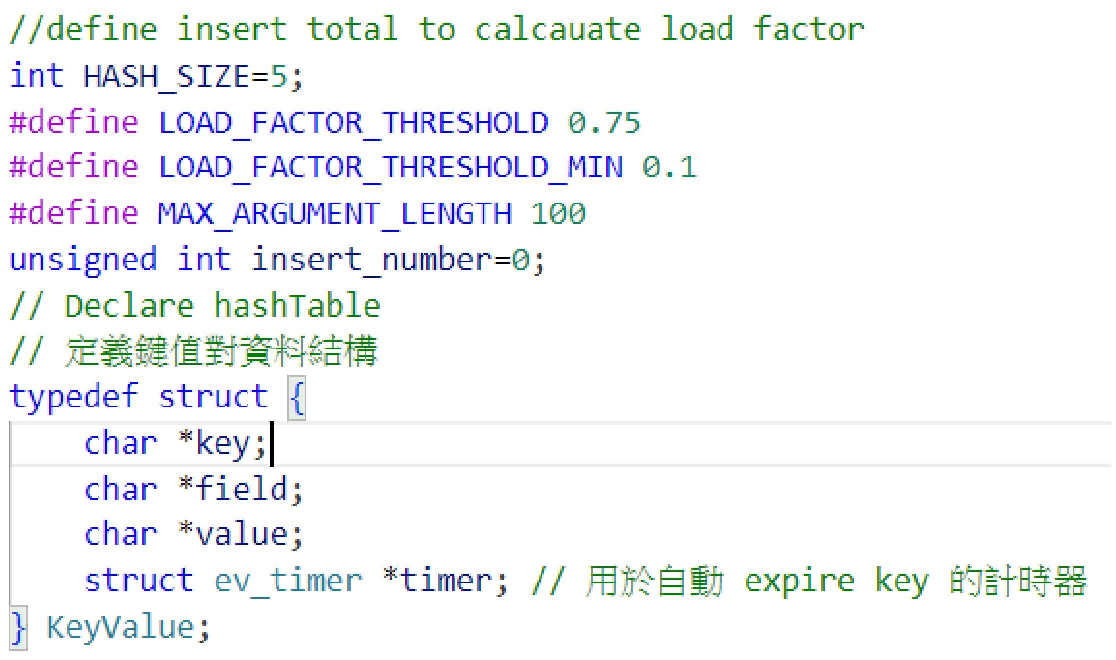
HW6

EE3B 110305504 黃聖倫

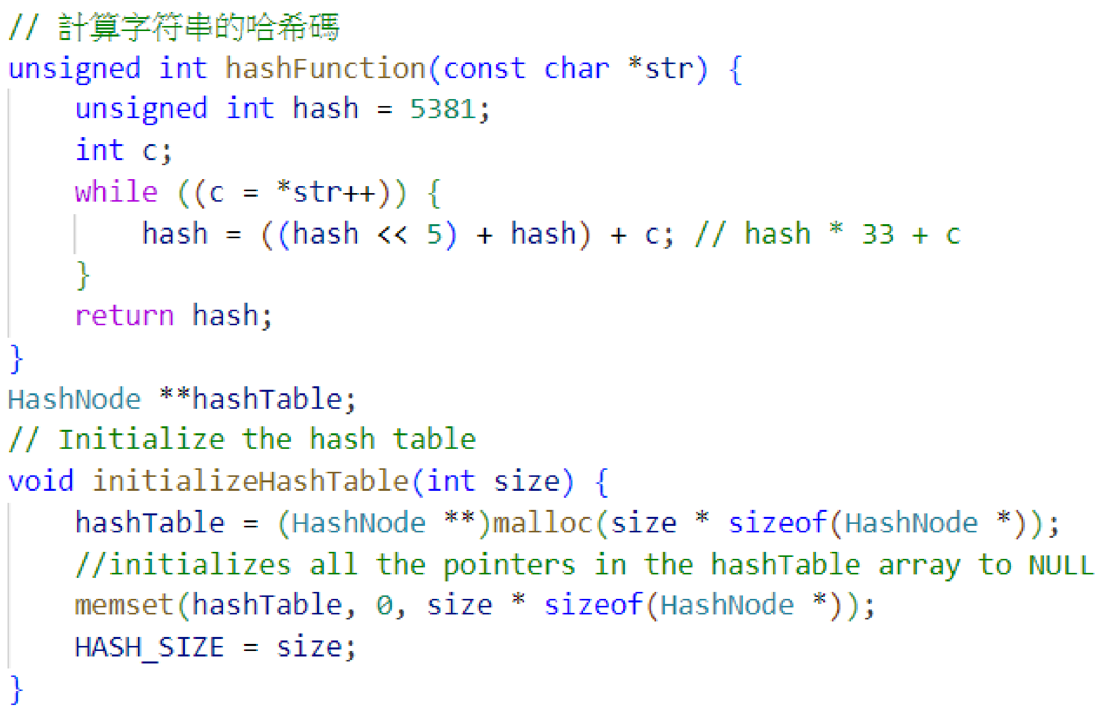
1.程式架構



```c
//define insert total to calcauate load factor
int HASH_SIZE=5;
#define LOAD_FACTOR_THRESHOLD 0.75
#define LOAD_FACTOR_THRESHOLD_MIN 0.1
#define MAX_ARGUMENT_LENGTH 100
unsigned int insert_number=0;
// Declare hashTable
// 定義鍵值對資料結構
typedef struct {
    char *key;
    char *field;
    char *value;
    struct ev_timer *timer; // 用於自動 expire key 的計時器
} KeyValue;
```

圖一

定義 INSERT_NUMBER 為 global variable，此為已存入 hashtable 的數量。將 key,field,value 和 callback function 的 timer 定義在 KeyValue 這個 structure 裡面。



```c
// 計算字符串的哈希碼
unsigned int hashFunction(const char *str) {
    unsigned int hash = 5381;
    int c;
    while ((c = *str++)) {
        hash = ((hash << 5) + hash) + c; // hash * 33 + c
    }
    return hash;
}
HashNode **hashTable;
// Initialize the hash table
void initializeHashTable(int size) {
    hashTable = (HashNode **)malloc(size * sizeof(HashNode *));
    //initializes all the pointers in the hashTable array to NULL
    memset(hashTable, 0, size * sizeof(HashNode *));
    HASH_SIZE = size;
}
```

圖二

圖二定義 hashfunction，為 hash*33+c。
C 為將指針 str 往後移動一個位置，指向下一個字元。
並定義 hashtable 為指標的指標和初始化 hashtable 根據圖一的 hashsize。

```c
void resizeHashTable(int newSize) {
    HashNode **newTable = (HashNode **)malloc(newSize * sizeof(HashNode *));
    memset(newTable, 0, newSize * sizeof(HashNode *));

    // Rehash all existing elements
    for (int i = 0; i < HASH_SIZE; i++) {
        HashNode *current = hashTable[i];
        while (current != NULL)
        {
            HashNode *next = current->next;
            unsigned int newIndex = hashFunction(current->data->key) % newSize;
            current->next = newTable[newIndex];
            newTable[newIndex] = current;

            current = next;
        }
    }
    // Free the old hash table
    free(hashTable);
    // Update hashTable pointer
    hashTable = newTable;
    HASH_SIZE = newSize;
}
```

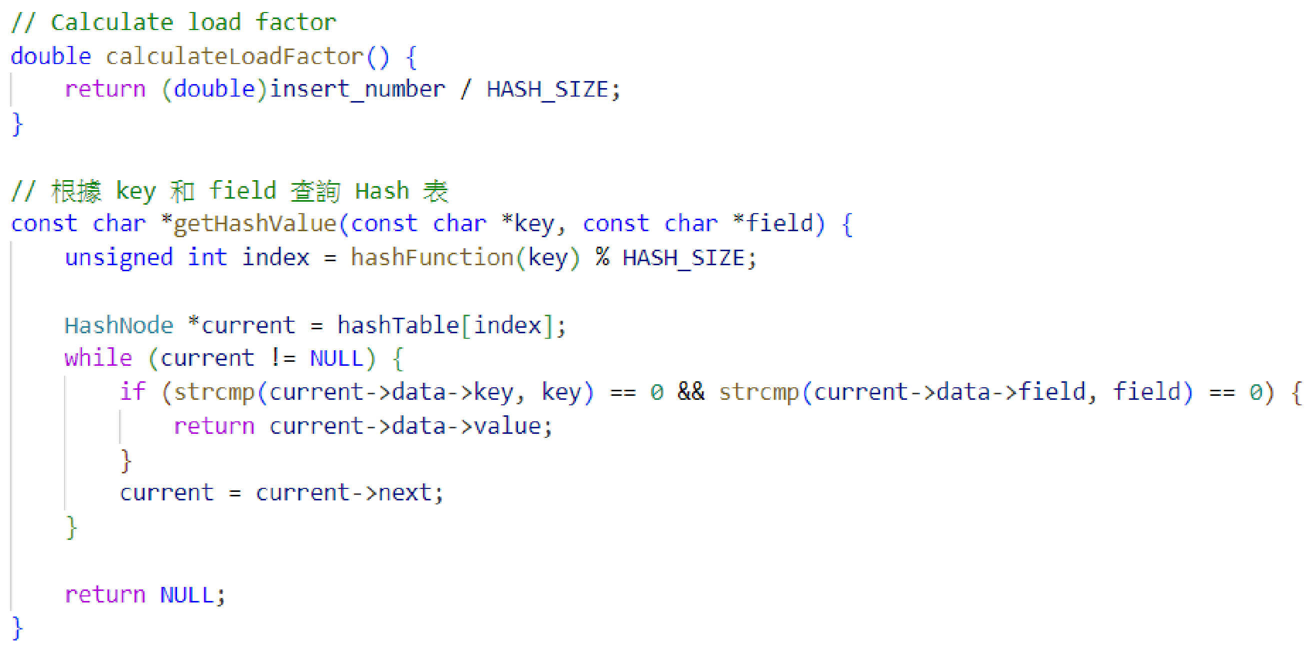
圖三

此為當 load_factor 滿足擴大和縮小時，會呼叫的函式。先創好新的 hashtable，再將舊的 hashtable 的 key,value 一個一個存入到新的 hashtable，最後釋放舊的 hashtable 的記憶體，並將 hashtable 和他的 size 更新。

```c
// 插入鍵值對到 Hash 表
void insertKeyValue(const char *key, const char *field, const char *value) {
    unsigned int index = hashFunction(key) % HASH_SIZE;

    HashNode *newNode = (HashNode *)malloc(sizeof(HashNode));
    KeyValue *newKeyValue = (KeyValue *)malloc(sizeof(KeyValue));

    newKeyValue->key = strdup(key);
    newKeyValue->field = strdup(field);
    newKeyValue->value = strdup(value);

    newNode->data = newKeyValue;
    newNode->next = NULL; // 將新節點的下一個節點設為 NULL
    insert_number+=1;
    // 如果哈希桶為空，將新節點設為頭節點
    if (hashTable[index] == NULL) {
        hashTable[index] = newNode;
    }
    else {
        // 否則，遍歷串列找到尾部，並將新節點插入到尾部
        HashNode *current = hashTable[index];
        while (current->next != NULL)
        {
            current = current->next;
        }
        current->next = newNode;
    }
}
```

圖四

圖四為通過 hashfunction 算出在 hashtable 的位置，並插入在那個位置 linked list 的最後。

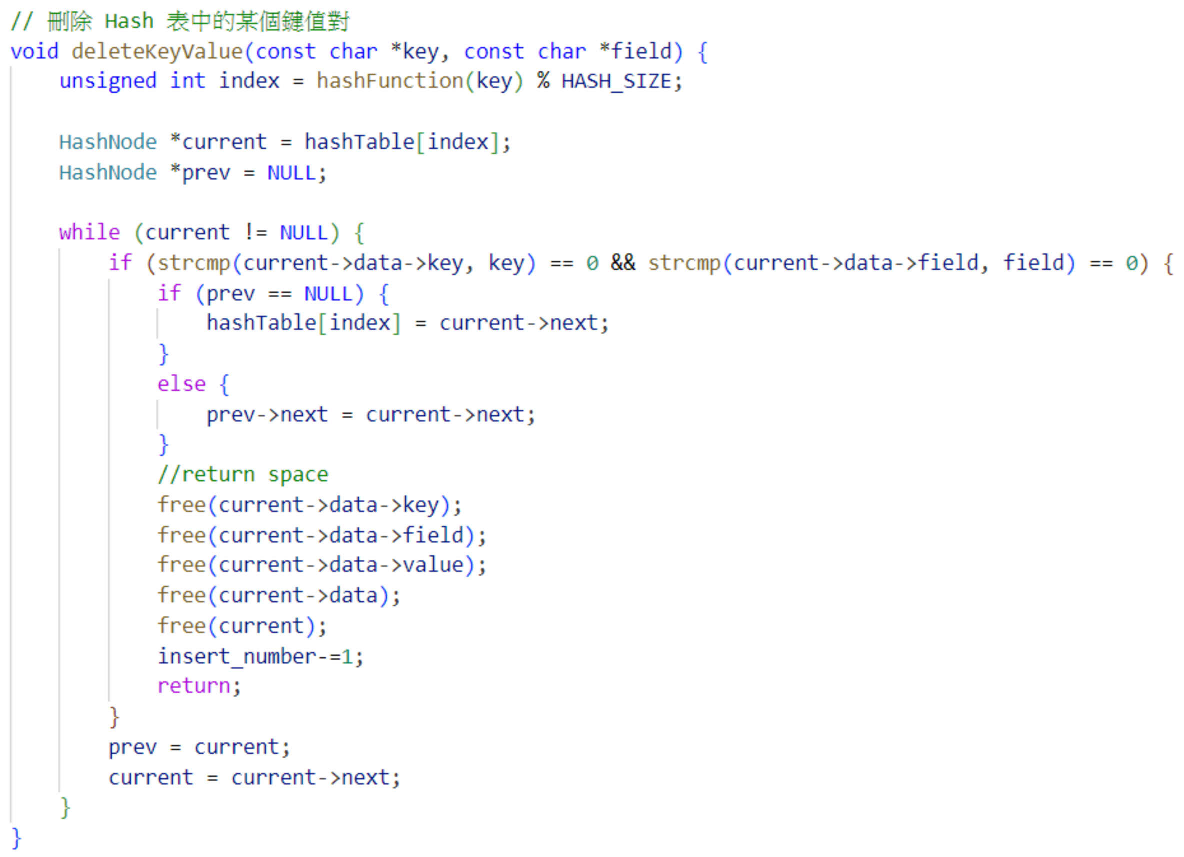


```c
// Calculate load factor
double calculateLoadFactor() {
    return (double)insert_number / HASH_SIZE;
}

// 根據 key 和 field 查詢 Hash 表
const char *getHashValue(const char *key, const char *field) {
    unsigned int index = hashFunction(key) % HASH_SIZE;

    HashNode *current = hashTable[index];
    while (current != NULL) {
        if (strcmp(current->data->key, key) == 0 && strcmp(current->data->field, field) == 0) {
            return current->data->value;
        }
        current = current->next;
    }

    return NULL;
}
```

圖五

Calculateloadfactor 算出 load_factor 的大小。

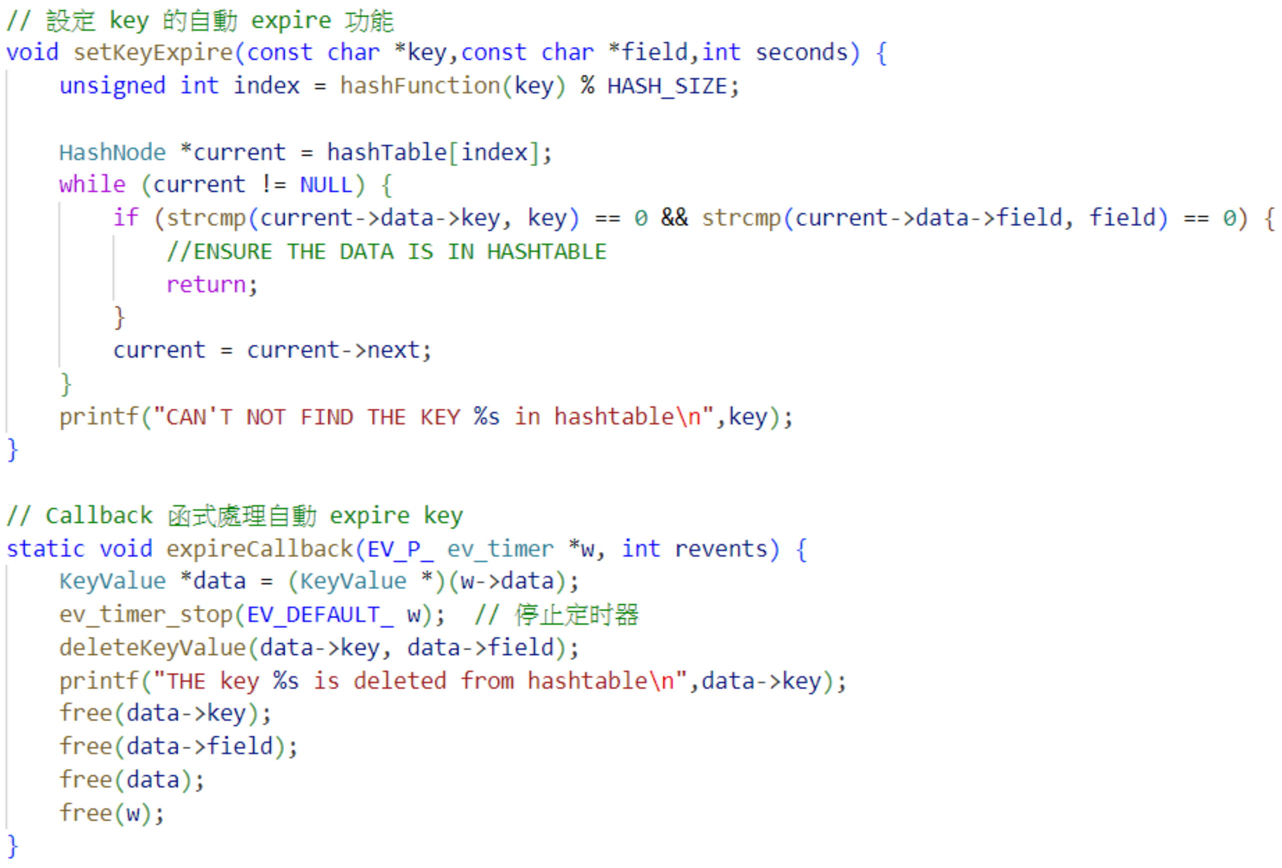
getHashValue 則找尋是否在對應的位址可以找到正確的 key。



```c
// 刪除 Hash 表中的某個鍵值對
void deleteKeyValue(const char *key, const char *field) {
    unsigned int index = hashFunction(key) % HASH_SIZE;

    HashNode *current = hashTable[index];
    HashNode *prev = NULL;

    while (current != NULL) {
        if (strcmp(current->data->key, key) == 0 && strcmp(current->data->field, field) == 0) {
            if (prev == NULL) {
                hashTable[index] = current->next;
            }
            else {
                prev->next = current->next;
            }
            //return space
            free(current->data->key);
            free(current->data->field);
            free(current->data->value);
            free(current->data);
            free(current);
            insert_number-=1;
            return;
        }
        prev = current;
        current = current->next;
    }
}
```

圖六

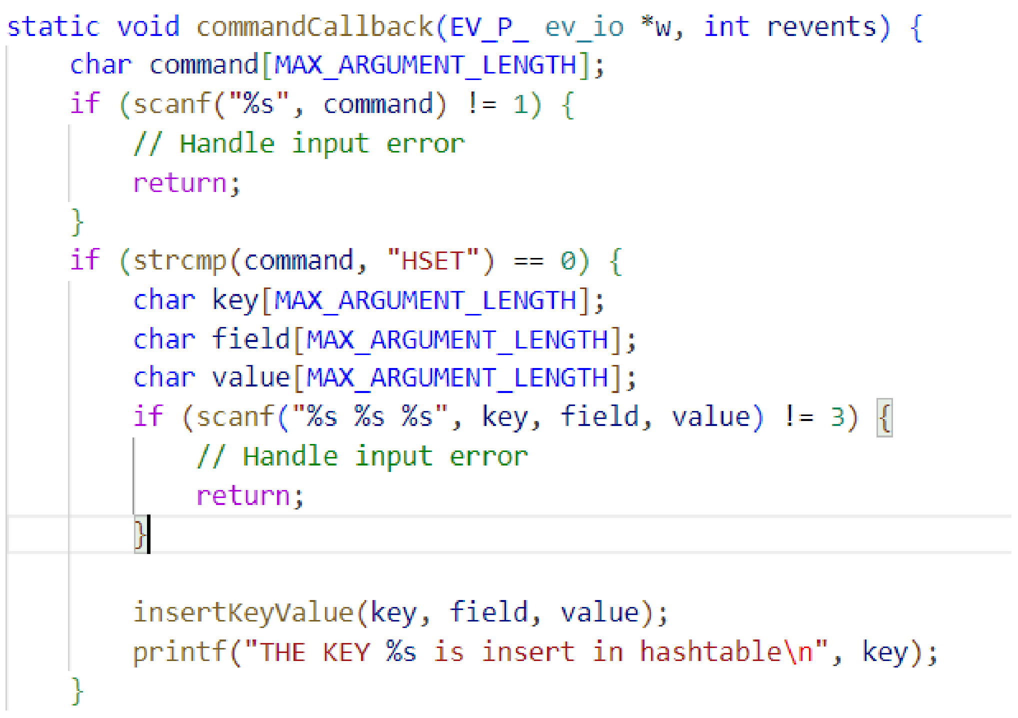
deleteKeyValue 將對應的 key 刪除，並將 linked list 連接起來。並歸回 key 的記憶體。



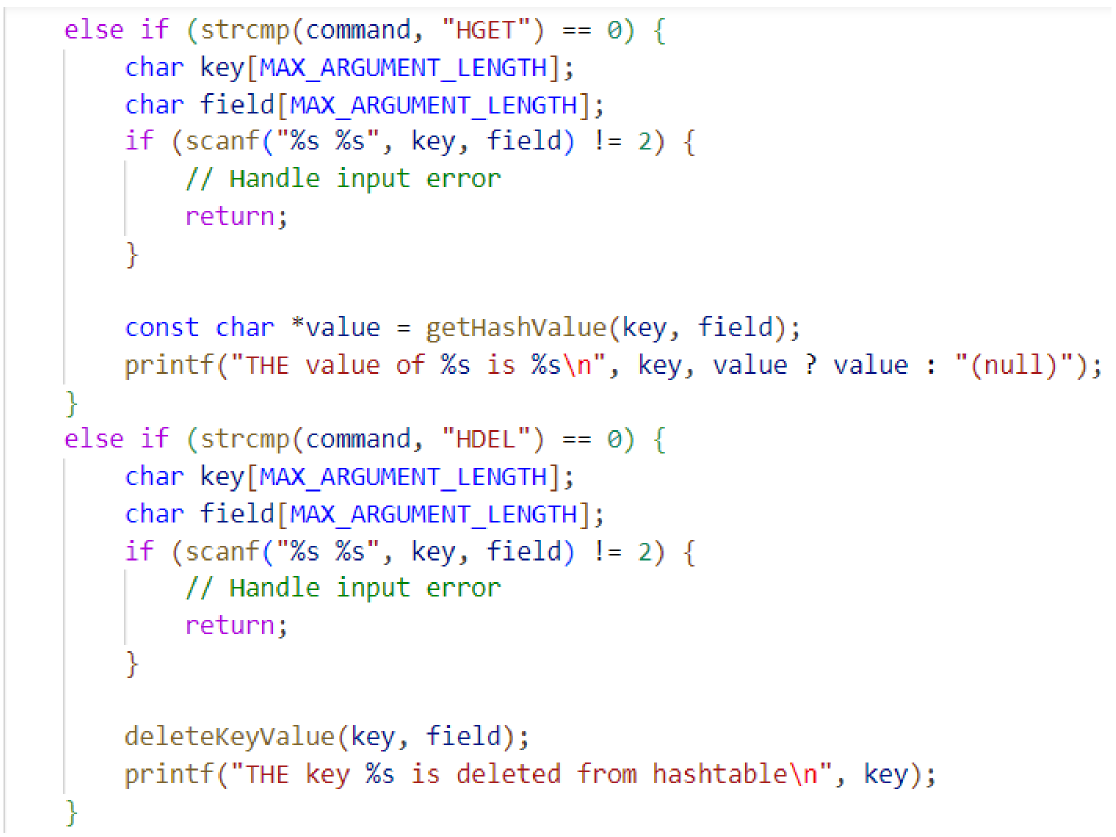
```c
// 設定 key 的自動 expire 功能
void setKeyExpire(const char *key,const char *field,int seconds) {
    unsigned int index = hashFunction(key) % HASH_SIZE;

    HashNode *current = hashTable[index];
    while (current != NULL) {
        if (strcmp(current->data->key, key) == 0 && strcmp(current->data->field, field) == 0) {
            //ENSURE THE DATA IS IN HASHTABLE
            return;
        }
        current = current->next;
    }
    printf("CAN'T NOT FIND THE KEY %s in hashtable\n",key);
}

// Callback 函式處理自動 expire key
static void expireCallback(EV_P_ ev_timer *w, int revents) {
    KeyValue *data = (KeyValue *)(w->data);
    ev_timer_stop(EV_DEFAULT_ w);  // 停止定时器
    deleteKeyValue(data->key, data->field);
    printf("THE key %s is deleted from hashtable\n",data->key);
    free(data->key);
    free(data->field);
    free(data);
    free(w);
}
```

圖七

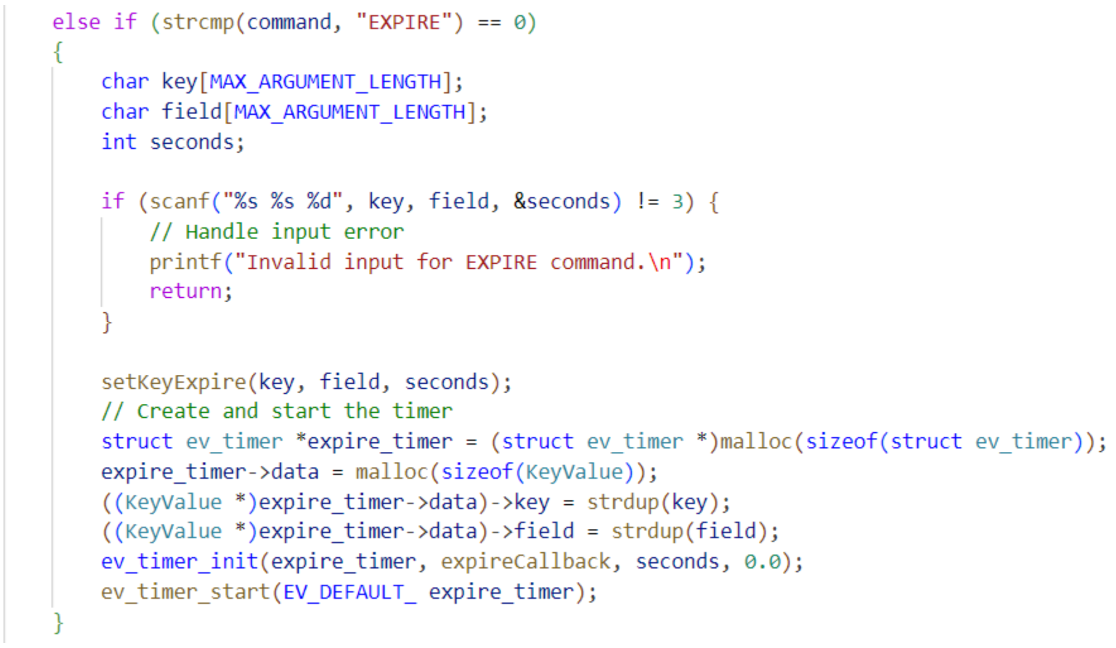
setKeyExpire 找尋 hashtable 是否存在對應的 key，沒有的話印出找不到。expireCallback，則是將 libev 的 timer 倒數完後將 key 刪除。



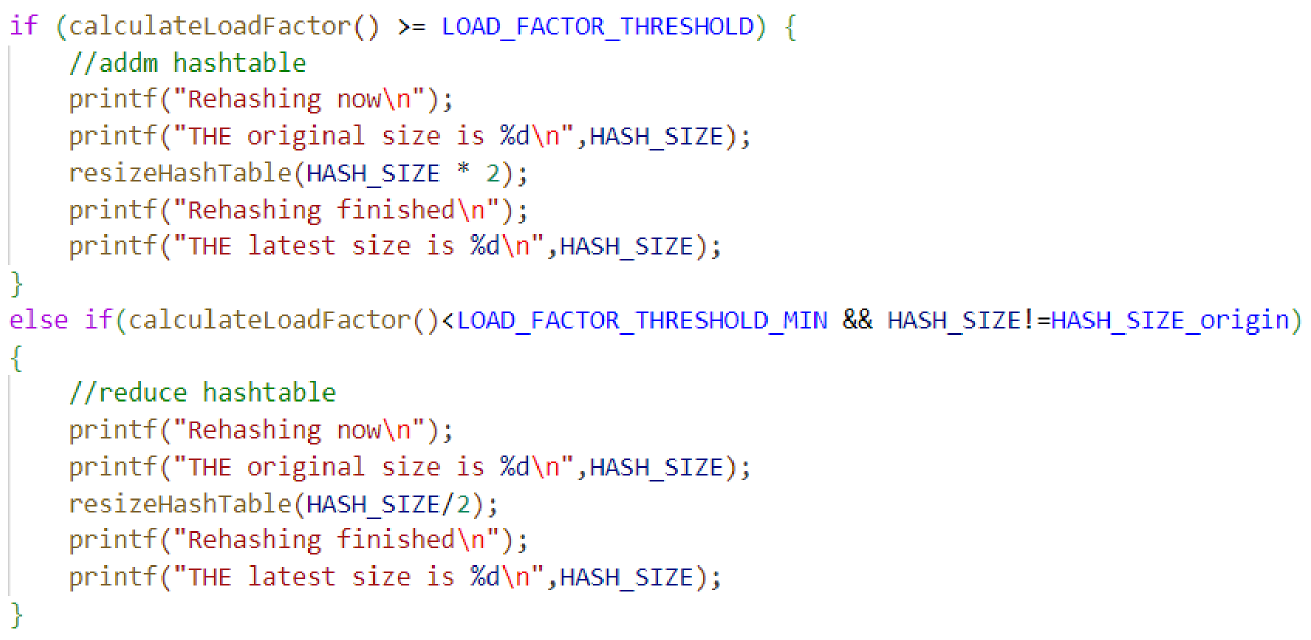
```c
static void commandCallback(EV_P_ ev_io *w, int revents) {
    char command[MAX_ARGUMENT_LENGTH];
    if (scanf("%s", command) != 1) {
        // Handle input error
        return;
    }
    if (strcmp(command, "HSET") == 0) {
        char key[MAX_ARGUMENT_LENGTH];
        char field[MAX_ARGUMENT_LENGTH];
        char value[MAX_ARGUMENT_LENGTH];
        if (scanf("%s %s %s", key, field, value) != 3) {
            // Handle input error
            return;
        }

        insertKeyValue(key, field, value);
        printf("THE KEY %s is insert in hashtable\n", key);
    }
```



```c
else if (strcmp(command, "HGET") == 0) {
    char key[MAX_ARGUMENT_LENGTH];
    char field[MAX_ARGUMENT_LENGTH];
    if (scanf("%s %s", key, field) != 2) {
        // Handle input error
        return;
    }

    const char *value = getHashValue(key, field);
    printf("THE value of %s is %s\n", key, value ? value : "(null)");
}
else if (strcmp(command, "HDEL") == 0) {
    char key[MAX_ARGUMENT_LENGTH];
    char field[MAX_ARGUMENT_LENGTH];
    if (scanf("%s %s", key, field) != 2) {
        // Handle input error
        return;
    }

    deleteKeyValue(key, field);
    printf("THE key %s is deleted from hashtable\n", key);
}
```



```c
else if (strcmp(command, "EXPIRE") == 0)
{
    char key[MAX_ARGUMENT_LENGTH];
    char field[MAX_ARGUMENT_LENGTH];
    int seconds;

    if (scanf("%s %s %d", key, field, &seconds) != 3) {
        // Handle input error
        printf("Invalid input for EXPIRE command.\n");
        return;
    }

    setKeyExpire(key, field, seconds);
    // Create and start the timer
    struct ev_timer *expire_timer = (struct ev_timer *)malloc(sizeof(struct ev_timer));
    expire_timer->data = malloc(sizeof(KeyValue));
    ((KeyValue *)expire_timer->data)->key = strdup(key);
    ((KeyValue *)expire_timer->data)->field = strdup(field);
    ev_timer_init(expire_timer, expireCallback, seconds, 0.0);
    ev_timer_start(EV_DEFAULT_ expire_timer);
}
```

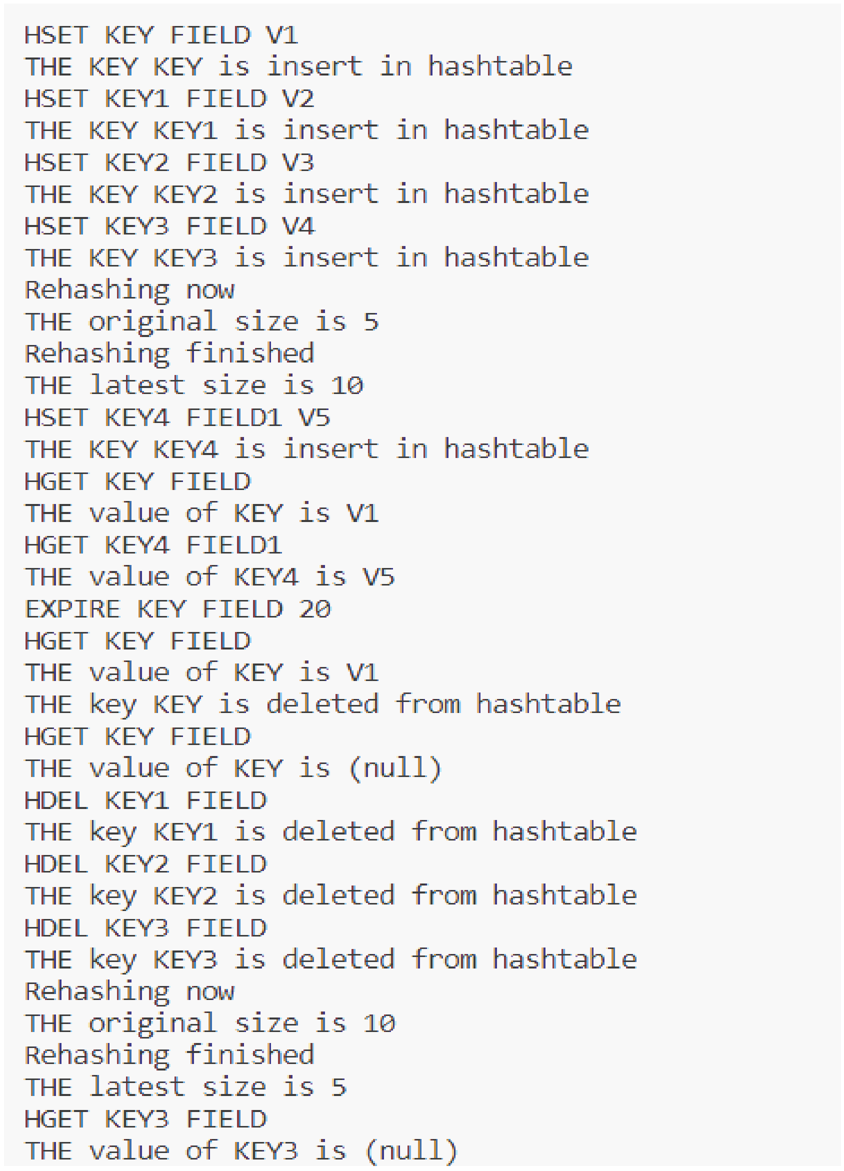


```c
if (calculateLoadFactor() >= LOAD_FACTOR_THRESHOLD) {
    //addm hashtable
    printf("Rehashing now\n");
    printf("THE original size is %d\n",HASH_SIZE);
    resizeHashTable(HASH_SIZE * 2);
    printf("Rehashing finished\n");
    printf("THE latest size is %d\n",HASH_SIZE);
}
else if(calculateLoadFactor()<LOAD_FACTOR_THRESHOLD_MIN && HASH_SIZE!=HASH_SIZE_origin)
{
    //reduce hashtable
    printf("Rehashing now\n");
    printf("THE original size is %d\n",HASH_SIZE);
    resizeHashTable(HASH_SIZE/2);
    printf("Rehashing finished\n");
    printf("THE latest size is %d\n",HASH_SIZE);
}
```

圖八

這個 function 則是使用 libev 的 callback function，判斷是否 HSET HGET HDEL EXPIRE，並且每次操作完後通過 loadfactor 判斷是否需要 rehash。如需要 REHASH，判斷放大，則 HASHTABLE 變兩倍，縮小也是為原本的 1/2。

程式碼範例輸出



```
HSET KEY FIELD V1
THE KEY KEY is insert in hashtable
HSET KEY1 FIELD V2
THE KEY KEY1 is insert in hashtable
HSET KEY2 FIELD V3
THE KEY KEY2 is insert in hashtable
HSET KEY3 FIELD V4
THE KEY KEY3 is insert in hashtable
Rehashing now
THE original size is 5
Rehashing finished
THE latest size is 10
HSET KEY4 FIELD1 V5
THE KEY KEY4 is insert in hashtable
HGET KEY FIELD
THE value of KEY is V1
HGET KEY4 FIELD1
THE value of KEY4 is V5
EXPIRE KEY FIELD 20
HGET KEY FIELD
THE value of KEY is V1
THE key KEY is deleted from hashtable
HGET KEY FIELD
THE value of KEY is (null)
HDEL KEY1 FIELD
THE key KEY1 is deleted from hashtable
HDEL KEY2 FIELD
THE key KEY2 is deleted from hashtable
HDEL KEY3 FIELD
THE key KEY3 is deleted from hashtable
Rehashing now
THE original size is 10
Rehashing finished
THE latest size is 5
HGET KEY3 FIELD
THE value of KEY3 is (null)
```

圖九

先插入四筆資料，因為放大條件為 loadfacor 超過 0.75，且因為初始 hashtable 大小為 5。所以當 HSET KEY3 後會顯示 REHASH，將 HASHTABLE 大小改為 10。

之後再 HSET 一個不同 FIELD 的 KEY。之後使用 HGET 可得到之前 HSET 的 VALUE 值。

接著測試 EXPIRE， EXPIRE KEY FIELD，並將秒數設為 20 秒。

之後先使用 HGET 可發現還可以得到 KEY 的值。

之後等超過 20S，他會顯示已將 KEY 刪除，再使用 HGET 可發現無法找到 KEY 的 VALUE。

最後測試縮小 HASHTABLE，將之前 HSET 的 KEY 刪除，因為之前引入 5 個 KEY，且 EXPIRE 一個，所以剩四個。但縮小條件為 HASHTABLE 的 1/5 資料，

也就是當資料小於兩筆則縮小 HASHTABLE，因此刪除三個資料後就符合 REHASH 的條件。
之後測試通過以刪除的 KEY 來找尋發現 KEY3 找不到。
以上為我的程式範例輸出，可知與預期皆符合，謝謝。