TP n°24 - Fonctions de hachage - Table de hachage

Dans toute la suite, on pourra utiliser la librairie C `stdint.h` (au programme de MPII) qui contient la définition des types suivants :

`int8_t :` entier signé 8 bits, équivalent de `int` sur les architectures classiques

`int32_t :` entier signé 32 bits, équivalent de `int` sur les architectures classiques

`int64_t :` entier signé 64 bits, équivalent de `long int` sur les architectures classiques

`uint8_t :` entier non signé 8 bits (`u` comme *unsigned*), équivalent de `char` ou de `short int` sur les architectures classiques

`uint32_t :` entier non signé 32 bits, équivalent de `unsigned int` sur les architectures classiques

`uint64_t :` entier non signé 64 bits, équivalent de `unsigned long int` sur les architectures classiques

Question : Dans quel fichier sont définis ces types ? Trouvez les dans votre arborescence de fichier Linux. Une bibliothèque implémentant une structure de données de type liste de couple clé-valeur est disponible sur Moodle sous la forme de deux fichiers `mylist.c` et `mylist.h`. Par défaut, cette bibliothèque gère des couples clé-valeur dont les clés sont des chaînes de caractères et les valeurs des `double`. On gardera cette implémentation pour l'ensemble de ce TP.

Allez voir le contenu de ces deux fichiers pour vous remettre en tête les primitives de liste et leur implémentation par maillons chaînés.

Toutes les fonctions de ce TP seront écrites dans un fichier nommé `NOM_hashtbl.c`.

Exercice 1 (Fonction de hachage).
On utilisera, dans toute la suite de ce TP, la fonction de hachage suivante permettant de hacher des chaînes de caractères de taille quelconque :

$$\begin{array}{rccl} h: & \mathcal{S} & \longrightarrow & [\![0, m-1]\!] \\ & c_1 \dots c_r & \mapsto & \displaystyle\sum_{j=1}^{r} \text{code\_ascii}(c_j) \times b^j \bmod m \end{array}$$

où m et b seront des entiers qui seront choisis par l'utilisateur de notre bibliothèque.

1. Implémenter cette fonction de hachage en C sous la forme d'une fonction `int hash_function(int b, int n_buckets, char *key`. L'algorithme implémenté doit être de **complexité temporelle linéaire en la taille de la chaîne hachée**.
2. Implémenter une fonction `void test_hash_function(int b, int n_buckets)` qui demande en boucle à l'utilisateur de rentrer une chaîne de caractères de taille quelconque, récupère cette chaîne de caractère, calcul son empreinte par la fonction de hachage et l'affiche à l'écran.
3. Utilisez votre votre fonction de test avec $b = 31$ et $n\_buckets = 8$ et amusez vous à regarder les empreintes générées. Vous testerez des chaînes de caractères de plus en plus longue. Que se passe-t-il ?
4. Montrez que l'utilisation d'entiers 64 bits `int64_t` pour représenter les empreintes ne corrige pas le problème.
5. Proposez une remédiation simple à ce problème qui limite le nombre de collisions.

Dans toute la suite, on fixe $b = 31$. Dans le code, on pourra utiliser la directive de pré-processing `#define` pour fixer cette valeur, ou bien stocker cette valeur dans le segment statique `RODATA`. Attention, le préprocesseur effectue un simple remplacement syntaxique : faites attention pour qu'il ne remplace que

les chaînes aux bons endroits.

Exercice 2 (Implémentation des primitives de table de hachage en C).

1. Récupérer la bibliothèque `mylist` dont le code est mis à disposition sur Moodle.
2. En utilisant la bibliothèque `mylist` mise à disposition, définir un type C `hashtbl` permettant d'implémenter des tables de hachage. Vous créerez un script Shell `NOM_compile_hashtbl.sh` pour faciliter la compilation séparée des différents fichiers source.
3. Implémenter les primitives des tables de hachage :

   `hashtbl_create :` constructeur, alloue une structure de données vide

   `hashtbl_free :` destructeur, libère l'espace mémoire alloué à la structure de données

   `hashtbl_find :` accesseur, permet de savoir si une entrée est présente dans la structure de données. Il faudra gérer le cas où la valeur n'est pas trouvée

   `hashtbl_add :` ajout d'une valeur dans la table, c'est-à-dire calcul de l'empreinte de sa clé puis ajout dans le seau correspondant à cette empreinte.

   `hashtbl_remove :` suppression d'une entrée dans la table, c'est-à-dire calcul de l'empreinte de sa clé puis localisation et suppression de la valeur dans le seau

   `hashtbl_replace :` remplacement d'une entrée dans la table, c'est-à-dire calcul de l'empreinte de sa clé puis localisation et remplacement de la valeur par une nouvelle valeur si la clé est présente, ou bien ajout d'une entrée avec la valeur souhaitée sinon.

   Vous mettrez en œuvre les principes de la programmation défensive.

   Vous testerez chacune de vos fonctions sur de petits exemples, directement en dur dans une fonction de test appelée par la fonction principale `main`.
4. Assurez vous que la complexité temporelle de chacune des primitives de transformation implémentées correspond bien à celle donnée en cours.

Exercice 3 (Statistiques sur les tables de hachage).

Écrire une fonction `hashtbl_stats` qui renvoie le nombre d'entrées présentes dans la table et affiche :

- le facteur d'équilibrage théorique ;
- la taille maximale s des seaux ;
- le diagramme en bâtons des tailles des seaux, c'est-à-dire qui affiche, pour chaque taille q potentielle, le nombre de seaux de taille q.

Exercice 4 (Application : hachage sur des volumes de données plus importants).

1. Implémentez une fonction `hashtbl *hash_file(char *path)` qui hache les chaînes de caractères correspondant aux lignes d'un fichier et stocke une entrée pour chaque ligne avec le numéro de ligne comme valeur. Attention, la bibliothèque `mylist` stocke des valeurs de type `double`.
2. La distribution Linux que vous utilisez met à disposition des dictionnaires de plusieurs langues dans le dossier `/usr/share/dict`. Allez y faire un tour. Hachez le dictionnaire français avec la fonction de hachage proposez dans ce TP. Combien y-a-t'il de mots dans ce dictionnaire ? On propose un nombre de seaux égal à 350033. Cela vous parait-il pertinent ? Quel coût de recherche optimal d'un mot dans ce dictionnaire peut-on espérer avec ce nombre de seaux ? Quel est le coût réel ?
3. Modifier la valeur de b à 10 et observez maintenant le nombre de collisions. Qu'en pensez-vous ?
4. Créer une bibliothèque `myhashtbl.h` et `myhashtbl.c` que vous pourrez réutiliser dans tous les prochains TP.

Pour s'amuser... Continuez à jouer avec la valeur de b et essayez de comprendre ce qui se passe... Y-a-t-il une raison théorique au fait que l'on ait choisi la valeur $b = 31$?

5. HORS-PROGRAMME, pour ceux qui se sentent très à l'aise sur tout le reste : modularité, fonction d'ordre supérieur en C. Améliorez votre bibliothèque pour que l'utilisateur puisse choisir la fonction de hachage qu'il souhaite utiliser. Vous implémenterez la fonction de hachage sommative (somme des codes ASCII de tous les caractères de la chaîne) et la fonction naïve n'utilisant que le code du premier caractère et donnerez le choix à l'utilisateur.

   Pour déclarer une variable *addr_f* de type pointeur vers une fonction qui prend en entrée deux valeurs de type `type_entree1` et `type_entree2` et renvoie une valeur de type `type_sortie` on écrira par exemple ceci :

   ```
   type_sortie (*addr_f)(type_entree1, type_entree2);
   ```

   Pour stocker l'adresse de l'une des fonctions `ma_fonction` écrites dans notre code dans ce pointeur de fonction, on écrira, par exemple :

   ```
   f = &ma_fonction
   ```

   On pourra utiliser un type `enum` pour désigner les différents types de fonctions de hachage disponibles.