PROG
TP#7.II

2020-11-16

Instructions de rendu

Ce TP est à réaliser en monôme. Vous devez envoyer votre travail sous la forme d'une archive à pierre.karpman@univ-grenoble-alpes.fr au plus tard le 4/12/2020 à 18 :00. Celle-ci doit contenir :

— Votre programme répondant aux questions ce sujet, ainsi que de la première partie du TP (https://www-ljk.imag.fr/membres/Pierre.Karpman/pc2020_tp7.pdf). Il doit être possible de facilement exécuter toutes les expériences permettant de répondre aux questions.
— Les instructions de compilation (idéalement avec un Makefile).
— Un court compte-rendu en format PDF qui détaille vos réponses.

Tables de Hachage II : Table de hachage

Le but de cet exercice est d'implémenter une structure de donnée d'*ensemble* grâce à une table de hachage.

Un ensemble $\mathcal{S}$ supporte trois opérations :

— L'insertion, qui ajoute un élément (accompagné d'éventuelles données auxiliaires) à $\mathcal{S}$ s'il ne s'y trouvait pas déjà, et ne fait rien (et éventuellement renvoie une erreur) sinon.
— La recherche, qui cherche si un élément se trouve dans $\mathcal{S}$ et le renvoie (ainsi que ses données auxiliaires) si c'est le cas.
— La suppression.

La structure de table de hachage que nous proposons d'implémenter repose sur les principes suivants :

— Chaque élément de la table est identifié par une *clef* unique.
— La table est composée de N compartiments (*buckets*) permettant de stocker un « petit nombre » d'éléments.
— Chaque élément est stocké dans le compartiment identifié par le haché de sa clef $\in [\![0, N-1]\!]$.

On utilisera les structures de données suivantes, et `hash339` pour fonction de hachage :

```
struct elem
{
        void *key;
        size_t keylen;
        void *data;
        size_t datalen;
        uint64_t hash_key;
};
```

```c
struct bucket
{
        struct elem **elems;
        size_t len; // taille du bucket
        size_t count; // nombre d'éléments présents
};

struct ht
{
        struct bucket **table;
        size_t tablen;
        size_t base_bucketlen; // taille initiale des buckets lors de leur
        ↪  création
        uint32_t k; // clef de la fonction de hachage
};
```

Q.1 : Implémentez les fonctions suivantes d'initialisation et de destruction d'une table de hachage :

1. `struct ht* setup_ht(size_t tablen, size_t base_bucketlen)` qui initialise une table de hachage. Est-il nécessaire à ce point d'initialiser tous les compartiments ?
2. `void free_bucket(struct bucket *b)`, qui libère la mémoire utilisée par un compartiment, sans détruire les éventuels éléments qui y sont stockés.
3. `void free_bucket_full(struct bucket *b)`, qui libère la mémoire utilisée par un compartiment et les éventuels éléments qui y sont stockés, en faisant la supposition que ces derniers ont été alloués avec `malloc` (ou une fonction similaire). Pourquoi cette dernière supposition est-elle importante ?
4. `void free_ht(struct ht *tab)`, qui libère la mémoire occupée par une table de hachage, mais pas celle des éventuels éléments qu'elle contient.
5. `void free_ht_full(struct ht *tab)`, qui libère la mémoire occupée par une table de hachage ainsi que celle des éventuels éléments qu'elle contient (avec la même supposition que ci-dessus).

Testez vos fonctions, et vérifiez avec `valgrind` que vous n'avez pas de fuite mémoire ou d'accès mémoire illégaux[1].

Q.2 : Implémentez les fonctions suivantes d'insertion, recherche, et suppression d'éléments :

1. `int insert(struct ht *tab, void *data, size_t datalen, void *key, size_t keylen)`. Cette fonction ajoute un élément dans l'ensemble représenté par la table de hachage `tab`, et ne fait rien si cet élément est déjà présent. Définissez clairement votre politique d'allocation (et éventuelle désallocation) mémoire. L'utilisation de `realloc` pourra être utile.
2. `struct elem *retrieve(struct ht *tab, void *key, size_t keylen)`, qui recherche un élément de la table de hachage et le renvoie s'il est présent.
3. `int delete(struct ht *tab, void *key, size_t keylen)` qui supprime un élément présent dans la table de hachage (c'est à dire qui le retire de la table, mais sans le désallouer).
4. Optionnel : Faites en sorte (sans changer les structures) que le champ `data` puisse-t-être utilisé à la place de `key`, laissé vide.

1. Pour cela, lancez votre programme `prog` (optionnellement compilé avec une option `-g`) avec la commande `$> valgrind ./prog`.

Testez vos fonctions, et vérifiez avec `valgrind` que vous n'avez pas de fuite mémoire ou d'accès mémoire illégaux.

N.B. Vous êtes autorisés à supposer que les allocations mémoire n'échoueront jamais.

Q.3 : Utilisez votre table de hachage pour rechercher efficacement des collisions pour la fonction `fun` suivante :

```
uint32_t fun(uint64_t x)
{
    uint32_t hi = x >> 32;
    uint32_t lo = x & 0xFFFFFFFF;

    hi += lo;
    lo  = (lo << 5) ^ (lo >> 27);
    lo ^= hi;

    lo = (lo << 11) ^ (lo >> 21);

    return (lo + hi);
}
```

C'est à dire qu'on cherche `x`, `y` $\neq$ `x` t.q. `fun(x)` $=$ `fun(y)`.

1. En supposant que `fun` se comporte comme une fonction aléaoire, pour combien de valeurs devez vous calculer `fun` avant d'espérer trouver une collision ?
2. En vous basant sur le *paradoxe des anniversaires*, quelle taille N de table de hachage vous semble appropriée ? (Justifiez votre choix, y compris relativement au comportement attendu de la fonction de hachage utilisée pour cette taille.)
3. Donnez le nombre moyen de valeurs (aléatoires) testées avant de trouver une collision, ainsi que le minimum et le maximum, pour un échantillon de taille moyenne (par ex. quelques dizaines de tests).

Q.4 : Refaites les tests de la question précédente avec une fonction de votre choix à la place de `fun`.