Hash Table：Chaining

Posted by Chiu CC (http://alrightchiu.github.io/SecondRound/author/chiu-cc.html) on 5 25, 2016

先備知識與注意事項

本篇文章將延續Hash Table：Intro(簡介) (http://alrightchiu.github.io/SecondRound/hash-tableintrojian-jie.html)的議題，介紹**Chaining**來解決**Collision**。

其中將會用到Linked list的概念，若不熟悉請參考：

- Linked List: Intro(簡介) (http://alrightchiu.github.io/SecondRound/linked-list-introjian-jie.html)
- Linked List: 新增資料、刪除資料、反轉 (http://alrightchiu.github.io/SecondRound/linked-list-xin-zeng-zi-liao-shan-chu-zi-liao-fan-zhuan.html)

目錄

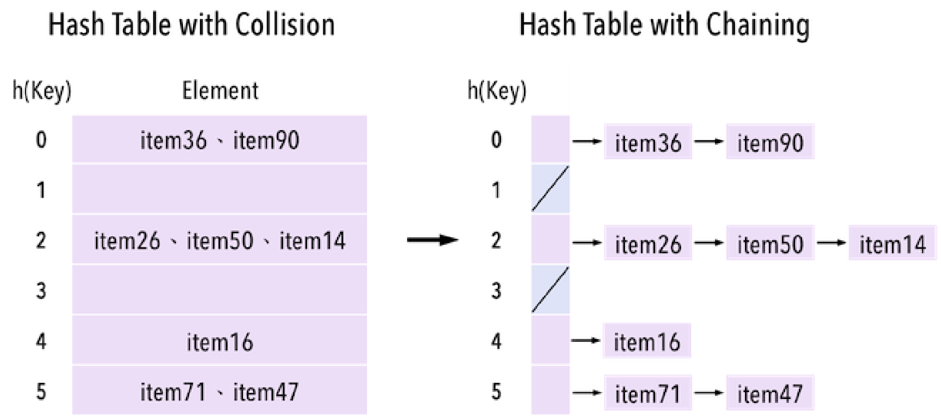
Chaining的概念

如果利用Division Method (http://alrightchiu.github.io/SecondRound/hash-tableintrojian-jie.html#dm)實作Hash Function：

$$h(Key) = Key \bmod m$$

若選擇$m = 6$，那麼對於**Key**為$14, 16, 26, 36, 47, 50, 71, 90$的item，進行**Hashing**後將會有如圖一的**Collision**發生。

解決的辦法，就是將被分配到同一個slot的item用**Linked list**串起來，這就是**Chaining**。



圖一：。

有了Linked list處理被分配到同一個slot的item，Hash Table的三項資料處理分別修正成：

Insert：

- 先利用Hash Function取得**Table**的**index**。
- 接著，只要在每一個slot的list之**front**加入item，即可保證在$O(1)$的時間複雜度完成。
  - 參考：Linked list:push_front() (http://alrightchiu.github.io/SecondRound/linked-list-xin-zeng-zi-liao-shan-chu-zi-liao-fan-zhuan.html#front)

Search：

- 先利用Hash Function取得**Table**的**index**。
- 再利用Linked list的**traversal**尋找item。

Delete：

- 先利用Hash Function取得**Table**的**index**。
- 再利用Linked list的**traversal**尋找欲刪除的item。

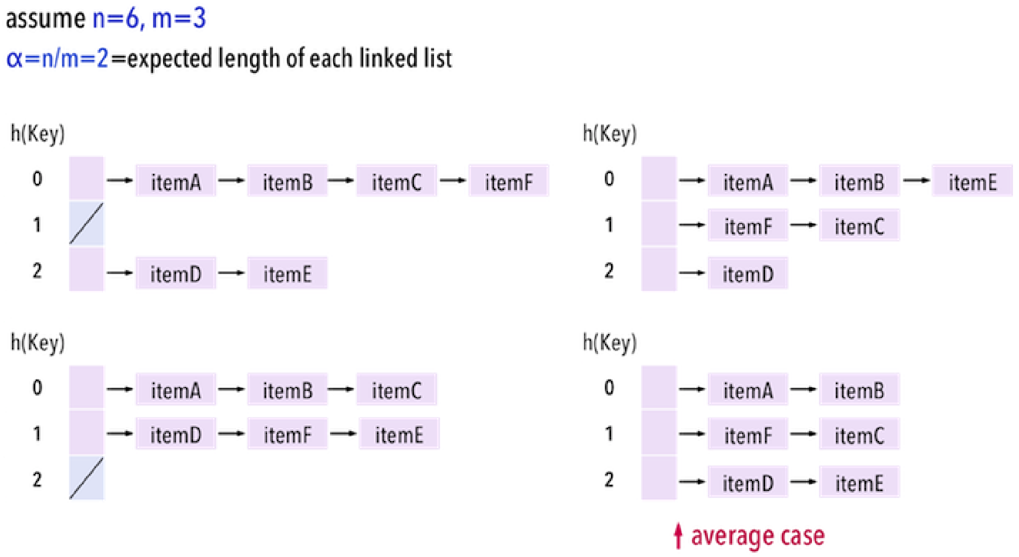
關於**Search**與**Delete**的時間複雜度：

worst case：$O(n)$，所有item都被很遜的Hash Function分配到同一個slot。

average case：$O(1+\alpha)$，其中$\alpha=\frac{n}{m}$稱為**load factor**，其物理意義為：

- 「資料數量(n)」與「slot個數(m)」的比例。
- 也會是list的**expected length**(list的平均長度)。

以圖二為例，若$m=3, n=6$，那麼$\alpha=n\backslash\mathrm{m}=2$，也就是「平均」每個list會被分配到$\alpha=2$個item，所以**Search**的時間複雜度就是list長度$O(\alpha)$，再加上Hash Function的時間複雜度$O(1)$，便得到$O(1+\alpha)$。



圖二：以上四種皆為可能的情形，其中右下圖為平均的/期望值結果。

若「資料數量(n)」與「slot個數(m)」的比例具有$n = O(m)$的關係，再加上一個不會把所有item分配到同一個slot的正常Hash Function，那麼可以想像，**Search**與**Delete**的時間可以在接近

$$O(1+\alpha) = O(1+constant) = O(1)$$

的情況下完成。
詳細證明請參考：Adnan Aziz：Hash Tables (http://users.ece.utexas.edu/~adnan/360C/hash.pdf)

程式碼

以下提供兩份基本的Hash Table實作方法：

第一份用標準模板函式庫(STL)的 `std::vector<std::list<struct>>` 處理Hash Table和**Chaining**。
重點放在： `Insert()` 、 `Delete()` 、 `Search()` 與 `Prehashing()` 上。

第二份很老實地用**pointer**串出Linked list，重點將放在 `TableDoubling()` 、 `TableShrinking()` 與 `Rehashing()` 上。

偷懶：使用STL

以下範例程式碼包含了：

`struct dict` 為自定義的**dictionary**，其中 `key` 與 `value` 皆為 `std::string` 。

`class HashChain_std` 表示以 `std::vector<std::list<dict>>` 建立Hash Table，其中包含了：

- `Insert()` 、 `Delete()` 、 `Search()` 、 `DisplayTable()` 等基本函式。
  - 涉及Linked list中的 `push_front()` 與**traversal**。
  - **traversal**以 `std::list::iterator` 處理。
- `PreHashing()` ：目的是把資料形態為 `string` 的key轉換成 `int` 。範例程式定義成：
  - 挑選一個正整數作為「底數(a)」(稍後將進行指數運算)，將字串先經由ASCII編碼轉換成正整數，再乘上a^{index}。

  - 若字串 `key` 為「Jordan」，$a = 9$，便轉換出
    $ASCII(J) \times 9^5 + ASCII(o) \times 9^4 + ASCII(r) \times 9^3 + ASCII(d) \times 9^2 + ASCII(a) \times 9^1 + ASCII(n) \times 9^0$
    $= 74 \times 9^5 + 111 \times 9^4 + 114 \times 9^3 + 100 \times 9^2 + 97 \times 9 + 110$
    $= 5190086$
- `HashFunction()`：此處使用**Division method**，將 `PreHashing()` 得到的整數除以Table大小(m)後取餘數。

`main()` 中以「假設NBA官網要建立球員資料，記錄每個球員所屬的球隊」為例示範Hash Table應用。

```
// C++ code
#include <iostream>
#include <vector>
#include <list>
#include <string>

using std::vector;
using std::list;
using std::string;
using std::cout;
using std::endl;

struct dict{                        // self-defined dictionary
    string key;                     //  key  for Name (eg:Jordan)
    string value;                   // value for Team (eg:Bulls)
    dict():key(""),value(""){};
    dict(string Key, string Value):key(Key),value(Value){};
};

class HashChain_std{
private:
    int size,                // size of table
        count;               // count: number of data

    vector<list<dict> > table;            // hash table with linked list

    int PreHashing(string key_str);       // turn string_type_key to int_type_key
    int HashFunction(string key_str);     // using Division method

public:
    HashChain_std(){};
    HashChain_std(int m):size(m),count(0){
        table.resize(size);               // allocate memory for each slot
    }

    void Insert(dict data);
    void Delete(string key);
    string Search(string key);
    void DisplayTable();
};

string HashChain_std::Search(string key_str){
    // two steps: 1. get index from hash function
    //            2. traversal in linked list
    int index = HashFunction(key_str);
    for (list<dict>::iterator itr = table[index].begin(); itr != table[index].end(); itr++) {
        if ((*itr).key == key_str) {
            return (*itr).value;
        }
    }
    return "...\nno such data";
}

void HashChain_std::Delete(string key_str){
    // two steps: 1. get index from hash function
    //            2. traversal in linked list
    int index = HashFunction(key_str);
    for (list<dict>::iterator itr = table[index].begin(); itr != table[index].end(); itr++) {
        if ((*itr).key == key_str) {
            table[index].erase(itr);
        }
    }
}

void HashChain_std::Insert(dict data){
    // two steps: 1. get index from hash function
    //            2. insert data at the front of linked list
    int index = HashFunction(data.key);
    table[index].push_front(data);
}

int HashChain_std::PreHashing(string key_str){
    // if   key_str = Jordan, exp = 9
    // then key_int = ASCII(J)*9^5+ASCII(o)*9^4+ASCII(r)*9^3
    //                +ASCII(d)*9^2+ASCII(a)*9^1+ASCII(n)*9^0

    int exp = 9,        // choose randomly
        key_int = 0,
        p = 1;
```

```
    for (int i = (int)key_str.size()-1; i >= 0; i--) {
        key_int += key_str[i]*p;
        p *= exp;
    }
    return key_int;
}

int HashChain_std::HashFunction(string key_str){

    return (PreHashing(key_str) % this->size);     // Division method
}

void HashChain_std::DisplayTable(){

    for (int i = 0; i < table.size(); i++) {
        cout << "slot#" << i << ": ";
        for (list<dict>::iterator itr = table[i].begin(); itr != table[i].end(); itr++) {
            cout << "(" << (*itr).key << "," << (*itr).value << ") ";
        }
        cout << endl;
    }
    cout << endl;
}

int main() {

    HashChain_std hash(5);
    hash.Insert(dict("T-Mac","Magic"));
    hash.Insert(dict("Bryant","Lakers"));
    hash.Insert(dict("Webber","Kings"));
    hash.Insert(dict("Arenas", "Wizards"));
    hash.Insert(dict("Davis","Clippers"));
    hash.Insert(dict("Kidd","Nets"));
    hash.DisplayTable();

    cout << "T-Mac is in " << hash.Search("T-Mac") << ". " << endl;
    cout << "Arenas is in " << hash.Search("Arenas") << ". " << endl;

    hash.Delete("Kidd");
    hash.Delete("T-Mac");
    cout << "\nAfter deleing Kidd and T-Mac:\n";
    hash.DisplayTable();

    return 0;
}
```

output:

```
slot#0: (Kidd,Nets) (Bryant,Lakers)
slot#1: (Arenas,Wizards)
slot#2: (Webber,Kings)
slot#3: (T-Mac,Magic)
slot#4: (Davis,Clippers)

T-Mac is in Magic.
Arenas is in Wizards.

After deleing Kidd and T-Mac:
slot#0: (Bryant,Lakers)
slot#1: (Arenas,Wizards)
slot#2: (Webber,Kings)
slot#3:
slot#4: (Davis,Clippers)
```

不偷懶：用pointer串出Linked list

以下的範例程式碼紮紮實實用pointer串出Linked list，其中的 `Insert()` 、 `Delete()` 、 `Search()` 、 `DisplayTable()` 與上一小節大同小異，只是要加入Linked list的手法(改變**pointer**指向)。

`HashFunction()` 採取**Multiplication method**，詳細討論請參考：[Hash Table：Intro(簡介)/Multiplication Method (http://alrightchiu.github.io/SecondRound/hash-tableintrojian-jie.html#mm)](http://alrightchiu.github.io/SecondRound/hash-tableintrojian-jie.html#mm)。

比較酷的是因應**load factor($\alpha = \frac{n}{m}$)**改變Table大小，以及改變之後把node從舊的Table搬到新的Table的 `Rehashing()`：

`TableDoubling()`：當$\alpha = \frac{n}{m} > 1$時，表示資料量大於slot數量，就把Table大小m加倍(並配置m加倍後的Table)，如此在理論上可以儘量避免**Collision**發生，增加搜尋資料的效率。

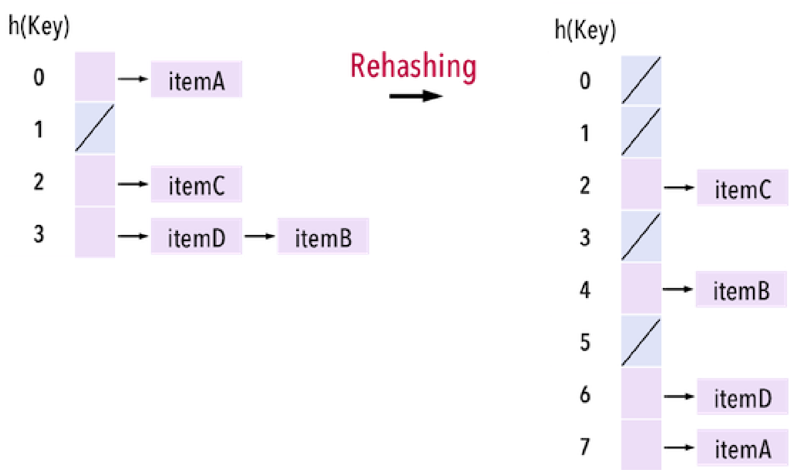
- 為什麼選擇Table大小加倍(`size*=2`)，而不是加一(`size++`)、加二？
  因為「加倍」的時間複雜度比較低，請參考：MIT 6.006：Lecture 9: Table Doubling, Karp-Rabin，影片6分44秒開始 (http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-006-introduction-to-algorithms-fall-2011/lecture-videos/lecture-9-table-doubling-karp-rabin/)。

`TableShrinking()`：當$\alpha = \frac{n}{m} < \frac{1}{4}$時，表示資料量減少到Table大小$m$的$\frac{1}{4}$，就把Table大小$m$減半(並配置$m$減半後的Table)，以節省記憶體空間。

- 為什麼選擇$\alpha = \frac{n}{m} < \frac{1}{4}$而不是$\alpha = \frac{n}{m} < \frac{1}{2}$？
  為了避免資料量在「臨界點」增增減減，造成不斷地動態配置記憶體(成本相當高)。
- 例如：起初$n = 8, m = 8$，現增加一筆資料，變成$n = 9, m = 8$，將會觸發一次 `TableDoubling()`，變成$n = 9, m = 16$。
  若接下來連續刪除兩筆資料，變成$n = 7, m = 16$，因為$\frac{n}{m} < \frac{1}{2}$，將會觸發一次 `TableShrinking()`，變成$n = 7, m = 8$。
  若接下來又連續增加兩筆資料，又將觸發一次 `TableDoubling()` ...依此類推，為了避免這種不斷配置記憶體的情況發生，寧可犧牲一點記憶體空間，等到$\frac{n}{m} < \frac{1}{4}$再觸發 `TableShrinking()`，重新為Table配置新的記憶體位置。

`Rehashing()`：當 `TableDoubling()` / `TableShrinking()` 增加/減半Table大小m後，需要把舊的Table上的資料(node)搬到新的Table上，過程將會透過Hash Function根據各筆資料的**key**重新分配一次**index**(因此稱為**Rehashing**)，此**index**即為資料在新的Table上的位置，如圖三。

- 範例程式碼採取直接改變node之pointer的做法，不另外配置新的記憶體空間。



圖三：。

`main()` 以「唱片行老闆想要以編號來整理各種樂風(genre)」示範Hash Table。

```
// C++ code
#include <iostream>
#include <vector>
#include <string>
#include <math.h>       // floor()

using std::vector;
using std::string;
using std::cout;
using std::endl;

struct Node{
    int key;                     // number
    string value;                // genre
    Node *next;                  // pointer to remember memory address of next node

    Node():key(0),value(""),next(0){};
    Node(int Key, string Value):key(Key),value(Value),next(0){};
    Node(Node const &data):key(data.key),value(data.value),next(data.next){};
};

class HashChainNode{
private:
    int size,                  // size: size of table, count: number of data
        count;                 // count/size = load factor
    Node **table;              // pointer to pointer, hash table

    int HashFunction(int key);      // Multiplication method
    void TableDoubling();
    void TableShrinking();
    void Rehashing(int size_orig);

public:
    HashChainNode(){};
    HashChainNode(int m):size(m),count(0){
        table = new Node *[size];           // allocate the first demension of table
        for (int i = 0; i < size; i++) {    // initialization
            table[i] = 0;                   // ensure every slot points to NULL
        }
    }
    ~HashChainNode();

    void Insert(Node data);         // consider TableDoubling()
    void Delete(int key);           // consider TableShrinking()
    string Search(int key);
    void DisplayTable();
};

void HashChainNode::Insert(Node data){

    count++;
    if (count > size) {         // consider load factor
        TableDoubling();        // if n/m > 1, then double the size of table
    }

    int index = HashFunction(data.key);   // get index of slot
    Node *newNode = new Node(data);       // create new node to store data

    // push_front()
    if (table[index] == NULL) {           // eg: list: (empty), add4
        table[index] = newNode;           // eg: list: 4->NULL
    }
    else {                                // eg: list: 5->9->NULL  , add 4
        Node *next = table[index]->next;  //     list: 5->4->9->NULL
        table[index]->next = newNode;
        newNode->next = next;
    }
}

void HashChainNode::Delete(int key){

    int index = HashFunction(key);        // get index of slot
    Node *current = table[index],         // use two pointer for traversal in list
         *previous = NULL;

    while (current != NULL && current->key != key) {
        previous = current;               // traversal in list, 3 cases:
        current = current->next;          // 1. data not found
    }                                     // 2. data found at first node in list
                                          // 3. data found at other position in list
```

```
    if (current == NULL) {                     // eg: list:5->2->9->NULL, want to delete 3
        cout << "data not found.\n\n";
        return;
    }
    else {
        if (previous == NULL) {                // eg: list:5->2->9->NULL, want to delete 5
            table[index] = current->next;      // after deleting 5, list:2->9->NULL
        }                                      // current points to 5

        else {                                 // eg: list:5->2->9->NULL, want to delete 2
            previous->next = current->next;    // after deleting 2, list:5->9->NULL
        }                                      // current points to 2
        delete current;
        current = 0;
    }

    count--;
    if (count < size/4) {      // consider load factor
        TableShrinking();      // if n/m < 4, then shrink the table
    }
}

string HashChainNode::Search(int key){

    int index = HashFunction(key);      // get index of slot
    Node *current = table[index];       // current points to the first node in list

    while (current != NULL) {           // traversal in list
        if ( current->key == key) {
            return current->value;
        }
        current = current->next;
    }
    return "...\nno such data";
}

int HashChainNode::HashFunction(int key){
    // Multiplication method
    double A = 0.6180339887,
           frac = key*A-floor(key*A);
    return floor(this->size*frac);
}

void HashChainNode::TableDoubling(){

    int size_orig = size;    // size_orig represents the original size of table
    size *= 2;               // double the size of table
    Rehashing(size_orig);;   // create new table with new larger size
}

void HashChainNode::TableShrinking(){

    int size_orig = size;    // size_orig represents the original size of table
    size /= 2;               // shrink the size of table
    Rehashing(size_orig);    // create new table with new smaller size
}

void HashChainNode::Rehashing(int size_orig){

    Node **newtable = new Node *[size];    // allocate memory for new table
    for (int i = 0; i < size; i++) {       // initializetion
        newtable[i] = 0;                   // ensure every node in slot points to NULL
    }

    for (int i = 0; i < size_orig; i++) {  // visit every node in the original table

        Node *curr_orig = table[i],        // curr_orig: current node in original table
             *prev_orig = NULL;            // prev_orig: following curr_orig

        while (curr_orig != NULL) {      // traversal in list of each slot in original table

            prev_orig = curr_orig->next; // curr_orig will be directly move to new table
                                         // need prev_orig to keep pointer in original table

            int index = HashFunction(curr_orig->key);    // get index of slot in new table

            // push_front(), do not allocate new memory space for data
            // directly move node in original table to new table
            if (newtable[index] == NULL) {       // means newtable[index] is empty
                newtable[index] = curr_orig;
```

```
                newtable[index]->next = 0;       // equivalent to curr_orig->next = 0;
            }
            // if there is no initialization for newtable, segmentation faults might happen
            // because newtable[index] might not point to NULL
            // but newtable[index] is empty
            else {                                   // if newtable[index] is not empty
                Node *next = newtable[index]->next;  // push_front()
                newtable[index]->next = curr_orig;
                curr_orig->next = next;
            }
            curr_orig = prev_orig;          // visit the next node in list in original table
        }
    }
    delete [] table;                // release memory of original table
    this->table = newtable;         // point table of object to new table
}

HashChainNode::~HashChainNode(){

    for (int i = 0; i < size; i++) {    // visit every node in table
                                        // and release the memory of each node
        Node *current = table[i];       // point *current to first node in list
        while (current != NULL) {       // traversal in list
            Node *previous = current;
            current = current->next;
            delete previous;
            previous = 0;
        }
    }
    delete [] table;
}

void HashChainNode::DisplayTable(){

    for (int i = 0; i < size; i++) {    // visit every node in table
        cout << "#slot#" << i << ": ";
        Node *current = table[i];
        while (current != NULL) {
            cout << "(" << current->key << "," << current->value << ") ";
            current = current->next;
        }
        cout << endl;
    }
    cout << endl;
}

int main(){

    HashChainNode hash(2);

    hash.Insert(Node(12,"post rock"));
    hash.Insert(Node(592,"shoegaze"));
    cout << "After inserting key(12),key(592):\n";
    hash.DisplayTable();
    hash.Insert(Node(6594,"blues"));        // evoke TableDoubling()
    cout << "After inserting key(6594), evoke TableDoubling():\n";
    hash.DisplayTable();
    hash.Insert(Node(7,"folk"));
    cout << "After inserting key(7):\n";
    hash.DisplayTable();
    hash.Insert(Node(123596,"hiphop"));     // evoke TableDoubling()
    cout << "After inserting key(123596), evoke TableDoubling():\n";
    hash.DisplayTable();
    hash.Insert(Node(93,"soul"));
    hash.Insert(Node(2288,"indie"));
    hash.Insert(Node(793,"jazz"));
    cout << "After inserting key(93),key(2288),key(793):\n";
    hash.DisplayTable();
    hash.Insert(Node(8491,"electro"));      // evoke TableDoubling()
    cout << "After inserting key(8491), evoke TableDoubling():\n";
    hash.DisplayTable();
    hash.Insert(Node(323359,"pop"));
    cout << "After inserting key(323359):\n";
    hash.DisplayTable();

    cout << "Searching: genre(8491) is " << hash.Search(8491) << ".\n\n";
    cout << "Searching: genre(7) is " << hash.Search(7) << ".\n\n";

    hash.Delete(7);
    cout << "After deleting key(7):\n";
    cout << "Searching: genre(7) is " << hash.Search(7) << ".\n\n";
```

```
    hash.Delete(592);
    cout << "After deleting key(592):\n";
    hash.DisplayTable();

    cout << "Want to  delete key(592) again:\n";
    hash.Delete(592);

    hash.Delete(123596);
    hash.Delete(323359);
    hash.Delete(793);
    hash.Delete(93);
    cout << "After deleting key(123596),key(323359),key(793),key(93):\n";
    hash.DisplayTable();

    hash.Delete(6594);      // evoke TableShrinking()
    cout << "After deleting key(6594), evoke TableShrinking():\n";
    hash.DisplayTable();

    return 0;
}
```

output:

```
After inserting key(12),key(592):
#slot#0: (12,post rock)
#slot#1: (592,shoegaze)

After inserting key(6594), evoke TableDoubling():
#slot#0:
#slot#1: (12,post rock) (6594,blues)
#slot#2:
#slot#3: (592,shoegaze)

After inserting key(7):
#slot#0:
#slot#1: (12,post rock) (7,folk) (6594,blues)
#slot#2:
#slot#3: (592,shoegaze)

After inserting key(123596), evoke TableDoubling():
#slot#0:
#slot#1:
#slot#2: (7,folk) (6594,blues)
#slot#3: (12,post rock)
#slot#4: (123596,hiphop)
#slot#5:
#slot#6:
#slot#7: (592,shoegaze)

After inserting key(93),key(2288),key(793):
#slot#0: (2288,indie) (793,jazz)
#slot#1:
#slot#2: (7,folk) (6594,blues)
#slot#3: (12,post rock) (93,soul)
#slot#4: (123596,hiphop)
#slot#5:
#slot#6:
#slot#7: (592,shoegaze)

After inserting key(8491), evoke TableDoubling():
#slot#0: (2288,indie)
#slot#1: (793,jazz)
#slot#2:
#slot#3:
#slot#4:
#slot#5: (7,folk) (6594,blues)
#slot#6: (12,post rock)
#slot#7: (93,soul)
#slot#8: (123596,hiphop)
#slot#9:
#slot#10:
#slot#11: (8491,electro)
#slot#12:
#slot#13:
#slot#14: (592,shoegaze)
#slot#15:

After inserting key(323359):
#slot#0: (2288,indie)
#slot#1: (793,jazz)
#slot#2:
#slot#3:
#slot#4:
#slot#5: (7,folk) (6594,blues)
#slot#6: (12,post rock)
#slot#7: (93,soul)
#slot#8: (123596,hiphop)
#slot#9:
#slot#10:
#slot#11: (8491,electro)
#slot#12:
#slot#13: (323359,pop)
#slot#14: (592,shoegaze)
#slot#15:

Searching: genre(8491) is electro.

Searching: genre(7) is folk.

After deleting key(7):
Searching: genre(7) is ...
no such data.
```

```
After deleting key(592):
#slot#0: (2288,indie)
#slot#1: (793,jazz)
#slot#2:
#slot#3:
#slot#4:
#slot#5: (6594,blues)
#slot#6: (12,post rock)
#slot#7: (93,soul)
#slot#8: (123596,hiphop)
#slot#9:
#slot#10:
#slot#11: (8491,electro)
#slot#12:
#slot#13: (323359,pop)
#slot#14:
#slot#15:

Want to  delete key(592) again:
data not found.

After deleting key(123596),key(323359),key(793),key(93):
#slot#0: (2288,indie)
#slot#1:
#slot#2:
#slot#3:
#slot#4:
#slot#5: (6594,blues)
#slot#6: (12,post rock)
#slot#7:
#slot#8:
#slot#9:
#slot#10:
#slot#11: (8491,electro)
#slot#12:
#slot#13:
#slot#14:
#slot#15:

After deleting key(6594), evoke TableShrinking():
#slot#0: (2288,indie)
#slot#1:
#slot#2:
#slot#3: (12,post rock)
#slot#4:
#slot#5: (8491,electro)
#slot#6:
#slot#7:
```

以上是以**Chaining**解決**Collision**之介紹。

參考資料：

- Introduction to Algorithms, Ch11 (http://www.amazon.com/Introduction-Algorithms-Edition-Thomas-Cormen/dp/0262033844)
- Fundamentals of Data Structures in C++, Ch8 (http://www.amazon.com/Fundamentals-Data-Structures-Ellis-Horowitz/dp/0929306376)
- Abdullah Ozturk：Simple Hash Map (Hash Table) Implementation in C++ (https://medium.com/@aozturk/simple-hash-map-hash-table-implementation-in-c-931965904250#.du6lwge1u)
- Pumpkin Programmer：C++ Tutorial: Intro to Hash Tables (http://pumpkinprogrammer.com/2014/06/21/c-tutorial-intro-to-hash-tables/)
- Adnan Aziz：Hash Tables (http://users.ece.utexas.edu/~adnan/360C/hash.pdf)
- MIT 6.006：Lecture 9: Table Doubling, Karp-Rabin (http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-006-introduction-to-algorithms-fall-2011/lecture-videos/lecture-9-table-doubling-karp-rabin/)
- Linked List: Intro(簡介) (http://alrightchiu.github.io/SecondRound/linked-list-introjian-jie.html)

- Linked List: 新增資料、刪除資料、反轉 (http://alrightchiu.github.io/SecondRound/linked-list-xin-zeng-zi-liao-shan-chu-zi-liao-fan-zhuan.html)

Hash Table系列文章

Hash Table：Intro(簡介) (http://alrightchiu.github.io/SecondRound/hash-tableintrojian-jie.html)
Hash Table：Chaining (http://alrightchiu.github.io/SecondRound/hash-tablechaining.html)
Hash Table：Open Addressing (http://alrightchiu.github.io/SecondRound/hash-tableopen-addressing.html)

回到目錄：

目錄：演算法與資料結構 (http://alrightchiu.github.io/SecondRound/mu-lu-yan-suan-fa-yu-zi-liao-jie-gou.html)

tags: C++ (http://alrightchiu.github.io/SecondRound/tag/c.html), Dictionary (http://alrightchiu.github.io/SecondRound/tag/dictionary.html), Hash Table (http://alrightchiu.github.io/SecondRound/tag/hash-table.html), Linked List (http://alrightchiu.github.io/SecondRound/tag/linked-list.html),