資料結構 HW_4

110303581 陳俊瑋

- Mylib.h

下面的部分從新加入的函式開始。

```
typedef struct TreeNode {
    char key[KEY_SIZE];
    struct TreeNode** value;
    int numValues;
} TreeNode;

typedef struct TreeDatabase{
    TreeNode** tree;
    int numOfTree;
} TreeDatabase;
```

這邊為定義 tree 節點的結構。TreeNode** value 為一指標陣列，每個指標都指向一個子節點(value)。

TreeDatabase 是定義一個資料庫，擁有一指標陣列，每個指標都指向一顆樹的樹根。numOfTree 是用來追蹤樹的數量。

```
TreeDatabase* createTreeDatabase(){
    TreeDatabase* db = (TreeDatabase*) malloc(sizeof(TreeDatabase));
    db->tree = NULL;
    db->numOfTree = 0;
    return db;
}
TreeNode* createTreeNode(const char* key) {
    TreeNode* node = (TreeNode*)malloc(sizeof(TreeNode));
    strncpy(node->key, key, KEY_SIZE);
    node->value = NULL;
    node->numValues = 0;
    return node;
}
```

createTreeDatabase 函式是將資料庫初始化。

而 createTreeNode 函式是創建一個節點。

```c
//創建樹根
void makeATree(TreeDatabase* db, const char* key) {
    TreeNode* tree = createTreeNode(key);
    db->tree = (TreeNode**)realloc(db->tree, (db->numOfTree + 1) *
sizeof(TreeNode*));
    db->tree[db->numOfTree] = tree;
    db->numOfTree++;
    printf("Tree of [%s] has made.\n", key);
}
```

此函式為創建一樹的樹根，創建一節點之後用 `realloc` 函式將資料庫的指標陣列(指向多個樹)擴大。

```c
// 加節點_1
void addTreeNode_1(TreeDatabase* db, const char* treeName, const char*
value) {
    TreeNode* root = NULL;
    for(int i = 0; i < db->numOfTree; i++){
        if(strcmp(db->tree[i]->key, treeName) == 0){    //找到該樹
            // printf("Found the Tree.\n");
            root = db->tree[i];
            break;
        }
    }
    if(root == NULL){
        printf("Cannot find the Tree.\n");
        return;
    }

    TreeNode* parent = root;
    TreeNode* child = createTreeNode(value);

    if (parent->value == NULL) {
        // 初始化 value
        parent->value = (TreeNode**)malloc(sizeof(TreeNode*));
        parent->value[0] = child;
    }
    else {
        parent->value = (TreeNode**)realloc(parent->value, (parent-
>numValues + 1) * sizeof(TreeNode*));
        parent->value[parent->numValues] = child;
    }

    // 更新值的数量
    parent->numValues++;
    printf("[%s] has add in [%s]\n", value, treeName);
}
```

此函式為加一節點成為樹根的子節點。因為我將整棵樹的輸出分開，一次增加

一節點，又因為增加節點有兩種可能，一為加在樹根下方成為樹根子節點:

```
Enter a command: maketree Class:A
Tree of [Class:A] has made.
Enter a command: addtreenode1 Class:A Name:Mike
[Name:Mike] has add in [Class:A]
```

此情況為 Name:Mike 加在 Class:A 這樹根的下方，成為樹根 Class:A 的子節點。addTreeNode_1 為處理這情況的函式。

而第二種情況為此樹中尋找 key，加在該 key 的下方，成為其子節點:

```
Enter a command: addtreenode2 Class:A Name:Mike Number:01
[Number:01] has add in [Name:Mike]
```

此情況為 Number:01 加在 Name:Mike 這 key 的下方，成為節點 Name:Mike 的子節點。addTreeNode_2 為處理這情況的函式。

回到 addTreeNode_1，先用 for 迴圈找到樹之後，將 parent 訂為樹根，child 訂為要插入之子節點，如果原本 parent 沒有子節點，將 child 定為第一個子節點；反之，重新定義 parent 的 value 的空間，將 child 放在最後。

```
// 加節點_(遞迴)
void addTreeNodeRecursive(TreeNode* parent, const char* nodeName, const
char* value) {
    if (strcmp(parent->key, nodeName) == 0) {
        // 找到指定的節點，將值添加到這個節點下
        TreeNode* child = createTreeNode(value);

        if (parent->value == NULL) {
            // 初始化 value
            parent->value = (TreeNode**)malloc(sizeof(TreeNode*));
            parent->value[0] = child;
        }
        else {
            parent->value = (TreeNode**)realloc(parent->value, (parent-
>numValues + 1) * sizeof(TreeNode*));
            parent->value[parent->numValues] = child;
        }

        // 更新值的数量
        parent->numValues++;
        printf("[%s] has add in [%s]\n", value, nodeName);
        return;
    }

    // 在子樹中遞迴搜索
```

```c
    for (int i = 0; i < parent->numValues; i++) {
        addTreeNodeRecursive(parent->value[i], nodeName, value);
    }
}
// 加節點_2
void addTreeNode_2(TreeDatabase* db, const char* treeName, const char*
nodeName, const char* value) {
    TreeNode* root = NULL;
    for (int i = 0; i < db->numOfTree; i++) {
        if (strcmp(db->tree[i]->key, treeName) == 0) {
            // printf("Found the Tree.\n");
            root = db->tree[i];
            addTreeNodeRecursive(root, nodeName, value);
            return;
        }
    }

    // 若未找到樹
    printf("Cannot find the Tree.\n");
}
```

此函式為剛剛所提到的第二種情況的加節點。先用 `addTreeNode_2` 函式找尋樹，把該樹的指標傳給 `addTreeNodeRecursive` 遞迴函式，去找尋對應的 key 值。之後在對應之 key 值的下方加上子節點(value)，方法與上個函式類似。

```c
// 列印節點(遞迴)
void printTreeRecursive(TreeNode* node, int numOfSpace) {
    if (node != NULL) {
        printf("%*sKey: (%s)\n", numOfSpace, " ", node->key);
        for (int i = 0; i < node->numValues; i++) {
            // printf("numValues:%d\n", node->numValues);
            printf("%*sValue: (%s)\n", numOfSpace+4, " ", node-
>value[i]->key);
            printTreeRecursive(node->value[i], numOfSpace+4);  // 遞迴印
出子樹
        }
    }
}
// 列印節點
void printTree(TreeDatabase* db, const char* treeName) {
    for (int i = 0; i < db->numOfTree; i++) {
        if (strcmp(db->tree[i]->key, treeName) == 0) {
            TreeNode* node =  db->tree[i];
            printTreeRecursive(node, 0);
        }
    }
}
```

這兩個函式是印出整個 tree 的所有。先用 `printTree` 函式找尋樹，把該樹的指標傳給 `printTreeRecursive` 遞迴函式。印出的順序類似前序，先印 key 值，在找其 value，如果 value 下面還有，則繼續打印 value 的 value。而當該節點沒有子節點時，是還會在印出 `Key: (%s)\n` 這行的(後面有範例)。

而 `printf("%*sKey: (%s)\n", numOfSpace, " ", node->key);`是前面的`%*s` 是每次遞迴時增加前面印出的空格，以便閱讀。 *為空格的數量，也就是 `numOfSpace`。

```
// 找尋樹節點(遞迴)
TreeNode* findTreeNodeRecursive(TreeNode* root, const char* nodeName) {
    if (root == NULL) {
        return NULL;
    }

    if (strcmp(root->key, nodeName) == 0) { //如果本節點是要找到節點
        return root;
    }

    // 該節點不是要找的節點
    for (int i = 0; i < root->numValues; i++) { //看下一個 value
        TreeNode* foundNode = findTreeNodeRecursive(root->value[i],
nodeName); //看其子樹(value)
        if (foundNode != NULL) {
            return foundNode;
        }
    }

    return NULL;
}
// 找尋樹節點
void findTreeNode(TreeDatabase* db, const char* treeName,  const char*
nodeName) {
    TreeNode* root = NULL;
    TreeNode* node = NULL;
    for (int i = 0; i < db->numOfTree; i++) {
        if(strcmp(db->tree[i]->key, treeName) == 0){    //找到該樹
            // printf("Found the Tree.\n");
            root = db->tree[i];
            node = findTreeNodeRecursive(root, nodeName);
            break;
        }
    }
    if (node != NULL) {
```

```
        printf("Values under node [%s]:\n", nodeName);
        for (int j = 0; j < node->numValues; j++) {
            printf("  (%s)\n", node->value[j]->key);
        }
        return;
    }

    printf("Node [%s] not found.\n", nodeName);
}
```

這函式是找尋節點並將其所有子節點(value)都列印出來，一樣也是先用 `findTreeNode` 函式找尋樹，把該樹的指標傳給 `findTreeNodeRecursive` 遞迴函式去做找尋。

```
void freeNode(TreeNode* node) {
    if (node == NULL) {
        return;
    }

    // 遞迴釋放子節點的內存
    for (int i = 0; i < node->numValues; i++) {
        freeNode(node->value[i]);
    }

    // 釋放節點自身的內存
    free(node->value);
    free(node);
}
```

此函式為釋放單個節點的函式，後面的函式會用到所以先定義。

```
// 刪除樹節點(遞迴)
void deleteNodeRecursive(TreeNode* root, const char* nodeName) {
    if (root == NULL) {
        return;
    }

    // 在 value 中遞迴搜索
    for (int i = 0; i < root->numValues; i++) {
        if (strcmp(root->value[i]->key, nodeName) == 0) {   // 找到該節點
            // 刪除該節點及所有 value
            freeNode(root->value[i]);
            // 將其他 value 往前搬
            for (int j = i; j < root->numValues - 1; j++) {
                root->value[j] = root->value[j + 1];
            }
            root->numValues--;
```

```c
            root->value = (TreeNode**)realloc(root->value, root-
>numValues * sizeof(TreeNode*));
            return;
        }
        else {// 找不到該節點
            // 在 value 中繼續遞迴搜索
            deleteNodeRecursive(root->value[i], nodeName);
        }
    }
}
// 刪除樹節點
void deleteTreeNode(TreeDatabase* db, const char* treeName, const char*
nodeName) {
    for (int i = 0; i < db->numOfTree; i++) {
        TreeNode* root = db->tree[i];
        if (strcmp(root->key, treeName) == 0) { //找到該樹
            printf("Found the Tree.\n");
            deleteNodeRecursive(root, nodeName);
            printf("Node [%s] and its values have been deleted.\n",
nodeName);
            return;
        }
    }

    printf("Cannot find the Tree.\n");
}
```

此函式為刪除一個節點，用的方法與之前找尋節點的方法相同。一樣是先用 `deleteTreeNode` 函式找尋樹，把該樹的指標傳給 `deleteNodeRecursive` 遞迴函式去找尋。找到之後用剛剛定義的 `freeNode` 函式刪除該節點，並且將子節點(value)往前搬。

```c
void freeTreeDatabase(TreeDatabase* db) {
    if (db == NULL) {
        return;
    }

    // 釋放每個樹的內存
    for (int i = 0; i < db->numOfTree; i++) {
        freeNode(db->tree[i]);
    }

    // 釋放樹陣列的內存
    free(db->tree);

    // 釋放資料庫結構本身的內存
    free(db);
```

```
}
```

此函式為釋放整個 Tree 的資料庫，放在程式最後釋回空間以結束程式。

輸出結果:

```
KEY:(get/set/update/del)
LIST:(lpush/rpush/lpop/rpop/llen/lrange)
TREE:(maketree/addtreenode/printtree)
EXIT:0
Enter a command: maketree Class:A
Tree of [Class:A] has made.
Enter a command: addtreenode1 Class:A Name:Mike
[Name:Mike] has add in [Class:A]
Enter a command: addtreenode2 Class:A Name:Mike Number:01
[Number:01] has add in [Name:Mike]
Enter a command: addtreenode2 Class:A Name:Mike Score:100
[Score:100] has add in [Name:Mike]
Enter a command: addtreenode1 Class:A Name:Jason
[Name:Jason] has add in [Class:A]
Enter a command: addtreenode2 Class:A Name:Jason Number:02
[Number:02] has add in [Name:Jason]
Enter a command: addtreenode2 Class:A Name:Jason Score:90
[Score:90] has add in [Name:Jason]
Enter a command: printtree Class:A
 Key: (Class:A)      //Class:A 的節點有 Name:Mike 與 Name:Jason
    Value: (Name:Mike)
    Key: (Name:Mike)      //Name:Mike 的節點有 Number:01 與 Score:100
        Value: (Number:01)
        Key: (Number:01)
        Value: (Score:100)
        Key: (Score:100)  //雖然 Score:100 後面沒東西但還是印出 Key:
    Value: (Name:Jason)
    Key: (Name:Jason)
        Value: (Number:02)
        Key: (Number:02)
        Value: (Score:90)
        Key: (Score:90)
Enter a command: maketree Class:B     //也可以創建另一個 tree
Tree of [Class:B] has made.
Enter a command: addtreenode1 Class:B Name:Wei
[Name:Wei] has add in [Class:B]
Enter a command: addtreenode1 Class:B Name:Una
[Name:Una] has add in [Class:B]
Enter a command: printtree Class:B
 Key: (Class:B)
    Value: (Name:Wei)
    Key: (Name:Wei)
    Value: (Name:Una)
    Key: (Name:Una)
```

```
Enter a command: findtreenode Class:A Name:Mike  //找尋 Name:Mike 其 value
Values under node [Name:Mike]:
  (Number:01)
  (Score:100)
Enter a command: deltreenode Class:A Name:Mike //刪除 Name:Mike 和其子節點
Found the Tree.
Node [Name:Mike] and its values have been deleted.
Enter a command: findtreenode Class:A Name:Mike
Node [Name:Mike] not found.
Enter a command: 0
END
```