資料結構 HW_6

110303581 陳俊瑋

- **Mylib.h**

下面的部分從新加入的部分開始。

```
//HASH
typedef struct NodeForHash {
    char key[KEY_SIZE];
    char value[VALUE_SIZE];
    struct NodeForHash *next;
} NodeForHash;

// 定義雜湊表
typedef struct HashTable {
    NodeForHash **table;
    char field[FIELD_SIZE];
    int size;
    int table_size;
} HashTable;

typedef struct DatabaseForHash {
    HashTable **hashTables;  // 指向哈希表陣列的指標
    int numHashTables;       // 哈希表的數量
} DatabaseForHash;
```

這邊先定義 `NodeForHash` 的結構，為每筆 key-value 的節點，類似一個 Linked List 的節點。`HashTable` 結構裡面有一個我們哈希表的 table 指標陣列，還有 table 的 field(字段)，而 `size` 為資料數量，`table_size` 為 table 的大小。之後再定義 `DatabaseForHash` 為一指標陣列和一個用來追蹤此指標陣列的 int `numHashTables`。

- **Mylib.c**

下面的部分從新加入的部分開始。

```
// 初始化雜湊表
void initHashTable(HashTable *hashTable, const char *field, int
table_size) {
    strcpy(hashTable->field, field);
    hashTable->table_size = table_size;
    hashTable->table = (NodeForHash **)malloc(sizeof(NodeForHash *) *
table_size);
    for (int i = 0; i < table_size; i++) {
        hashTable->table[i] = NULL;
    }
```

```
    hashTable->size = 0;  // 初始化項目數量
    printf("[%s] has set in the database.\n", field);
}

// 初始化雜湊表資料庫
DatabaseForHash* createDatabaseForHash() {
    DatabaseForHash* db =
(DatabaseForHash*)malloc(sizeof(DatabaseForHash));
    db->hashTables = NULL;
    db->numHashTables = 0;
    return db;
}
```

`initHashTable` 是用來初始化一個哈希表，給予空間，同時賦予 field，而 `createDatabaseForHash` 是用來初始化資料庫。

```
HashTable* searchForField(DatabaseForHash* db, const char *field){
    for (int i = 0; i < db->numHashTables; i++) {
        if (strcmp(db->hashTables[i]->field, field) == 0) {   //找到
Field
        // printf("Found [%s].\n", field);
            return db->hashTables[i];
        }
    }
    // 找不到 Field
     // printf("Cannot find [%s].\n", field);
    return NULL;
}

// 雜湊函數
int hashFunction(const char *key, const int table_size) {
    int hash = 0;
    for (int i = 0; key[i] != '\0'; i++) {
        hash += key[i];
    }
    return hash % table_size;
}
```

`searchForField` 函式是用 field 來找尋對應的哈希表。找到則回傳指標，沒有找到則回傳 `NULL`。

`hashFunction` 是哈希表的函數換算，我是將 key 字串的 ASCII code 相加然後處以哈希表的大小然後取餘數。

```
void insertItemForHash(HashTable *hashTable, const char *key, const
char *value) {
    int index = hashFunction(key, hashTable->table_size);

    NodeForHash *newNode = createNodeForHash(key, value);

    // 將新節點插入到槽的開頭(Linked List)
    newNode->next = hashTable->table[index];
    hashTable->table[index] = newNode;

    // 更新項目數量
    hashTable->size++;

    printf("Set [%s: %s] Successfully.\n", key, value);

    // 在插入後檢查 Load Factor
    checkLoadFactor(hashTable);
}
```

`insertItemForHash` 函式是用來加入進哈希表的，如果有 hash collision 就用 chaining，也就是 Linked List 串聯。`checkLoadFactor` 是用來檢查 Load Factor 的，會在後面提到。而在檔案裡，`insertItemForHash` 下方有一個函式，`insertItemForHash_withoutLF` 與這函式的差異只是不會一直檢查 Load Factor。

```
// 檢查 Load Factor，調整哈希表大小
void checkLoadFactor(HashTable *hashTable) {
    float loadFactor = (float)hashTable->size / hashTable->table_size;
    printf("(Load Factor of [%s]: %f)\n", hashTable->field, loadFactor);

    // 根據需要調整哈希表大小
    if (loadFactor > LOAD_FACTOR_THRESHOLD) {
        printf("Enlarge Hash Table Size.\n");
        int newSize = hashTable->table_size * 2;
        resizeHashTable(hashTable, newSize);  // 放大哈希表
    } else if (loadFactor < MIN_LOAD_FACTOR_THRESHOLD && hashTable-
>table_size > INITIAL_TABLE_SIZE) {
        printf("Shrink Hash Table Size.\n");
        int newSize = hashTable->table_size / 2;
        resizeHashTable(hashTable, newSize);  // 縮小哈希表，且避免過度縮
小
    }
}

// 調整哈希表的大小
```

```
void resizeHashTable(HashTable *ht, int newSize) {
    HashTable newHashTable;
    initHashTable(&newHashTable, ht->field, newSize);

    // 將所有現有項目重新插入到新表中
    for (int i = 0; i < ht->table_size; i++) {
        NodeForHash *current = ht->table[i];
        while (current != NULL) {
            insertItemForHash_withoutLF(&newHashTable, current->key,
current->value);
            current = current->next;
        }
    }

    // 釋放舊表
    freeHashTable(ht);

    // 更新哈希表的大小
    *ht = newHashTable;

    float loadFactor = (float) ht->size / ht->table_size;
    printf("(Load Factor of [%s]: %f)\n", ht->field, loadFactor);
}

void freeHashTable(HashTable *hashTable) {
    for (int i = 0; i < hashTable->table_size; i++) {
        NodeForHash *current = hashTable->table[i];
        while (current != NULL) {
            NodeForHash *next = current->next;
            free(current);
            current = next;
        }
        hashTable->table[i] = NULL; // 將該槽設為空，避免懸空指標
    }
}
```

`checkLoadFactor` 是用來檢查 Load Factor 的，並判斷說有沒有大於或是小於閥值(`LOAD_FACTOR_THRESHOLD` 和 `MIN_LOAD_FACTOR_THRESHOLD`)，然後進行大小除 2 或乘 2 的動作。此外也會確保說不會縮小到小於原本初始的 Table 大小(`INITIAL_TABLE_SIZE` 為 10)。

`resizeHashTable` 會先用新的大小創建一個新的哈希表，給予一樣的 field，然後再將原先的資料一一放進新表，並且釋放舊表。之後將原本指向舊表的指標更新改指為新表。而 `freeHashTable` 是只將該指標目前指向的東西釋放，不會釋放該指標的空間(也就是釋放`*hashTable` 所指向的東西，但不會釋放

`*hashTable` 這個指標。

```
NodeForHash* searchHashNode(const HashTable *hashTable, const char
*key) {
    int index = hashFunction(key, hashTable->table_size);

    NodeForHash *pointer = hashTable->table[index];

    // 在 Linked List 中查找
    while (pointer != NULL) {
        if (strcmp(pointer->key, key) == 0) {
            // 找到了
            return pointer;
        }
        pointer = pointer->next;
    }

    // 沒有找到
    return NULL;
}
```

這函數是走訪整個哈希表，也用 while 走訪鏈結的部分。找到則回傳該節點的指標，找不到則回傳 `NULL`。

HSET:

```
void HSET(DatabaseForHash *db, const char *key, const char *field,
const char *value){
    HashTable* ht = searchForField(db, field);

    // 找不到 Field
    if(ht == NULL){
        db->hashTables = (HashTable **)realloc(db->hashTables,
sizeof(HashTable *) * (size_t)(db->numHashTables + 1));
        db->hashTables[db->numHashTables] = (HashTable
*)malloc(sizeof(HashTable));
        // 初始化一個 Hash
        initHashTable(db->hashTables[db->numHashTables], field,
INITIAL_TABLE_SIZE);
        db->numHashTables++;
        ht = db->hashTables[db->numHashTables-1];

        insertItemForHash(ht, key, value);
        return;
    }
```

```
    NodeForHash *pointer = searchHashNode(ht, key);
    if(pointer == NULL){
        // 新的 key
        insertItemForHash(ht, key, value);
    }
    else{
        // 更新 value
        strcpy(pointer->value, value);
        printf("Update [%s] Successfully.\n", key);
    }

}
```

HSET 函式會先找尋該哈希表，該哈希表不存在則初始化一個哈希表並且插入一個 key-value。如果該哈希表存在，則找尋該節點，找不到就創建一個新節點，找到就更新該 value 值。

HGET:

```
void HGET(DatabaseForHash *db, const char *key, const char *field){
    HashTable* ht = searchForField(db, field);

    if(ht == NULL){
        printf("Not Found [%s] in Database.\n", field);
        return;
    }
    else{
        NodeForHash *pointer = searchHashNode(ht, key);
        if (pointer == NULL){
            printf("Not Found [%s] in [%s] hashtable.\n", key, field);
            return;
        }
        else {
            printf("Key: [%s]  Value: [%s]\n", key, pointer->value);
            return;
        }

    }
}
```

該函數為一樣先找尋對應的哈希表，找到之後在找尋對應的 key 的節點，之後輸出 key 和 value。

HDEL:

```
void HDEL(DatabaseForHash *db, const char *key, const char *field,
const int isEXPIRE) {
    HashTable* ht = searchForField(db, field);

    if (ht == NULL) {
        printf("Not Found [%s] in [%s] hashtable.\n", key, field);
        return;
    }

    int index = hashFunction(key, ht->table_size);
    NodeForHash *current = ht->table[index];
    NodeForHash *prev = NULL;

    while (current != NULL) {
        if (strcmp(current->key, key) == 0) {
            // 找到要刪除的節點
            if (prev == NULL) {
                // 節點在 Linked List 的開頭
                ht->table[index] = current->next;
            }
            else {
                // 節點在 Linked List 中間或尾巴
                prev->next = current->next;
            }

            // 釋放節點
            free(current);

            // 更新項目數量
            ht->size--;

            if(isEXPIRE == 1){
                printf("\n");
                printf("Deleted [%s] Successfully.\n", key);
            }
            else{
                printf("Deleted [%s] Successfully.\n", key);
            }

            // 在刪除後檢查 Load Factor
            checkLoadFactor(ht);
            // printf("%d", ht->table_size);
            return;
        }

        prev = current;
        current = current->next;
```

```
    }

    // 找不到要刪除的節點
    printf("Key [%s] not found in [%s].\n", key, field);
}
```

這函式與前個函式相似，先找尋哈希表在找對應的節點，但由於可能會涉及鏈結的刪除，需要兩個指標進行操作，並且在刪除後檢查 Load Factor 看是否要縮小哈希表。

而 `isEXPIRE` 只是判斷是不是 EXPIRE 函式呼叫它的，此目的是因為後面的 EXPIRE 函式輸出結果需要先分行，不然排版會有點醜。

```
void freeHashDatabase(DatabaseForHash *db) {
    // 釋放每一個哈希表的記憶體
    for (int i = 0; i < db->numHashTables; i++) {
        freeHashTable(db->hashTables[i]);
        free(db->hashTables[i]);
    }

    // 釋放哈希表的陣列
    free(db->hashTables);

    // 釋放資料庫本身
    free(db);
}
```

此函式為釋放整個哈希表的資料庫。

EXPIRE:

```
void EXPIRE(DatabaseForHash* db, const char* key, const int time){
    HANDLE threadHandle;
    DWORD threadId;
    EXPIRE_input *input = (EXPIRE_input*)malloc(sizeof(EXPIRE_input));
    if (input == NULL) {
        fprintf(stderr, "Error allocating memory for input\n");
        return;
    }

    input->db = db;
    strcpy(input->key, key);
    input->time = time;
    printf("[%s] will be expired in [%d]s.\n", key, time);
    threadHandle = CreateThread(NULL, 0, countDownToDEL, input, 0,
NULL);   //創建 Thread
    if (threadHandle == NULL) {
        printf("Error creating thread\n");
        free(input);
        return;
    }
}

DWORD WINAPI countDownToDEL(LPVOID lpParam) {
    EXPIRE_input* input = (EXPIRE_input*)lpParam;
    Sleep(input->time * 1000);    //線程休眠一段時間(此函式的單位為毫秒所以
要乘以 1000)
    // printf("%d", input->time);
    for (int i = 0; i < input->db->numHashTables; i++) {    //將所有有包
含該 key 名稱的 field 都刪除
        char *field = input->db->hashTables[i]->field;
```

```
        HDEL(input->db, input->key, field, 0);
    }

    printf("Enter a command: ");
    free(input);
    return 0;
}
```

此函式會根據使用者傳入的秒數，決定過幾秒會刪除。由於我發現 redis 的 EXPIRE 函式是會將所有有包含該 key 名稱的 field 都刪除，所以我也用一個 for 迴圈達成這樣的操作。

而我是用 Windows API 來創建 Thread 來達成讓程式在後台讀秒，不會阻礙使用者輸入 command。

由於 `CreateThread` 這函式他只給一個位置放入參數，所以我創建了一個結構叫做 `EXPIRE_input` 來塞入所有的參數。

`EXPIRE_input` 結構為下:

```
typedef struct EXPIRE_input {
    DatabaseForHash *db;
    char key[KEY_SIZE];
    int time;
} EXPIRE_input;
```

輸出結果:

```
KEY:(get/set/update/del)
LIST:(lpush/rpush/lpop/rpop/llen/lrange)
SET:(zadd/zcard/zcount/zinterstore/zunionstore/zrange/zrangebyscore/zra
nk/zrem/zremrangebylex/zremrangebyrank/zremrangebyscore)
HASH:(hset/hget/hdel)
EXIT:0
Enter a command: hset 1 field1 Value1
[field1] has set in the database.
Set [1: Value1] Successfully.
(Load Factor of [field1]: 0.100000)
Enter a command: hset 2 field1 Value2
Set [2: Value2] Successfully.
(Load Factor of [field1]: 0.200000)
Enter a command: hset 3 field1 Value3
Set [3: Value3] Successfully.
(Load Factor of [field1]: 0.300000)
Enter a command: hset 4 field1 Value4
Set [4: Value4] Successfully.
(Load Factor of [field1]: 0.400000)
Enter a command: hset 5 field1 Value5
Set [5: Value5] Successfully.
(Load Factor of [field1]: 0.500000)
Enter a command: hset 6 field1 Value6
Set [6: Value6] Successfully.
(Load Factor of [field1]: 0.600000)
Enter a command: hset 7 field1 Value7
Set [7: Value7] Successfully.
(Load Factor of [field1]: 0.700000)
Enter a command: hset 8 field1 Value8
Set [8: Value8] Successfully.
(Load Factor of [field1]: 0.800000)
Enter a command: hset 9 field1 Value9
Set [9: Value9] Successfully.
(Load Factor of [field1]: 0.900000)
Enter a command: hset 10 field1 Value10
Set [10: Value10] Successfully.
(Load Factor of [field1]: 1.000000)   //超過閥值(0.9)，放大哈希表
Enlarge Hash Table Size.
[field1] has set in the database.
Set [2: Value2] Successfully.
Set [3: Value3] Successfully.
Set [4: Value4] Successfully.
Set [5: Value5] Successfully.
Set [6: Value6] Successfully.
Set [7: Value7] Successfully.
Set [8: Value8] Successfully.
Set [10: Value10] Successfully.
```

```
Set [9: Value9] Successfully.
Set [1: Value1] Successfully.
(Load Factor of [field1]: 0.500000)
Enter a command: hset 1 field1 Value1111
Update [1] Successfully.
Enter a command: hget 1 field1
Key: [1]  Value: [Value1111]
Enter a command: hdel 1 field1    //刪除節點
Deleted [1] Successfully.
(Load Factor of [field1]: 0.450000)
Enter a command: hdel 2 field1
Deleted [2] Successfully.
(Load Factor of [field1]: 0.400000)
Enter a command: hdel 3 field1
Deleted [3] Successfully.
(Load Factor of [field1]: 0.350000)
Enter a command: hdel 4 field1
Deleted [4] Successfully.
(Load Factor of [field1]: 0.300000)
Enter a command: hdel 5 field1
Deleted [5] Successfully.
(Load Factor of [field1]: 0.250000)
Enter a command: hdel 6 field1
Deleted [6] Successfully.
(Load Factor of [field1]: 0.200000)
Enter a command: hdel 7 field1
Deleted [7] Successfully.
(Load Factor of [field1]: 0.150000)       //小於閥值(0.2)，縮小哈希表
Shrink Hash Table Size.
[field1] has set in the database.
Set [8: Value8] Successfully.
Set [9: Value9] Successfully.
Set [10: Value10] Successfully.
(Load Factor of [field1]: 0.300000)
Enter a command: expire 10 5          //設定 5 秒後過期
[10] will be expired in [5]s.
Enter a command: hget 10 field1
Key: [10]  Value: [Value10]           //5 秒內依然可以得到數值
Enter a command:
Deleted [10] Successfully.
(Load Factor of [field1]: 0.200000)
Enter a command: hget 10 field1       //5 秒後數值已刪除
Not Found [10] in [field1] hashtable.
Enter a command:
```