資料結構 HW_3

110303581 陳俊瑋

- **Mylib.h**

  其前半段的內容與 HW1 的前半段幾乎一樣，也就是定義 KeyValue 和 DataBase 的結構，還有定義四個操作(CRUD)的函數。由於第一份作業已經提過就不在贅述細節。下面的部分從新加入的函式開始。

```
//LIST
typedef struct Node
{
    char key[KEY_SIZE];
    char value[VALUE_SIZE];
    struct Node* prev;
    struct Node* next;
} Node;

Node *createNode(const char* key, const char* value)
{
    Node *newNode = (Node *)malloc(sizeof(Node));
    if (newNode == NULL)
    {
        perror("Memory allocation failed");
        exit(1);
    }

    strncpy(newNode->key, key, KEY_SIZE);
    strncpy(newNode->value, value, VALUE_SIZE);
    newNode->prev = NULL;
    newNode->next = NULL;
    return newNode;
}
```

此段為定義 linked list 的結構，而 `createNode` 函數的功能為創建一個節點，並將節點的指標返回。

```
// 加入 PUSH
void insertRight_node(Node** head, Node** bottom, const char* key,
const char* value)
{
    Node* newNode = createNode(key, value);

    if (*head == NULL)
    {
```

```
        *head = newNode;
        *bottom = newNode;
    }
    else
    {
        (*bottom)->next = newNode;
        newNode->prev = *bottom;
        *bottom = newNode;
    }

    printf("Insert [%s] in [%s] Successfully.\n", value,key);
}

void insertLeft_node(Node** head, Node** bottom, const char* key, const
char* value)
{
    Node* newNode = createNode(key, value);

    if (*head == NULL)
    {
        *head = newNode;
        *bottom = newNode;
    }
    else
    {
        (*head)->prev = newNode;
        newNode->next = *head;
        *head = newNode;
    }

    printf("Insert [%s] in [%s] Successfully.\n", value,key);
}
```

此段為 LPUSH（insertLeft_node)和 RPUSH（insertRight_node)的函式，會呼叫到上一個函數 createNode。當鏈結為空時，直接把 head 和 bottom 指向新節點。不為空時，LPUSH 將節點加在 head 這一側，RPUSH 加在 bottom 那一側。

```
void lpop(Node** head, Node** bottom, const char* key, int num)
{
    if(num == 0){
        num = 1;
    }

    if (*head == NULL)
    {
        printf("Database is Empty. Cannot delete Value.\n");
```

```
        return;
    }
    else
    {
        Node* Pointer = *head;
        int counter = 0;
        while (Pointer != NULL && counter < num)
        {
            if (strcmp(Pointer->key, key) == 0)
            {
                counter++;

                if (Pointer == *head) //刪除第一個
                {
                    *head = Pointer->next;
                    if (*head != NULL) {
                        (*head)->prev = NULL;
                    }
                }
                else if (Pointer == *bottom) //刪除最後一個
                {
                    *bottom = Pointer->prev;
                    if (*bottom != NULL) {
                        (*bottom)->next = NULL;
                    }
                }
                else
                {
                    if (Pointer->prev != NULL)
                    {
                        Pointer->prev->next = Pointer->next; //Pointer的
前一個指向下一個
                    }
                    if (Pointer->next != NULL)
                    {
                        Pointer->next->prev = Pointer->prev; //Pointer的
下一個指向前一個
                    }
                }

                Node* tmp = Pointer;
                Pointer = Pointer->next;
                tmp->prev = NULL;
                tmp->next = NULL;
                printf("Delete [%s] in [%s] Successfully.\n", tmp-
>value, key);
                free(tmp);
            }
```

```
            else{
                Pointer = Pointer->next;
            }
        }

        if (counter == 0)
        {
            printf("Key not found. Cannot delete.\n");
            return;
        }
        else
        {
            return;
        }
    }
}

void rpop(Node** head, Node** bottom, const char* key, int num)
{
    if(num == 0){
        num = 1;
    }

    if (*head == NULL)
    {
        printf("Database is Empty. Cannot delete Value.\n");
        return;
    }
    else
    {
        Node* Pointer = *bottom;
        int counter = 0;
        while (Pointer != NULL && counter < num)
        {
            if (strcmp(Pointer->key, key) == 0)
            {
                // printf("counter: %d\n", counter);
                counter++;

                if (Pointer == *head) //刪除第一個
                {
                    *head = Pointer->next;
                    if (*head != NULL) {
                        (*head)->prev = NULL;
                    }
                }
                else if (Pointer == *bottom) //刪除最後一個
                {
```

```
                    *bottom = Pointer->prev;
                    if (*bottom != NULL) {
                        (*bottom)->next = NULL;
                    }
                }
                else
                {
                    if (Pointer->prev != NULL)
                    {
                        Pointer->prev->next = Pointer->next; //Pointer的
前一個指向下一個
                    }
                    if (Pointer->next != NULL)
                    {
                        Pointer->next->prev = Pointer->prev; //Pointer的
下一個指向前一個
                    }
                }

                Node* tmp = Pointer;
                Pointer = Pointer->prev;
                tmp->prev = NULL;
                tmp->next = NULL;
                printf("Delete [%s] in [%s] Successfully.\n", tmp-
>value, key);
                free(tmp);
            }
            else{
                Pointer = Pointer->next;
            }
        }

        if (counter == 0){
            printf("Key not found. Cannot delete.\n");
        }
        return;
    }
}
```

此段為 LPOP 和 RPOP 的函式定義，redis 目前的版本有支援可以多次 POP，所以我多定義了 num 來給使用者輸入要 POP 幾個數值。最外面的 while 迴圈 `while (Pointer != NULL && counter < num)`是在讓 Pointer 找尋直到最後，而 counter 是當每找到一個對應的 key 時，就會累加一遍，直到大於使用者給的 num 為止。

由於我只有定義一個 linked list，所以在 POP 的時候要先找尋要 POP 的 key (`if (strcmp(Pointer->key, key) == 0)`)。LPOP 是從 head 開始找尋，而 RPOP 是從 bottom 開始找尋。

因為是雙向 linked list，所以刪除頭、尾和中間要分三種寫法。
刪除頭要將 head 改指向下一個，然後再將 head 現在指向的數值的 prev 指向 NULL，而刪除尾則是反過來。至於刪除中間的就是將目前指到的數據的前一個指向下一個，然後下一個指向前一個。
最後，再將 tmp 指向要刪除的數據，然後 free 掉。

```
void llen(Node* head, const char* key)
{
    if (head == NULL)
    {
        printf("Database is Empty. Cannot get Length.\n");
        return;
    }
    else
    {
        Node* Pointer = head;
        int num = 0;
        while (Pointer != NULL)
        {
            if (strcmp(Pointer->key, key) == 0)
            {
                num++;
            }
            Pointer = Pointer->next;
        }

        if(num != 0){
            printf("The length of [%s] is [%d]\n", key, num);
            return;
        }
    }
    printf("Not Found [%s] in Database. Cannot get Length.\n", key);
}
```

此段函數為 LLEN 的函數。是將 Pointer 從 head 開始，每有找到對應的 key 就計算一次，最後回傳數字。

```
// 列印範圍 lrange
void lrange(Node* head, Node* bottom, const char* key, int start, int
stop)
{
    if (head == NULL)
    {
        printf("Database is Empty. Cannot get Values.\n");
        return;
    }
    else
    {
        Node* Pointer1 = head;
        int num = 0;
        while (Pointer1 != NULL)
        {
            if (strcmp(Pointer1->key, key) == 0)
            {
                num++;
            }
            Pointer1 = Pointer1->next;
        }

        // printf("num: %d\n", num);

        if(num != 0){
            if(stop < 0){
                stop = num + stop; //把 stop 改成到哪裡結束
            }
            if(start < 0){
                start = num + start; //把 start 改成到哪裡開始
            }
            // printf("start: %d\n", start);
            // printf("stop: %d\n", stop);

            if(start <= stop){
                int counter = 0;
                Node* Pointer2 = head;
                while (Pointer2 != NULL){
                    if (strcmp(Pointer2->key, key) == 0){
                        if(counter >= start && counter <= stop){
                        printf("%d) %s\n", counter+1, Pointer2->value);
                        }
                        counter++;
                    }
                    Pointer2 = Pointer2->next;
                }
            }
            // 當 start < stop
```

```
            else{
                int counter = num-1;
                Node* Pointer2 = bottom;
                while (Pointer2 != NULL){
                    if (strcmp(Pointer2->key, key) == 0){
                        if(counter >= stop && counter <= start){
                        printf("%d) %s\n", counter+1, Pointer2->value);
                        }
                        counter--;
                    }

                    Pointer2 = Pointer2->prev;
                }
            }
            return;
        }
        else{
            printf("Not Found [%s] in Database. Cannot get Values.\n",
key);
        }
    }
}
```

此段為 LRANGE 的函數。Redis 目前版本的 LRANGE 是可以輸入要列印的範圍，以 0 為第一個，-1 為最後一個，-2 為倒數第二，也可以反著列印，像是 lrange num -1 0 這樣。所以我加上 start 和 stop 這兩個整數給使用者輸入列印範圍。

一開始的 while 先把這 linked list 有幾個對應的 key 的數量先找出來，數值為 num。這步驟的目的是為了有-1(最後一個)這種表達方式要進行換算。例如 stop 等於-1，就將加上 num 然後賦值給 stop，這樣 stop 就會為最後一個的索引(num-1)。

換算完 start 和 stop 後，再用一個 counter 從 0 開始數，有對應的 key 就將 counter 加 1。由於先前使用者會輸入範圍，我是判斷當 counter 數到範圍內時才會 print 數值。

後面的 `read_node`、`updata_node`、`del_node` 只是以備不時之需，可忽略。

```
void freeList(Node* head) {
    while (head != NULL) {
        Node* Pointer = head;
        head = head->next;
        Pointer->prev = NULL;
        Pointer->next = NULL;
        free(Pointer);
    }
}
```

最後這函式為釋放整個 linked list 的記憶體，由於是 linked list，需要每個節點走訪一一釋放。

- hw2.c

```c
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <malloc.h>
#include "mylib.h"

int main()
{
    // KEY-VALUE
    Database* db = createDatabase();
    char input[INPUT_SIZE];

    // LIST
    Node* head = NULL;
    Node* bottom = NULL;
    char input_for_list[INPUT_SIZE];
```

前面先是 include 會用到的函式。KeyValue 的形式不變，加上了雙向 Linked List 所需要的 head 和 bottom。`input_for_list` 是用來存輸入的陣列，之後會將其拆解，判斷指令。

```c
do
    {
        printf("Enter a command
(get/set/update/del/lpush/rpush/lpop/rpop/llen/lrange/0): ");
        fgets(input, sizeof(input), stdin);
        input[strcspn(input, "\n")] = '\0';

        strcpy(input_for_list, input);

        char* command_ptr;
        char* key_ptr;

        // pointer
        command_ptr = strtok(input_for_list, " ");
        key_ptr = strtok(NULL, " ");
```

上面的部分是將 input_for_list 等於 input，目的只是保留原本的 input，因為 `strtok` 函數判斷空格拆解陣列然後就回不去原本的樣子了。

而 `command_ptr` 指向 `input_for_list` 以第一個空格切割前的字串，`key_ptr` 則為剩下的字串再進行切割。

中間有段程式碼為判斷 KeyValue 的指令和執行，由於跟第一次作業類似，就不在贅述。

```
            // LIST
            // LPUSH
            else if (strcmp(command_ptr, "lpush") == 0)
            {
                char *value_ptr = strtok(NULL, " ");
                while (value_ptr != NULL)
                {
                    insertLeft_node(&head, &bottom, key_ptr, value_ptr);
                    value_ptr = strtok(NULL, " ");
                }
            }
            // RPUSH
            else if (strcmp(command_ptr, "rpush") == 0)
            {
                char *value_ptr = strtok(NULL, " ");
                while (value_ptr != NULL)
                {
                    insertRight_node(&head, &bottom, key_ptr,
value_ptr);
                    value_ptr = strtok(NULL, " ");
                }
            }
```

這邊開始為 Linked List 的指令判斷和操作。`value_ptr` 為剩下的字串再進行切割，也就是 PUSH 指令的 Value 部分。會一直切割直到變成 NULL。而每次切割下來的 value 都會用 `insertLeft_node` 或是 `insertRight_node` 添加到 Linked List 裡面。

```
            // LPOP
            else if (strcmp(command_ptr, "lpop") == 0)
            {
                int num = atoi(strtok(NULL, " "));
                lpop(&head, &bottom, key_ptr, num);
            }

            // RPOP
            else if (strcmp(command_ptr, "rpop") == 0)
            {
                int num = atoi(strtok(NULL, " "));
                rpop(&head, &bottom, key_ptr, num);
            }
```

此段為 POP 的指令判斷和操作。num 為再切割的部分，也就是要一次 POP 幾個數據。其中 `atoi` 是將字串轉成整數型態。

```
            // LLEN
            else if (strcmp(command_ptr, "llen") == 0)
            {
                llen(head, key_ptr);
            }

            // LRANGE
            else if (strcmp(command_ptr, "lrange") == 0)
            {
                int start = atoi(strtok(NULL, " "));
                int stop = atoi(strtok(NULL, " "));
                lrange(head, bottom, key_ptr, start, stop);
            }

            else
            {
               printf("Invalid command\n");
            }

        }
    }
```

此段為 LLEN 和 LRANGE。其中也是要將 `start` 和 `stop` 轉為整數型態。

```
    free(db->data);
    free(db);
    freeList(head);
```

最後就釋放空間，而 Linked List 的部分用先前定義的 `freeList` 函式來釋放。

輸出結果:

```
Enter a command
(get/set/update/del/lpush/rpush/lpop/rpop/llen/lrange/0): lpush num
three
two one
Insert [three] in [num] Successfully.
Insert [two] in [num] Successfully.
Insert [one] in [num] Successfully.
Enter a command
(get/set/update/del/lpush/rpush/lpop/rpop/llen/lrange/0): rpush num
four five six
Insert [four] in [num] Successfully.
Insert [five] in [num] Successfully.
Insert [six] in [num] Successfully.
Enter a command
(get/set/update/del/lpush/rpush/lpop/rpop/llen/lrange/0): lrange num 0
-1
1) one
2) two
3) three
4) four
5) five
6) six
Enter a command
(get/set/update/del/lpush/rpush/lpop/rpop/llen/lrange/0): lrange num 0
-2
1) one
2) two
3) three
4) four
5) five
Enter a command
(get/set/update/del/lpush/rpush/lpop/rpop/llen/lrange/0): lrange num -3
0
4) four
3) three
2) two
1) one
Enter a command
(get/set/update/del/lpush/rpush/lpop/rpop/llen/lrange/0): llen num
The length of [num] is [6]
Enter a command
(get/set/update/del/lpush/rpush/lpop/rpop/llen/lrange/0): lpop num
Delete [one] in [num] Successfully.
Enter a command
(get/set/update/del/lpush/rpush/lpop/rpop/llen/lrange/0): rpop num
Delete [six] in [num] Successfully.
```

```
Enter a command
(get/set/update/del/lpush/rpush/lpop/rpop/llen/lrange/0): lrange num 0
-1
1) two
2) three
3) four
4) five
Enter a command
(get/set/update/del/lpush/rpush/lpop/rpop/llen/lrange/0): lpop num 2
Delete [two] in [num] Successfully.
Delete [three] in [num] Successfully.
Enter a command
(get/set/update/del/lpush/rpush/lpop/rpop/llen/lrange/0): rpop num 2
Delete [five] in [num] Successfully.
Delete [four] in [num] Successfully.
Enter a command
(get/set/update/del/lpush/rpush/lpop/rpop/llen/lrange/0): lrange num 0
-1
Database is Empty. Cannot get Values.
Enter a command
(get/set/update/del/lpush/rpush/lpop/rpop/llen/lrange/0): 0
END
```