HW4

EE3B 110305504 黃聖倫

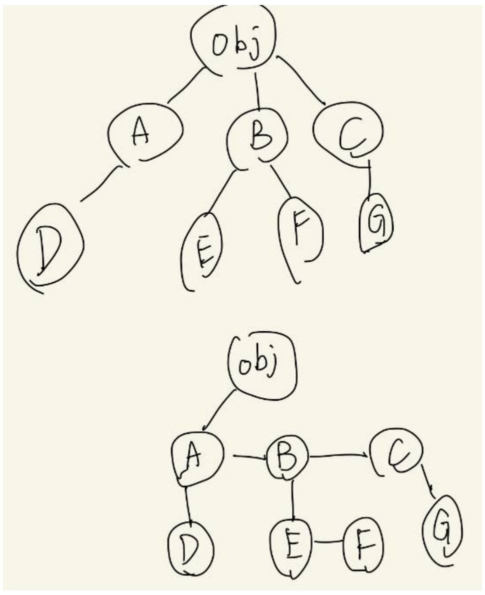
1. 程式架構

   參考

   1. https://github.com/DaveGamble/cJSON/
   2. https://www.json.org/json-en.html

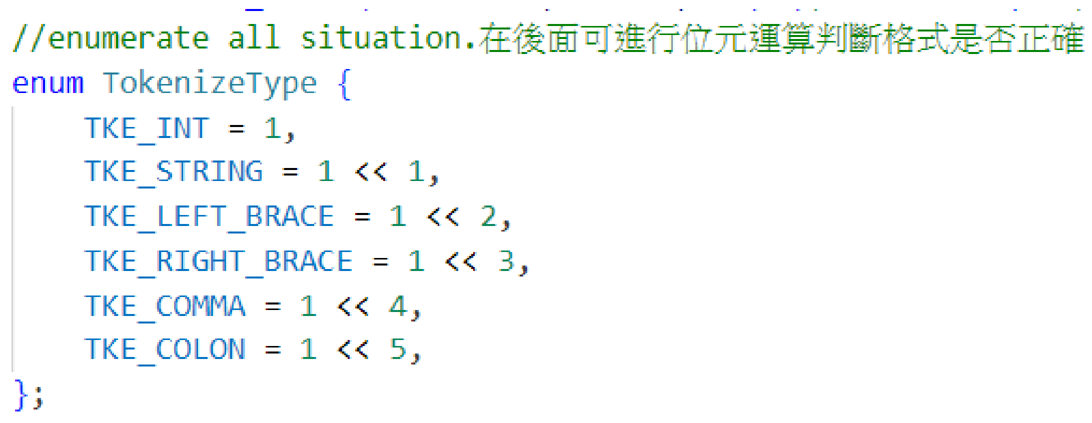
主要的 JSON PARSER 參考第一個開源去做修改。(因為 json 格是很難去分解，需要考慮多種情況，因此我的 parser 主要是經過參考 1 去修正，以達到 parser 作用。)但因為 json 格式的 array，我認為需要考慮更多情況，因此此次作業沒有辦法處理 array 的情況。

儲存格式則為下圖(概念圖)



圖一

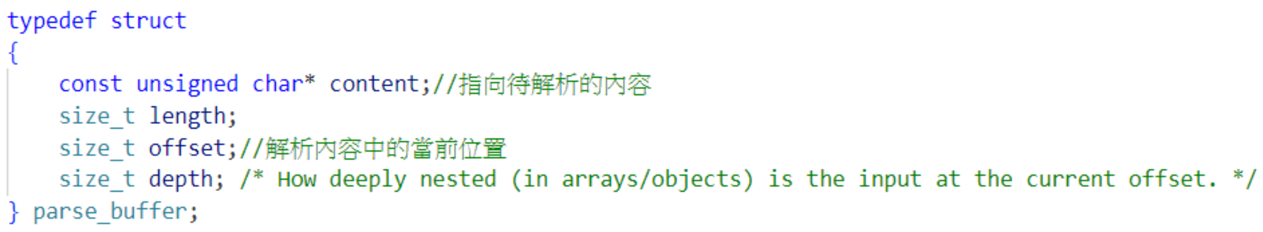
將 DEGREE 分支度大於 2 的，轉為二元樹。



```
//enumerate all situation.在後面可進行位元運算判斷格式是否正確
enum TokenizeType {
    TKE_INT = 1,
    TKE_STRING = 1 << 1,
    TKE_LEFT_BRACE = 1 << 2,
    TKE_RIGHT_BRACE = 1 << 3,
    TKE_COMMA = 1 << 4,
    TKE_COLON = 1 << 5,
};
```

圖二

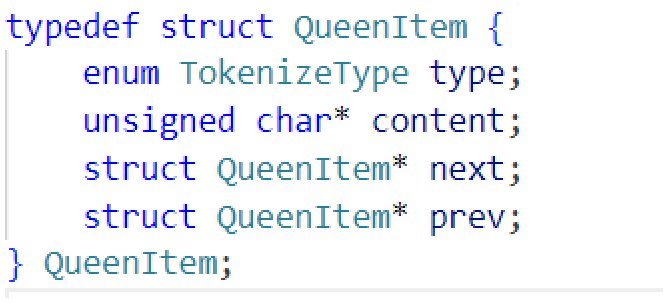
使用位元位置去判斷是否符合 json 格式。



```c
typedef struct
{
    const unsigned char* content;//指向待解析的內容
    size_t length;
    size_t offset;//解析內容中的當前位置
    size_t depth; /* How deeply nested (in arrays/objects) is the input at the current offset. */
} parse_buffer;
```

圖三

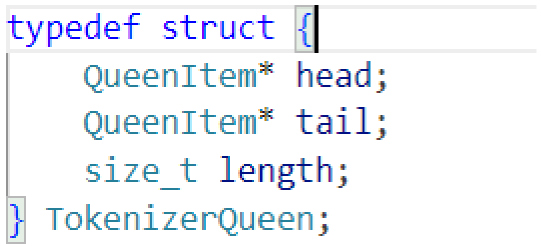
定義輸入文檔。並記錄現在解析的位置，字串長度等。



```c
typedef struct QueenItem {
    enum TokenizeType type;
    unsigned char* content;
    struct QueenItem* next;
    struct QueenItem* prev;
} QueenItem;
```

圖四

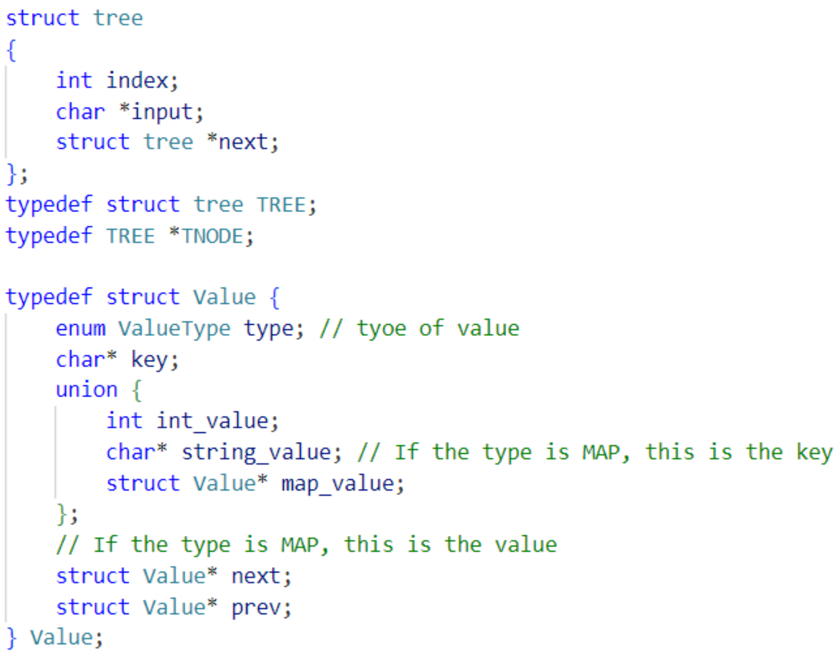
使用雙向鏈結針對不同輸入進行存儲及讀取。



```c
typedef struct {
    QueenItem* head;
    QueenItem* tail;
    size_t length;
} TokenizerQueen;
```

圖五

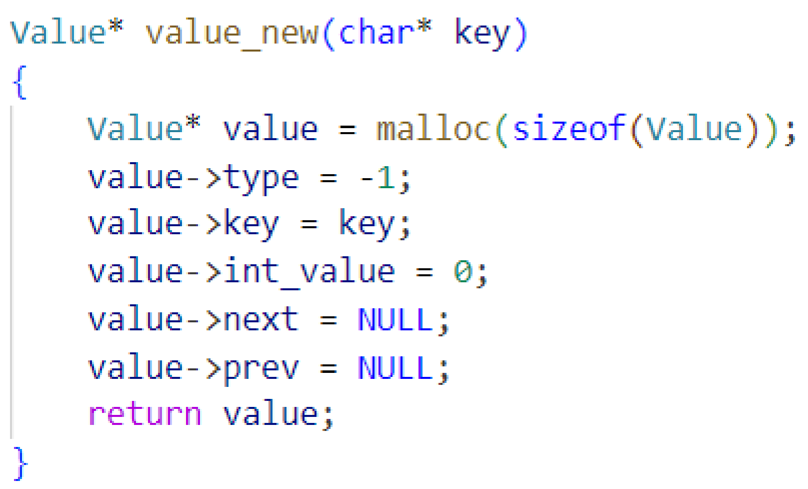
為 json 格式儲存方法。通過 doubly linked list 去建樹。結構如圖一所示。



```c
struct tree
{
    int index;
    char *input;
    struct tree *next;
};
typedef struct tree TREE;
typedef TREE *TNODE;

typedef struct Value {
    enum ValueType type; // tyoe of value
    char* key;
    union {
        int int_value;
        char* string_value; // If the type is MAP, this is the key
        struct Value* map_value;
    };
    // If the type is MAP, this is the value
    struct Value* next;
    struct Value* prev;
} Value;
```

圖六

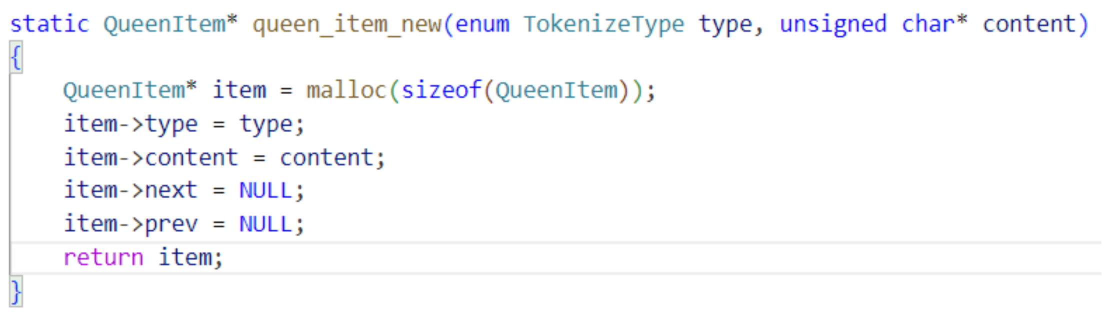
圖六為針對後續判斷格式的初始化定義。有可能是 int string 或是{}裡的 keyvalue。



```c
Value* value_new(char* key)
{
    Value* value = malloc(sizeof(Value));
    value->type = -1;
    value->key = key;
    value->int_value = 0;
    value->next = NULL;
    value->prev = NULL;
    return value;
}
```

圖七

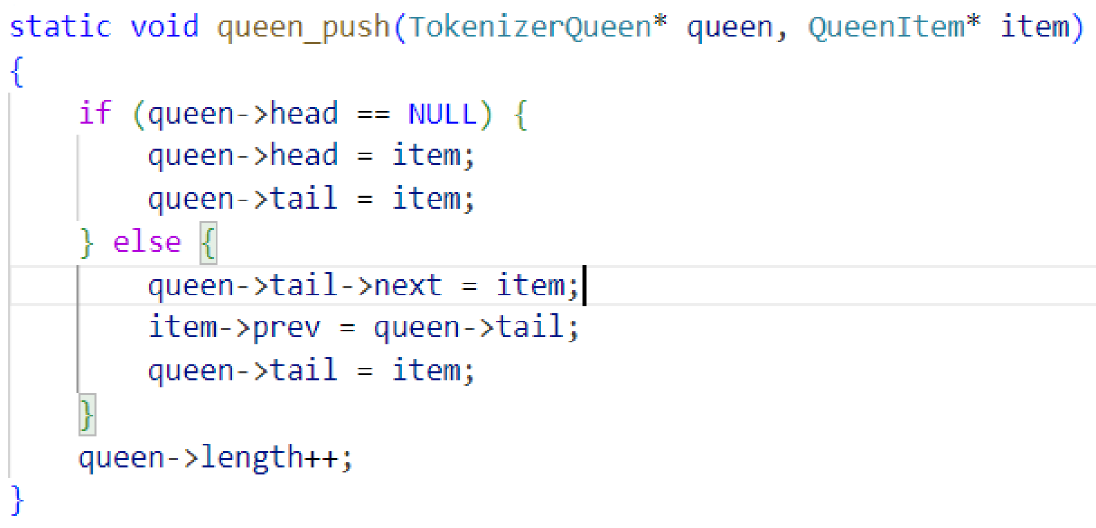
建立 value 得一個結構，並對他初始化。



```c
static QueenItem* queen_item_new(enum TokenizeType type, unsigned char* content)
{
    QueenItem* item = malloc(sizeof(QueenItem));
    item->type = type;
    item->content = content;
    item->next = NULL;
    item->prev = NULL;
    return item;
}
```

圖八

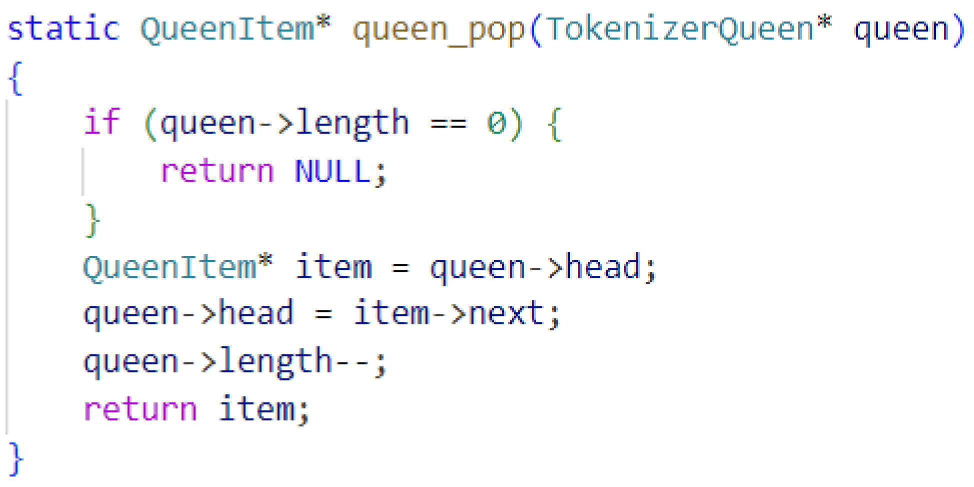
儲存輸入進來的 key or value。並定義 type。還未指向任何位置。



```c
static void queen_push(TokenizerQueen* queen, QueenItem* item)
{
    if (queen->head == NULL) {
        queen->head = item;
        queen->tail = item;
    } else {
        queen->tail->next = item;
        item->prev = queen->tail;
        queen->tail = item;
    }
    queen->length++;
}
```

圖九

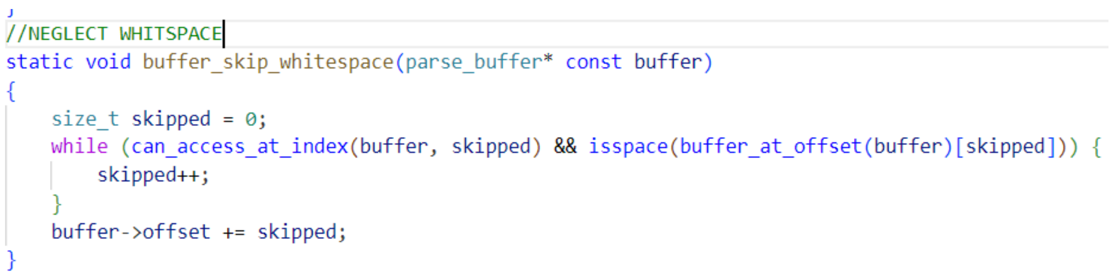
將一個新的 QueenItem 加入到 TokenizerQueen 的鏈表中



```c
static QueenItem* queen_pop(TokenizerQueen* queen)
{
    if (queen->length == 0) {
        return NULL;
    }
    QueenItem* item = queen->head;
    queen->head = item->next;
    queen->length--;
    return item;
}
```

圖十

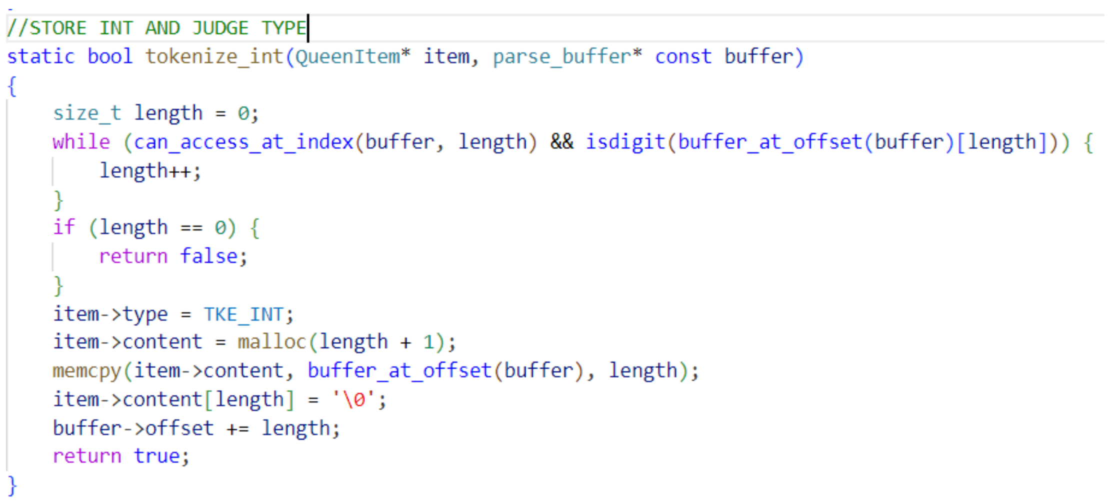
將資料從 TokenizerQueen 裡面取出。



```c
//NEGLECT WHITSPACE
static void buffer_skip_whitespace(parse_buffer* const buffer)
{
    size_t skipped = 0;
    while (can_access_at_index(buffer, skipped) && isspace(buffer_at_offset(buffer)[skipped])) {
        skipped++;
    }
    buffer->offset += skipped;
}
```

圖十一

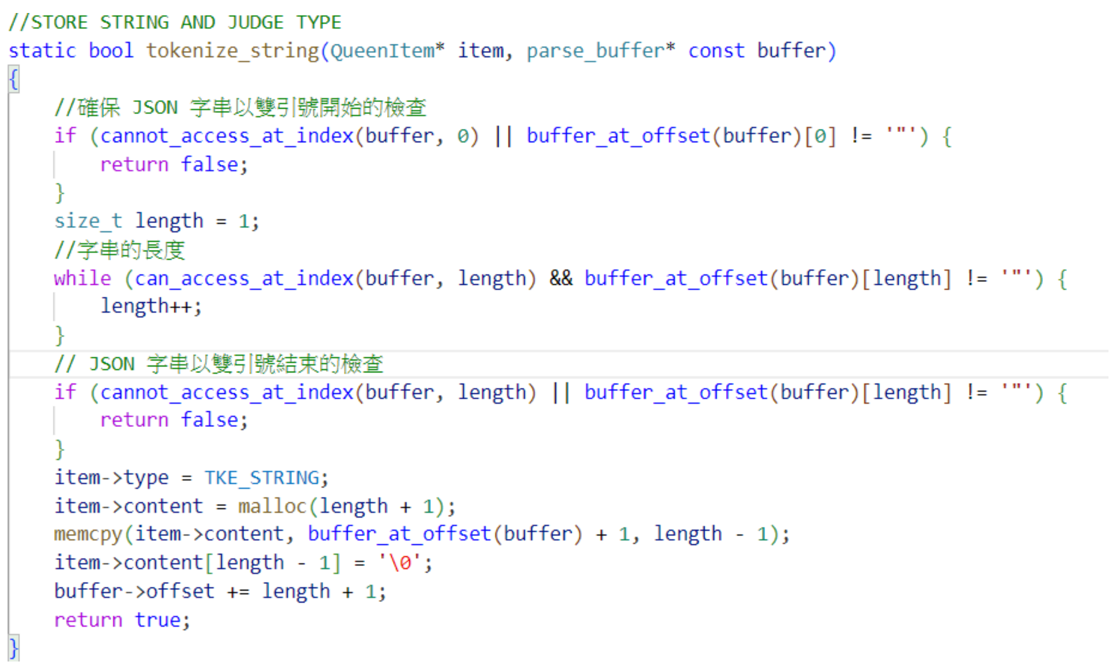
忽略空白，並 OFFSET+1。



```c
//STORE INT AND JUDGE TYPE
static bool tokenize_int(QueenItem* item, parse_buffer* const buffer)
{
    size_t length = 0;
    while (can_access_at_index(buffer, length) && isdigit(buffer_at_offset(buffer)[length])) {
        length++;
    }
    if (length == 0) {
        return false;
    }
    item->type = TKE_INT;
    item->content = malloc(length + 1);
    memcpy(item->content, buffer_at_offset(buffer), length);
    item->content[length] = '\0';
    buffer->offset += length;
    return true;
}
```

圖十二

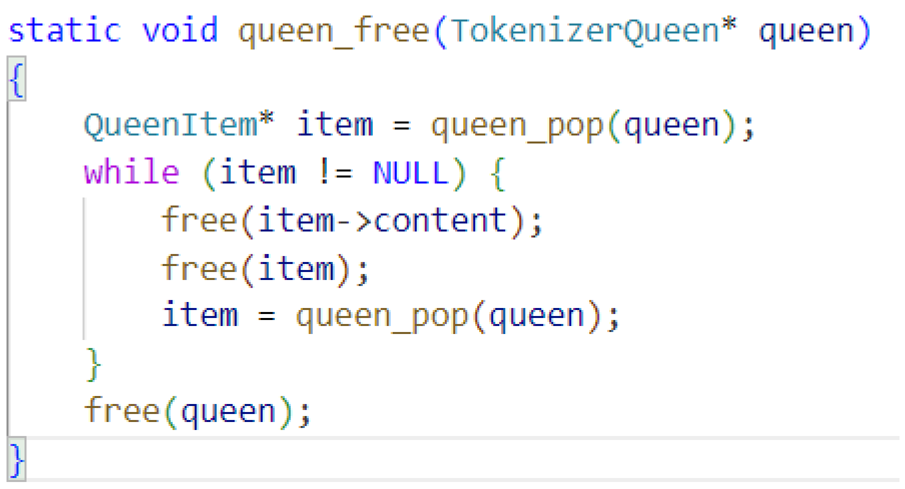
當判斷為 INT 時，分析 INT 長度(增加 OFFSET，使其可讀取下次的符號)。並將她得 TYPE 進行定義。後面找尋時會需要用到。



```c
//STORE STRING AND JUDGE TYPE
static bool tokenize_string(QueenItem* item, parse_buffer* const buffer)
{
    //確保 JSON 字串以雙引號開始的檢查
    if (cannot_access_at_index(buffer, 0) || buffer_at_offset(buffer)[0] != '"') {
        return false;
    }
    size_t length = 1;
    //字串的長度
    while (can_access_at_index(buffer, length) && buffer_at_offset(buffer)[length] != '"') {
        length++;
    }
    // JSON 字串以雙引號結束的檢查
    if (cannot_access_at_index(buffer, length) || buffer_at_offset(buffer)[length] != '"') {
        return false;
    }
    item->type = TKE_STRING;
    item->content = malloc(length + 1);
    memcpy(item->content, buffer_at_offset(buffer) + 1, length - 1);
    item->content[length - 1] = '\0';
    buffer->offset += length + 1;
    return true;
}
```

圖十三

與上述 INT 同理。但多了必須以雙引號包住字串的判斷式。

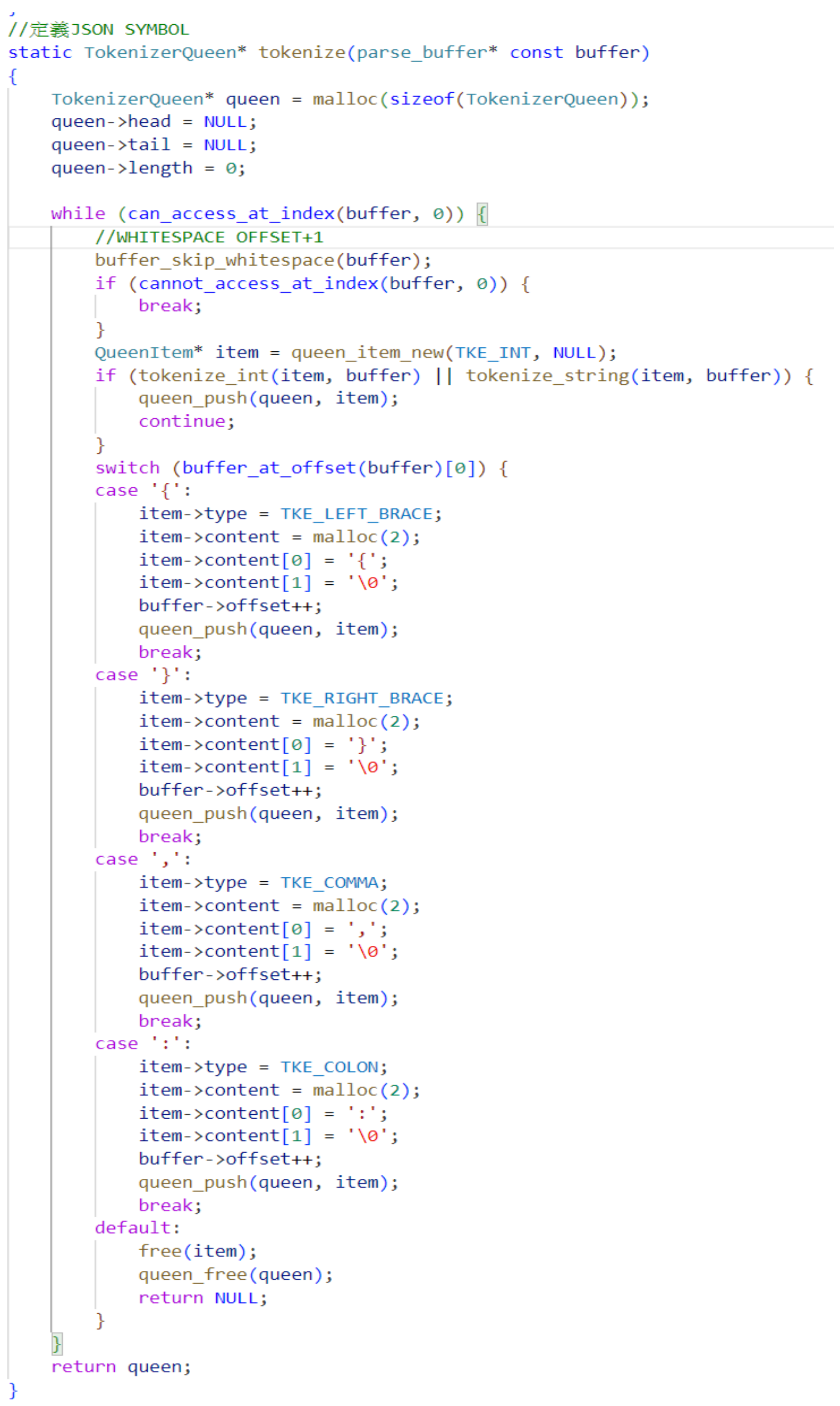


```c
static void queen_free(TokenizerQueen* queen)
{
    QueenItem* item = queen_pop(queen);
    while (item != NULL) {
        free(item->content);
        free(item);
        item = queen_pop(queen);
    }
    free(queen);
}
```

圖十四

將從 LINKED LIST 取出的資料歸還記憶體。

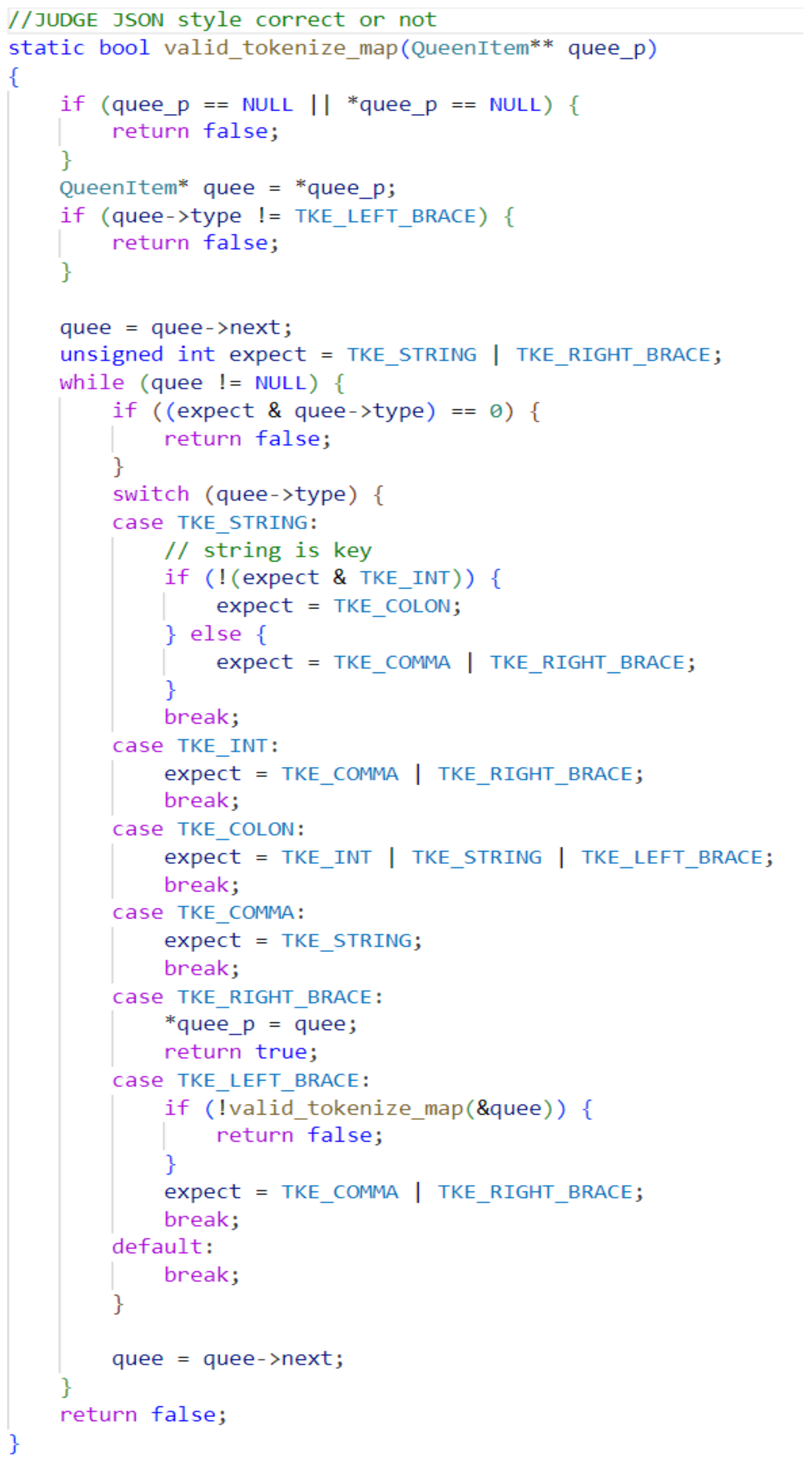
此函式主要將 SYMBOL 進行定義並存入 LINKED LIST。如果是數字 OR 字串則跳過這一次判斷。



```c
//定義JSON SYMBOL
static TokenizerQueen* tokenize(parse_buffer* const buffer)
{
    TokenizerQueen* queen = malloc(sizeof(TokenizerQueen));
    queen->head = NULL;
    queen->tail = NULL;
    queen->length = 0;

    while (can_access_at_index(buffer, 0)) {
        //WHITESPACE OFFSET+1
        buffer_skip_whitespace(buffer);
        if (cannot_access_at_index(buffer, 0)) {
            break;
        }
        QueenItem* item = queen_item_new(TKE_INT, NULL);
        if (tokenize_int(item, buffer) || tokenize_string(item, buffer)) {
            queen_push(queen, item);
            continue;
        }
        switch (buffer_at_offset(buffer)[0]) {
        case '{':
            item->type = TKE_LEFT_BRACE;
            item->content = malloc(2);
            item->content[0] = '{';
            item->content[1] = '\0';
            buffer->offset++;
            queen_push(queen, item);
            break;
        case '}':
            item->type = TKE_RIGHT_BRACE;
            item->content = malloc(2);
            item->content[0] = '}';
            item->content[1] = '\0';
            buffer->offset++;
            queen_push(queen, item);
            break;
        case ',':
            item->type = TKE_COMMA;
            item->content = malloc(2);
            item->content[0] = ',';
            item->content[1] = '\0';
            buffer->offset++;
            queen_push(queen, item);
            break;
        case ':':
            item->type = TKE_COLON;
            item->content = malloc(2);
            item->content[0] = ':';
            item->content[1] = '\0';
            buffer->offset++;
            queen_push(queen, item);
            break;
        default:
            free(item);
            queen_free(queen);
            return NULL;
        }
    }
    return queen;
}
```

圖十五

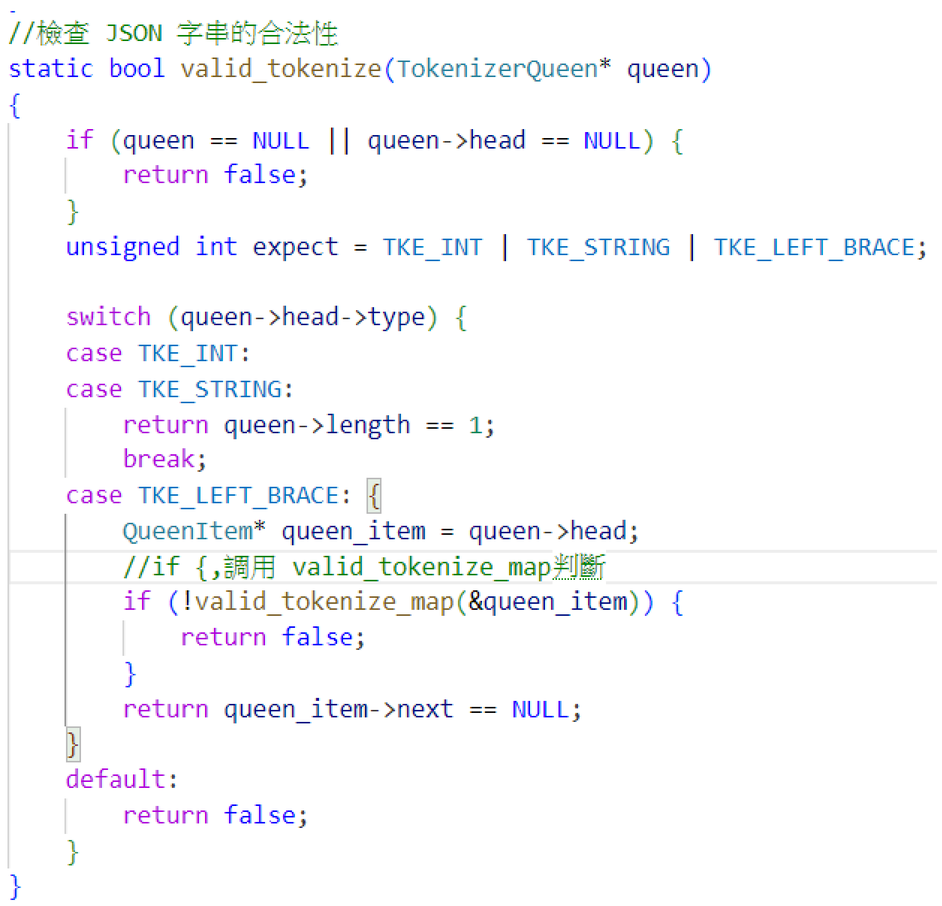
此函式主要將 SYMBOL 進行定義並存入 LINKED LIST。如果是數字 OR 字串則跳過這一次判斷。



```c
//JUDGE JSON style correct or not
static bool valid_tokenize_map(QueenItem** quee_p)
{
    if (quee_p == NULL || *quee_p == NULL) {
        return false;
    }
    QueenItem* quee = *quee_p;
    if (quee->type != TKE_LEFT_BRACE) {
        return false;
    }

    quee = quee->next;
    unsigned int expect = TKE_STRING | TKE_RIGHT_BRACE;
    while (quee != NULL) {
        if ((expect & quee->type) == 0) {
            return false;
        }
        switch (quee->type) {
        case TKE_STRING:
            // string is key
            if (!(expect & TKE_INT)) {
                expect = TKE_COLON;
            } else {
                expect = TKE_COMMA | TKE_RIGHT_BRACE;
            }
            break;
        case TKE_INT:
            expect = TKE_COMMA | TKE_RIGHT_BRACE;
            break;
        case TKE_COLON:
            expect = TKE_INT | TKE_STRING | TKE_LEFT_BRACE;
            break;
        case TKE_COMMA:
            expect = TKE_STRING;
            break;
        case TKE_RIGHT_BRACE:
            *quee_p = quee;
            return true;
        case TKE_LEFT_BRACE:
            if (!valid_tokenize_map(&quee)) {
                return false;
            }
            expect = TKE_COMMA | TKE_RIGHT_BRACE;
            break;
        default:
            break;
        }

        quee = quee->next;
    }
    return false;
}
```

圖十六

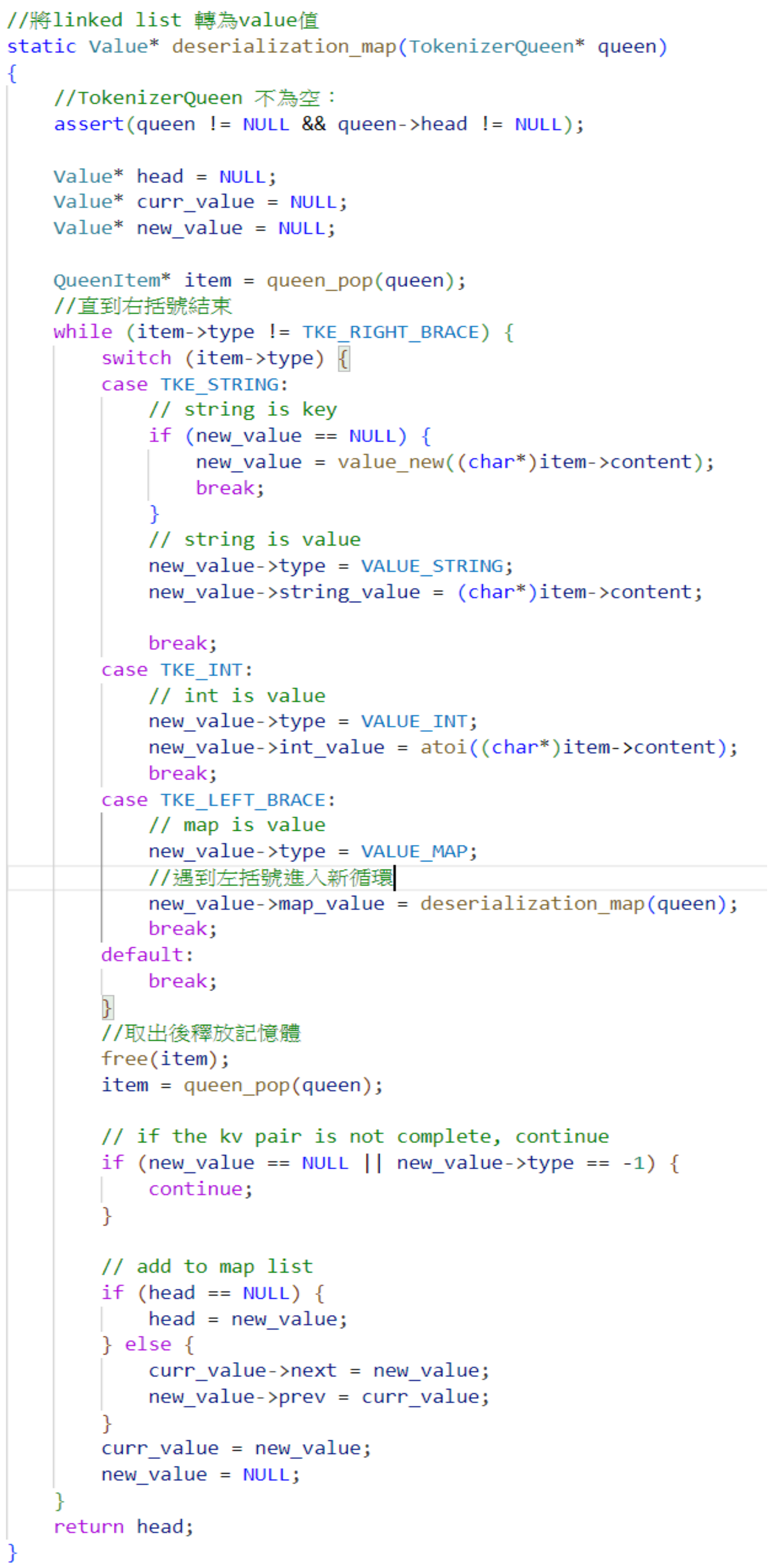
將參考資料二所寫，json 格式規則進行預測。用前面所說的位元比較來達成。如遇到” { “時，則在進入此函式一次。



```c
//檢查 JSON 字串的合法性
static bool valid_tokenize(TokenizerQueen* queen)
{
    if (queen == NULL || queen->head == NULL) {
        return false;
    }
    unsigned int expect = TKE_INT | TKE_STRING | TKE_LEFT_BRACE;

    switch (queen->head->type) {
    case TKE_INT:
    case TKE_STRING:
        return queen->length == 1;
        break;
    case TKE_LEFT_BRACE: {
        QueenItem* queen_item = queen->head;
        //if {,調用 valid_tokenize_map判斷
        if (!valid_tokenize_map(&queen_item)) {
            return false;
        }
        return queen_item->next == NULL;
    }
    default:
        return false;
    }
}
```

圖十七

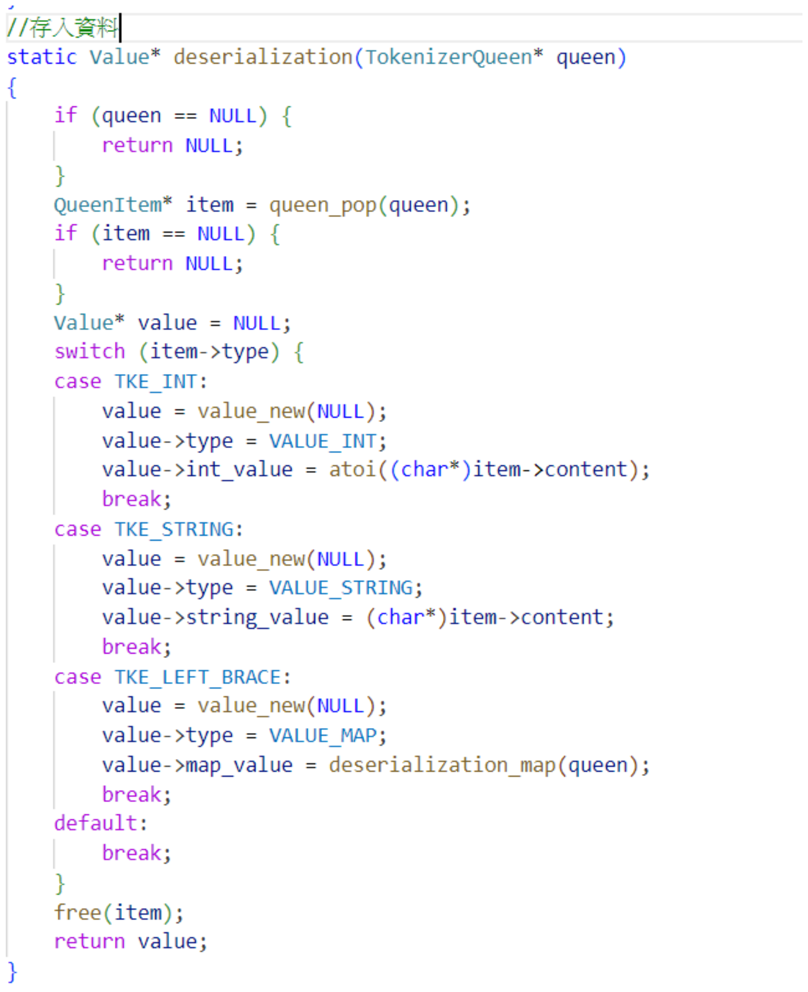
expect 初始化為 TKE_INT、TKE_STRING 和 TKE_LEFT_BRACE 這三種 Token 的組合。這些可以是 JSON 字串的開始。
如果頭部 Token 的類型是 TKE_INT 或 TKE_STRING，且 Token 的數量為 1，表示 JSON 字串有效。因為 json int string 可以直接是單個值。如果是{}包住，則有多層需要操作。



```c
//將linked list 轉為value值
static Value* deserialization_map(TokenizerQueen* queen)
{
    //TokenizerQueen 不為空：
    assert(queen != NULL && queen->head != NULL);

    Value* head = NULL;
    Value* curr_value = NULL;
    Value* new_value = NULL;

    QueenItem* item = queen_pop(queen);
    //直到右括號結束
    while (item->type != TKE_RIGHT_BRACE) {
        switch (item->type) {
        case TKE_STRING:
            // string is key
            if (new_value == NULL) {
                new_value = value_new((char*)item->content);
                break;
            }
            // string is value
            new_value->type = VALUE_STRING;
            new_value->string_value = (char*)item->content;

            break;
        case TKE_INT:
            // int is value
            new_value->type = VALUE_INT;
            new_value->int_value = atoi((char*)item->content);
            break;
        case TKE_LEFT_BRACE:
            // map is value
            new_value->type = VALUE_MAP;
            //遇到左括號進入新循環
            new_value->map_value = deserialization_map(queen);
            break;
        default:
            break;
        }
        //取出後釋放記憶體
        free(item);
        item = queen_pop(queen);

        // if the kv pair is not complete, continue
        if (new_value == NULL || new_value->type == -1) {
            continue;
        }

        // add to map list
        if (head == NULL) {
            head = new_value;
        } else {
            curr_value->next = new_value;
            new_value->prev = curr_value;
        }
        curr_value = new_value;
        new_value = NULL;
    }
    return head;
}
```

圖十八

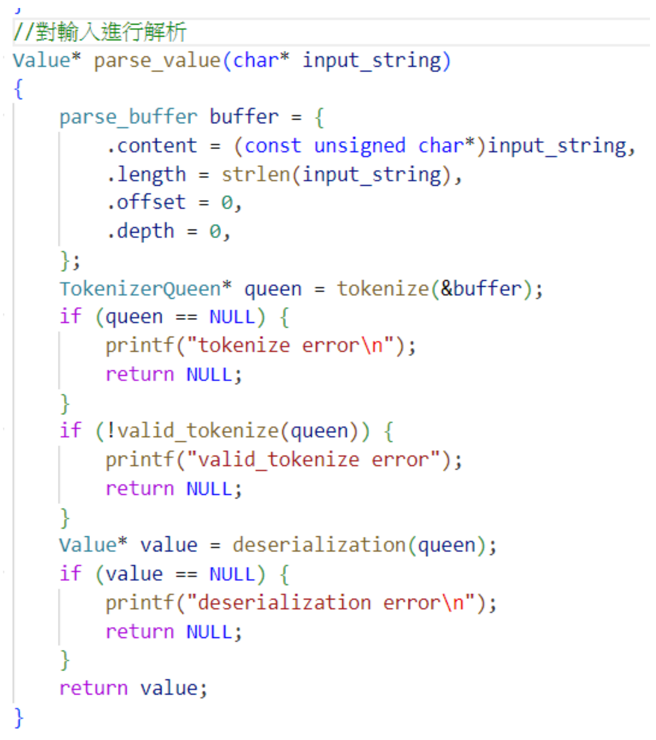
將前面所儲存 linked list 進行分析，改成 value 形式，以利於後面轉成樹。
根據 linked list 所讀取去進行存取。如果遇到” } “迴圈終止。遇到” { “，進行遞迴重新呼叫。



```
//存入資料
static Value* deserialization(TokenizerQueen* queen)
{
    if (queen == NULL) {
        return NULL;
    }
    QueenItem* item = queen_pop(queen);
    if (item == NULL) {
        return NULL;
    }
    Value* value = NULL;
    switch (item->type) {
    case TKE_INT:
        value = value_new(NULL);
        value->type = VALUE_INT;
        value->int_value = atoi((char*)item->content);
        break;
    case TKE_STRING:
        value = value_new(NULL);
        value->type = VALUE_STRING;
        value->string_value = (char*)item->content;
        break;
    case TKE_LEFT_BRACE:
        value = value_new(NULL);
        value->type = VALUE_MAP;
        value->map_value = deserialization_map(queen);
        break;
    default:
        break;
    }
    free(item);
    return value;
}
```

圖十九

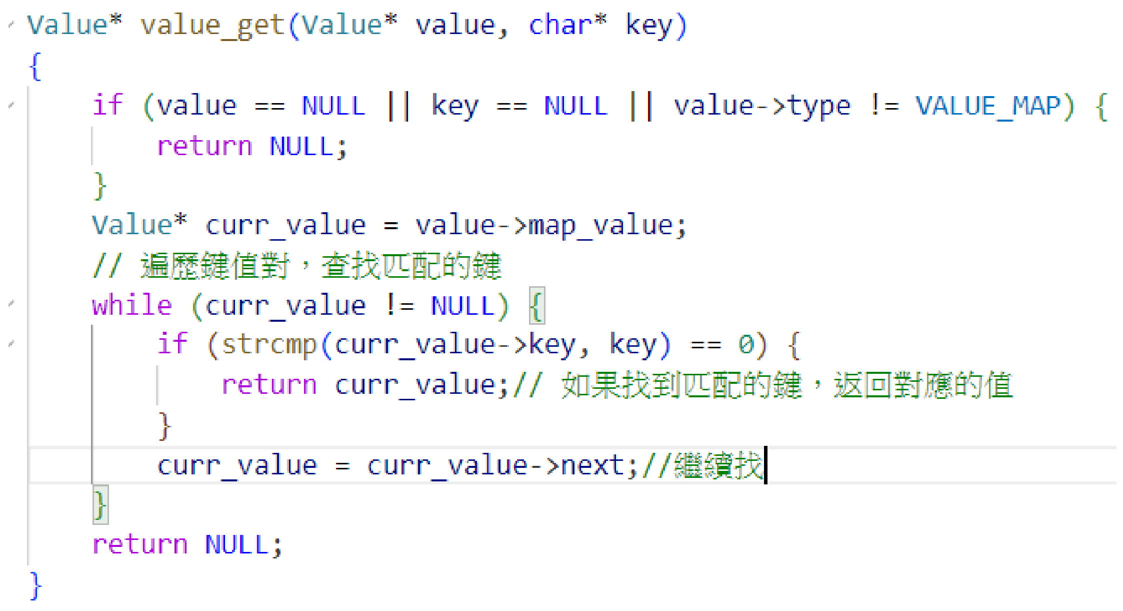
將數值通過 deserialization_map 的型態去判斷現在的格式。因為寫如時都是用 string 存，所以需要將 int 進行轉換，所以使用 atoi。
如果遇到"{"代表不只一層，所以會需要再次進行呼叫，通過遞迴完成目標。



```
//對輸入進行解析
Value* parse_value(char* input_string)
{
    parse_buffer buffer = {
        .content = (const unsigned char*)input_string,
        .length = strlen(input_string),
        .offset = 0,
        .depth = 0,
    };
    TokenizerQueen* queen = tokenize(&buffer);
    if (queen == NULL) {
        printf("tokenize error\n");
        return NULL;
    }
    if (!valid_tokenize(queen)) {
        printf("valid_tokenize error");
        return NULL;
    }
    Value* value = deserialization(queen);
    if (value == NULL) {
        printf("deserialization error\n");
        return NULL;
    }
    return value;
}
```

圖二十

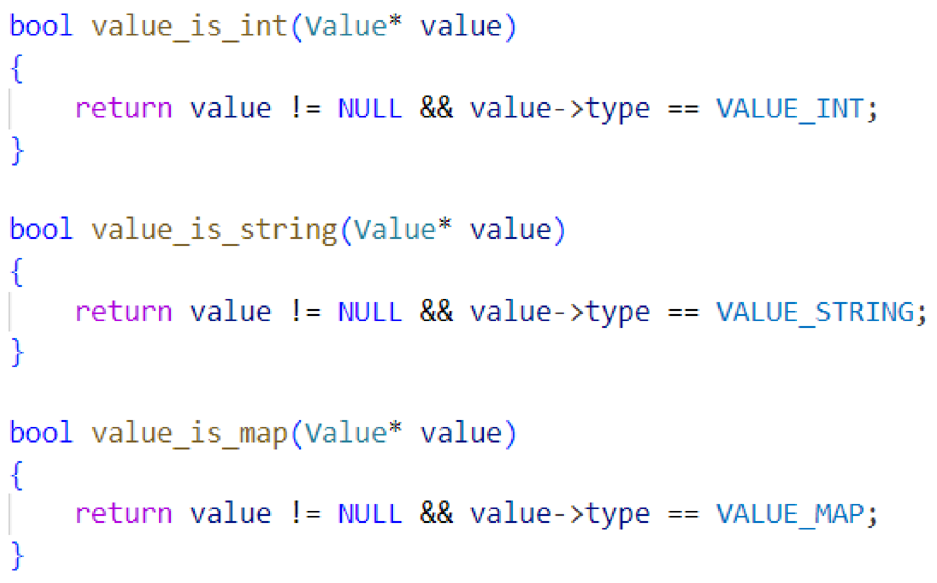
此 function 會在後續 get 樹的資料說明輸入的 json 格式哪裡錯誤。



```
Value* value_get(Value* value, char* key)
{
    if (value == NULL || key == NULL || value->type != VALUE_MAP) {
        return NULL;
    }
    Value* curr_value = value->map_value;
    // 遍歷鍵值對，查找匹配的鍵
    while (curr_value != NULL) {
        if (strcmp(curr_value->key, key) == 0) {
            return curr_value;// 如果找到匹配的鍵，返回對應的值
        }
        curr_value = curr_value->next;//繼續找
    }
    return NULL;
}
```

圖二十一

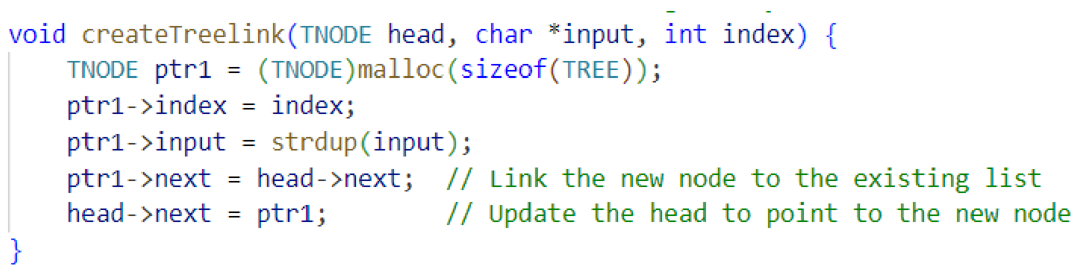
通過 linked list 遍歷，找尋符合的 key。



```
bool value_is_int(Value* value)
{
    return value != NULL && value->type == VALUE_INT;
}

bool value_is_string(Value* value)
{
    return value != NULL && value->type == VALUE_STRING;
}

bool value_is_map(Value* value)
{
    return value != NULL && value->type == VALUE_MAP;
}
```

圖二十二

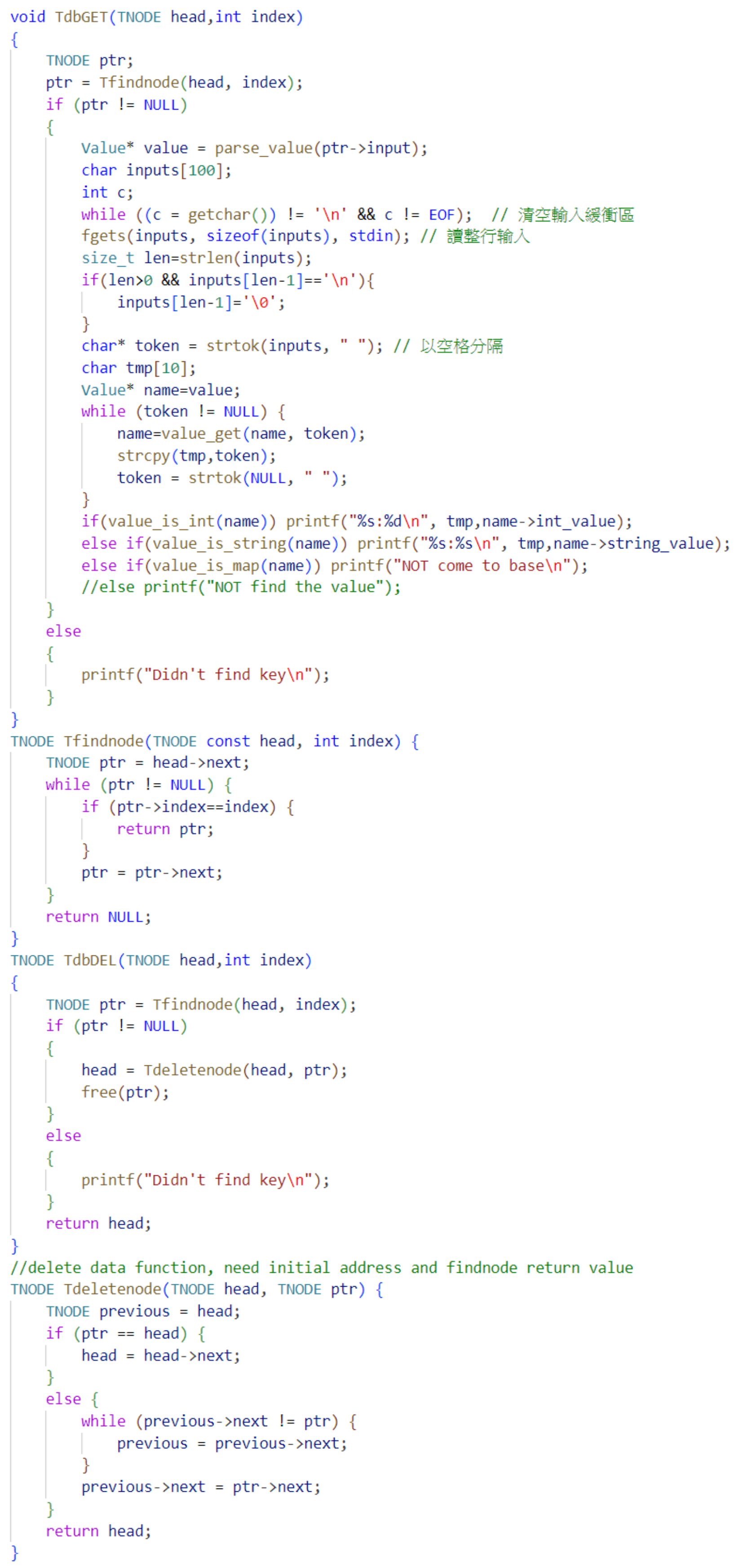
判斷是否是 int string or map。



```
void createTreelink(TNODE head, char *input, int index) {
    TNODE ptr1 = (TNODE)malloc(sizeof(TREE));
    ptr1->index = index;
    ptr1->input = strdup(input);
    ptr1->next = head->next;  // Link the new node to the existing list
    head->next = ptr1;        // Update the head to point to the new node
}
```

圖二十三

創建 linked list 存不同樹。



```c
void TdbGET(TNODE head,int index)
{
    TNODE ptr;
    ptr = Tfindnode(head, index);
    if (ptr != NULL)
    {
        Value* value = parse_value(ptr->input);
        char inputs[100];
        int c;
        while ((c = getchar()) != '\n' && c != EOF);  // 清空輸入緩衝區
        fgets(inputs, sizeof(inputs), stdin); // 讀整行輸入
        size_t len=strlen(inputs);
        if(len>0 && inputs[len-1]=='\n'){
            inputs[len-1]='\0';
        }
        char* token = strtok(inputs, " "); // 以空格分隔
        char tmp[10];
        Value* name=value;
        while (token != NULL) {
            name=value_get(name, token);
            strcpy(tmp,token);
            token = strtok(NULL, " ");
        }
        if(value_is_int(name)) printf("%s:%d\n", tmp,name->int_value);
        else if(value_is_string(name)) printf("%s:%s\n", tmp,name->string_value);
        else if(value_is_map(name)) printf("NOT come to base\n");
        //else printf("NOT find the value");
    }
    else
    {
        printf("Didn't find key\n");
    }
}
TNODE Tfindnode(TNODE const head, int index) {
    TNODE ptr = head->next;
    while (ptr != NULL) {
        if (ptr->index==index) {
            return ptr;
        }
        ptr = ptr->next;
    }
    return NULL;
}
TNODE TdbDEL(TNODE head,int index)
{
    TNODE ptr = Tfindnode(head, index);
    if (ptr != NULL)
    {
        head = Tdeletenode(head, ptr);
        free(ptr);
    }
    else
    {
        printf("Didn't find key\n");
    }
    return head;
}
//delete data function, need initial address and findnode return value
TNODE Tdeletenode(TNODE head, TNODE ptr) {
    TNODE previous = head;
    if (ptr == head) {
        head = head->next;
    }
    else {
        while (previous->next != ptr) {
            previous = previous->next;
        }
        previous->next = ptr->next;
    }
    return head;
}
```

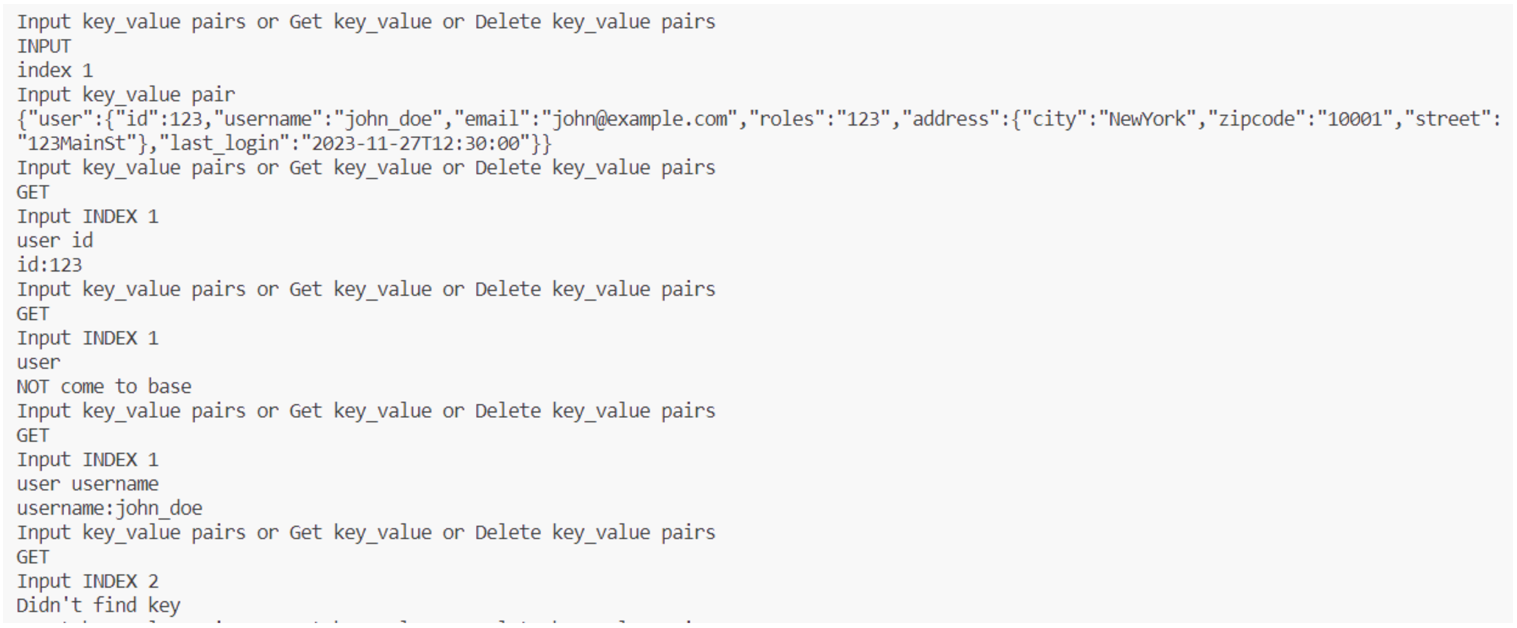
圖二十四

以上函式和 hw1,hw3 寫法相似。這裡只是 struct 定義為 TREE。TREE * TNODE。其他操作與 LINKED LIST 刪除，查找一樣。

程式輸出範例

測資一

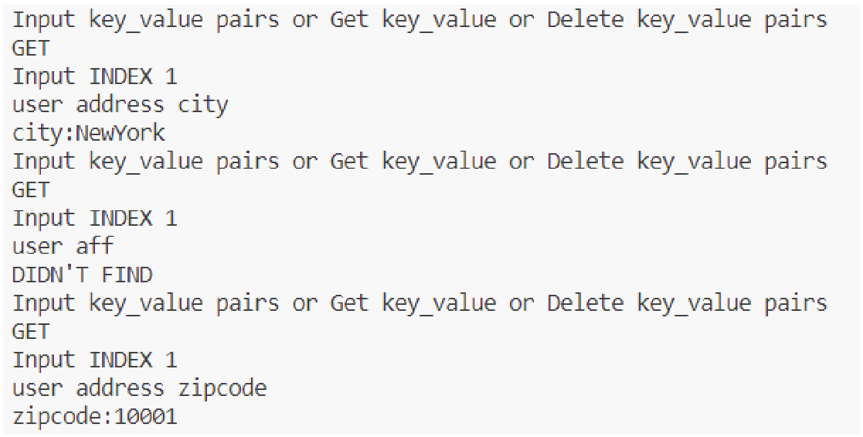
{"user":{"id":123,"username":"john_doe","email":"john@example.com","roles":"123","address":{"city":"NewYork","zipcode":"10001","street":"123MainSt"},"last_login":"2023-11-27T12:30:00"}}



```
Input key_value pairs or Get key_value or Delete key_value pairs
INPUT
index 1
Input key_value pair
{"user":{"id":123,"username":"john_doe","email":"john@example.com","roles":"123","address":{"city":"NewYork","zipcode":"10001","street":
"123MainSt"},"last_login":"2023-11-27T12:30:00"}}
Input key_value pairs or Get key_value or Delete key_value pairs
GET
Input INDEX 1
user id
id:123
Input key_value pairs or Get key_value or Delete key_value pairs
GET
Input INDEX 1
user
NOT come to base
Input key_value pairs or Get key_value or Delete key_value pairs
GET
Input INDEX 1
user username
username:john_doe
Input key_value pairs or Get key_value or Delete key_value pairs
GET
Input INDEX 2
Didn't find key
```

有三種選項。INPUT GET DEL。每個都需要你打出你想要 TREE 的位置(INDEX)。此外這裡的 DEL 是刪除整個樹，之前下課問過教授，樹要實現 JSON 刪除不好刪除，所以把整棵樹刪除即可。

從上圖可知我們先 INPUT 上面的測資。在通過 GET，一定要一直輸入直到得到唯一 KEY。第一個我們先 GET user 的 id，可以得到 123。如果只輸入 user，會輸出 not come to base，代表現在還為一對一。假如輸入 user username 可得到 john_doe。



```
Input key_value pairs or Get key_value or Delete key_value pairs
GET
Input INDEX 1
user address city
city:NewYork
Input key_value pairs or Get key_value or Delete key_value pairs
GET
Input INDEX 1
user aff
DIDN'T FIND
Input key_value pairs or Get key_value or Delete key_value pairs
GET
Input INDEX 1
user address zipcode
zipcode:10001
```

從上圖可知當輸入為錯的 key 時，會顯示 didin’t find。其他會輸出 key:value。

```
Input key_value pairs or Get key_value or Delete key_value pairs
DEL
Input INDEX 1
Input key_value pairs or Get key_value or Delete key_value pairs
GET
Input INDEX 1
Didn't find key
```

當輸入 DEL，和目前 TREE 所在 INDEX 時，會將資料刪除。
通過 GET 可得知檔案已刪除。

測資二

{"name":"zhang","age":18,"address":{"city":"TAIPEI","street":"haha"}}

```
Input key_value pairs or Get key_value or Delete key_value pairs
INPUT
index 1
Input key_value pair
{"user":{"id":123,"username":"john_doe","email":"john@example.com","roles":"123","address":{"city":"NewYork","zipcode":"10001","
street":"123MainSt"},"last_login":"2023-11-27T12:30:00"}}
Input key_value pairs or Get key_value or Delete key_value pairs
INPUT
index 2
Input key_value pair
{"name":"zhang","age":18,"address":{"city":"TAIPEI","street":"haha"}}
Input key_value pairs or Get key_value or Delete key_value pairs
GET
Input INDEX 2
name
name:zhang
Input key_value pairs or Get key_value or Delete key_value pairs
GET
Input INDEX 1
user username
username:john_doe
Input key_value pairs or Get key_value or Delete key_value pairs
DEL
Input INDEX 1
Input key_value pairs or Get key_value or Delete key_value pairs
GET 1
Input INDEX Didn't find key
Input key_value pairs or Get key_value or Delete key_value pairs
GET
Input INDEX 2
address city
city:TAIPEI
```

一次輸入兩筆資料，可發現可以同時查詢兩個 tree 的數據。並對 index=1 的 tree 刪除後再 get 無法找到。但還是可以對還未刪除 indeex=2 的樹得到其資料。
以上可知結果與預期符合。