'est-à-dire des déclarations de structures et d'en-têtes des fonctions nécessaires à l'implémentation du type table de hachage. Vous ne devez pas écrire le code dans cette partie.

Question 1 – Proposez une structure *ht_item* permettant de stocker la paire clé, valeur d'un élément de la table de hachage. La clé (*key*) et la valeur (*value*) seront toutes les deux de type chaine de caractères.

Question 2 – Proposez une structure *ht_hash_table* permettant de stocker :

- Sa taille (*size*), c'est-à-dire le nombre de cases du tableau support de la table de hachage,
- Son nombre d'éléments effectivement stockés (*amount*)
- Un tableau de pointeurs sur des éléments *hsht_key_value_pair* (*kv_items*). Rappel, une variable tableau est en fait un pointeur sur le premier élément du tableau.

Question 3 – Proposez une **déclaration** d'une fonction d'initialisation d'un couple clé, valeur c'est-à-dire d'un élément *hsht_key_value_pair*. Cette fonction, nommée

hsht_new_key_value_pair devra allouer la mémoire nécessaire – vous en écrirez le code en partie 2 – à un nouvel élément et retourner le pointeur sur ce nouvel élément. Les valeurs de la clé et de la valeur seront passées en paramètres de la fonction. On veillera à ne pas modifier ces valeurs (qu'on peut donc les considérer comme constantes) et on en stockera une copie dans le nouvel item *hsht_key_value_pair*.

Question 4 – Proposez une **déclaration** de fonction de suppression d'un item *hsht_key_value_pair* dont le pointeur est passé en paramètre i.e. une fonction qui libère l'espace mémoire utilisé par la clé, par la valeur et par l'item en lui-même. On nommera cette fonction *hsht_del_key_value_pair*. Elle ne retourne rien.

Question 5 – Proposez une **déclaration** de fonction d'initialisation d'une nouvelle table de hachage que l'on nommera *hsht_new* i.e. une fonction qui alloue l'espace mémoire nécessaire pour potentiellement stocker *size* item *hsht_key_value_pair* et retourne un pointeur sur cette nouvelle hashtable.

Question 6 – Proposez une **déclaration** de fonction de suppression d'une *hsht_del* i.e. une fonction qui libère l'espace mémoire utilisé.

Partie 2 : Première implémentation hashtable.c, hsht_main.c et makefile

Nous allons maintenant donner du code à l'interface décrite en partie 1, proposer un programme de test de notre solution et gérer le tout via un makefile.

Question 7 – Donnez une implémentation dans un fichier *hashtable.c* permettant de correspondre à l'interface déclarée dans le fichier *hashtable.h* proposé dans la partie précédente. Pour cela, on précise les points suivants :

a) Pour la fonction *hsht_new_key_value_pair* : Pour effectuer la copie des valeurs de clé et de valeur dans le nouvel item *hsht_key_value_pair* vous pourrez vous servir de la fonction *strdup*[1] qui permet de dupliquer une chaîne de caractères en lui allouant une nouvelle zone mémoire (ne pas utiliser *strcpy*, ou *strncpy* car la zone mémoire de la copie doit dans notre cas, être allouée).
b) Pour la fonction *hsht_new_hashtable*, il faudra penser à allouer l'espace mémoire pour la hashtable en elle-même, mais aussi pour le tableau de pointeurs sur les items. Les cases du tableau de pointeurs sur *key-value-pair* devront toutes être à NULL ce qui stipule que l'emplacement est vide/libre. Dans le cas où la valeur de *size*, passée en paramètre serait de 0, on créera par défaut une table de 53[2] cases. Regardez les différentes fonctions d'allocation mémoire à votre disposition afin de choisir la/les bonnes.
c) Pour la fonction de suppression de la table de hachage, il faut libérer tout l'espace mémoire, c'est-à-dire, tous les items *hsht_key_value_pair* stockés dans le tableau, mais aussi le tableau lui-même, puis enfin la structure même de la table.

Question 8 – Proposez, dans un fichier hsht_main.c un main, dans lequel vous testerez au fur et à mesure l'ensemble des codes produits. A ce stade, vous pouvez créer une nouvelle table de hachage, un nouvel item, en vérifiez le contenu champs par champs, puis détruire tous les éléments créés.

Question 9 – Proposez un make file permettant de :

[1] p. 103 « The GNU C Library Reference Manual », Sandra Loosemore with, Richard M. Stallman, Roland McGrath, Andrew Oram, and Ulrich Drepper, Version 2.38, 1239 pages, disponible en PDF, en ligne à l'adresse https://www.gnu.org/software/libc/manual/pdf/libc.pdf, et consultable en version HTML à l'adresse https://sourceware.org/glibc/manual/latest/html_mono/libc.html.
[2] Il est recommandé de toujours dimensionner les tables de hachages avec un nombre premier.

a) compiler séparément les différents fichiers,
b) générer l'exécutable correspondant à votre main hsht_main.c
c) nettoyer les fichiers inutiles entre les différentes opérations de compilation.

Partie 3 : Fonction de hachage et gestion des collisions

Pour mémoire, une fonction de hachage est toujours constituée de deux parties. Une partie code de hachage et une partie fonction de compression destinée à permettre le rangement dans la table de la clé ayant le code de hachage calculé (Cf. Transparent 186 du cours).

Question 10 – Supposons une fonction de hachage $h_1(S) = \sum_{i=0}^{n-1} ascii(c_i)$ avec S la chaîne de n caractères c_i et $ascii(c)$ le code ASCII d'un caractère c. Expliquez pourquoi cette fonction de hachage n'est pas valide. Proposez une solution mathématique qui permette de la rendre valide, puis expliquez pourquoi cette fonction n'est pas optimale.

Question 11 – Supposons une fonction de hachage $h_2(S) = \sum_{i=0}^{n-1} \alpha^i * ascii(c_i)\ mod\ M$ avec S la chaîne de n caractères c_i, $ascii(c)$ le code ASCII d'un caractère c, M la taille (*size*) de la table de hachage et α un nombre premier supérieur à la taille de l'alphabet utilisé. Écrire une fonction *hsht_hash* implémentant la fonction h_2. Vous veillerez à la déclarer dans l'interface hashtable.h et à l'implémenter dans le fichier hashtable.c.

Question 12 – Calculer à la main et remplir le tableau en utilisant h_2 puis vérifier l'implémentation de votre fonction :

S	α	M	Index
"abc"	397	10	
"abcd"	397	10	
"paradigme"	397	10	
"de"	227	53	
"est"	227	53	
"salut"	227	53	

Question 13 – Une fonction de hachage peut avoir un nombre infini d'entrées mais a forcément un nombre fini de sortie. Ainsi $\exists s_1, s_2 \mid s_1 \neq s_2\ et\ h_2(s_1) = h_2(s_2)$ avec s1 et s2 des chaînes de caractères. C'est ainsi que les collisions se produisent. Listez et décrivez les méthodes possibles pour empêcher les collisions.

Partie 4 : Méthodes pour gérer la table de hachage

Dans cette partie nous choisissons d'utiliser la technique de l'adressage ouvert c'est-à-dire qu'en cas de collision les couples clé-valeur ayant le même indice, donc en collision, seront stockés dans une autre alvéole (case) libre du tableau de pointeurs sur *hsht_key_value_pair*. L'alvéole sera recherchée par une méthode dîtes de sondage par double hachage. Nous proposons ci-dessous une implémentation de la fonction de hachage que vous pouvez intégrer à votre code.

```
int hsht_give_hash(const char* S, const int M, const int attempt,
                   const int alpha_1, const int alpha_2) {
   const int hash_a = hsht_hash(s, alpha_1, M);
   const int hash_b = hsht_hash(s, alpha_2, M);
   return (hash_a + (attempt * (hash_b + 1))) % M;
}
```

Cette fonction renvoie un hash de la chaîne de caractère *S* en fonction de M le nombre maximum d'éléments dans la *hsht_hash_table,* de alpha_1 et alpha_2 (avec alpha_1 différent de alpha_2) deux nombres premiers supérieurs ou égaux à 397 et un nombre de tentative *attempt* égale au nombre de fois où l'on a eu une collision. Donc lors du premier appel pour calculer l'indice où ranger la chaine S, on aura attempt=0. Si l'indice retourné est celui d'une alvéole déjà occupée de la table de hachage, on calculera un nouvel indice avec attempt = 1 et ainsi de suite tant qu'on ne tombe pas sur une alvéole libre. C'est ce comportement que nous vous demandons d'implémenter dans la suite.

Question 14 – Proposez une implémentation de la procédure *hsht_insert* qui permet d'insérer dans une *hsht_hash_table* donnée une nouvelle paire constituée d'une clé *key* et d'une valeur *value,* donnés tous deux en paramètre.

Question 16 – Proposez une implémentation de la fonction *hsht_search* qui renvoie la valeur *value* de type chaine de caractère associée à la clé *key* passée en paramètre de la fonction, dans une *hsht_hash_table donnée*. S'il n'y a aucune valeur associée à la clé recherchée, *hsht_search* renvoie NULL.

Question 15 – Proposez une implémentation de la procédure *hsht_delete* qui permet de supprimer le couple clé-valeur, associé à la clé *key* donnée en paramètre dans la *hsht_hash_table donnée*.
Attention, si le couple clé valeur que nous souhaitons supprimer fait partie d'une chaîne de collision, l'ôter de la table de hachage peut poser un problème en « cassant » la chaine de collision et ainsi rendre impossible la recherche d'éléments liés par cette même chaîne (c'est-à-dire ayant le même hash que les autres éléments liés). Pour éviter cela, plus que de simplement supprimer la paire clé-valeur en libérant la mémoire, on marque la case comme ayant été supprimée en pointant un élément *hsht_key_value_pair* spécifique nommé HSHT_DELETED_PAIR constitué de deux NULL.
On aura donc la déclaration :
hsht_key_value_pair HSHT_DELETED_PAIR = {NULL, NULL};

Question 16 – Modifiez si nécessaire vos implémentations des fonctions *hsht_insert* et *hsht_search* pour prendre en compte la représentation des paires supprimées introduite dans la question précédente.

Question 17 – Proposez une implémentation de la procédure *hsht_uptade* qui met à jour la valeur *value* associée à la clé *key,* toutes deux passées en paramètre dans une *hsht_hash_table* également donnée.

Partie 5 : Optimisation - Redimensionnement

Dans cette partie, nous allons traiter les problèmes liés à l'optimisation c'est-à-dire au maintien de la plus grande performance d'insertion et de recherche des paires clé-valeur dans la table de hachage. Cette efficacité peut s'exprimer à l'aide de la complexité qui – si on considère que le calcul de l'indice de la clé après hachage compte pour une opération – est, dans le meilleur des cas, c'est-à-dire sans collision, en $O(1)$. La complexité tend vers $O(n)$ lorsque le facteur de charge de notre table de hachage est proche de 1.
Ce facteur de charge se définit par :

$$fdc = \frac{nb\ clés}{nb\ places\ total}$$

Il donne un aperçu du remplissage de notre table de hachage.
Pour diminuer le facteur de charge deux solutions s'offrent à nous :

1) Diminuer le nombre de clés => revient à supprimer des éléments.

2) Augmenter le nombre de places => revient à augmenter la taille maximale de la table de hachage.

Nous allons nous concentrer sur la deuxième solution. Lorsque le facteur de charge atteint 0.7 les collisions deviennent trop fréquentes. Nous allons considérer que lorsque le facteur de charge dépasse ce seuil nous augmentons la taille de la table de hachage. Notons que de la même manière, si le facteur de charge est inférieur à 0.1, la table de hachage peut/doit être diminuée, pour, cette fois optimiser l'espace mémoire.

Question 18 – Proposez une fonction *hsht_extend* qui prend en entrée une table de hachage et qui retourne une table de hachage redimensionnée avec une taille 2 fois supérieure à l'ancienne. N'oubliez pas de libérer l'espace mémoire de l'ancienne table de hachage proprement.

Question 19 – Implémentez dans votre code, partout où il parait nécessaire, l'utilisation de la fonction *hsht_extend* en fonction du facteur de charge.

Question 20 – Dans la littérature, il est montré que la taille optimale d'une table de hachage est un nombre premier. En prenant cette information en considération améliorez *hsht_extend* pour que la taille de la table redimensionnée, en fonction du facteur de charge, soit 2 fois supérieure, ou 2 fois inférieure à l'ancienne table tout en étant un nombre premier. N'hésitez pas à spécialiser vos fonctions selon que vous augmentiez ou diminuiez la taille de la table et à définir des fonctions outils vous permettant de trouver les nombres premiers immédiatement supérieur à un entier donné.