POLYTECH MARSEILLE Aix*Marseille Université **HUGO3**

Algorithmique

TD n°9 : les tables de hachage

Exercice 1 – Comparaison de différents types de tables

On considère l'ensemble de 13 mots dont les valeurs hachées par une fonction de hachage f sont décrites ci-dessous :

	mot	valeur hachée (en hexadécimal)
01	« le »	FF2E
02	« cours »	178DD38
03	« de »	75EA33
04	« types »	35CE5
05	« et »	9AA8BF1
06	« structures »	2738
07	« est »	A4C74
08	« absolument »	1CA4C74
09	« génial »	14D26
10	« j'adore »	5A38
11	« faire »	1BAE5
12	« ses »	65B4EE5
13	« TD/TME »	8C74

Pour chaque type de table de hachage ci-dessous, indiquez ce que l'on obtiendrait si l'on partait d'une table vide de taille 16 et que l'on insérait dans l'ordre tous les mots ci-dessus avec une fonction de hachage $f(k) = k \bmod 16$.

1. Table de hachage avec résolution des collision par chaînage,
2. Table de hachage avec adressage ouvert et probing linéaire,
3. Table de hachage avec adressage ouvert et probing quadratique $h(k) = f(k) + \frac{i}{2} + \frac{i^2}{2}$.

Pour les adressages ouverts, calculez le nombre de probes à effectuer en moyenne pour obtenir un élément donné dans la table. Déduisez-en le meilleur des deux probings.

Exercice 2 – fonction de hachage

On considère l'ensemble de couples (clé,valeur) suivant :

	mot	clé (en décimal)
01	« le »	123
02	« cours »	22
03	« de »	88
04	« types »	33
05	« et »	4
06	« structures »	28
07	« est »	73
08	« absolument »	7
09	« génial »	15

Q 2.1 Calculez les valeurs hachées des clés obtenues pour chaque couple par les fonctions ci-dessous :

- `int f1 (int x) { return x; }`
- `int f2 (int x) { return 10 * x; }`
- `int f3 (int x) { return 2 * x; }`
- `int f4 (int x) { return x == 0 ? 0 : 8 * f4 (x/10) + x % 10;}`

Q 2.2 Supposons que l'on veuille insérer les mots ci-dessus dans une table de hachage de longueur 10. On veut hacher les clés des mots en utilisant la fonction $f(x) = g(x) \bmod 10$, où $g \in \{\texttt{f1},\texttt{f2},\texttt{f3},\texttt{f4}\}$. Quelle est le meilleur choix pour g ? Expliquez pourquoi.

Exercice 3 – Implantation

Dans cet exercice, on veut implanter une table de hachage avec résolution des collisions par chaînage (listes simplement chaînées de couples (clé,valeur)) et fonction de hachage multiplicative. Pour la suite de l'exercice, on supposera que le tableau contenant les listes chaînées d'éléments a une taille correspondant à une puissance de 2 (ceci afin d'obtenir un algorithme rapide, comme nous l'avons vu en cours). On définit les types suivants :

```c
// on définit le type hash_func_t comme une fonction de int * int -> int
// la fonction prend en paramètres 2 arguments : k et m
typedef int (hash_func_t) (int,int);

// les couples (clé,valeur) stockées dans la table de hash
typedef struct {
  int   cle;
  float valeur;
} bucket;

// on crée une structure pour des listes simplement chaînées de buckets
#include "list_bucket.h"

typedef struct {
  list_bucket_t* tab; // tableau des listes chaînées (clé,valeur)
  int m;              // log_size de la taille de la table qu'on a allouée
  hash_func_t*  hash; // fonction de hachage
} hash_table_t;
```

On veut implanter une fonction de hachage $h(k)$ du type :

$$h(k) = \lfloor M \times ((k \times A) \bmod 1) \rfloor,$$

où M représente la taille de la table de hachage (une puissance de 2, $M = 2^m$), $A \approx 0,618$ est le nombre d'or, et $(k \times A)$ mod 1 signifie la partie décimale de $(k \times A)$. Soit B un entier tel que $A \approx B \times 2^{-n}$, pour un entier $n > 0$. Alors $k \times A \approx k \times B \times 2^{-n}$. Il est facile de montrer que les n derniers bits de cette expression correspondent à la partie décimale de $k \times A$. Si ET représente un « et logique », $((k \times B)\ ET\ (2^n - 1)) \times 2^{-n}$ permet donc d'obtenir la partie décimale de $k \times A$. Puisque $h(k)$ s'obtient en multipliant cette partie décimale par $M = 2^m$,

$$h(k) = \lfloor ((k \times B) ET (2^n - 1)) \times 2^{m-n} \rfloor.$$

En C, le ET logique s'écrit `&`[1]. De plus, le calcul d'une puissance de 2 peut être effectué par des décalages de bits grâce aux fonctions `<<` (décalage à gauche) et `>>` (décalage à droite). Ainsi, 2^k est égal à `1 << k` pour k positif.

Dans la suite, on supposera que $n = 15$ (donc $2^n - 1 = 32767$) et que $m < n$. Pour $n = 15$, $B = 20252$. De plus, la multiplication par 2^{m-n} s'obtient en effectuant $n - m$ décalages vers la droite (`>>`).

Q 3.1 Écrivez une fonction `h`, qui prend deux entiers k et m et qui renvoie la valeur de $h(k)$ telle que décrite ci-dessus. Vous vérifierez que $m < n$ et que $m > 2$. Dans le cas contraire, vous lèverez une exception. *Indice* : si vous avez obtenu la bonne fonction $h(\cdot)$, pour $m = 10$, vous devriez avoir :

$h(1) = 632 \quad h(2) = 241 \quad h(10) = 184 \quad h(100) = 823 \quad h(531) = 184 \quad h(1573) = 184 \quad h(1052) = 184.$

Q 3.2 Écrivez une fonction `log_size` telle que `log_size(size)` renvoie le nombre de bits nécessaires pour stocker l'entier `size`. Ainsi `log_size(1024)` ou `log_size(1000)` renverraient 10.

Q 3.3 Écrivez une fonction `create_hash_table` qui prend en argument une taille `size` et qui renvoie une table de hachage vide de taille M, la plus petite puissance de 2 supérieure ou égale à `size`.

Q 3.4 Écrivez une fonction `find_elt` telle que `find_elt (table,k)` renvoie la valeur de l'élément de clé `k` si celui-ci existe et lève une exception sinon. Vous supposerez qu'il y a au plus un élément par clé.

Q 3.5 Écrivez une fonction `exists_elt` telle que `exists_elt(table,k)` renvoie un booléen indiquant si la table de hachage contient la clé `k`.

Q 3.6 Écrivez une fonction `insert_elt` telle que `insert_elt(table,k,v)` rajoute l'élément de clé `k` et de valeur `v` à la table de hachage passée en premier argument si la clé `k` n'existe pas déjà dans la table, et lève une exception sinon.

Q 3.7 Écrivez une fonction `display_table` prenant en argument une table de hachage et affichant le contenu de celle-ci. Par exemple, une table contenant les couples clé-valeur : (1052,5.5), (1573,2.5), (531,3.5) afficherait la chaîne de caractères`[ (1052,2.500000) (1573,2.500000) (531,2.500000) ]`. Attention : les couples peuvent apparaître dans n'importe quel ordre.

Q 3.8 Écrivez une fonction `delete_elt` telle que `delete_elt(table,k)` supprime l'élément de clé `k` si celui-ci existe et lève une exception sinon.

1. Attention : ne confondez pas « `&&` » qui est un ET logique entre 2 booléens et « `&` » qui réalise un ET entre des suites de bits.

Exercice 4 – dictionnaire

Dans cet exercice, on se propose de comparer la gestion d'un dictionnaire par liste chaînée et par table de hachage. Le but est de voir à quel point l'utilisation d'une table de hachage accélère les recherches dans le dictionnaire. Ceci justifie l'emploi des tables dans des logiciels tels que les compilateurs.

Q 4.1 Écrivez une fonction `create_dico_list`, prenant en paramètre un entier `m` et qui renvoie une liste d'associations contenant des couples (nombre sous forme d'entier, nombre sous forme de chaîne de caractères (d'au plus 10 caractères)), pour tous les nombres de 0 à `m`. L'ordre de ces couples dans la liste n'a aucune importance. Précisons que vous pouvez exploiter la fonction `sprinf`, qui s'utilise comme un `printf` à ceci près qu'elle a un argument en plus (son premier argument) qui est une chaîne de caractères s et que l'affichage, au lieu d'avoir lieu à l'écran, est réalisé dans la chaîne s.

Q 4.2 Écrivez de même une fonction `create_dico_hash`, prenant en paramètre un entier `m` et qui renvoie une table de hachage de taille `m` (cf. l'exercice précédent) et la remplit avec des chaînes de caractères correspondant à tous les nombres de 0 à `m`. Les clés de ces chaînes sont les nombres eux-mêmes.

Q 4.3 Écrivez une fonction `get_element_list` qui, étant donné une liste créée par la fonction `create_dico_list` et un nombre `nb`, renvoie la chaîne de caractères correspondant au nombre `nb` si celle-ci appartient à la liste, et lève une exception sinon.

De même, écrivez une fonction `get_element_hash` réalisant la même opération avec une table de hachage.

Q 4.4 On veut maintenant comparer la vitesse de recherche d'informations dans une liste et dans une table de hachage. Pour cela, écrivez une fonction `time_search_list` telle que `time_search_list(m)` crée une liste de couples de taille `m` en utilisant la fonction `create_dico_list` et recherche les chaînes correspondant à `m` nombres tirés au hasard entre 0 et `m` (utilisez la fonction `rand()` à cet effet). Faites un programme qui lance cette fonction. Exécutez sous Linux ce dernier avec la commande `time mon_programme`. La commande `time` vous indiquera, après l'exécution, le nombre de secondes d'exécution du programme.

Q 4.5 Faites de même avec la fonction `time_search_hash` pour les tables de hachage.

Q 4.6 Comparez les temps obtenus et exclamez-vous « y a pas photo ! ».