Hash Table：Open Addressing

Posted by Chiu CC (http://alrightchiu.github.io/SecondRound/author/chiu-cc.html) on 5 29, 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**Open Addressing**來解決**Collision**。

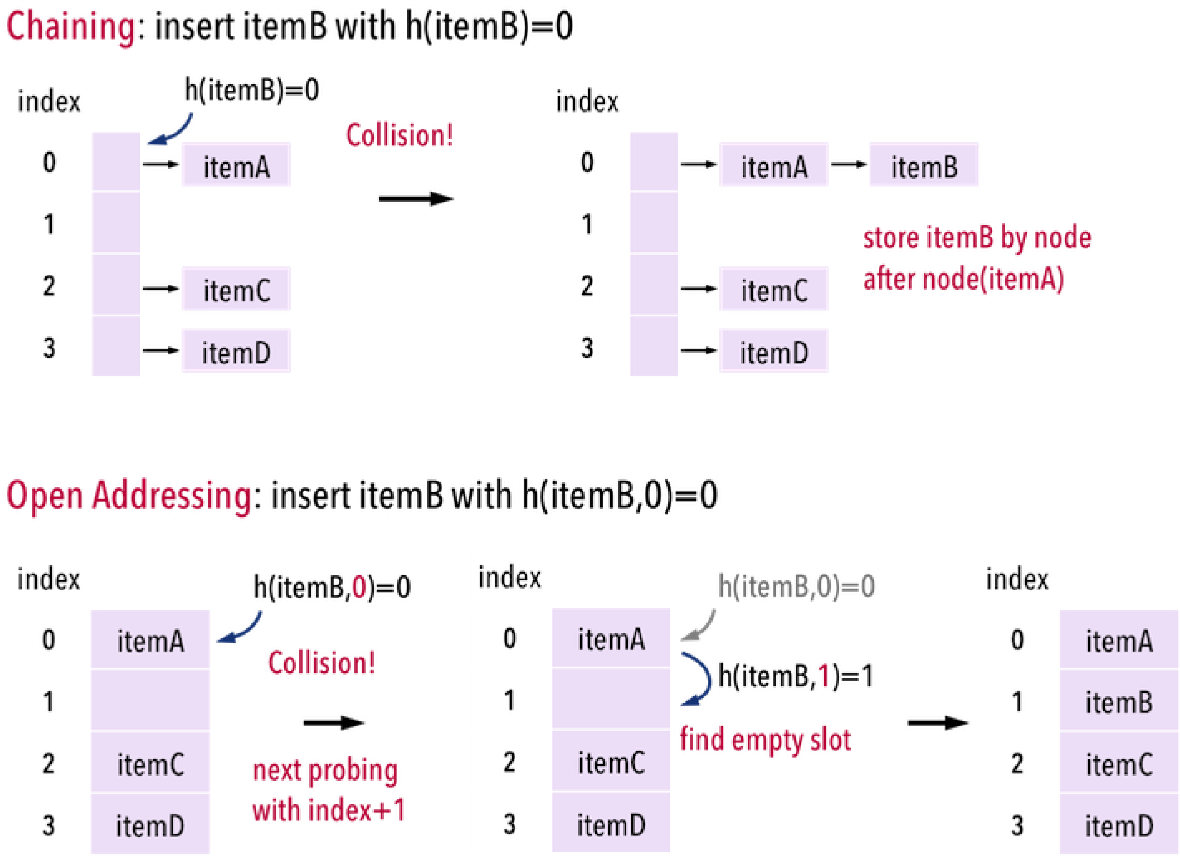
目錄

Open Addressing的概念

當發生**Collision**時，**Chaining**會將所有被Hash Function分配到同一格slot的資料透過Linked list串起來，像是在書桌的抽屜下面綁繩子般，把所有被分配到同一格抽屜的物品都用繩子吊在抽屜下面。

相較於**Chaining**提供額外空間(node)來存放被分配到相同slot的資料，**Open Addressing**則是將每筆資料都放在書桌(Table)本身配備的抽屜(slot)，一格抽屜只能放一個物品，如果抽屜都滿了，就得換張書桌(重新配置記憶體給新的Table)。

因此，**load factor**($\alpha = \frac{n}{m}$)不能超過1。



圖一：Chaining vs Open Addressing。

既然沒有額外的空間存放資料，當Hash Function把具有不同**Key**的資料分配到同一個slot時怎麼辦呢？

那就繼續「尋找下一格空的slot」，直到

1. 終於找到空的slot，或者
2. 所有slot都滿了為止

如圖一，這種「尋找下一格空的slot」的方式就稱為**Probing**。
(probe有探測的意思，在這裏可以解讀成：不斷探測下一個slot是否為空的。)

利用Probing

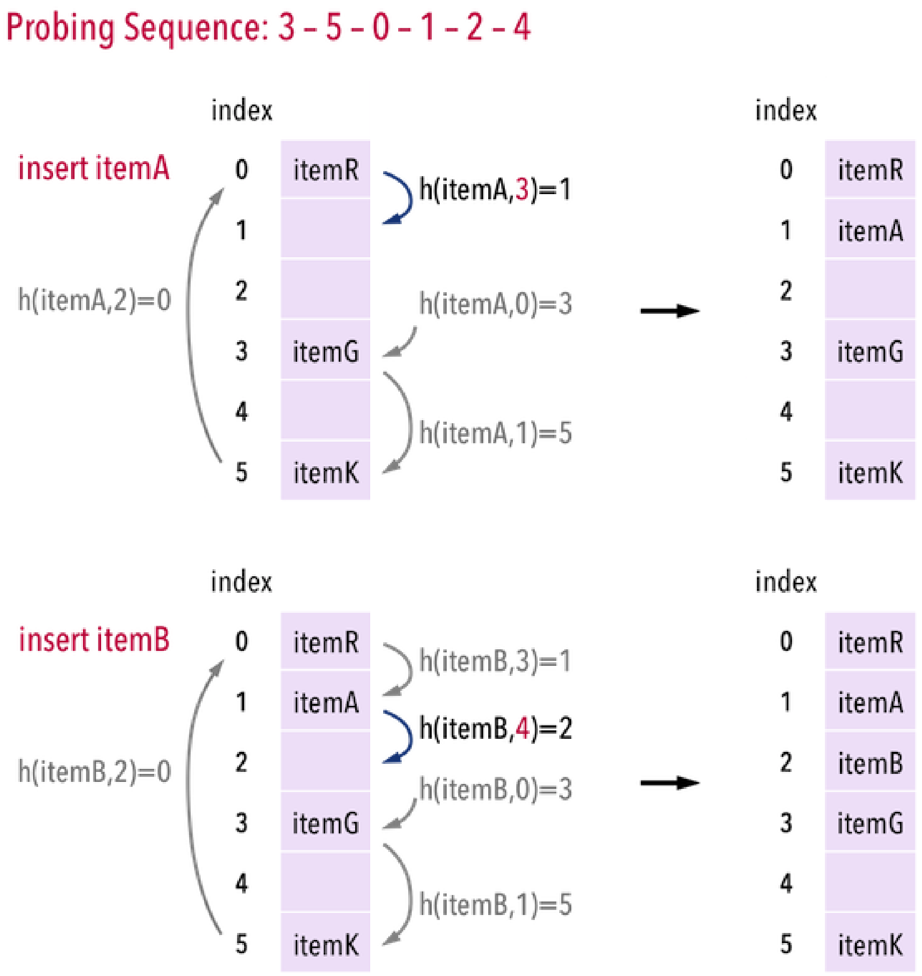
Probing就是「尋找下一格空的slot」，如果沒找到，就要繼續「往下找」，因此，**Probing**的精髓就是要製造出「往下找的順序」，這個順序盡可能越不規則越好，如此可確保Hash Function不會一直找到同一個slot：

- 假設Table大小為6，對於itemA而言，**Probing**順序是$\{3,5,0,1,2,4\}$，就表示itemA會先被分配到第3個slot，如果第3個slot已經有其他item，就往下找第5個slot；如果第5個slot也有item了，就再找第0個slot，依此類推，按照其**Probing**順序逐一檢查是否有空的slot，如果6個slot都滿了，便宣告失敗，需要調整Table大小。

- 若有另一個itemB，其**Probing**順序與itemA完全相同($\{3,5,0,1,2,4\}$)，那麼在加入(insert)itemB時，必須再完整經歷一次itemA的過程，見圖二：
  - 假設itemA最後被放在第1個slot，表示前面的第3、第5、第0個slot都已經有item，那麼在加入itemB時，就需要經歷「4次」失敗，才終於在第2個slot找到位置存放。

可以想像的是，若對於所有item都只有一種**Probing**順序，那麼在加入(insert)第一個item時，只需要$O(1)$的時間，但是再繼續加入item時，就必須考慮「現有的item數」，時間複雜度增加為$O(\#items\ in\ slots)$。

由此可見，**Probing**順序會影響到Hash Table的操作(**insert**、**delete**、**search**)之時間複雜度。



圖二：。

圖二中，**Probing**之Hash Function的定義域(domain)有兩個參數，一個是**Key**，另一個是**Probing**的「次數」，而值域(range)即為Table的**index**範圍：

$$h : U \times \{0, 1, \ldots, m-1\} \rightarrow \{0, 1, \ldots m-1\}$$

- U是所有可能的**Key**的宇集合(universal set)。
- **Probing**的次數最多不會超過Table大小m，定義從第0次到第$m-1$次。
- 隨著「次數增加」，Hash Function的值域可以視為$\{0, 1, \ldots, m-1\}$的排列組合(permutation)，亦即：

$$\{h(k,0), h(k,1)\ldots, h(k,m-1)\} = permutation\ of\ \{0, 1, \ldots, m-1\}$$

如此便可確保**Probing**會檢查Table中的每一個slot。

接下來介紹三種常見的**Probing method**：

1. Linear Probing
2. Quadratic Probing
3. Double Hashing

特別注意，**Probing**的Hash Function與**Chaining**的Hash Function略有不同(雖然都稱為Hash Function)：

1. **Chaining**使用的Hash Function只有一個參數，就是資料的**Key**。
2. **Open Addressing**使用的Hash Function有兩個參數，一個是資料的**Key**，另一個是**Probing**的「次數」。

Linear Probing

Linear Probing定義為：

$$h(k,i) = (h'(k) + i) \bmod m$$

其中：

- $h'(k)$即可視為**Chaining**用的Hash Function，其值域(range)在$0 \sim m-1$之間：
  - $h'(k)$可以使用Division Method、Multiplication Method或Universal Hashing等等。
  - $h'(k)$的結果就是**Probing**的起點。
- i表示**Probing**的「次數」，在此因為i的係數為1，因此i也會是影響到**Probing**順序的參數。

由於i是線性成長，產生的**Probing Sequence**也會是線性的：

- 由於$i = 0 \sim m-1$，對每個Key而言，**Probing Sequence**為

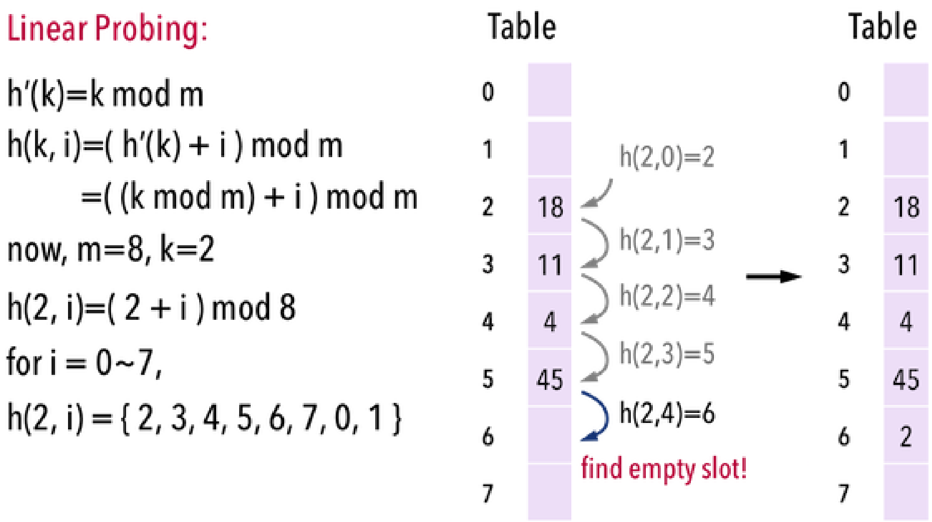
$$\{h'(k), h'(k)+1, h'(k)+2, \ldots, h'(k)+(m-1)\} \bmod m$$

- 以上數列確實是一種$\{0, 1, \ldots, m-1\}$的排列組合(permutation)。
  觀察方法：因為 $\bmod m$會循環，所以可以先忽略$h'(k)$，看剩下的值是否沒有重複地涵蓋了$0 \sim m-1$的每一個值。

意思是若第1個slot滿了，就找第2個slot；若第2個slot滿了，就找第3個slot，依此類推，當目前的slot已經有item時，就找繼續找「下一個**index**」的slot，檢查其是否是空的。

舉例來說，若$m = 8$，$h'(k) = k \bmod m$，$h(k,i) = ((k \bmod m) + i) \bmod m$：

- $k = 7$，$h'(7) = 7$，**Probing Sequence**為$\{7,0,1,2,3,4,5,6\}$
- $k = 13$，$h'(13) = 5$，**Probing Sequence**為$\{5,6,7,0,1,2,3,4\}$
- $k = 50$，$h'(50) = 2$，**Probing Sequence**為$\{2,3,4,5,6,7,0,1\}$
- $k = 74$，$h'(74) = 2$，**Probing Sequence**為$\{2,3,4,5,6,7,0,1\}$



圖三：。

缺點

由以上可以觀察出，**Linear Probing**只有m個**Probing Sequence**，而且很規律：

- 只要「起點」決定，**Probing Sequence**就決定(i不斷加一)。
- $h'(k) = k \bmod m$會產生$0 \sim m-1$的值，一共是m個值作為起點。

因為**Linear Probing**是「找下一個**index**」的slot，所以如果Table中某個區塊已經擠滿了item，若有某個**Key**又被$h'(k)$分配到該區塊的附近，就會使得該區塊「越來越擠」，這種現象稱為**primary clustering**：

- 如圖三，`Table[2,3,4,5]`已經有item，再來一個$k = 2$，經過**Probing**被分配到`Table[6]`，使得`Table[2,3,4,5,6]`的區塊變得更擠。

這使得之後被分配到這個區塊的item之**insert**、**delete**、**search**的時間複雜度將會受到「前面擋住的item數量」影響。

- 圖三的$k = 2$經過5次**Probing**才順利找到空的slot存進Hash Table。

所以，如果能夠產生一個「比較不規律」的**Probing Sequence**，例如$\{0,1,3,6,2,7,5,4\}$，就能儘量避免把「某個區塊變得更擠」，減少**primary clustering**發生的可能。

接下來介紹的**Quadratic Probing**即可產生「比較亂的」**Probing Sequence**。

Quadratic Probing

Quadratic Probing定義為：

$$h(k,i)=(h'(k)+c_1 i+c_2 i^2) \bmod m\,, c_2\neq 0$$

與**Linear Probing**相比，多了i的二次項及係數($c_2 i^2$)，並且一次項也多了係數c_1，有機會產生「較為分散」的**Probing Sequence**。

不過並不是任意的c_1、c_2及m都能夠使$h(k,i)$產生$\{0,1,\ldots,m-1\}$的排列組合(permutation)，以下提供兩種常見的選法：

第一種：

選擇$c_1=c_2=0.5, m=2^P$：

$$h(k,i)=(h'(k)+0.5i+0.5i^2) \bmod m$$

產生的**Probing Sequence**為：

$$\{h'(k),h'(k)+1,h'(k)+3,h'(k)+6,h'(k)+10,h'(k)+15,h'(k)+21,h'(k)+28...\} \bmod m$$

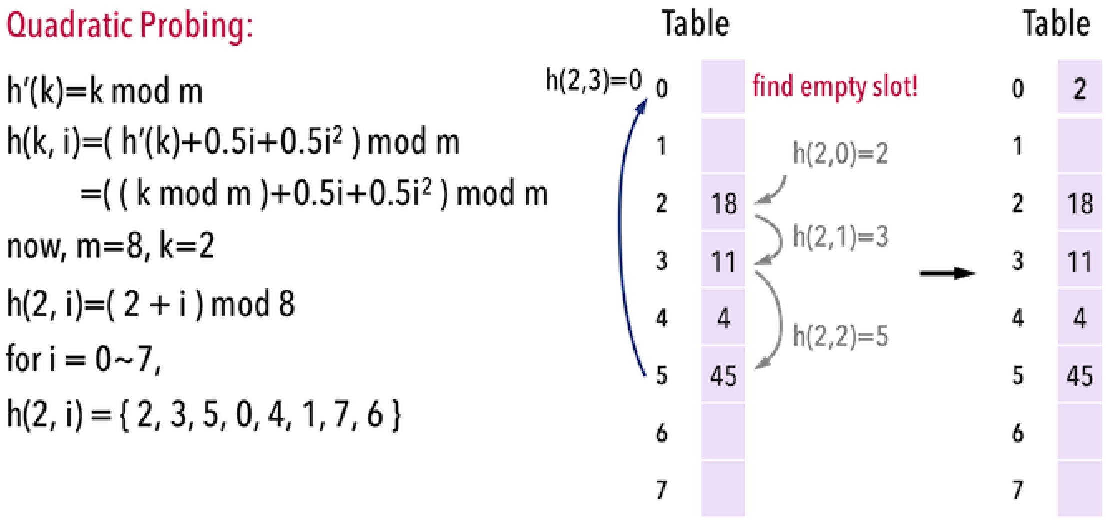
每一次在找下一格slot時，「跨距」都會再增加一，第一次跨「1格」，下一次跨「2格」，再下一次跨「3格」，依此類推。

例如，若考慮$m=8$，**Probing Sequence**為：

$$\{h'(k),h'(k)+1,h'(k)+3,h'(k)+6,h'(k)+2,h'(k)+7,h'(k)+5,h'(k)+4\} \bmod 8$$

確認其確實是$\{0,1,\ldots,7\}$的排列組合，並且是比**Linear Probing**更跳躍的方式找下一格slot。
幾個範例如下，考慮$h'(k)=k \bmod 8$，見圖四(a)：

- $k=2$，$h'(k)=2$，**Probing Sequence**：$\{2,3,5,0,4,1,7,6\}$
- $k=6$，$h'(k)=6$，**Probing Sequence**：$\{6,7,1,4,0,5,3,2\}$
- $k=9$，$h'(k)=1$，**Probing Sequence**：$\{1,2,4,7,3,0,6,5\}$
- $k=25$，$h'(k)=1$，**Probing Sequence**：$\{1,2,4,7,3,0,6,5\}$



圖四(a)：。

第二種：

$$h(k,i) = (h'(k) - (-1)^i \lceil \frac{i}{2} \rceil^2) \bmod m$$
$$where\ m\ be\ prime\ and\ m = 4n+3,\ for\ some\ n$$

產生的**Probing Sequence**為：

$\{h'(k), h'(k)+1, h'(k)-1, h'(k)+4, h'(k)-4, h'(k)+9, h'(k)-9, h'(k)+16, h'(k)-16...\} \bmod m$

這種方式很有創意地在正負i^2跳躍，並且要求：

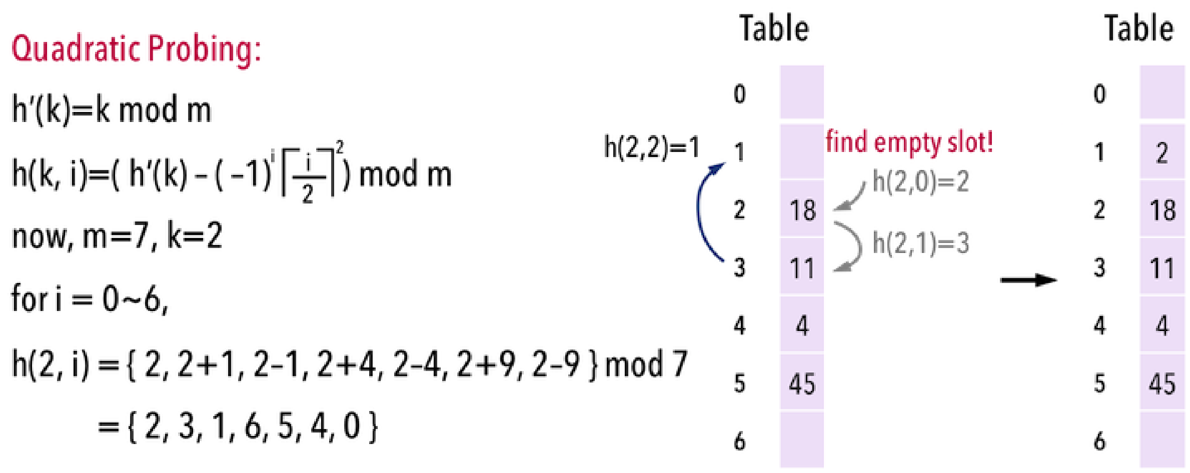
1. m是質數；
2. 滿足$m = 4n+3$，其中n是某些能夠另m成為質數的正整數。
   - 例如：$(m,n) = (7,1), (11,2), (19,4), (23,5)...$。

例如，選擇$m = 7$，那麼**Probing Sequence**即為：

$$\{h'(k), h'(k)+1, h'(k)-1, h'(k)+4, h'(k)-4, h'(k)+9, h'(k)-9\} \bmod 7$$
$$= \{h'(k), h'(k)+1, h'(k)+6, h'(k)+4, h'(k)+3, h'(k)+2, h'(k)+5\} \bmod 7$$

確認其確實是$\{0,1,\ldots,6\}$的排列組合，並且是比**Linear Probing**更跳躍的方式找下一格slot。
幾個範例如下，考慮$h'(k) = k \bmod 7$，見圖四(b)：

- $k = 2$，$h'(k) = 2$，**Probing Sequence**：$\{2,3,1,6,5,4,0\}$
- $k = 6$，$h'(k) = 6$，**Probing Sequence**：$\{6,0,5,3,2,1,4\}$
- $k = 19$，$h'(k) = 5$，**Probing Sequence**：$\{5,6,4,2,1,0,3\}$



圖四(b)：。

整體而言，**Quadratic Probing**的優缺點：

優點：

透過較為跳躍的方式找下一格空的slot，**Quadratic Probing**可以有效避免**primary clustering**。

缺點：

並不是任意的c_1, c_2, m都可以產生$\{0, 1, \ldots, m-1\}$的排列組合(permutation)，所以參數需要慎選。

並且，如同**Linear Probing**，一旦「起點$h'(k)$」決定好，**Probing Sequence**就決定好了，因此：

- 同樣只有m個不同的**Probing Sequence**。
- 如果$h'(k_1) = h'(k_2)\,, k_1 \neq k_2$，那麼這兩個item會有同樣的**Probing**順序。
- 可以想像的是，若$h'(k)$一直把item分配到同一格slot起點，那麼較晚加入Table的item之**insert**、**search**、**delete**的時間複雜度仍然會增加，這稱為**secondary clustering**。

Double Hashing

根據上面兩種**Probing Method**可以看出，透過調整「次數i」的函式形式，也就調整了「跨距」方式，便能夠製造出「較為分散」的**Probing Sequence**。

而**Double Hashing**就直接加入「第二個Hash Function」來影響「次數i」，定義為：

$$h(k, i) = (h_1(k) + ih_2(k)) \bmod m$$

並且要求，$h_2(k)$一定要與m互質($h_2(k)$ be relatively prime to m)。

這裡有項神奇的事實：若a與b互質，那麼$(a \times i) \bmod b,\ for\ i = 0, 1 \ldots, b-1$，正好可以形成$\{0, 1, \ldots, b-1\}$的排列組合(permutation)。

舉例來說，若$a = 5, b = 8$，那麼：

- $i = 0$，$(5 \times 0 = 0) \bmod 8 = 0$
- $i = 1$，$(5 \times 1 = 5) \bmod 8 = 5$
- $i = 2$，$(5 \times 2 = 10) \bmod 8 = 2$
- $i = 3$，$(5 \times 3 = 15) \bmod 8 = 7$
- $i = 4$，$(5 \times 4 = 20) \bmod 8 = 4$
- $i = 5$，$(5 \times 5 = 25) \bmod 8 = 1$
- $i = 6$，$(5 \times 6 = 30) \bmod 8 = 6$
- $i = 7$，$(5 \times 7 = 35) \bmod 8 = 3$

因此，要求$h_2(k)$與m互質確實可以產生$\{0, 1, \ldots, m-1\}$的排列組合。

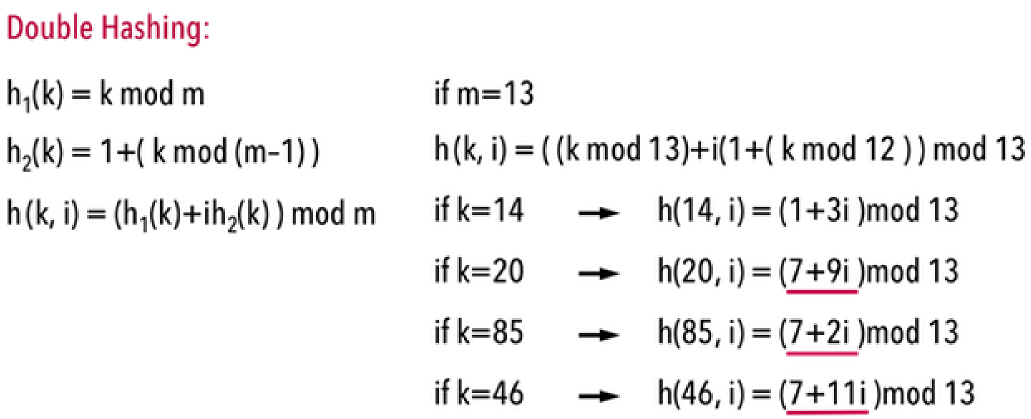
以下提供幾種常見的$h_2(k)$選擇：

第一種：$m = 2^P$，$h_2(k)$的值域(range)為奇數(odd number)，$h_2(k) = 2n + 1$。

- 因為奇數一定與2^P互質，所以滿足。
- 不過由於$h_2(k)$的值域(range)只有奇數(odd number)，可能不能視為在**Chainine**所使用的Hash Function。

第二種：m是質數，$h_2(k)$的值域(range)介於$1 \sim m-1$，$0 < h_2(k) < m-1$。

- 若m是質數，那麼m必定與$1 \sim m-1$的任何數互質。
- 由圖五的範例可以觀察出，即使不同的**Key**被$h_1(k)$分配到相同的起點slot，但是$h_2()$有很大的機會調整出不同的「跨距」(i的係數)，使得兩個不同**Key**的item有不同的**Probing Sequence**。



圖五：。

第三種：m是質數，$h_2(k) = R - k$，R也是質數，並且$R < m$。

- 與第二種類似的概念產生m與$h_2(k)$互質，不過可以製造出不同的**Probing Sequence**。

Double Hashing的優點：

因為同時有兩個Hash Function($h_1(k), h_2(k)$)，若兩者個值域都是$\{0, 1, \ldots, m-1\}$，那麼**Double Hashing**一共可以產生m^2種不同的**Probing Sequence**，因此可以大大減緩**clustering**。

- 除非$(h_1(k_1), h_2(k_1)) = (h_1(k_2), h_2(k_2))$的情況才會發生兩個**Key**有完全相同的**Probing Sequence**，機率較低。

程式碼

範例程式碼簡單地以**Quadratic Probing**($c_1 = c_2 = 0.5$，$m = 2^P$)實作出**Open Addressing**的Hash Table。

關於**Rehashing**、調整Table大小的議題與Hash Table：Chaining (http://alrightchiu.github.io/SecondRound/hash-tablechaining.html#ll)的方法大同小異，不過**load factor**可能要限制得更嚴謹(請看下一小節的挑論)，這裡就不再贅述。

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

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

struct dict{
    int key;
    string value;
    dict():key(0),value(""){};
    dict(int k, string s):key(k),value(s){};
};

class HashOpenAddress{
private:
    int size, count;
    dict *table;

    int QuadraticProbing(int key, int i);

    // TableDoubling()
    // TableShrinking()
    // Rehashing()

public:
    HashOpenAddress():size(0),count(0),table(0){};
    HashOpenAddress(int m):size(m),count(0){
        table = new dict[size];
    }
    void Insert(int key, string value);
    void Delete(int key);
    string Search(int key);
    void Display();
};

string HashOpenAddress::Search(int key){

    int i = 0;
    while (i != size) {
        int j = QuadraticProbing(key, i);
        if (table[j].key == key) {
            return table[j].value;
        }
        else {
            i++;
        }
    }
    return "...data not found\n";
}

void HashOpenAddress::Delete(int key){

    int i = 0;
    while (i != size) {
        int j = QuadraticProbing(key, i);
        if (table[j].key == key) {
            table[j].key = 0;
            table[j].value = "";
            count--;
            return;
        }
        else {
            i++;
        }
    }
    cout << "...data not found\n";
```

```
}

void HashOpenAddress::Insert(int key, string value){

    int i = 0;
    while (i != size) {
        int j = QuadraticProbing(key, i);
        if (table[j].value == "") {
            table[j].key = key;
            table[j].value = value;
            count++;
            return;
        }
        else {
            i++;
        }
    }
    cout << "Hash Table Overflow\n";
}

int HashOpenAddress::QuadraticProbing(int key, int i){
    // c1 = c2 = 0.5
    return ((int)( (key % size) + 0.5*i + 0.5*i*i ) % size);
}

void HashOpenAddress::Display(){

    for (int i = 0; i < size; i++) {
        cout << "slot#" << i << ": (" << table[i].key
             << "," << table[i].value << ")" << endl;
    }
    cout << endl;
}
int main(){

    HashOpenAddress hash(8);          // probing sequence:
    hash.Insert(33, "blue");          // 1,2,4,7,3,0,6,5 -> 1
    hash.Insert(10, "yellow");        // 2,3,5,0,4,1,7,6 -> 2
    hash.Insert(77, "red");           // 5,6,0,3,7,4,2,1 -> 5
    hash.Insert(2, "white");          // 2,3,5,0,4,1,7,6 -> 3
    hash.Display();
    hash.Insert(8, "black");          // 0,1,3,6,2,7,5,4 -> 0
    hash.Insert(47, "gray");          // 7,0,2,5,1,6,4,3 -> 7
    hash.Insert(90, "purple");        // 2,3,5,0,4,1,7,6 -> 4
    hash.Insert(1, "deep purple");    // 4,5,7,2,6,3,1,0 -> 6
    hash.Display();
    hash.Insert(15, "green");         // hash table overflow

    cout << "number#90 is " << hash.Search(90) << "\n\n";

    hash.Delete(90);
    cout << "after deleting (90,purple):\n";
    cout << "number#90 is " << hash.Search(90) << "\n";

    hash.Insert(12, "orange");        // 4,5,7,2,6,3,1,0 -> 4
    hash.Display();

    return 0;
}
```

output:

```
slot#0: (0,)
slot#1: (33,blue)
slot#2: (10,yellow)
slot#3: (2,white)
slot#4: (0,)
slot#5: (77,red)
slot#6: (0,)
slot#7: (0,)

slot#0: (8,black)
slot#1: (33,blue)
slot#2: (10,yellow)
slot#3: (2,white)
slot#4: (90,purple)
slot#5: (77,red)
slot#6: (1,deep purple)
slot#7: (47,gray)

Hash Table Overflow

number#90 is purple

after deleting (90,purple):
number#90 is ...data not found

slot#0: (8,black)
slot#1: (33,blue)
slot#2: (10,yellow)
slot#3: (2,white)
slot#4: (12,orange)
slot#5: (77,red)
slot#6: (1,deep purple)
slot#7: (47,gray)
```

比較Open Addressing與Chaining

time complxity

對於**Open Addressing**：

- insert：要找到空的slot才能 `insert()` 。
    - 找到空的slot，也就是沒有找到與**Key**相符合的item，稱為**Unsuccessful Search**。
- delete：要找到與**Key**相符合的item才能 `delete()` 。
    - 找到與**Key**相符合的item，稱為**Successful Search**。

以上兩者都需要進行 `search` 。

對於**Chaining**：

- insert：可以透過Linked list的 `push_front()` 以$O(1)$完成。
- delete：需要在Linked list進行**traversal**，如同 `search` 。
- 不論是「搜尋成功」還是「搜尋不成功」，時間複雜度都與Linked list的長度有關。

表一是**Open Addressing**與**Chaining**針對「搜尋」的時間複雜度比較，分成「搜尋成功」與「搜尋不成功」：

	Open Addressing	**Chaining**
Unsuccessful Search	$\frac{1}{1-\alpha}$	$1+\alpha$
Successful Search	$\frac{1}{\alpha}\ln\frac{1}{1-\alpha}$	$1+\alpha$

表一：Open Addressing

關於**Open Addressing**之時間複雜度證明，請參考：Joost-Pieter Katoen：Hashing/page35、36 (http://fmt.cs.utwente.nl/courses/adc/lec5.pdf)。

效率：考慮load factorα

以**Open Addressing**之**Unsuccessful Search**為例，$O(\frac{1}{1-\alpha})$，根據其時間複雜度可以觀察出，當**load factor**$\alpha=\frac{n}{m}$趨近於1時(Table快被放滿)，那麼時間複雜度會趨近無限大：$\lim_{\alpha\to 1} O(\frac{1}{1-\alpha}) \to O(\infty)$，這種情況便不適合使用**Open Addressing**。

不過**Open Addressing**使用Array存放資料，不需要頻繁使用動態記憶體配置(`new` / `delete` / `malloc` / `free`)，所以如果**load factor**沒有超過0.5(有些使用0.7)，那麼**Open Addressing**會是不錯的選擇。

memory使用

Chaining使用Linked list，每個node裡面會帶一個**pointer**記錄下一個node的記憶體位置，因此會比純粹使用Array存放資料的**Open Addressing**多花一點記憶體空間。

不過前面提到，**Open Addressing**考慮**load factor**儘量不要超過0.5，因此將有近一半的記憶體位置閒置。

所以這兩種處理**Collision**的方法沒有絕對的好壞，應該要是情況而定。

以上是以**Open Addressing**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)

- 林清池：Algorithms Chapter 11 Hash Tables (http://www.cs.ntou.edu.tw/lincc/courses/al99/pdf/Algorithm-Ch11-Hash%20Tables.pdf)
- Collision Resolution: Open Addressing (http://faculty.kfupm.edu.sa/ICS/saquib/ICS202/Unit30_Hashing3.pdf)
- Jan-Georg Smaus：8 Hashing: Open addressing (http://gki.informatik.uni-freiburg.de/teaching/ss11/theoryI/08_Open_Addressing.pdf)
- Joost-Pieter Katoen：Hashing/page35、36 (http://fmt.cs.utwente.nl/courses/adc/lec5.pdf)

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),

(https://github.com/alrightchiu)