

Implémentation d'une table de hachage parallèle

Raphaël Prévost | Wargan Solutions

3 juin 2013

Résumé

La table de hachage est une structure de données généralement utilisée pour implémenter tableaux associatifs, caches, ou plus généralement associer une clé à une valeur.

Dans le cadre de notre projet, nous avons choisi de réaliser un cache de données reposant sur cette structure, particulièrement intéressante dans notre cas pour ses temps d'accès relativement constants par rapport au nombre d'entrées dans la table.

Cependant, la majorité des implémentations libres ne sont pas optimisées pour une utilisation parallèle. Nous avons donc décidé de programmer cette structure nous-mêmes, en nous basant sur les travaux publiés sur le sujet. Nous décrirons dans cet article le cheminement que nous avons suivi afin de mener à bien cette implémentation.

Première partie

Dans l'intimité d'une table de hachage

La table de hachage n'est pas une structure complexe : c'est en réalité un vulgaire tableau. Tout l'art et la difficulté de son implémentation se résume à deux problèmes principaux : le premier consiste à trouver une fonction injective (fonction de hachage) associant *efficacement* à toute clé un emplacement quelconque dans le tableau.

Le second est plus subtil et procède directement du premier : notre tableau ne disposant que d'un nombre fini d'emplacements, il est nécessaire de gérer le cas où la fonction de hachage attribue le même emplacement à plusieurs clés différentes (collision). C'est le problème de la gestion de ces collisions qui est le plus ardu, et il existe de nombreuses façons de le traiter. Nous allons les passer en revue concisément dans les paragraphes suivants.

1 Deux écoles principales : «*chaining*» contre «*open addressing*»

Globalement, on peut diviser les algorithmes de résolution de collisions en deux familles principales, le «*chaining*» favorisant plutôt la montée en charge, et l'«*open addressing*» privilégiant une consommation mémoire réduite.

1.1 «*Chaining*»

Cet algorithme consiste à régler effectivement la collision en stockant les deux paires clé-valeur conflictuelles dans le même emplacement. Ceci est rendu possible en définissant chaque élément du tableau comme le point d'entrée d'une liste chaînée.

Cette méthode présente les avantages d'être simple à implémenter et de conserver de bonnes performances même lorsque le tableau se remplit ; en effet, le temps requis pour retrouver une clé augmente linéairement avec le nombre de collisions.



En revanche, elle a tous les inconvénients des listes chaînées : dommageable pour le cache du processeur, suppression des éléments avec une complexité linéaire, et consommation mémoire accrue par la nécessité de stocker les pointeurs.

De nombreuses implémentations se basent sur cet algorithme, car il n'impose pas de redimensionner la table ; en effet, il est toujours possible d'y insérer une nouvelle clé au prix d'une certaine perte de performance.

1.2 «*Open addressing*»

L'alternative, «*open addressing*», enregistre la clé conflictuelle directement dans un emplacement disponible arbitraire dans le tableau.

Les coordonnées de cet autre emplacement sont calculées soit à partir de l'emplacement où la clé aurait dû être insérée, soit en utilisant une autre fonction de hachage.

Quand le tableau se remplit, il peut être nécessaire d'effectuer ces calculs un certain nombre de fois avant de trouver un emplacement libre.

Cette méthode est plus efficace au niveau de la gestion de la mémoire, puisqu'il n'est pas nécessaire d'en allouer et que les données sont disposées idéalement pour le cache processeur. Les performances se dégradent toutefois de manière très marquée dès que la table atteint un certain seuil de charge.

2 Les hybrides

Il existe d'autres algorithmes empruntant certains principes propres aux deux méthodes présentées précédemment, en les combinant pour compenser leurs faiblesses respectives. Nous nous sommes particulièrement intéressés à ces derniers, dans le but de réaliser une implémentation à la fois rapide et efficace.

2.1 «*Cuckoo hashing*»

Cette méthode, dérivée de l'«*open addressing*» est exposée dans un article de recherche éponyme de Rasmus Pagh et Flemming Rodler publié en 2001.

Le principe consiste à utiliser deux fonctions de hachage, afin de disposer de deux emplacements possibles pour chaque clé. Lors de l'insertion d'une clé, si les deux emplacements sont occupés, l'algorithme va purement et simplement «éjecter» la clé déjà en place, comme un bébé coucou éjecte les œufs du nid qu'il parasite, pour la remplacer par les nouvelles données.

La clé éjectée est ensuite réinsérée dans le tableau suivant le même algorithme. On voit cependant que ce principe boucle infiniment quand le tableau devient saturé. Il est possible de retarder cet instant en utilisant un plus grand nombre de fonctions de hachage, augmentant ainsi le nombre d'emplacements potentiels.

Cet algorithme a pour principal intérêt d'offrir des temps d'insertion et de lecture constants, ainsi qu'une bonne utilisation de la mémoire (dépendante toutefois du nombre de fonctions de hachage utilisées).

2.2 «*Cuckoo hashing*» et panier

Dans un article d'Adam Kirsh, Michael Mitzenmacher et Udi Wieder publié en 2008, l'idée est avancée de pallier les défauts du «*cuckoo hashing*» en adjoignant un «panier» à la table de hachage, de façon à recueillir les clés «tombées du nid».

Ce «panier» se présente sous la forme d'un tableau secondaire de taille fixe, dans lequel sont insérées temporairement les clés pour lesquelles un emplacement n'a pu être trouvé. Ce procédé permet de retarder efficacement l'agrandissement de la table, sans utiliser plus de fonctions de hachage.



Finalement, le «*cuckoo hashing*» permet d'améliorer grandement les performances du modèle «*open addressing*» en offrant des performances constantes en dépit du nombre d'entrées dans la table.

Son inconvénient majeur, l'impossibilité d'insérer de nouvelles clés au delà d'une certaine charge, peut être contourné en augmentant le nombre de fonctions de hachage et par l'adjonction d'un «panier» pour recueillir les clés problématiques.

Examinons maintenant les solutions permettant d'améliorer les algorithmes basés sur le «*chaining*».

2.3 «*Chaining*» interne et agrégation

Les améliorations apportées aux tables utilisant le «*chaining*» concernent d'avantage la consommation et la disposition des données en mémoire plutôt que les performances.

Une optimisation courante consiste à inclure le premier maillon de la liste chaînée directement dans l'emplacement réservé du tableau, afin de permettre une meilleure utilisation du cache dans le cas où il n'y a pas de collisions.

Il est également possible de remplacer la liste par un tableau dynamique, afin de préserver le cache du processeur et d'économiser la mémoire qui autrement serait utilisée pour stocker les pointeurs de la liste.

D'autres algorithmes vont plus loin et n'utilisent pas de réelles listes : les éléments sont directement placés dans le tableau lui-même, ce qui est appelé «*chaining*» interne. Cette méthode se rapproche de l'«*open addressing*», mais sans présenter l'inconvénient de calculer la position des emplacements alternatifs.

Deuxième partie

Notre implémentation

On a vu qu'en adoptant une approche hybride entre «*chaining*» et «*open addressing*», il est possible de réaliser des algorithmes avec des performances et une occupation mémoire plus équilibrées. Nous nous sommes donc efforcés d'effectuer une synthèse des différents travaux que nous avons évoqué précédemment dans le but de trouver un compromis satisfaisant entre performances brutes et utilisation de la mémoire.

Le «*cuckoo hashing*» nous a particulièrement intéressés du fait de ses temps d'accès constants ; nous l'avons donc utilisé comme base tout en cherchant à pallier le problème des insertions impossibles dont cet algorithme est affligé quand la table devient trop pleine.

1 Algorithme de résolution des collisions

La solution que nous avons retenue est, sans surprise, un «*cuckoo hashing*» adjoint d'un «panier». Le «*cuckoo hashing*» permet une bonne utilisation de l'espace de stockage de la table, tout en permettant un recyclage automatique des emplacements occupés par des clés supprimées. Avec une fonction de hachage de qualité raisonnable, le «panier» reste en pratique de taille suffisamment réduite pour n'avoir qu'un impact négligeable sur les performances, tout en permettant une bien meilleure montée en charge.

Le «panier» statique exposé dans l'article que nous avons mentionné a cependant l'inconvénient de ne pas éliminer totalement le cas où une clé ne peut être enregistrée, surtout s'il est de petite taille. Cela oblige parfois à effectuer un agrandissement prématuré de la table, qui reste une opération coûteuse. De grande taille, le panier rend ces redimensionnements forcés



inexistants au prix d'un certain gaspillage mémoire. Le panier dynamique nous est donc apparu la solution la plus performante.

Nous avons également cherché à optimiser l'accès aux données du «panier». Afin d'éviter de devoir faire une recherche linéaire pour retrouver un élément placé dans le «panier», nous avons eu recours à un «*chaining*» interne entre l'emplacement où la clé aurait dû être insérée, et sa position réelle dans le panier.

Notre première version du «panier» utilisait un tableau dynamique, mais nous avons déterminé expérimentalement qu'une liste chaînée apportait un gain de performances significatif à la fois lors de l'insertion et de la recherche grâce à la diminution du nombre d'allocations mémoire.

Ce changement a néanmoins un inconvénient : dans le cas pathologique où tous les emplacements possibles du tableau sont occupés et déjà liés à une place dans le «panier», la recherche linéaire plus lente ne peut être évitée. En pratique, cette situation est si rare que l'impact sur les performances est virtuellement nul.

2 Structure de la table de hachage

Notre première implémentation se présentait sous la forme d'un tableau de structures contenant :

- La taille codée sur 16 bits de la clé. Une taille nulle indique un emplacement supprimé.
- L'index lui aussi codé sur 16 bits d'un éventuel doublon dans le panier.
- Un pointeur sur un emplacement mémoire contenant la clé. Celle-ci est directement préfixée par un pointeur sur la valeur qui lui est associée.

Le panier était implémenté comme un tableau dynamique de ces mêmes structures, limité à 65535 éléments au maximum. En pratique, cette limite ne devrait pas être atteinte (le panier ne comporte généralement pas plus d'une dizaine d'éléments pour un million de couples clé-valeur).

Cette disposition mémoire permettait de tirer partie du cache processeur, et d'avoir une surcharge mémoire raisonnable de 32 bits par clé pour une occupation mémoire totale de 64 bits par emplacement (96 bits sur architecture 64 bits).

Après différents tests, nous avons modifié ce programme initial afin d'améliorer les performances.

La table de hachage a été remplacée par un simple tableau de pointeurs, réduisant ainsi la taille des emplacements. Les données ont été déplacées pour être stockées avec la clé dans un bloc mémoire dynamiquement alloué.

Le «panier» quant à lui a été changé en une liste chaînée, permettant ainsi une croissance illimitée. Le «*chaining*» fut réimplémenté en conséquence pour utiliser un pointeur au lieu d'un index.

Ces modifications ont amélioré les performances d'environ 20% en lecture et 10% en insertion, mais l'occupation mémoire excédentaire a augmenté de 32 bits par clé.

Afin de permettre une énumération aisée des éléments stockés dans notre table de hachage, nous avons ajouté un pointeur supplémentaire à la structure contenant les données. Ainsi, une liste chaînée de tous les éléments est construite au gré des insertions, permettant de les énumérer dans l'ordre suivant lequel ils ont été insérés.

Cela nous a permis d'optimiser du même coup le redimensionnement de la table, en ne parcourant que les éléments de la liste au lieu de la table entière. Grâce à cette modification, l'insertion est 20% plus rapide.



Troisième partie

Le défi du parallélisme

Afin de rendre l'accès à cette table parallélisable, nous lui avons ajouté un verrou de type lecteur/écrivain. Ce type de verrou autorise plusieurs processus à lire les données simultanément tant qu'aucun d'entre eux n'écrit dans la table.

Lorsqu'un processus écrit, tous les autres doivent attendre que l'insertion soit terminée avant de pouvoir accéder de nouveau à la table.

Cette solution est simple mais peu satisfaisante ; en effet, rien ne justifie de bloquer tous les processus lecteurs dans le cas où un processus écrivain modifie une clé à laquelle ils ne tentent pas d'accéder.

Nous avons donc cherché à améliorer la granularité de la synchronisation en modifiant la structure de la table.

1 Accès parallèles

Nous avons implémenté notre table de hachage dans le but de l'utiliser au sein d'un serveur, c'est à dire un environnement où toutes les ressources doivent être simultanément accessibles en parallèle par de nombreux processus.

Ce point est d'autant plus crucial que l'avènement des processeurs disposant de multiples cœurs permet des gains de performance très importants pour peu qu'on utilise des structures favorisant un parallélisme d'échelle.

Il n'était donc pas possible de se limiter au schéma de verrouillage primitif évoqué dans la partie précédente, car sa mauvaise granularité empêche sans légitimité les accès concurrents à des éléments indépendants.

Nous avons choisi de modifier la structure originale de notre table de hachage pour la diviser en segments, permettant d'obtenir une granularité de verrouillage plus fine de la synchronisation.

Le niveau le plus précis imposait d'attribuer un verrou à chaque emplacement du tableau, ce qui nous a paru bien trop coûteux en occupation mémoire.

La segmentation permet d'augmenter le nombre d'accès concurrents à moindre coût et d'ouvrir d'autres perspectives intéressantes que nous allons détailler.

2 Une table de hachage segmentée

Plutôt que de rejeter notre implémentation initiale et de tout réécrire, nous avons choisi de redéfinir ce que nous appelions précédemment table de hachage, **segment**, et de construire ce que nous appellerons dans la suite de ce document **table de hachage** au dessus de la structure existante.

La nouvelle table de hachage est donc une collection de segments indépendants ; cela pose *a priori* un nouveau problème : comment décider dans quel segment chaque clé doit être insérée ?

Il existe de nombreuses réponses à cette question, comme les filtres de Bloom ou les tables de hachage à multiples niveaux, qui nous ont paru d'une complexité indésirable dans ce contexte où les performances brutes sont primordiales.

Nous avons choisi la solution, plus simple, de composer la table de hachage d'un nombre fixe de 256 segments, et de distribuer les clés dans ces derniers suivant leur somme CRC8.

Le CRC8 est un algorithme de somme de contrôle permettant d'associer à toute clé un entier codé sur 8 bits, que nous utilisons ici comme index de segment. Le CRC8 étant un calcul extrêmement simple et rapide, cela permet de n'ajouter qu'un minimum de complexité à l'algorithme existant.



Il aurait été possible d'économiser de la mémoire en utilisant un nombre dynamique de segments, mais cette solution impose de réinsérer toutes les clés enregistrées à chaque ajout ou suppression d'un segment. En effet, changer le nombre des ces derniers modifie la distribution des clés.

Cette opération coûteuse verrouillerait de surcroît toute la table pendant son déroulement, allant à l'encontre de notre objectif de parallélisme. Le coût mémoire de la solution retenue est plus important mais reste constant. Nous allons voir que cette solution améliore également les performances générales de l'implémentation sous-jacente.

3 La dissolution des collisions

L'utilisation de segments nous a permis de favoriser les accès concurrents aux données de la table de hachage, mais présente également une propriété intéressante.

En distribuant les clés suivant leur CRC8, cet algorithme diminue la probabilité de collision dans chaque segment, en allégeant leur charge respective et en opérant une première répartition pseudo-aléatoire.

Il en résulte des temps d'insertion et d'accès plus rapides du fait de la réduction du nombre de collisions, «diluées» par la présélection à laquelle sont soumises les clés.

Au final, l'implémentation complète présente des temps d'accès plus avantageux en utilisant plusieurs processus parallèles, ce qui correspond à l'objectif que nous nous étions fixé. Dans le cas où la table ne doit être utilisée que par un unique processus, il reste toutefois plus avantageux d'utiliser un simple segment comme table de hachage afin de réduire l'empreinte mémoire.

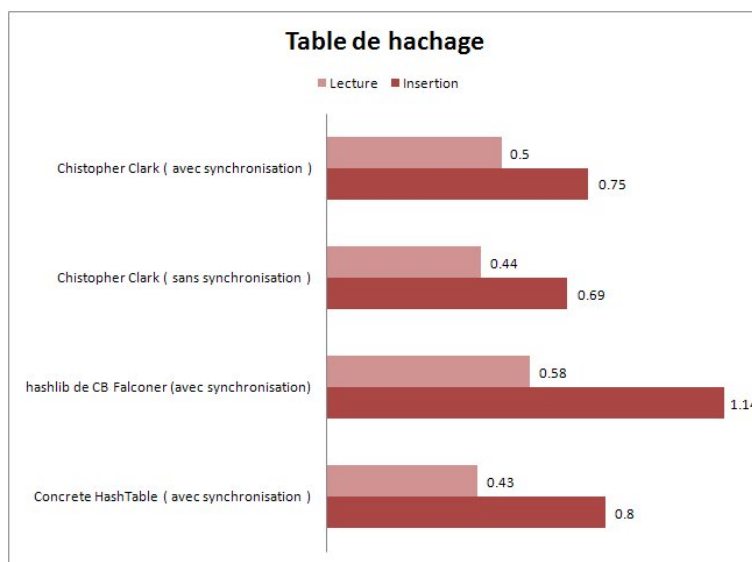
A Annexe : Quelques chiffres

Notre implémentation finale a des performances compétitives ; insérer 800 000 clés est effectué en 0.800 secondes, les relire en 0.430 secondes sur un Intel Core Duo 2GHz. Le remplacement de toutes les clés est un peu plus long, prenant 0.560 secondes. La suppression est réalisée en 0.470 secondes.

À titre de comparaison, la bibliothèque hashlib de C. B. Falconer modifiée afin d'être synchronisée, utilisant la même fonction de hachage et les mêmes données de test réalise l'insertion des 800 000 éléments en 1.140 secondes, la relecture en 0.500 secondes et le remplacement en 0.700 secondes.

Une autre implémentation de référence, celle de Christopher Clark, insère les 800 000 clés en 0.750 secondes et les retrouve en 0.500 secondes, avec synchronisation. Sans verrous, ses temps sont plus rapides : 0.690 secondes en insertion et 0.440 secondes en lecture.

Tous ces résultats ont été obtenus avec les mêmes options de compilation, la même fonction de hachage (lookup3 de Bob Jenkins) et le même échantillon de données.





B Annexe : Bibliographie

1. *Cuckoo Hashing*, Pagh, Rodler, 2001.
<http://citeseer.ist.psu.edu/pagh01cuckoo.html>
2. *More Robust Hashing : Cuckoo Hashing with a Stash*, Kirsch, Mitzenmacher, Wieder, 2008.
<http://research.microsoft.com/users/uwieder/papers/stash-full.pdf>
3. *Hash Functions and Block Ciphers*, Jenkins
<http://burtleburtle.net/bob/hash/>



C Annexe : Implémentation

```

1  /*****
2  * Concrete Server
3  * Copyright (c) 2005–2009 Princesse Avis Co., Ltd
4  *
5  * This library is free software; you can redistribute it and/or
6  * modify it under the terms of the GNU Lesser General Public
7  * License as published by the Free Software Foundation; either
8  * version 2.1 of the License, or (at your option) any later version.
9  *
10 * This library is distributed in the hope that it will be useful,
11 * but WITHOUT ANY WARRANTY; without even the implied warranty of
12 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
13 * Lesser General Public License for more details.
14 *
15 * You should have received a copy of the GNU Lesser General Public
16 * License along with this library; if not, write to the Free Software
17 * Foundation, Inc., 51 Franklin Street, 5th Floor, Boston, MA 02110–1301 USA
18 *
19 *****/
20
21 #ifndef M_HASHTABLE_H
22
23 #define M_HASHTABLE_H
24
25 #ifdef _ENABLE_HASHTABLE
26
27 #include "m_core_def.h"
28 #include "m_random.h"
29
30 /* required server build configuration */
31 #ifndef _ENABLE_RANDOM
32 #error "Concrete: the hashtable module requires the builtin PRNG."
33 #endif
34
35 #define CACHE_HASHFNCOUNT      5      /* number of hash functions to use */
36 #define CACHE_CUCKOORETRY       8      /* retries if the bucket is full */
37 #define CACHE_GROWTHRATIO       1.6    /* threshold to grow the table */
38
39 /** @defgroup cache module::cache */
40
41 typedef struct m_cache {
42     pthread_rwlock_t *_lock;
43     size_t _bucket_size;
44     size_t _bucket_count;
45     char **_idx_head;
46     char **_idx_tail;
47     char **_bucket;
48     char *_basket;
49     m_random *_hash_seed;
50     void (*_freeval)(void *);
51     uint32_t _seed[CACHE_HASHFNCOUNT];
52 } m_cache;
53
54 /**
55  * @ingroup cache
56  * @struct m_cache
57  *
58  * This structure supports the implementation of a thread safe hash table.
59  *
60  * Each key/value pair is stored in this structure with an overhead of
61  * sizeof(size_t) + sizeof(char *) bytes.
62  *
63  * @Note Even if this structure is thread safe, it is not recommended

```



```

64  * to use it in a context where concurrency is important, due to its
65  * simplistic locking scheme. For good concurrency, it is better to use
66  * the @ref m_hashtable structure and the related API.
67  *
68  */
69
70  typedef struct m_hashtable {
71      m_cache *_segment[256];
72  } m_hashtable;
73
74  /* ----- */
75
76  public m_cache *cache_alloc(void (*freeval)(void *));
77
78  /* ----- */
79
80  public inline void *cache_push(m_cache *h, const char *k, size_t l, void *v);
81
82  /* ----- */
83
84  public inline void *cache_add(m_cache *h, const char *k, size_t l, void *v);
85
86  /* ----- */
87
88  public void *cache_findexec(m_cache *h, const char *key, size_t len,
89                             void *(*function)(void *));
90
91  /* ----- */
92
93  public inline void *cache_find(m_cache *h, const char *key, size_t len);
94
95  /* ----- */
96
97  public void cache_foreach(m_cache *h,
98                           int (*function)(const char *, size_t, void *));
99
100 /* ----- */
101
102 public void *cache_pop(m_cache *h, const char *key, size_t len);
103
104 /* ----- */
105
106 public m_cache *cache_free(m_cache *h);
107
108 /* ----- */
109 /* Segmented hash table */
110 /* ----- */
111
112 public m_hashtable *hashtable_alloc(void (*freeval)(void *));
113
114 /* ----- */
115
116 public inline void *hashtable_insert(m_hashtable *h, const char *key,
117                                     size_t len, void *val);
118
119 /* ----- */
120
121 public inline void *hashtable_update(m_hashtable *h, const char *key,
122                                     size_t len, void *val);
123
124 /* ----- */
125
126 public inline void *hashtable_remove(m_hashtable *h, const char *key,
127                                     size_t len);
128

```



```
129 /* ----- */
130
131 public inline void *hashtable_findexec(m_hashtable *h, const char *key,
132                                       size_t len, void *(*function)(void *));
133
134 /* ----- */
135
136 public void hashtable_foreach(m_hashtable *h,
137                               int (*function)(const char *, size_t, void *));
138
139 /* ----- */
140
141 public inline void *hashtable_find(m_hashtable *h, const char *key, size_t len);
142
143 /* ----- */
144
145 public inline m_hashtable *hashtable_free(m_hashtable *h);
146
147 /* ----- */
148
149 /* _ENABLE_HASHTABLE */
150 #endif
151
152 #endif
```



```

1  /*****
2  *  Concrete Server
3  *  Copyright (c) 2005–2009 Princesse Avis Co., Ltd
4  *
5  *  This library is free software; you can redistribute it and/or
6  *  modify it under the terms of the GNU Lesser General Public
7  *  License as published by the Free Software Foundation; either
8  *  version 2.1 of the License, or (at your option) any later version.
9  *
10 *  This library is distributed in the hope that it will be useful,
11 *  but WITHOUT ANY WARRANTY; without even the implied warranty of
12 *  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
13 *  Lesser General Public License for more details.
14 *
15 *  You should have received a copy of the GNU Lesser General Public
16 *  License along with this library; if not, write to the Free Software
17 *  Foundation, Inc., 51 Franklin Street, 5th Floor, Boston, MA 02110–1301 USA
18 *
19 *****/
20
21 #include "m_hashtable.h"
22
23 /* _____ */
24 #ifdef _ENABLE_HASHTABLE
25 /* _____ */
26
27 /* crc8 lookup table (Maxim/Dallas 1 wire) */
28 static uint8_t _crc_lut[256] = {
29     0x00, 0x5e, 0xbc, 0xe2, 0x61, 0x3f, 0xdd, 0x83,
30     0xc2, 0x9c, 0x7e, 0x20, 0xa3, 0xfd, 0x1f, 0x41,
31     0x9d, 0xc3, 0x21, 0x7f, 0xfc, 0xa2, 0x40, 0x1e,
32     0x5f, 0x01, 0xe3, 0xbd, 0x3e, 0x60, 0x82, 0xdc,
33     0x23, 0x7d, 0x9f, 0xc1, 0x42, 0x1c, 0xfe, 0xa0,
34     0xe1, 0xbf, 0x5d, 0x03, 0x80, 0xde, 0x3c, 0x62,
35     0xbe, 0xe0, 0x02, 0x5c, 0xdf, 0x81, 0x63, 0x3d,
36     0x7c, 0x22, 0xc0, 0x9e, 0x1d, 0x43, 0xa1, 0xff,
37     0x46, 0x18, 0xfa, 0xa4, 0x27, 0x79, 0x9b, 0xc5,
38     0x84, 0xda, 0x38, 0x66, 0xe5, 0xbb, 0x59, 0x07,
39     0xdb, 0x85, 0x67, 0x39, 0xba, 0xe4, 0x06, 0x58,
40     0x19, 0x47, 0xa5, 0xfb, 0x78, 0x26, 0xc4, 0x9a,
41     0x65, 0x3b, 0xd9, 0x87, 0x04, 0x5a, 0xb8, 0xe6,
42     0xa7, 0xf9, 0x1b, 0x45, 0xc6, 0x98, 0x7a, 0x24,
43     0xf8, 0xa6, 0x44, 0x1a, 0x99, 0xc7, 0x25, 0x7b,
44     0x3a, 0x64, 0x86, 0xd8, 0x5b, 0x05, 0xe7, 0xb9,
45     0x8c, 0xd2, 0x30, 0x6e, 0xed, 0xb3, 0x51, 0x0f,
46     0x4e, 0x10, 0xf2, 0xac, 0x2f, 0x71, 0x93, 0xcd,
47     0x11, 0x4f, 0xad, 0xf3, 0x70, 0x2e, 0xcc, 0x92,
48     0xd3, 0x8d, 0x6f, 0x31, 0xb2, 0xec, 0x0e, 0x50,
49     0xaf, 0xf1, 0x13, 0x4d, 0xce, 0x90, 0x72, 0x2c,
50     0x6d, 0x33, 0xd1, 0x8f, 0x0c, 0x52, 0xb0, 0xee,
51     0x32, 0x6c, 0x8e, 0xd0, 0x53, 0x0d, 0xef, 0xb1,
52     0xf0, 0xae, 0x4c, 0x12, 0x91, 0xcf, 0x2d, 0x73,
53     0xca, 0x94, 0x76, 0x28, 0xab, 0xf5, 0x17, 0x49,
54     0x08, 0x56, 0xb4, 0xea, 0x69, 0x37, 0xd5, 0x8b,
55     0x57, 0x09, 0xeb, 0xb5, 0x36, 0x68, 0x8a, 0xd4,
56     0x95, 0xcb, 0x29, 0x77, 0xf4, 0xaa, 0x48, 0x16,
57     0xe9, 0xb7, 0x55, 0x0b, 0x88, 0xd6, 0x34, 0x6a,
58     0x2b, 0x75, 0x97, 0xc9, 0x4a, 0x14, 0xf6, 0xa8,
59     0x74, 0x2a, 0xc8, 0x96, 0x15, 0x4b, 0xa9, 0xf7,
60     0xb6, 0xe8, 0x0a, 0x54, 0xd7, 0x89, 0x6b, 0x35
61 };
62
63 #define _key(b) ((b) + sizeof(uint16_t))
64 #define _len(b) (*((uint16_t *) (b)))
65 /* if your arch requires aligned reads, you can modify this macro */

```



```

66 #define _pad(b)  (*((uint16_t *) (b)))
67 #define _ptr(b)  (*((char **) ((b) + sizeof(uint16_t) + _pad(b)))
68 #define _val(b)  \
69  ((void *) *((int **) ((b) + sizeof(uint16_t) + _pad(b) + sizeof(int *))))
70
71 #define _set_len(b, l)  \
72 do { *((uint16_t *) (b)) = (l); } while (0)
73
74 #define _set_key(b, k, l)  \
75 do { memcpy((b) + sizeof(uint16_t), (k), (l)); } while (0)
76
77 #define _set_ptr(b, p)  \
78 do { *((char **) ((b) + sizeof(uint16_t) + _pad(b))) = (p); } while (0)
79
80 #define _cpy_ptr(b0, b1)  \
81 do { \
82     *((char **) ((b0) + sizeof(uint16_t) + _pad(b0))) = \
83     *((char **) ((b1) + sizeof(uint16_t) + _pad(b1))); \
84 } while (0)
85
86 #define _clr_ptr(b)  \
87 do { *((char **) ((b) + sizeof(uint16_t) + _pad(b))) = NULL; } while (0)
88
89 #define _set_val(b, p)  \
90 do { \
91     *((int **) ((b) + sizeof(uint16_t) + _pad(b) + sizeof(int *))) = (p); \
92 } while (0)
93
94 /* ----- */
95
96 static inline uint32_t _hash(const char *data, size_t len, uint32_t initval)
97 {
98     /* Bob Jenkins' lookup3 hash function, we use it for the bucket index */
99     register uint32_t a, b, c;          /* internal state */
100     union { const void *ptr; size_t i; } u; /* needed for Mac Powerbook G4 */
101
102     #if (defined(__BYTE_ORDER) && defined(__LITTLE_ENDIAN) && \
103         __BYTE_ORDER == __LITTLE_ENDIAN) || \
104         (defined(i386) || defined(__i386__) || defined(__i486__) || \
105          defined(__i586__) || defined(__i686__) || defined(vax) || \
106          defined(MIPSEL))
107         #define HASH_LITTLE_ENDIAN 1
108         #define HASH_BIG_ENDIAN 0
109     #elif (defined(__BYTE_ORDER) && defined(__BIG_ENDIAN) && \
110           __BYTE_ORDER == __BIG_ENDIAN) || \
111           (defined(sparc) || defined(POWERPC) || defined(mc68000) || \
112            defined(sel))
113         #define HASH_LITTLE_ENDIAN 0
114         #define HASH_BIG_ENDIAN 1
115     #else
116         #define HASH_LITTLE_ENDIAN 0
117         #define HASH_BIG_ENDIAN 0
118     #endif
119
120     #define rot(x, k) (((x) << (k)) | ((x) >> (32 - (k))))
121
122     #define mix(a, b, c) \
123     { \
124         a -= c; a ^= rot(c, 4); c += b; \
125         b -= a; b ^= rot(a, 6); a += c; \
126         c -= b; c ^= rot(b, 8); b += a; \
127         a -= c; a ^= rot(c,16); c += b; \
128         b -= a; b ^= rot(a,19); a += c; \
129         c -= b; c ^= rot(b, 4); b += a; \
130     }

```



```

131
132 #define final(a, b, c) \
133 { \
134     c ^= b; c -= rot(b,14); \
135     a ^= c; a -= rot(c,11); \
136     b ^= a; b -= rot(a,25); \
137     c ^= b; c -= rot(b,16); \
138     a ^= c; a -= rot(c, 4); \
139     b ^= a; b -= rot(a,14); \
140     c ^= b; c -= rot(b,24); \
141 }
142
143 /* set up the internal state */
144 a = b = c = 0x52661314 + ((uint32_t) len) + initval; /* wo ai ni Lulu */
145
146 u.ptr = data;
147
148 if (HASH_LITTLE_ENDIAN && ((u.i & 0x3) == 0)) {
149     const uint32_t *k = (const uint32_t *) data; /* read 32-bit chunks */
150     #ifdef DEBUG
151     const uint8_t *k8;
152     #endif
153
154     /* all but last block: aligned reads and affect 32 bits of (a, b, c) */
155     while (len > 12) {
156         a += k[0]; b += k[1]; c += k[2];
157         mix(a, b, c);
158         len -= 12; k += 3;
159     }
160
161     /* - handle the last (probably partial) block
162     *
163     * "k[2]&0xffffffff" actually reads beyond the end of the string, but
164     * then masks off the part it's not allowed to read. Because the
165     * string is aligned, the masked-off tail is in the same word as the
166     * rest of the string. Every machine with memory protection I've seen
167     * does it on word boundaries, so is OK with this. But VALGRIND will
168     * still catch it and complain. The masking trick does make the hash
169     * noticeably faster for short strings (like English words).
170     */
171     #ifndef DEBUG
172
173     switch(len) {
174         case 12: c += k[2]; b += k[1]; a += k[0]; break;
175         case 11: c += k[2] & 0xffffffff; b += k[1]; a += k[0]; break;
176         case 10: c += k[2] & 0xffff; b += k[1]; a += k[0]; break;
177         case 9 : c += k[2] & 0xff; b += k[1]; a += k[0]; break;
178         case 8 : b += k[1]; a += k[0]; break;
179         case 7 : b += k[1] & 0xffffffff; a += k[0]; break;
180         case 6 : b += k[1] & 0xffff; a += k[0]; break;
181         case 5 : b += k[1] & 0xff; a += k[0]; break;
182         case 4 : a += k[0]; break;
183         case 3 : a += k[0] & 0xffffffff; break;
184         case 2 : a += k[0] & 0xffff; break;
185         case 1 : a += k[0] & 0xff; break;
186         case 0 : return c; /* zero length strings require no mixing */
187     }
188
189     #else /* make valgrind happy */
190
191     k8 = (const uint8_t *) k;
192     switch(len) {
193         case 12: c += k[2]; b += k[1]; a += k[0]; break;
194         case 11: c += ((uint32_t) k8[10]) << 16; /* fall through */
195         case 10: c += ((uint32_t) k8[9]) << 8; /* fall through */

```



```

196         case 9 : c += k8[8];                               /* fall through */
197         case 8 : b += k[1]; a += k[0]; break;
198         case 7 : b += ((uint32_t) k8[6]) << 16;          /* fall through */
199         case 6 : b += ((uint32_t) k8[5]) << 8;           /* fall through */
200         case 5 : b += k8[4];                               /* fall through */
201         case 4 : a += k[0]; break;
202         case 3 : a += ((uint32_t) k8[2]) << 16;          /* fall through */
203         case 2 : a += ((uint32_t) k8[1]) << 8;           /* fall through */
204         case 1 : a += k8[0]; break;
205         case 0 : return c;
206     }
207
208     #endif /* !valgrind */
209
210 } else if (HASH_LITTLE_ENDIAN && ((u.i & 0x1) == 0)) {
211     const uint16_t *k = (const uint16_t *) data; /* read 16-bit chunks */
212     const uint8_t *k8;
213
214     /* - all but last block: aligned reads and different mixing */
215     while (len > 12) {
216         a += k[0] + (((uint32_t) k[1]) << 16);
217         b += k[2] + (((uint32_t) k[3]) << 16);
218         c += k[4] + (((uint32_t) k[5]) << 16);
219         mix(a, b, c);
220         len -= 12; k += 6;
221     }
222
223     /* - handle the last (probably partial) block */
224     k8 = (const uint8_t *) k;
225     switch(len) {
226         case 12: c += k[4] + (((uint32_t) k[5]) << 16);
227                 b += k[2] + (((uint32_t) k[3]) << 16);
228                 a += k[0] + (((uint32_t) k[1]) << 16);
229                 break;
230         case 11: c += ((uint32_t) k8[10]) << 16;          /* fall through */
231         case 10: c += k[4];
232                 b += k[2] + (((uint32_t) k[3]) << 16);
233                 a += k[0] + (((uint32_t) k[1]) << 16);
234                 break;
235         case 9 : c += k8[8];                               /* fall through */
236         case 8 : b += k[2] + (((uint32_t) k[3]) << 16);
237                 a += k[0] + (((uint32_t) k[1]) << 16);
238                 break;
239         case 7 : b += ((uint32_t) k8[6]) << 16;          /* fall through */
240         case 6 : b += k[2];
241                 a += k[0] + (((uint32_t) k[1]) << 16);
242                 break;
243         case 5 : b += k8[4];                               /* fall through */
244         case 4 : a += k[0] + (((uint32_t) k[1]) << 16);
245                 break;
246         case 3 : a += ((uint32_t) k8[2]) << 16;          /* fall through */
247         case 2 : a += k[0];
248                 break;
249         case 1 : a += k8[0];
250                 break;
251         case 0 : return c; /* zero length requires no mixing */
252     }
253
254 } else if (HASH_BIG_ENDIAN && ((u.i & 0x3) == 0)) {
255     const uint32_t *k = (const uint32_t *) data; /* read 32-bit chunks */
256     #ifdef DEBUG
257     const uint8_t *k8;
258     #endif
259
260     /* all but last block: aligned reads and affect 32 bits of (a, b, c) */

```



```

261     while (len > 12) {
262         a += k[0]; b += k[1]; c += k[2];
263         mix(a, b, c);
264         len -= 12; k += 3;
265     }
266
267     /* - handle the last (probably partial) block */
268     #ifndef DEBUG
269
270     switch(len) {
271         case 12: c += k[2]; b += k[1]; a += k[0]; break;
272         case 11: c += k[2] & 0xfffff00; b += k[1]; a += k[0]; break;
273         case 10: c += k[2] & 0xffff0000; b += k[1]; a += k[0]; break;
274         case 9 : c += k[2] & 0xff000000; b += k[1]; a += k[0]; break;
275         case 8 : b += k[1]; a += k[0]; break;
276         case 7 : b += k[1] & 0xfffff00; a += k[0]; break;
277         case 6 : b += k[1] & 0xffff0000; a += k[0]; break;
278         case 5 : b += k[1] & 0xff000000; a += k[0]; break;
279         case 4 : a += k[0]; break;
280         case 3 : a += k[0] & 0xfffff00; break;
281         case 2 : a += k[0] & 0xffff0000; break;
282         case 1 : a += k[0] & 0xff000000; break;
283         case 0 : return c; /* zero length strings require no mixing */
284     }
285
286     #else /* make valgrind happy */
287
288     k8 = (const uint8_t *) k;
289     switch(len) {
290         case 12: c += k[2]; b += k[1]; a += k[0]; break;
291         case 11: c += ((uint32_t) k8[10]) << 8; /* fall through */
292         case 10: c += ((uint32_t) k8[9]) << 16; /* fall through */
293         case 9 : c += ((uint32_t) k8[8]) << 24; /* fall through */
294         case 8 : b += k[1]; a += k[0]; break;
295         case 7 : b += ((uint32_t) k8[6]) << 8; /* fall through */
296         case 6 : b += ((uint32_t) k8[5]) << 16; /* fall through */
297         case 5 : b += ((uint32_t) k8[4]) << 24; /* fall through */
298         case 4 : a += k[0]; break;
299         case 3 : a += ((uint32_t) k8[2]) << 8; /* fall through */
300         case 2 : a += ((uint32_t) k8[1]) << 16; /* fall through */
301         case 1 : a += ((uint32_t) k8[0]) << 24; break;
302         case 0 : return c;
303     }
304
305     #endif /* !VALGRIND */
306
307     } else { /* need to read the key one byte at a time */
308         const uint8_t *k = (const uint8_t *) data;
309
310         /* all but the last block: affect some 32 bits of (a, b, c) */
311         while (len > 12) {
312             a += k[0];
313             a += ((uint32_t) k[1]) << 8;
314             a += ((uint32_t) k[2]) << 16;
315             a += ((uint32_t) k[3]) << 24;
316             b += k[4];
317             b += ((uint32_t) k[5]) << 8;
318             b += ((uint32_t) k[6]) << 16;
319             b += ((uint32_t) k[7]) << 24;
320             c += k[8];
321             c += ((uint32_t) k[9]) << 8;
322             c += ((uint32_t) k[10]) << 16;
323             c += ((uint32_t) k[11]) << 24;
324             mix(a, b, c);
325             len -= 12; k += 12;

```




```

326     }
327
328     /* - last block: affect all 32 bits of (c) */
329     switch(len) {
330         case 12: c += ((uint32_t) k[11]) << 24;
331         case 11: c += ((uint32_t) k[10]) << 16;
332         case 10: c += ((uint32_t) k[9]) << 8;
333         case 9 : c += k[8];
334         case 8 : b += ((uint32_t) k[7]) << 24;
335         case 7 : b += ((uint32_t) k[6]) << 16;
336         case 6 : b += ((uint32_t) k[5]) << 8;
337         case 5 : b += k[4];
338         case 4 : a += ((uint32_t) k[3]) << 24;
339         case 3 : a += ((uint32_t) k[2]) << 16;
340         case 2 : a += ((uint32_t) k[1]) << 8;
341         case 1 : a += k[0]; break;
342         case 0 : return c;
343     }
344 }
345
346 final(a, b, c);
347
348 return c;
349 }
350
351 /* ----- */
352
353 static inline uint8_t _crc8(const char *string, size_t len)
354 {
355     register uint8_t crc = 0xff;
356     while (len --) crc = _crc_lut[*string ++ ^ crc];
357     return crc;
358 }
359
360 /* ----- */
361
362 static inline int _memcmp32(const char *a, const char *b, size_t len)
363 {
364     register unsigned int i = 0;
365
366     if (unlikely(len <= sizeof(uint32_t))) return memcmp(a, b, len);
367
368     for (i = 0; i < len - sizeof(uint32_t); i += sizeof(uint32_t)) {
369         if (*(uint32_t *) (a + i)) != *(uint32_t *) (b + i))
370             return -1;
371
372         if (i >= len - (i + sizeof(uint32_t)))
373             return 0;
374
375         if (*(uint32_t *) (a + len - (i + sizeof(uint32_t)))) !=
376             *(uint32_t *) (b + len - (i + sizeof(uint32_t))))
377             return -1;
378     }
379
380     return 0;
381 }
382
383 /* ----- */
384
385 static inline void *_cache_push(m_cache *h, char *key, int replace)
386 {
387     unsigned int i = 0, index = 0;
388     unsigned int retry = 0;
389     unsigned int hash_cache[CACHE_HASHFCOUNT];
390     char *tmp = NULL, *slot = NULL;

```



```

391     uint32_t mask = h->_bucket_size - 1;
392
393     _clr_ptr(key);
394
395     if (replace) {
396         /* look for the key in the buckets */
397         for (i = 0; i < CACHE_HASHFNCOUNT; i++) {
398             index = _hash(_key(key), _len(key), h->_seed[i]) & mask;
399
400             if ( (slot = h->_bucket[index]) ) {
401                 if (_len(key) == _len(slot)) {
402                     /* if keys are identical, replace the data and bail out */
403                     if (_memcmp32(_key(key), _key(slot), _len(key)) == 0) {
404                         switch (replace) {
405                             case 1: /* replace old values with new ones */
406                                 tmp = _val(slot); _set_val(slot, _val(key));
407                                 free(key - (2 * sizeof(char *)));
408                                 return tmp;
409                             case -1: /* keep old values and reject new ones */
410                                 tmp = _val(key);
411                                 free(key - (2 * sizeof(char *)));
412                                 return tmp;
413                         }
414                     }
415                 }
416             } else {
417                 /* got a free slot */
418                 h->_bucket[index] = key;
419                 h->_bucket_count++;
420                 return NULL;
421             }
422         }
423
424         /* couldn't find it, look in the basket */
425         for (slot = h->_basket; slot; slot = _ptr(slot)) {
426             if (_len(key) == _len(slot)) {
427                 if (_memcmp32(_key(key), _key(slot), _len(key)) == 0) {
428                     switch (replace) {
429                         case 1: /* replace old values with new ones */
430                             tmp = _val(slot); _set_val(slot, _val(key));
431                             free(key - (2 * sizeof(char *)));
432                             return tmp;
433                         case -1: /* keep old values and reject new ones */
434                             tmp = _val(key);
435                             free(key - (2 * sizeof(char *)));
436                             return tmp;
437                     }
438                 }
439             }
440         }
441     } else {
442
443         /* no replacement, look directly for a free slot */
444         for (i = 0; i < CACHE_HASHFNCOUNT; i++) {
445             index = _hash(_key(key), _len(key), h->_seed[i]) & mask;
446             if (!h->_bucket[index]) {
447                 /* this slot is free, we can insert */
448                 h->_bucket[index] = key;
449                 h->_bucket_count++;
450                 return NULL;
451             }
452         }
453     }
454 }
455

```



```

456     /* we did not find a free slot, try cuckoo hashing */
457     do {
458         /* get data off the last slot and replace them */
459         tmp = h->_bucket[index]; h->_bucket[index] = key; key = tmp;
460
461         /* get rid of tombstones */
462         if (! _len(key)) { /*free(key);*/ return NULL; }
463
464         /* keep the sibling */
465         _cpy_ptr(h->_bucket[index], key); _clr_ptr(key);
466
467         for (i = 0; i < CACHE_HASHFNCOUNT; i++) {
468             index = _hash(_key(key), _len(key), h->_seed[i]) & mask;
469
470             if (! h->_bucket[index]) {
471                 /* this slot is free, we can insert */
472                 h->_bucket[index] = key;
473                 h->_bucket_count++;
474                 return NULL;
475             }
476
477             hash_cache[i] = index;
478         }
479     } while (retry++ < CACHE_CUCKOORETRY);
480
481     /* cuckoo hashing did not help, store the key in the basket... */
482     _set_ptr(key, h->_basket); h->_basket = key;
483
484     /* try to get a sibling */
485     for (i = 0; i < CACHE_HASHFNCOUNT; i++) {
486         index = hash_cache[i];
487         if (_len(h->_bucket[index]) && ! _ptr(h->_bucket[index])) {
488             _set_ptr(h->_bucket[index], key);
489             break;
490         }
491     }
492 }
493
494 return NULL;
495 }
496
497 /* ----- */
498
499 static inline int _cache_resize(m_cache *h, uint32_t size)
500 {
501     unsigned int i = 0;
502     char **bucket = NULL, **prev = NULL, **next = NULL;
503     char *tmp = NULL;
504
505     if (unlikely(! h)) {
506         debug("_cache_resize(): bad parameters.\n");
507         return -1;
508     }
509
510     /* round the size to the next highest power of 2 */
511     size--; size |= size >> 1; size |= size >> 2;
512     size |= size >> 4; size |= size >> 8; size |= size >> 16;
513     size++; size += (size == 0);
514
515     if (size == h->_bucket_size) return 0;
516
517     free(h->_bucket);
518
519     /* try to resize the bucket array */
520     if (! (h->_bucket = calloc(size, sizeof(*bucket))) ) {

```



```

521     perror(ERR(_cache_resize, calloc));
522     return -1;
523 }
524
525 /* add the new hash functions */
526 for (i = 0; i < CACHE_HASHFCOUNT; i++)
527     h->_seed[i] = random_uint32(h->_hash_seed);
528
529 /* update the state */
530 h->_bucket_size = size; h->_bucket_count = 0; h->_basket = NULL;
531
532 /* rehash old buckets */
533 for (bucket = h->_idx_head; bucket; bucket = next) {
534     tmp = ((char *) bucket) + (2 * sizeof(char *));
535     next = (char **) bucket[1];
536
537     if (_len(tmp)) { _cache_push(h, tmp, 0); continue; }
538
539     prev = (char **) bucket[0];
540     if (prev) prev[1] = (char *) next; else h->_idx_head = next;
541     if (next) next[0] = (char *) prev; else h->_idx_tail = prev;
542     free(bucket);
543 }
544
545 return 0;
546 }
547
548 /* ----- */
549
550 public m_cache *cache_alloc(void (*freeval)(void *))
551 {
552     m_cache *h = malloc(sizeof(*h));
553
554     if (! h) {
555         perror(ERR(cache_alloc, malloc));
556         return NULL;
557     }
558
559     if (! (h->_lock = malloc(sizeof(*h->_lock))) ) {
560         perror(ERR(cache_alloc, malloc));
561         goto _err_lock;
562     }
563
564     if (pthread_rwlock_init(h->_lock, NULL) == -1) {
565         perror(ERR(cache_alloc, pthread_rwlock_init));
566         goto _err_init;
567     }
568
569     if (! (h->_hash_seed = random_init()) )
570         goto _err_init;
571
572     h->_bucket = NULL;
573     h->_bucket_count = h->_bucket_size = 0;
574     h->_basket = NULL;
575     h->_idx_head = h->_idx_tail = NULL;
576     h->_freeval = freeval;
577
578     if (_cache_resize(h, 2) == -1) {
579         debug("cache_alloc(): could not resize the hash table.\n");
580         goto _err_size;
581     }
582
583     return h;
584
585 _err_size:

```



```

586     free(h->_bucket);
587     free(h->_basket);
588     h->_hash_seed = random_fini(h->_hash_seed);
589 _err_init:
590     free(h->_lock);
591 _err_lock:
592     free(h);
593
594     return NULL;
595 }
596
597 /* ----- */
598
599 static void *_cache_add(m_cache *h, const char *key, size_t len, void *val,
600                        int replace)
601 {
602     char *mkey = NULL;
603     char **keyptr = NULL;
604
605     if (unlikely(! h || ! key || ! len)) {
606         debug("_cache_add(): bad parameters.\n");
607         return val;
608     }
609
610     /* replace the key by a dynamically allocated one */
611     if (! (keyptr = malloc(sizeof(uint16_t) + (4 * sizeof(int *)) + len)) ) {
612         perror(ERR(cache_push, malloc));
613         return val;
614     }
615
616     mkey = ((char *) keyptr) + (2 * sizeof(char *));
617
618     _set_len(mkey, len); _set_key(mkey, key, len); _set_val(mkey, val);
619
620     pthread_rwlock_wrlock(h->_lock);
621
622     /* expand the hashtable if the load is too important */
623     if (h->_bucket_count * CACHE_GROWTHRATIO >= h->_bucket_size)
624         _cache_resize(h, h->_bucket_count * CACHE_GROWTHRATIO);
625
626     if (! (val = _cache_push(h, mkey, replace)) ) {
627         if (h->_idx_tail) {
628             keyptr[0] = (char *) h->_idx_tail; keyptr[1] = NULL;
629             h->_idx_tail[1] = (char *) keyptr; h->_idx_tail = keyptr;
630         } else {
631             keyptr[0] = keyptr[1] = NULL;
632             h->_idx_head = h->_idx_tail = keyptr;
633         }
634     }
635
636     pthread_rwlock_unlock(h->_lock);
637
638     return val;
639 }
640
641 /* ----- */
642
643 public inline void *cache_push(m_cache *h, const char *k, size_t l, void *v)
644 {
645     return _cache_add(h, k, l, v, 1);
646 }
647
648 /* ----- */
649
650 public inline void *cache_add(m_cache *h, const char *k, size_t l, void *v)

```



```

651 {
652     return _cache_add(h, k, l, v, -1);
653 }
654
655 /* ----- */
656
657 public void *cache_findexec(m_cache *h, const char *key, size_t len,
658                             void *(*function)(void *))
659 {
660     register unsigned int i = 0, j = 0;
661     const char *ptr = NULL;
662     void *res = NULL;
663     uint32_t mask = 0;
664
665     if (unlikely(! h || ! key || ! len)) {
666         debug("cache_findexec(): bad parameters.\n");
667         return NULL;
668     }
669
670     pthread_rwlock_rdlock(h->_lock);
671
672     mask = h->_bucket_size - 1;
673
674     for (i = 0; i < CACHE_HASHFCOUNT; i++) {
675         j = _hash(key, len, h->_seed[i]) & mask;
676
677         if (! h->_bucket[j] || ! _len(h->_bucket[j])) continue;
678
679         if (_len(h->_bucket[j]) == len) {
680             if (_memcmp32(_key(h->_bucket[j]), key, len) == 0) {
681                 res = _val(h->_bucket[j]);
682                 if (function) res = function(res);
683                 pthread_rwlock_unlock(h->_lock);
684                 return res;
685             }
686         }
687
688         if (! (ptr = _ptr(h->_bucket[j])) )
689             continue;
690
691         if (_len(ptr) == len) {
692             if (_memcmp32(_key(ptr), key, len) == 0) {
693                 res = _val(ptr);
694                 if (function) res = function(res);
695                 pthread_rwlock_unlock(h->_lock);
696                 return res;
697             }
698         }
699     }
700
701     /* unlucky, scan the basket for orphan keys */
702     for (ptr = h->_basket; ptr; ptr = _ptr(ptr)) {
703         if (_len(ptr) == len) {
704             if (_memcmp32(_key(ptr), key, len) == 0) {
705                 res = _val(ptr);
706                 if (function) res = function(res);
707                 pthread_rwlock_unlock(h->_lock);
708                 return res;
709             }
710         }
711     }
712
713     pthread_rwlock_unlock(h->_lock);
714
715     return NULL;

```



```

716 }
717
718 /* ----- */
719
720 public inline void *cache_find(m_cache *h, const char *key, size_t len)
721 {
722     return cache_findexec(h, key, len, NULL);
723 }
724
725 /* ----- */
726
727 public void cache_foreach(m_cache *h,
728                          int (*function)(const char *, size_t, void *))
729 {
730     char **bucket = NULL, **next = NULL;
731     char *tmp = NULL;
732
733     if (! h || ! function) {
734         debug("cache_foreach(): bad parameters.\n");
735         return;
736     }
737
738     pthread_rwlock_wrlock(h->_lock);
739
740     for (bucket = h->_idx_head; bucket; bucket = next) {
741         tmp = ((char *) bucket) + (2 * sizeof(char *));
742         next = (char **) bucket[1];
743
744         if (_len(tmp)) {
745             if (function(_key(tmp), _len(tmp), _val(tmp)) == -1) {
746                 /* delete the record */
747                 _set_len(tmp, 0);
748             }
749         }
750     }
751
752     pthread_rwlock_unlock(h->_lock);
753
754     return;
755 }
756
757 /* ----- */
758
759 public void *cache_pop(m_cache *h, const char *key, size_t len)
760 {
761     register unsigned int i = 0, j = 0;
762     char *ptr = NULL, *tmp = NULL, *prev = NULL;
763     void *result = NULL;
764     uint32_t mask = 0;
765
766     if (unlikely(! h || ! key || ! len)) {
767         debug("cache_pop(): bad parameters.\n");
768         return NULL;
769     }
770
771     pthread_rwlock_wrlock(h->_lock);
772
773     mask = h->_bucket_size - 1;
774
775     for (i = 0; i < CACHE_HASHFNCCOUNT; i++) {
776         j = _hash(key, len, h->_seed[i]) & mask;
777
778         if (! h->_bucket[j] || !_len(h->_bucket[j])) continue;
779
780         if (_len(h->_bucket[j]) == len) {

```



```

821         if (_memcmp32(_key(h->_bucket[j]), key, len) == 0) {
822             /* remove from the bucket */
823             result = _val(h->_bucket[j]); ptr = _ptr(h->_bucket[j]);
824             /* a length of 0 indicates a tombstone */
825             _set_len(h->_bucket[j], 0);
826
827             if (ptr) {
828                 for (tmp = h->_basket; tmp; tmp = _ptr(tmp)) {
829                     if (_ptr(tmp) == ptr) {
830                         _set_ptr(tmp, _ptr(ptr)); _clr_ptr(ptr);
831                         h->_bucket[j] = ptr;
832                         break;
833                     }
834                 }
835             }
836
837             pthread_rwlock_unlock(h->_lock);
838             return result;
839         }
840     }
841 }
842
843
844 /* unlucky, scan the basket for orphan keys */
845 for (tmp = prev = h->_basket; tmp; prev = tmp, tmp = _ptr(tmp)) {
846     if (_len(tmp) == len) {
847         if (_memcmp32(_key(tmp), key, len) == 0) {
848             /* just remove the orphan sibling from the basket */
849             result = _val(tmp);
850             if (tmp == h->_basket) h->_basket = _ptr(tmp);
851             else { _set_ptr(prev, _ptr(tmp)); _set_len(tmp, 0); /*free(tmp);
852                    */ }
853
854             pthread_rwlock_unlock(h->_lock);
855
856             return result;
857         }
858     }
859 }

```




```

845     _cache_resize(h, h->_bucket_count * CACHE_GROWTHRATIO);
846
847     pthread_rwlock_unlock(h->_lock);
848
849     return NULL;
850 }
851
852 /* ----- */
853
854 public m_cache *cache_free(m_cache *h)
855 {
856     char **bucket = NULL, **next = NULL;
857     char *keyptr = NULL;
858
859     if (! h) return NULL;
860
861     for (bucket = h->_idx_head; bucket; bucket = next) {
862         next = (char **) bucket[1];
863         keyptr = ((char *) bucket) + (2 * sizeof(char *));
864         if (h->_freeval && _len(keyptr)) h->_freeval(_val(keyptr));
865         free(bucket);
866     }
867
868     free(h->_bucket);
869     random_fini(h->_hash_seed);
870     pthread_rwlock_destroy(h->_lock);
871     free(h->_lock); free(h);
872
873     return NULL;
874 }
875
876 /* ----- */
877
878 public m_hashtable *hashtable_alloc(void (*freeval)(void *))
879 {
880     unsigned int i = 0;
881
882     m_hashtable *h = malloc(sizeof(*h));
883
884     if (! h) {
885         perror(ERR(hash_alloc, malloc));
886         return NULL;
887     }
888
889     for (i = 0; i < 256; i++) h->_segment[i] = cache_alloc(freeval);
890
891     return h;
892 }
893
894 /* ----- */
895
896 public inline void *hashtable_insert(m_hashtable *h, const char *key,
897                                     size_t len, void *val)
898 {
899     if (! key || ! len) return NULL;
900
901     if (! h) return val;
902
903     return cache_add(h->_segment[_crc8(key, len)], key, len, val);
904 }
905
906 /* ----- */
907
908 public inline void *hashtable_update(m_hashtable *h, const char *key,
909                                     size_t len, void *val)

```



```

910 {
911     if (! key || ! len) return NULL;
912
913     if (! h) return val;
914
915     return cache_push(h->_segment[_crc8(key, len)], key, len, val);
916 }
917
918 /* ----- */
919
920 public inline void *hashtable_remove(m_hashtable *h, const char *key, size_t len)
921 {
922     if (! h || ! key || ! len) return NULL;
923
924     return cache_pop(h->_segment[_crc8(key, len)], key, len);
925 }
926
927 /* ----- */
928
929 public inline void *hashtable_findexec(m_hashtable *h, const char *key,
930                                       size_t len, void *(*function)(void *))
931 {
932     if (! h || ! key || ! len) return NULL;
933
934     return cache_findexec(h->_segment[_crc8(key, len)], key, len, function);
935 }
936
937 /* ----- */
938
939 public void hashtable_foreach(m_hashtable *h,
940                              int (*function)(const char *, size_t, void *))
941 {
942     unsigned int i = 0;
943
944     if (! h || ! function) return;
945
946     for (i = 0; i < 256; i++) {
947         if (h->_segment[i]->_idx_head)
948             cache_foreach(h->_segment[i], function);
949     }
950
951     return;
952 }
953
954 /* ----- */
955
956 public inline void *hashtable_find(m_hashtable *h, const char *key, size_t len)
957 {
958     if (! h || ! key || ! len) return NULL;
959
960     return cache_find(h->_segment[_crc8(key, len)], key, len);
961 }
962
963 /* ----- */
964
965 public inline m_hashtable *hashtable_free(m_hashtable *h)
966 {
967     unsigned int i = 0;
968
969     if (! h) return NULL;
970
971     for (i = 0; i < 256; i++) cache_free(h->_segment[i]); free(h);
972
973     return NULL;
974 }

```



```
975 |
976 | /* _____ */
977 | #else
978 | /* _____ */
979 |
980 | /* Hashtable support will not be compiled in the Concrete Library */
981 | __attribute__((unused)) static int __dummy__ = 0;
982 |
983 | /* _____ */
984 | #endif
985 | /* _____ */
```