Séances 4 et 5 : Table de hachage

En séance 4 et 5, on implémente notre propre table de hachage ! Ce TP, plus conséquent que ce que vous avez eu à implémenter jusqu'à présent, est **à faire par équipe de 3, constituée de membres de votre groupe de TP**.

Pas de note à la fin, l'objectif est bien ici de travailler encore une fois les pointeurs, et de comprendre deux-trois trucs sur les tables de hachage quand même, si nécessaire.

C'est aussi l'occasion de démarrer un travail en équipe et toute l'expérience que vous engrangerez ici vous sera utile lors du projet de fin d'année (lui aussi à faire en trinômes).

Présentation

Une table de hachage est une structure de données permettant d'implémenter efficacement des dictionnaires. Sous des hypothèses raisonnables, les opérations principales (insertion, recherche, suppression) sont à coût constant *O(1)* amorti.

Souvenons des mois derniers pour faire du lien entre nos cours

- Algorithmique et structures de données : Coût amorti et tables de hachage
- BPI : Structure de données versus types abstrait

La structure est en fait un tableau `T` de `m` listes chaînées (voir Fig. 1, tirée de *"Introduction to algorithms, second edition", Cormen et al., The MIT Press.*). Lors de l'insertion d'un couple `<cle, valeur>`, une *fonction de hachage* est appliquée à la clé, qui retourne une valeur entière `h(cle)`. L'insertion a alors lieu dans la liste chaînée d'index `h(cle)`. Chaque liste `T[k]` contient donc tous les éléments dits en collision, c'est-à-dire dont la valeur de hachage `h(cle)` est `k`.

L'archive de départ pour ce TP est disponible ici : hash.tar.gz

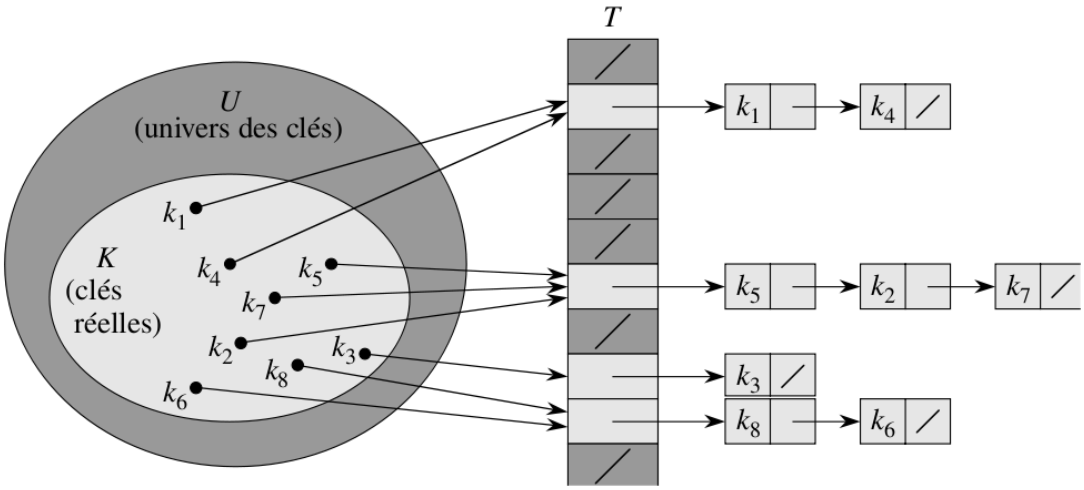

Fig. 1 : Structure d'une table de hachage `T`. Chaque liste chaînée en `T[k]` contient tous les éléments dont la valeur de hachage `h(cle)` est `k`. Par exemple `h(k1) = h(k4)` et `h(k5) = h(k2) = h(k7)`

Le nombre de cases disponible dans le tableau étant généralement plus petit que le nombre des valeurs de hachage, l'insertion a en fait lieu dans la liste positionnée en `h(cle)` modulo `m`. Les clés sont supposées uniques : une valeur n'est insérée que si elle n'est pas déjà présente dans la liste. Dans le cas général, aucun ordre n'est maintenu sur les listes de collision.

Les performances d'une table de hachage dépendent très fortement de la fonction de hachage. Dans le pire des cas, tous les éléments sont dans la même liste de collision et les coûts sont identiques à une liste chaînée simple. Une bonne fonction de hachage répartit au contraire les éléments de manière quasi-uniforme dans les listes de la table, tout en gardant un coût de calcul raisonnable.

Table de hachage statique

Le travail demandé consiste à utiliser une table de hachage pour implémenter un annuaire associant des numéros de téléphone à des noms (les clés). Dans une première approche, cette table sera *statique*, c'est-à-dire que sa taille sera fixée à la création de la table et ne changera plus ensuite. Dans un deuxième temps, on s'intéressera à la possibilité de redimensionner dynamiquement la table de hachage.

Les données manipulées sont des noms et numéros de téléphone représentés comme des chaînes de caractères. En C, une chaîne de caractères se termine toujours par le caractère `\0`. La bibliothèque standard du langage C fournit diverses fonctions de manipulation de chaînes de caractères, déclarées dans le fichier `string.h`.

Architecture logicielle

Cette section fixe les consignes quant à l'organisation de votre programme en différents modules. Vous implémenterez ici une *bibliothèque de gestion d'un annuaire*, qui prendra la forme d'une bibliothèque dynamique (fichier `.so`) embarquant toutes les fonctionnalités demandées.

Vous devez respecter scrupuleusement le découpage proposé et la spécification ci-dessous (pas de modification des prototypes).

Votre projet comportera *au moins* trois modules dont le contenu est décrit ci-après. L'arborescence de votre projet est aussi imposée et sera décrite à la fin de ce document.

Module contact

Un `contact` est une structure de données représentant un contact de l'annuaire, à savoir une association entre un nom et un numéro de téléphone.

Le module `contact` implémente les fonctionnalités nécessaires à la manipulation d'un chaînage de contacts par le module `directory`, décrit juste après, implémentant l'annuaire à proprement parler.

Vous pouvez voir ce module comme la spécialisation d'un paquetage de gestion de liste chaînée, qu'on peut aisément tester en dehors du projet (c'est-à-dire sans qu'il soit utilisé par un annuaire).

La spécification de ce module n'est pas distribuée, **c'est à vous d'identifier les fonctionnalités qui seront utiles pour construire un annuaire de contacts**. De la même manière, vous devrez mettre en place vous-même une batterie de tests validant votre implémentation.

Module directory

Ce module définit une structure d'annuaire (*directory* en anglais) `struct dir` comme un tableau de listes chaînées contenant des `contact` représentant une association entre un nom et un numéro de téléphone.

Les opérations demandées sur la structure d'annuaire sont détaillées dans l'entête `directory.h` :

```
1  /*
2    Structure de données représentant un annuaire.
3    Son contenu est détaillé dans directory.c.
4  */
5  extern struct dir;
6
7  /* Crée un nouvel annuaire contenant _len_ listes vides. */
8  struct dir* dir_create(uint32_t len);
9
10 /*
11   Insère un nouveau contact dans l'annuaire _dir_, construit à partir des
12 nom et
```

```
13    numéro passés en paramètre. Si il existait déjà un contact du même nom,
14  son
15    numéro est remplacé et la fonction retourne une copie de l'ancien
16  numéro.
17    Sinon, la fonction retourne NULL.
18  */
19  char* dir_insert(struct dir* dir, const char* name, const char* num);
20
21  /*
22    Retourne le numéro associé au nom _name_ dans l'annuaire _dir_. Si
23  aucun
24    contact ne correspond, retourne NULL.
25  */
26  const char* dir_lookup_num(struct dir* dir, const char* name);
27
28  /*
29    Supprime le contact de nom _name_ de l'annuaire _dir_. Si aucun contact
30  ne
31    correspond, ne fait rien.
32  */
33  void dir_delete(struct dir* dir, const char* name);
34
    /* Libère la mémoire associée à l'annuaire _dir_. */
    void dir_free(struct dir* dir);

    /* Affiche sur la sortie standard le contenu de l'annuaire _dir_. */
    void dir_print(struct dir* dir);
```

REMARQUES :

- La structure `struct dir` est simplement déclarée dans le fichier d'en-tête. L'implémentation de cette structure (c'est-à-dire *son contenu*) doit apparaître dans le code source implémentant le module `directory` (`directory.c`, par exemple). On sépare ainsi l'interface de l'implémentation. Ainsi, un programme de test qui fait appel aux fonctions déclarées dans `directory.h` permettra de tester *n'importe quelle implémentation du module* `directory` (comprenez : quels que soient les champs que vous définissez à l'intérieur de la structure `struct dict`), à condition que ce module implémente bien toutes les fonctionnalités exposées par le `.h`, bien entendu ;
- Le type `const char*` représente une chaîne de caractère constante (dont les caractères ne peuvent pas être modifiés). Plus généralement, le mot-clé `const` appliqué à un type pointeur sous la forme `const type*` indique que la mémoire référencée par le pointeur ne peut pas être modifiée. Conséquence directe : il est interdit de passer un paramètre de type `const type*` à une fonction f dont le prototype serait `void f(type*)` (comme `free` par exemple).

Module hash

La fonction de hachage des noms est donnée par l'algorithme suivant:

Algorithm 1 Fonction de hachage

Require: a string str
Ensure: an unsigned int corresponding to the hash value of the key.

```
hash ← 5381
c ← first character of str
while c ≠ '\0' do
    hash ← hash * 33 + c
    c ← next character of str
end while
return hash
```

Fig. 2 : Fonction de hachage classique et efficace, introduite par Daniel J. Bernstein. Noter l'utilisation des valeurs 5381 (nombre premier) et 33 (= 32 + 1).

Le module `hash` devra contenir l'implémentation de cette fonction de hachage. Bien qu'elle ne soit utilisée que par le module `directory`, on décide ici de placer cette fonction dans un module à part, ce qui permettrait d'étendre votre projet en ajoutant d'autres fonctions de hachage pour évaluer leur impact sur la façon dont se remplit votre table de hachage. Les fonctions de hachage sont aussi indépendantes du reste du projet et peuvent donc être testées de manière autonome.

Redimensionnement dynamique de la table de hachage

Lorsque le nombre de contacts grandit, le risque de collisions augmente et donc la longueur des listes et le coût des opérations. Inversement, si le nombre de contacts diminue, la table va utiliser inutilement de la mémoire. Dans cette 2ème étape, on souhaite donc disposer de fonctions **internes à votre bibliothèque** permettant le redimensionnement de la table afin de minimiser les coûts liés à une table très remplie, ou de réduire l'empreinte mémoire d'une table majoritairement vide. Ainsi, on :

- doublera la taille de la table lorsque le nombre de contacts se trouvant dans l'annuaire dépasse 75% du nombre d'entrées dans la table ;
- on divisera par deux la taille de la table sans toutefois descendre en dessous de 10 entrées lorsque le nombre de contacts se trouvant dans l'annuaire représente moins de 15% du nombre d'entrées dans la table.

Bien entendu, il faudra mettre en œuvre de nouveaux tests démontrant le bon fonctionnement de l'agrandissement et du rétrécissement dynamique de l'annuaire.

Squelette de code

Nous fournissons un squelette de code pour vous aider à démarrer.

Arborescence

Nous vous demandons de respecter l'arborescence fournie, qui contient :

- un répertoire `src/` qui regroupe les fichiers source (`.c`) permettant de construire la bibliothèque ;
- un répertoire `include/` qui regroupe les entêtes des modules de la bibliothèque ;
- un répertoire `obj/` qui accueillera les fichiers objets (`.o`) construits lors de la compilation du projet ;
- un répertoire `lib/` qui contiendra la bibliothèque dynamique `libdirectory.so`, produit final de votre projet ;
- un répertoire `tests/` qui contient le code source de programmes de tests ;
- un `Makefile` permettant de construire la bibliothèque à partir des sources et les exécutables de test.

On retrouve souvent ce type d'organisation, la règle communément admise étant de *"ne jamais compiler dans les sources"* (expression familière qui laisse entendre que générer des fichiers issus de la compilation dans le même répertoire que celui où se trouvent les fichiers source qui ont permis de les générer, c'est mal).

Compilation

Le `Makefile` fourni est capable de générer la bibliothèque dynamique `libdirectory.so` avec les contraintes suivantes :

- taper `make` dans le répertoire de départ lance la génération du fichier `lib/libdirectory.so` ;
- la compilation d'un fichier `src/toto.c` provoque la création d'un fichier objet `obj/toto.o` ;
- à l'édition des liens, on fusionne les fichiers objets situés dans le répertoire `obj/` pour créer la bibliothèque dynamique `libdirectory.so` dans le répertoire `lib/` ;
- la règle `clean` supprime les fichiers générés, c'est-à-dire le contenu des répertoires `obj/` et `lib/` ;
- la règle `tests` permet la compilation des programmes de tests (dont les sources se trouvent dans `tests/`).

Annexes

Génération d'une bibliothèque dynamique

Pour construire une bibliothèque dynamique, il suffit de :

- rajouter l'option `-fPIC` à la compilation (dans les `CFLAGS`) ;
- rajouter l'option `-shared` lors de l'édition des liens (dans les `LDFLAGS`).

Pour lier un programme de test à la bibliothèque dynamique, il suffit de :

- rajouter l'option `-Llib/ -ldirectory` lors de l'édition des liens, avec `lib/` correspondant au chemin (relatif ou absolu) dans lequel se trouve le fichier `libdirectory.so` ;
- rajouter le chemin absolu vers `libdirectory.so` à la variable d'environnement `LD_LIBRARY_PATH`, par exemple :

```
1  LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/home/bibi/projetC/lib ./mon_super_test
```

À la manière de `$PATH`, `$LD_LIBRARY_PATH` liste un ensemble de répertoires dans lesquels on va chercher des bibliothèques dynamiques (fichiers `.so`) à charger lors de l'exécution d'un programme. Vous pouvez afficher la liste des bibliothèques dynamiques chargées à l'exécution d'un programme `toto` à l'aide de la commande :

```
1  ldd ./toto
```

Vous trouvez ça pénible ? Trois choix s'offrent à vous :

- *exporter* la nouvelle valeur de la variable `LD_LIBRARY_PATH` pour qu'elle soit chargée à chaque fois que vous lancez un nouveau terminal, en rajoutant cette ligne au fichier `.bashrc` qui se trouve à la racine de votre compte (overkill!) (bien entendu, le chemin `/home/coco/boulot/C/hash/lib` est à remplacer par le vôtre) :

```
1  export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/home/coco/boulot/C/hash/lib
```

- OU ajouter une règle dans votre `Makefile` permettant de lancer votre programme principal qui intègre le positionnement de la variable d'environnement à la bonne valeur (vachement mieux, déjà) ;
- OU indiquer lors de la *compilation* du programme que la bibliothèque qu'il faudra charger à l'exécution se trouve dans `/home/toto/tps/hash/lib` en ajoutant l'option `-Wl,-rpath=/home/toto/tps/hash/lib` lors de la phase d'édition des liens (variable LDFLAGS dans le Makefile) (hey ouais, c'est un tips "ceinture noire", très utile pour briller dans les diners mondains).

Le `Makefile` du projet utilise cette dernière technique pour générer des programmes de tests exécutables quel que soit l'environnement dans lequel ils sont lancés.