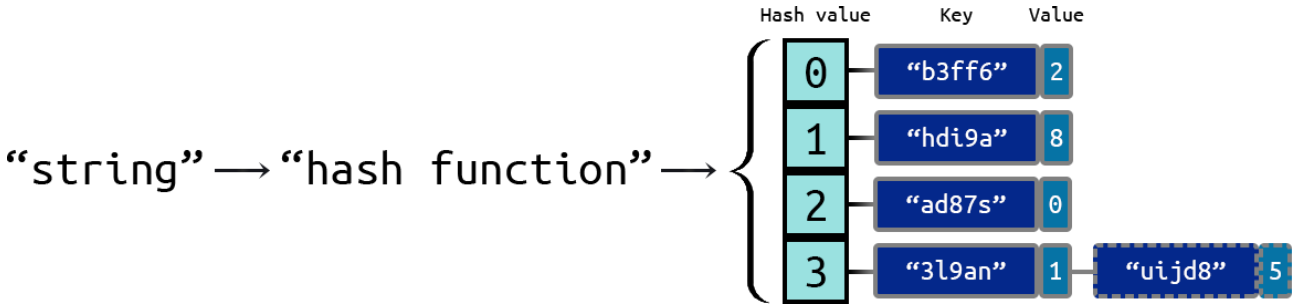

STRING HASH TABLE

INTEGRAZIONE 3 CFU: REALIZZAZIONE HASH TABLE PER STRINGHE

A cura di: *Andrea Fortunato*

Università degli Studi di Roma "Tor Vergata"
Ingegneria degli Algoritmi
A.A. 2021/2022

SOMMARIO

INTRODUZIONE

1. SCOPO DEL DOCUMENTO

Il seguente documento ha lo scopo di illustrare una descrizione della struttura dati sviluppata, analizzandone scelte progettuali, implementazione e test effettuati.

2. AMBIENTE DI SVILUPPO

L'applicazione String Hash Table è stata sviluppata e testata su ambiente Unix/Linux ma può essere compilata ed eseguita anche su Microsoft Windows, poiché non utilizza librerie "posix-only" (come può essere "pthread.h").

3. ARCHITETTURA DELLA STRUTTURA HASH TABLE

L'applicazione implementa una struttura dati che rappresenta un hash table per il mantenimento di dati formati da stringhe lunghe esattamente 64 caratteri (chiavi) e valori numerici associati ad esse.

Si può pensare, ad esempio, di voler memorizzare quante volte un determinato file viene visualizzato/aggiornato: si trasforma il nome del file nel corrispettivo hash SHA-256 (es. "algoritmi.pdf" --> "c5ef8a9301f5103b345f97ae55bd8db9401944b518e2fd709f4d234cfc19e7dc") e si passa quest'ultimo all'applicazione che, tramite una funzione di hash, è in grado di indicizzare la stringa in tempo costante.

La gestione delle collisioni viene effettuata tramite chaining-list, o liste di trabocco. Di seguito, viene riportato un esempio (per semplicità, le stringhe non saranno lunghe 64 caratteri):

```
hashtable[hash] --> (key1, value1) -> ... -> (keyn, valuen) -> NULL
hashtable[0]    --> ("3126c784", 52) -> NULL
hashtable[1]    --> NULL
hashtable[2]    --> NULL
hashtable[3]    --> ("364b039d", 0) -> NULL
hashtable[4]    --> ("d84562b1", 78) -> ("a82f963e", 63) -> NULL
hashtable[5]    --> ("4621f712", 11) -> NULL
hashtable[6]    --> NULL
hashtable[7]    --> NULL
hashtable[8]    --> NULL
hashtable[9]    --> ("af425227", 92) -> NULL
```

In questo esempio, le due chiavi "`d84562b1`" e "`a82f963e`" collidono.

IMPLEMENTAZIONE

1. STRUTTURA DATI PRINCIPALE: HASH TABLE

La struttura dati che rappresenta l'hash table è la seguente

```
typedef struct hashtable_t {
    unsigned int size;
    unsigned int different_entries;
    unsigned int collisions;

    struct hashtable_entry_t** table;
} hashtable;
```

ed è composta da 4 campi:

- size: dimensione dell'array che conterrà tutte le coppie (chiave, valore);
- different_entries: contatore che viene
  - incrementato quando si effettua l'inserimento di una chiave associata ad un valore hash non ancora presente nell'hash table;
  - decrementato quando si effettua la cancellazione dell'unica chiave associata ad un valore hash;
- collisions: contatore che viene
  - incrementato quando si effettua l'inserimento di una chiave associata ad un valore hash già presente nell'hash table e si verifica quindi una collisione;
  - decrementato quando si effettua la cancellazione di una chiave associata ad un valore hash che presenta almeno due chiavi e si elimina quindi una collisione;
- table: array di liste collegate contenenti nodi di tipo (chiave, valore). Ogni indice dell'array rappresenta un valore hash.

2. CALCOLO DEL VALORE HASH

Il primo passo, necessario per poter utilizzare l'hash table in questione, è quello di calcolare un valore hash associato alla chiave da inserire, aggiornare, cercare oppure cancellare.

In questo caso, per il calcolo del valore hash, è stato utilizzato un modo semplice ed efficace che permetta di convertire una stringa in un numero, il quale rappresenterà il valore hash cercato.

Di seguito viene mostrato lo pseudo codice dell'algoritmo:

```
Algorithm: GetHash
Input: hash table size (integer), key (string);

hash ← 0
for character ← key[0] to key[lenght(key)-1] do
    hash ← int(character) + hash * 33
    hash ← hash mod (hash table size)

Return hash
```

La funzione "`int(character)`" converte un carattere nel corrispettivo valore intero ASCII (es. 'A'-->65, 'a'-->97, '3'-->51, ecc...).

La costante "`33`" è stata scelta poiché corrisponde a 31, numero primo di Mersenne pari a 2^5-1, più 2 , numero primo che rende più efficiente il calcolo dal punto di vista dell'unicità.

Il passaggio "`hash ← hash mod (hash table size)`" è necessario poiché il valore dell'hash non può superare la dimensione della tabella hash, altrimenti si andrebbe ad accedere ad una zona di memoria non riservata a quest'ultima.

Per completezza, di seguito viene riportato anche il corrispondente codice C:

```c
unsigned int hashtable_gethash(unsigned int hashtable_size, char* key) {
    unsigned int hash = 0;

    for (char* ch = key; *ch != '\0'; ch++) {
        hash = ((int)(*ch) + (hash << 5) + hash) % hashtable_size;
    }

    return hash;
}
```

Per ottimizzare la velocità ed evitare di fare moltiplicazioni, viene utilizzata l'operazione "`(hash << 5) + hash)`" che corrisponde a "`hash * 32 + hash == hash * 33`", poiché "`hash << 5`" shifta a sinistra `hash` di 5 bit e ciò equivale a moltiplicarlo per 2^5=32.

3. INSERIMENTO, RICERCA ED ELIMINAZIONE

Le tre operazioni sono prettamente molto simili e si basano tutte sullo stesso concetto: viene calcolato il valore hash relativo alla chiave passata e viene

indicizzata, nella tabella hash, la cella con indice pari al valore hash appena calcolato. A questo punto, si verifica se la chiave è presente nella cella:

- presente:
  - inserimento: viene aggiornato il campo "valore" con quello appena passato;
  - ricerca: viene restituito il campo "valore";
  - eliminazione: viene eliminata la coppia (chiave, valore) e si cambia il valore della cella con la prossima coppia che segue quella appena eliminata (se presente, altrimenti NULL);
- non presente: si prosegue con lo scorrimento della lista di trabocco e si verifica, coppia per coppia, se la chiave analizzata corrisponde con quella passata. In caso venisse trovata si prosegue come al punto precedente, altrimenti si procede nel seguente modo:
  - inserimento: si crea la coppia (chiave, valore) e la si associa alla cella indicizzata se quest'ultima fosse NULL, altrimenti si aggiunge in coda alla lista di trabocco;
  - ricerca: la ricerca risulta infruttuosa;
  - eliminazione: non viene effettuata alcuna eliminazione.

MANUALE D'USO

1. COMPILAZIONE ED ESECUZIONE

Nella cartella sono presenti tre file:

- *stringhashtable.c*: file contenente il codice sorgente, opportunamente commentato, dell'intera implementazione;
- *Makefile*: file che facilita la compilazione del sorgente, utilizzando entrambi i flags "-Wall" e "-Wextra" per massimizzare il livello di controllo degli errori. È sufficiente digitare il comando

  `user@host:/directory/ $ make`

  per generare l'eseguibile `/directory/stringhashtable` lanciabile tramite il comando

  `user@host:/directory/ $ ./stringhashtable`
- *rnd_str.txt*: file contenente esattamente 100.000 stringhe randomiche di 64 caratteri tutte diverse l'una dall'altra.

Una volta avviato, verrà mostrato un semplice menù di scelta. Di seguito, vengono descritte le tre opzioni presenti:

1) Viene lanciata un'esecuzione utilizzando 12 stringhe differenti, ognuna lunga 10 caratteri. In ordine, vengono svolti i seguenti passaggi:
   a. Viene creata una tabella hash di dimensione pari a 16;
   b. Vengono inserite le 12 stringhe;
   c. Vengono eliminate 4 stringhe;
   d. Vengono modificati i valori associati a 3 stringhe.

   Ad ogni singolo step, viene inoltre stampata l'intera tabella hash in maniera facilmente leggibile, la quale evidenzia in particolare le liste di trabocco per le collisioni verificatesi.

2) Viene lanciata un'esecuzione che legge dal file "rnd_str.txt" riga per riga ed inserisce la stringa letta all'interno della tabella hash. Alla fine, viene stampata l'intera tabella come nel caso numero 1).
3) Permette di uscire dall'esecuzione.

PRESTAZIONI E COMMENTI FINALI

Il calcolo del valore hash richiede la lettura di ogni carattere della stringa e di conseguenza sarà sempre costante $\boldsymbol{O(64) = O(1)}$.

Una volta calcolato l'hash associato alla chiave, le tre operazioni principali, ovvero inserimento/aggiornamento, ricerca ed eliminazione, possono essere svolte in tempo costante $\boldsymbol{O(1)}$ grazie all'indicizzazione "`hashtable[hash]`".

Questo vale nel caso in cui la tabella hash sia ben dimensionata e ci siano poche collisioni: se per assurdo provassi ad inserire 10 chiavi in una tabella hash di dimensione 2, avrei almeno 8 collisioni ed ogni inserimento/aggiornamento, ricerca o eliminazione richiederebbe di scorrere una lunga lista di trabocco, arrivando a costare anche circa $O\left(\frac{n}{2}\right) = O(n)$, con n pari al numero di elementi presenti nella tabella hash.

RIFERIMENTI

1. Slide del Professore Filippone Salvatore, Tabelle Hash: https://uniroma2.sharepoint.com/:b:/s/FILIPPONE-8039871-INGEGNERIA_DEGLI_ALGORITMI/EYxP59XRm0pDtbpePtlqEq0BspJvI3U3cIBXGxzAQAM0ug?e=KjyVB1
2. Slide del Professore Esposito Matteo, Tabelle Hash: https://uniroma2.sharepoint.com/:b:/s/FILIPPONE-8039871-INGEGNERIA_DEGLI_ALGORITMI/EfrSpQx5495GllTyHOhpxRwBwPs3w8G781kmoIkiTrWCXw?e=AGTwBy
3. Hashing by Steven J. Zei, Hashing 101: the Fundamentals: https://www.cs.odu.edu/~zeil/cs361/latest/Public/hash/