# Hash tables – part I

Jean-Cédric Chappelier 2017

from former document by Laurent Bindschaedler + Jean-Cédric Chappelier 2016

Le but du travail de cette semaine est :

1. de prendre plein connaissance du cadre et des concepts du projet : commencez par lire le fichier de description principal ;
2. de créer les structures de données relatives aux tables de hachage locales ;
3. et éventuellement commencer la mise en place du cœur du projet (semaine prochaine).

## I. Implémentation d'une hashtable

Pour rappel (cours *Algorithms* (CS-250), BA3), une *hashtable* (« table de hachage ») est une structure de données qui permet un accès en temps (quasi-)constant ($O(1)$) à une valeur à l'aide de la « *valeur de hachage* » (*hash value*) d'une clé d'accès :

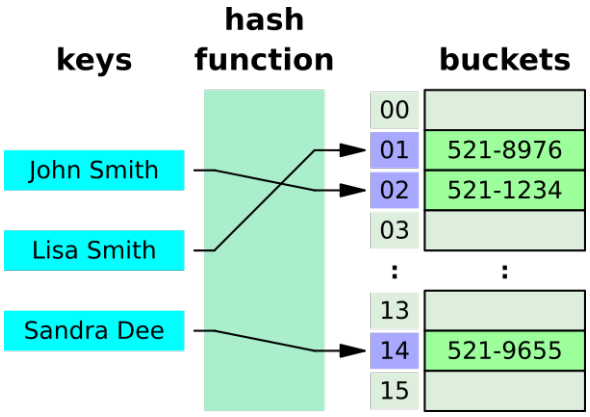


Fig. : *une hashtable lie une clé à une valeur à l'aide d'une fonction de hashing.* (CC) Jorge Stolfi@WikiMedia

La représentation concrète de la *hashtable* que nous vous proposons d'implémenter à terme dans ce projet ressemblera à ceci :

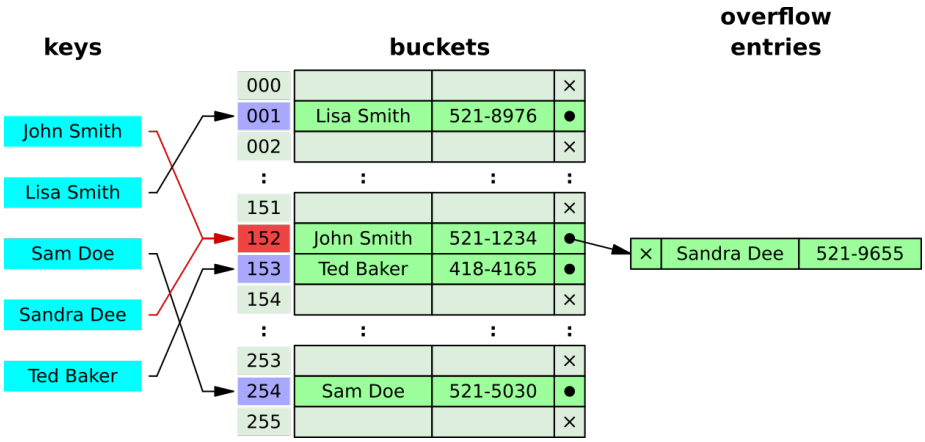


Fig. : *représentation d'une hashtable avec gestion des collisions par listes chaînées.* (CC) Jorge Stolfi@WikiMedia

mais pour le moment (jusqu'à la semaine 7) ce sera un simple tableau d'entiers.

Définissez (dans `hashtable.h`):

- le type de la clé est `pps_key_t`, pour le moment simplement `char` et plus tard (semaine 7) `const char*` lorsque nous les aurons vus en cours.
- et le type de la valeur, `pps_value_t`, pour le moment `int32_t` et plus tard (semaine 7) `const char*`.

Définissez ensuite les types :

- `bucket_t`, qui correspond à un contenu de *hashtable*, une « alvéole » (ou « position », « *buckets* »), pour le moment simple un alias sur `pps_value_t`,
- et `Htable_t` qui correspond à une *hashtable* contenant `size` « alvéoles », 256 dans la figure ci-dessus et `HTABLE_SIZE` pour nous dans ce projet.

`size` est une caractéristique de la *hashtable* (indiquant sa taille en nombre de position possibles ; notez que le vrai nombre d'entrées contenues dans la table peut être différent : certaines positions sont vides (exemple la 002 ci-dessus), certaines clés peuvent partager la même position (exemple « John Smith » et « Sandra Dee » ci-dessus ; on parle alors de « *collision* »)).

Définissez ensuite les fonctions suivantes qui permettent d'effectuer un ensemble d'opérations standard sur la *hashtable* (les prototypes dans `hashtable.h`, les implémentations dans `hashtable.c`):

- `error_code add_Htable_value(Htable_t, pps_key_t, pps_value_t)` qui prend une *hashtable* en argument ainsi qu'une clé et une valeur et qui stocke la valeur associée à la clé dans la *hashtable* ; on supposera pour le moment qu'il n'y a pas de collision ; cette fonction se contentera donc d'écrire la valeur sans autre modification ; (ce comportement sera modifié plus tard dans le semestre lorsque nous aurons vu les pointeurs et changerons le type des clés) ; si la table est `NULL`, elle doit retourner

`ERR_BAD_PARAMETER`, si tout fonctionne normalement, elle doit retourner `ERR_NONE` (les différentes erreurs et fonctions outils se trouvent dans `error.h`)

- `pps_value_t get_Htable_value(Htable_t, pps_key_t)` qui prend une *hashtable* en argument ainsi qu'une clé et qui retourne la valeur associée ; ce comportement sera modifié plus tard dans le projet pour avoir une valeur impossible qui sera utilisée comme code d'erreur ; le comportement n'est pas défini pour une clé qui n'a pas été ajoutée avant (vous pouvez retourner n'importe quoi) ;
- `size_t hash_function(pps_key_t, size_t)` qui calcule la valeur de hash d'une clé donnée, pour une taille de *hashtable* donnée (`size`) et retourne un index (numéro d'alvéole/bucket, de type `size_t`) ; pour le moment, il suffit de retourner la valeur de la clé modulo `size` ; si la taille est plus grande que celle de la table, la fonction doit retourner 0

N'hésitez pas à créer d'autres fonctions utilitaires si nécessaire !

Et n'oubliez pas de tester correctement les fonctionnalités de votre *hashtable* pour plusieurs combinaisons de tailles, clés et valeurs **avant** de passer à l'étape suivante. Cela vous évitera bien des tracas...

# II. Complément optionnel

Cette partie n'est pas strictement nécessaire pour le moment, ni d'ailleurs pour le reste du projet (on peut faire sans ; c'est moins bien, mais on *peut*).

Définissez un type `kv_pair_t` qui représente une paire (clé, valeur) ; c.-à-d. qui regroupe une clé et une valeur. Ce type pourra être utile lorsque vous aurez à manipuler conjointement des clés et leur valeur associée.

# III. Exemple et tests

Nous vous fournissons un fichier `test-hashtable.c` que nous vous conseillons fortement de **copier** dans un fichier `week04.c` à éditer pour y ajouter vos propre tests : remplacez la ligne

```
puts("Ecrivez ici vos tests et SUPPRIMEZ ce puts");
```

par tout ce qui vous semble nécessaire.

**Note :** n'éditez pas directement le fichier `test-hashtable.c` lui-même car il sera modifié (et écrasé) par nous pour nos propres soins. Travaillez donc plutôt sur une copie `week04.c` de ce fichier.
Et n'oubliez pas de mettre à jour votre `Makefile` !
[**fin de note**]

Vous pourriez par exemple tester que l'ajout d'une valeur à une clé donne la bonne valeur lorsqu'on la redemande :

```
Htable_t table;

const pps_key_t cle = 'c';
const pps_value_t valeur_mise = 42;
add_Htable_value(table, cle, valeur_mise);

const pps_value_t valeur_lue = get_Htable_value(table, cle);

ck_assert_int_eq(valeur_mise, valeur_lue);
```

**Note :** cet exemple n'est pas forcément parfait, ni même complet ; libre à vous de l'adapter comme bon vous semble.
[**fin de note**]

Pour les tests, nous utilisons la bibliothèque Check. Pour tester si deux `int` sont égaux, utilisez alors la « fonction » `ck_assert_int_eq` : `ck_assert_int_eq(a, b)`.

Nous avons également défini les « fonctions » suivantes dans `test.h` :

- `ck_assert_err_none(int erreur)` : teste si l'erreur `erreur` est `ERR_NONE` (c.-à-d. correspond à un retour de fonction sans erreur ; voir `erreur.h`) ;
- `ck_assert_bad_param(int erreur)` : teste si l'erreur `erreur` est `ERR_BAD_PARAMETER` (c.-à-d. correspond à une erreur d'une fonction ayant reçu un mauvais paramètre ; voir `erreur.h`) ;
- `ck_assert_ptr_nonnull(void* pointeur)` : teste si le pointeur `pointeur` n'est pas `NULL` ;
- `ck_assert_ptr_null(void* pointeur)` : teste si le pointeur `pointeur` est `NULL`.

Pour faire l'édition de lien avec la bibliothèque Check, il faut ajouter les options `-lcheck -lm -lrt -pthread` ; par exemple :

```
cc test-hashtable.o hashtable.o error.o -lcheck -lm -lrt -pthread -o test-hashtable
```

Sur certaines architectures, il faut aussi ajouter la bibliothèque `-lsubunit`.

# IV. Organisation du travail

**REMARQUE IMPORTANTE :** en raison de vos connaissances en C (synchronisation avec le cours), le travail de cette semaine-ci est assez léger ; celui de la semaine prochaine (on attend les pointeurs !) sera par contre *très conséquent* : ce sera toute la mise en place du cœur du projet (revoir si nécessaire le document général de présentation).

Mais vous pouvez néanmoins commencer sa mise en place sans plus attendre. Pour cela (vous aider à équilibrer la charge de travail), nous avons déjà mis en ligne le sujet de la semaine prochaine). Nous vous conseillons de le commencer gentiment en laissant si nécessaire de coté les *quelques* points qui nécessiteraient des pointeurs. L'essentiel peut déjà être mis en place dès cette semaine.

Libre à vous, donc, de vous organiser au mieux dans votre travail suivant vos contraintes.

# V. Rendu

Avant la soumission, veuillez retirer (ou commenter) tous les appels à `printf()` superflus que vous auriez pu ajouter.

Pour rendre le devoir, ajoutez les quatre fichiers suivants à un répertoire `done/` de votre dépôt GitHub **de groupe** : `Makefile`, `hashtable.h`, `hashtable.c` et `week04.c` puis « tagger » avec le tag `week04` et « pousser » le résultat vers GitHub (pour **le groupe**) : `git push --tags`.

Ce rendu ne sera pas évalué en tant de tel (c.-à-d. seul), mais devra faire partie du premier rendu intermédiaire du projet (délai : le dimanche 22 avril 23:59). Néanmoins, nous vous conseillons fortement de travailler régulièrement et faire systématiquement ces tags hebdomadaires (si votre travail est opérationnel). Cela vous aidera vous-mêmes à mesurer votre progression, d'autant plus que nous vous fournirons un retour systématique (dans votre dépôt GitHub) des résultats de certains tests que nous aurons fait tourner sur vos rendus (s'ils sont présents et taggés). Sachez donc profiter de ce compte-rendu hebdomadaire !

Et n'oubliez pas de faire le rendu (individuel) de la semaine passée avant dimanche soir : n'oubliez pas de tagger (`git tag week03`) et **pousser** votre tag vers GitHub (`git push --tags`) depuis votre dépôt *personnel*.