# Hash Table：Intro(簡介)

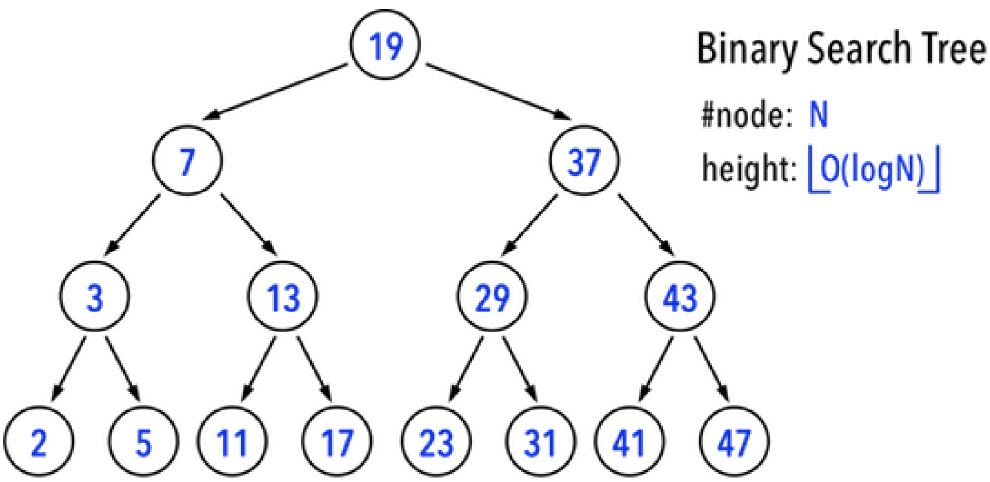
*Posted by Chiu CC (http://alrightchiu.github.io/SecondRound/author/chiu-cc.html) on 5 19, 2016*

## 先備知識與注意事項

在做資料處理時，常常需要「查詢資料」，譬如線上購物平台有會員登入時，首先確認輸入的帳號密碼是否在資料庫裡，如果是，便從資料庫裡找出此會員的資料，如購物記錄、暫存購物清單等等。

想到「查詢資料」，可能會想到能夠在時間複雜度為$O(\log N)$完成查詢的**平衡的**Binary Search Tree(二元搜尋樹) (http://alrightchiu.github.io/SecondRound/binary-search-tree-introjian-jie.html)，如圖一。

- 在圖一的BST中，要找到Key(17)的資料，需要比較$4 = \lfloor \log_2 15 \rfloor + 1$次，時間複雜度可以視為**height(樹高)**。




**圖一：若為平衡的BST，查詢資料之時間複雜度為$O(\log N)$。**

但是若資料量非常龐大(例如社交平台的註冊會員資料庫)，即使是$O(\log N)$也非常可觀。

如果能在時間複雜度為常數的$O(1)$完成查詢該有多好。

本篇文章便要介紹能夠在$O(1)$完成查詢的**Hash Table(雜湊表)**。

---

# 目錄



# 簡介：Dictionary(字典)

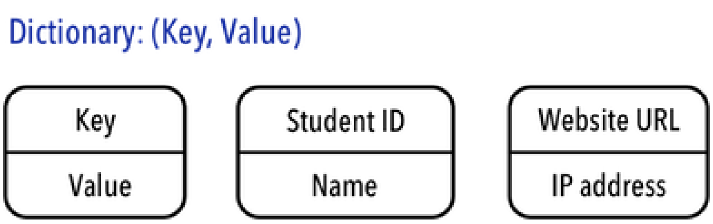
**Dictionary**是以「鍵值-資料對」(**Key-Value** pair)來描述資料的抽象資料形態(Abstract Data Type)。

舉例來說：

- 電話簿裡的Dictionary即是將「姓名」視為**Key**，「電話號碼」視為**Value**。
- 學校學籍系統的Dictionary將「學號」視為**Key**，「學生資料」(如姓名、修課記錄)視為**Value**。

所以，任何具有辨別功能、可以用在「查詢資料」的「符號」(像是姓名、學號、網址等等)都能夠視為**Key**。
而**Value**代表著較為廣義的「資料」，例如電話號碼、學籍資料、IP位置等等。

只要在系統輸入**Key**，便能找到相對應的**Value**，這就是**Dictionary**的基本概念。




**圖二：**。

一般**Dictionary**會支援三個操作：

1. 新增資料(insert)
2. 刪除資料(delete)
3. 查詢資料(search)

以上三個操作：

- 若以**平衡的BST**(例如AVL樹、Red Black Tree (http://alrightchiu.github.io/SecondRound/red-black-tree-introjian-jie.html))便能在$O(\log N)$完成。
- 「理想情況」的**Hash Table**希望能夠以$O(1)$完成。

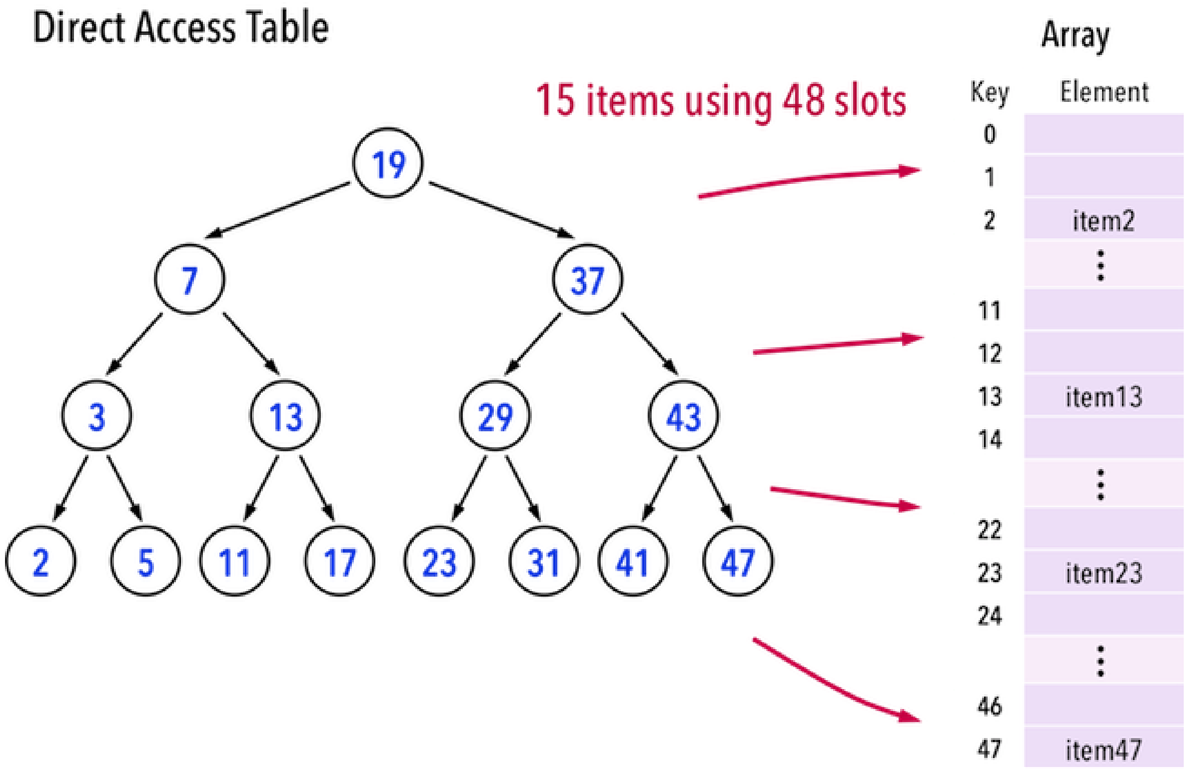
# 以Array實現的Direct Access Table

若要在$O(1)$對資料進行「新增」、「刪除」以及「查詢」，沒有比**Array(**矩陣**)**更適合的人選了：

- 若直接把**Key**當作Array的**index**，並將**Value**存放進Array，這樣的實作稱為**Direct Access Table**。

但是**Direct Access Table**有兩個重大缺陷：

1. **Key**一定要是「非負整數(non-negative integer)」，才能作為Array的**index**。
2. 若**Key**的「範圍」非常大，可是**Key**的「數量」相對很少，那麼會非常浪費記憶體空間。
    - 以圖三為例，因為**Key**的範圍從2到47，所以Array的大小(size)至少要「48」，因此15筆資料用了48單位的記憶體空間，也就浪費超過三分之二的記憶體空間。若**Key**的範圍更大，浪費的程度將非常可觀。




**圖三：。**

關於第一點「**Key**不是非負整數」的缺陷，可以先利用一個「一對一函數(one-to-one function)」將**Key**對應到非負整數，問題即可解決，稱為**prehash**。

例如，若**Key**是英文名字，那便利用「ASCII編碼」將字串轉換成非負整數。現要存放「T-MAC」與「KOBE」兩位球員的資料：

- 「T-MAC」$= 84 * 10^4 + 45 * 10^3 + 77 * 10^2 + 65 * 10 + 67 = 893417$
- 「KOBE」$= 75 * 10^3 + 79 * 10^2 + 66 * 10 + 69 = 83629$

以上的範例，至少需要一個大小為893418的Array，可是只有存放兩個英文名字，不划算。

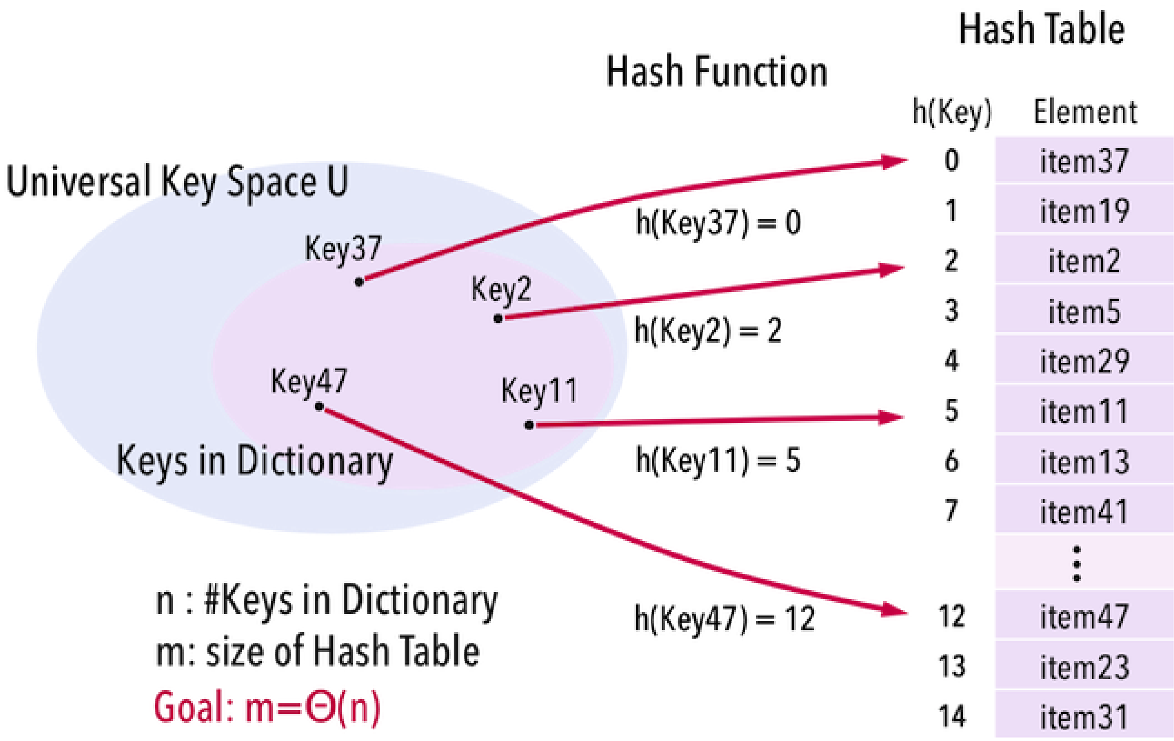
**Hash Table**的核心概念就是要解決第二個缺陷：避免記憶體空間浪費。

# Hash Table的概念

**Hash Table**希望能夠將存放資料的「Table」的大小(size)降到「真正會存放進Table的資料的數量」，也就是「有用到的**Key**的數量」：

- 若有用到的**Key**之數量為$n$，**Table**的大小為$m$，那麼目標就是$m = \Theta(n)$。

要達到這個目標，必須引入**Hash Function**，將**Key**對應到符合**Table**大小$m$的範圍內，$index = h(Key)$，即可成為**Hash Table**的**index**，如圖四。




**圖四：Hash Table和Direct Access Table的差別在於Hash Function。**

可惜事與願違，因為$|U| \gg m$，再加上**Hash Function**設計不易，所以很可能發生**Collision**。

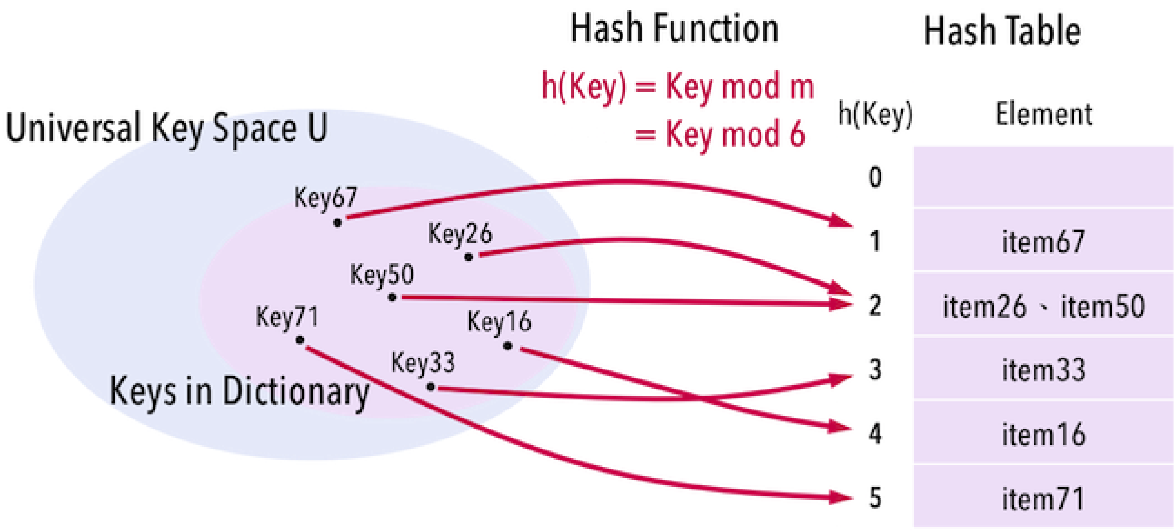
## 很可能發生Collision

**Collision**就是兩筆資料存進同一個**Table**之**slot**的情形，這將會使得查詢資料失敗(例如：使用item1的**Key**，卻回傳item2的資料)。

若以**Division Method**實作**Hash Function**，定義$h(Key) = Key \bmod m$，**Table**大小為$m = 6$，目前要處理的資料之**Key**有$67, 26, 50, 33, 16, 71$，那麼各個Key將被對應到的**index**如下，同時參考圖五：

- $h(67) = 67 \bmod 6 = 1$
- $h(26) = 26 \bmod 6 = 2$
- $h(50) = 50 \bmod 6 = 2$
- $h(33) = 33 \bmod 6 = 3$
- $h(16) = 16 \bmod 6 = 4$
- $h(71) = 71 \bmod 6 = 5$

「item26」與「item50」經過**Hash Function**後，同時想要將資料存進 `Table[2]` ，這就是**Collision**。




**圖五：。**

**Collision**在「可能使用到的**Key**」之數量遠大於**Table**大小(亦即$|U| \gg m$)的情況下，無可避免。

解決的辦法有二：

1. **Chaining**：使用Linked list把「Hashing到同一個slot」的資料串起來。
2. **Open Addressing**：使用**Probing Method**來尋找**Table**中「空的slot」存放資料。

這兩個方法將分別在後續文章介紹。

# Hash Function介紹

優秀的**Hash Function**$(h())$應具備以下特徵：

- 定義$h()$的定義域(domain)為整個**Key**的宇集合$U$，值域(range)應小於**Table**的大小$m$：

$$h : U \to \{0, 1, \ldots, m-1\},\ where\ |U| \gg m$$

- 盡可能讓**Key**在經過**Hash Function**後，在值域(也就是**Table**的**index**)能夠平均分佈(uniform distributed)，如此才不會有「兩筆資料存進同一個**Table**空格(稱為**slot**)」的情況。

若把**Table**想像成「書桌」，**slot**想像成書桌的「抽屜」，那麼為了要能更快速找到物品，當然是希望「每一個抽屜只放一個物品」，如此一來，只要拿著**Key**，透過**Hash Function**找到對應的抽屜(**Hash Function**的功能是指出「第幾個」抽屜，也就是抽屜的**index**)，就能保證是該**Key**所要找的物品。
反之，如果同一個抽屜裡有兩個以上的物品時，便有可能找錯物品。

以下介紹兩種**Hash Function**的基本款：

1. **Division Method**：$m$有限制，但是比較快。
2. **Multiplication Method**：$m$沒有限制，但是比較慢。

## Division Method

要把大範圍的$|U|$對應到較小範圍的$\{0, 1, \ldots, m-1\}$，最直覺的做法就是利用**Modulus**(mod)取餘數。
假設**Table**大小為$m$，定義**Hash Function**為：

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

例如，選定**Table**大小為$m = 8$，那麼以下的**Key**與**Table**之**index**將有對應關係如下：

- $h(14) = 14 \bmod 8 = 6$
  - 代表「編號14」的物品要放進「第6格」抽屜。
- $h(23) = 23 \bmod 8 = 7$
  - 代表「編號23」的物品要放進「第7格」抽屜。
- $h(46) = 46 \bmod 8 = 6$
  - 代表「編號46」的物品要放進「第6格」抽屜。
- $h(50) = 50 \bmod 8 = 2$
  - 代表「編號50」的物品要放進「第2格」抽屜。

### 優點

以**Division Method**實現**Hash Function**的優點就是非常快，只要做一次餘數(一次除法)運算即可。

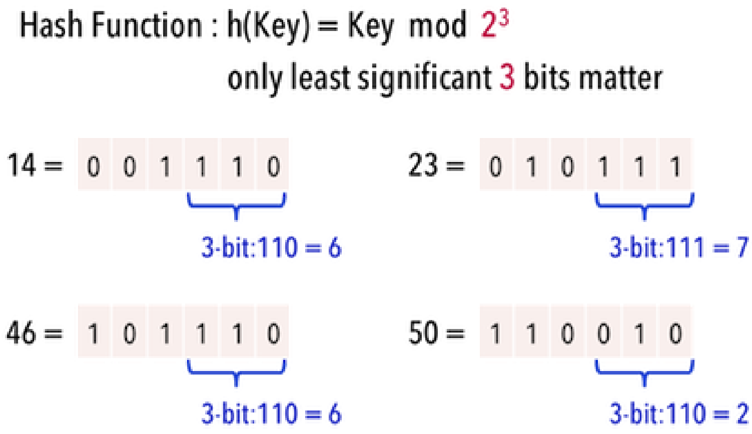
### 缺點

較為理想的**Table**大小$m$是「距離$2^p$夠遠」的質數，像是701。

換句話說，**Table**大小$m$必須慎選。

例如，要儘量避開「2的指數($2^p$)」，否則就只有「最低位的p-bit」會影響**Hash Function**的結果。

轉換成二進位會更容易看出，以$m=8=2^3$為例，$h(Key)=Key \bmod 2^3$的意思就是，只取「以二進位表示的**Key**的最低位的**3個bit**」來決定**Key**對應到的**Table**之**index**，見圖六。



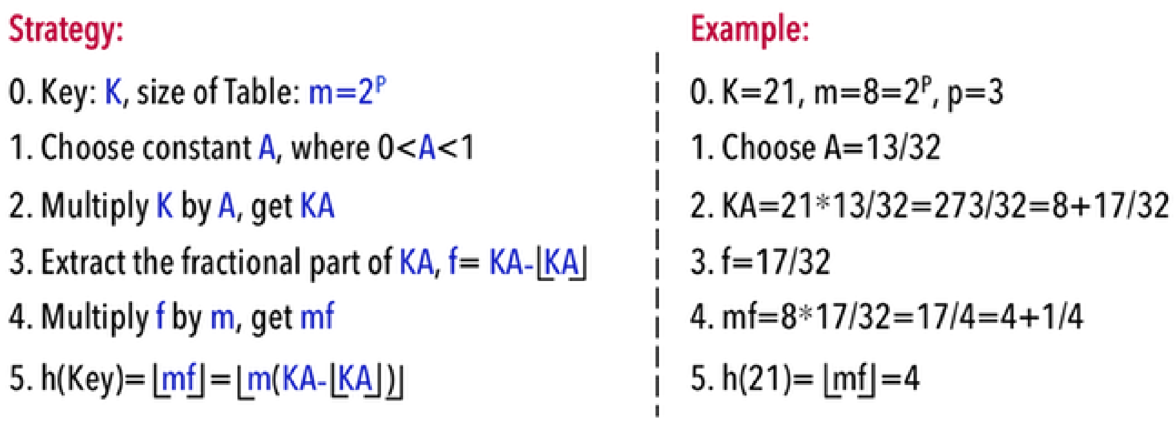

**圖六：。**

這種情況下，若有大量變數以相同的命名規則，例如「 `a_count` 、 `b_count` 、 `c_count` 」，很有可能在**prehash**(在**Direct Access Table**提過的「T-MAC」與「KOBE」)將字串轉換成**Key**時，得到「低位bit」完全相同的**key**，因為以上三個變數的結尾都是 `_count` ，那麼**Division Method**就會把這三個變數都放進同一個**slot**，造成**Collision**。

## Multiplication Method

由於在實際面對資料時，時常無法預先得知「**Key**的範圍」以及「在該範圍內**Key**的分佈情形」，在這個前提下，不需要避開特定$m$的**Multiplication Method**可能會比較優秀。

步驟如下：



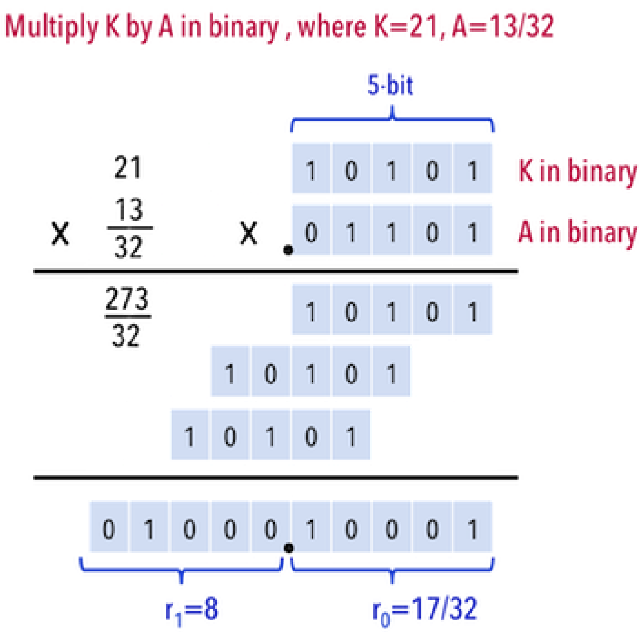

**圖七(a)：。**

而且這個將「**Key**乘上$A$、取小數點部分、再乘上$m$、再取整數部分」的**Hash Function**能夠儘量把更多的**Key**的bit納入考慮，來得到$h(Key)$。

轉換成二進位會更容易看出來。
先把圖七(a)中的「$constant\ A = \frac{13}{32}$」轉換成二進位：

$$\begin{aligned}\frac{13}{32} &= \frac{8}{32} + \frac{4}{32} + \frac{1}{32} \\ &= \frac{1}{4} + \frac{1}{8} + \frac{1}{32} \\ &= 0.01101_{(2)}\end{aligned}$$

再把「$K$乘上$A$」以二進位表示，見圖七(b)：




**圖七(b)：。**

得到的$r_0 = \frac{17}{32}$其實就是$KA$的小數部分(fractional part)，亦即$r_0 = KA - \lfloor KA \rfloor$，也就等於圖七(a)中的「f」。

那麼，接下來對「$KA$的小數部分乘上$m = 2^3$」，以二進位表示法解讀，就會是把「小數點往右移3位」：

$$r_0 = \frac{17}{32} = 0.10001_{(2)}$$
$$m \times r_0 = 2^3 \times \frac{17}{32} = \frac{17}{4}$$
$$= 100.01_{(2)}$$

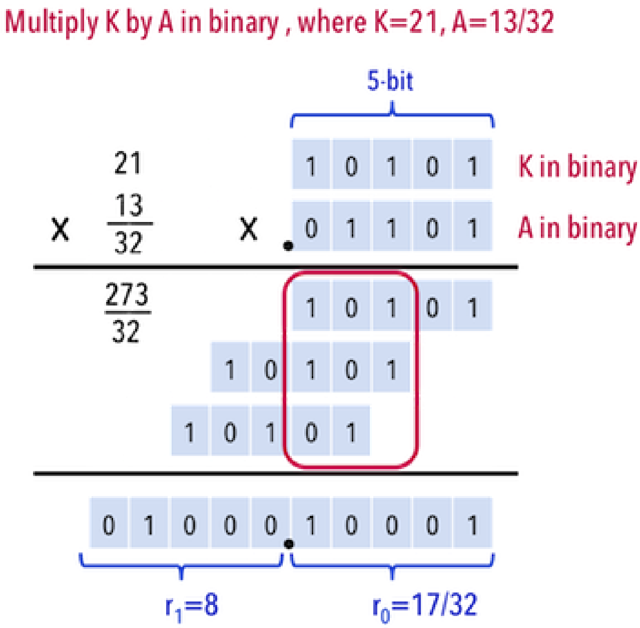
最後一步：$h(K) = \lfloor m(KA - \lfloor KA \rfloor) \rfloor$，其實就是取$m \times r_0$的整數部分，結果與圖七(a)吻合。

$$\lfloor m \times r_0 \rfloor = 100_{(2)} = 4$$

到這裡，再看一次圖七(b)，可以發現，決定出$h(21) = 4$的過程當中，以二進位表示的**Key** $= 21$的「每一個部分的bit」都用上了，二進位的4是由二進位的$Key = 21$之：

- 頭：101
- 中：101
- 尾：01

三者相加得到的，因為參與的「部位」變多了，那麼隨機性也就增加了，如此$h(Key)$便能得到較為隨機的結果。




**圖七(b)：。**

以上範例是將**Key**以5-bit表示，試想，若以32-bit表示，並且增加**Key**的範圍，那麼可以預期「參與並相加出最後$h(Key)$」的部分會更加「隨機」，也就能夠將不同**Key**「更隨機地」對應到不同的值(也就是不同的**slot**)，有效降低**Collision**的發生。

還有一個小議題是，如何找到「$constant\ A$」？
根據Knuth (https://en.wikipedia.org/wiki/Donald_Knuth)的說法，選擇黃金比例還不錯：

$$A = \frac{\sqrt{5}-1}{2} \approx 0.6180339887...$$

至於程式的實作上，利用**bit-shifting**會更有效率，請參考：Geoff Kuenning：Hash Functions (https://www.cs.hmc.edu/~geoff/classes/hmc.cs070.200101/homework10/hashfuncs.html)

最後，在Hashing的議題裡，還有兩位大魔王叫做**Universal Hashing**以及**Perfect Hashing**(有鑒於筆者可能一輩子都看不懂，所以這裡就放個連結)，據說可以產生最低限度的**Collision**，請參考：

- CMU14-451,Algorithms：Universal and Perfect Hashing (https://www.cs.cmu.edu/~avrim/451f11/lectures/lect1004.pdf)
- Wikipedia：Universal Hashing (https://en.wikipedia.org/wiki/Universal_hashing)
- Sarah Adel Bargal：Universal Hashing (http://cs-www.bu.edu/faculty/homer/537/talks/SarahAdelBargal_UniversalHashingnotes.pdf)

以上是**Hash Table**之基本概念介紹。

接下來兩篇將介紹**Chaining**與**Open Addressing**解決**Collision**。

---

## 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)
- Geoff Kuenning：Hash Functions (https://www.cs.hmc.edu/~geoff/classes/hmc.cs070.200101/homework10/hashfuncs.html)
- MIT 6.006：Lecture 8: Hashing with Chaining (http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-006-introduction-to-algorithms-fall-2011/lecture-

videos/lecture-8-hashing-with-chaining/)
- Wikipedia：Universal Hashing (https://en.wikipedia.org/wiki/Universal_hashing)
- CMU14-451,Algorithms：Universal and Perfect Hashing (https://www.cs.cmu.edu/~avrim/451f11/lectures/lect1004.pdf)
- Sarah Adel Bargal：Universal Hashing (http://cs-www.bu.edu/faculty/homer/537/talks/SarahAdelBargal_UniversalHashingnotes.pdf)
- Explanation from Ou-Yang,Hui (https://www.facebook.com/profile.php?id=100000181170314&fref=ts)

## 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), Intro (http://alrightchiu.github.io/SecondRound/tag/intro.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)