

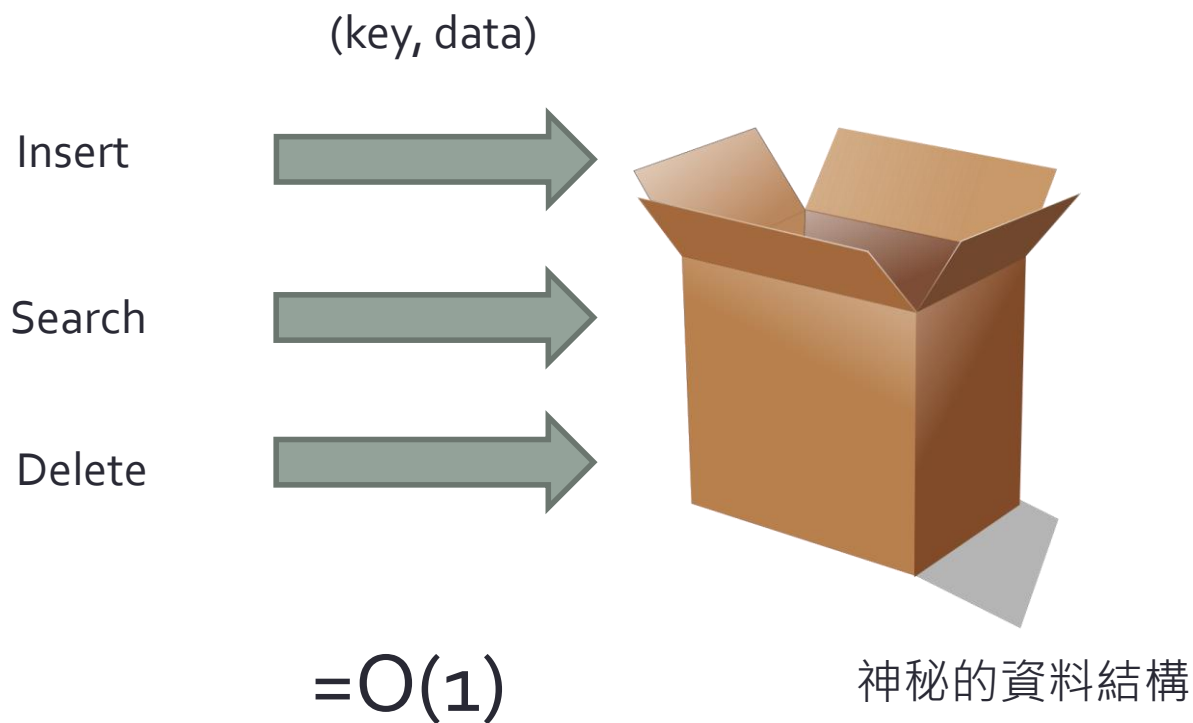
HASHING

Michael Tsai

2013/06/04



有沒有一種天方夜壇

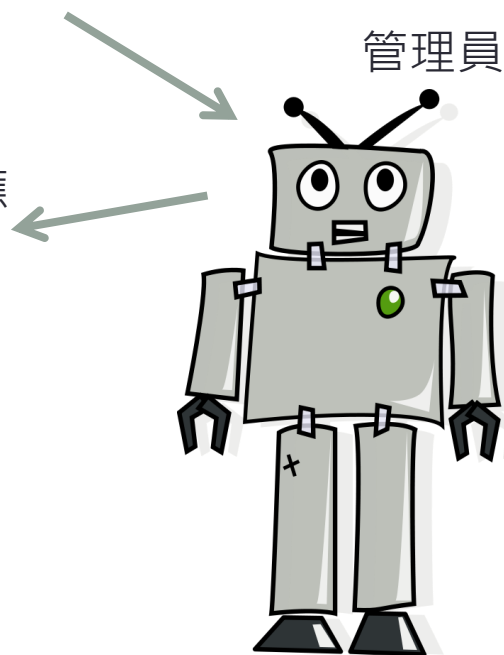


Hint: “以空間換取時間”

概念

問：“菜瓜布”的資料去哪找？
（“菜瓜布”，資料）

管理員：“菜瓜布”對應
到1028號櫃子



很多很多有編號的櫃子



如果箱子夠多, 則花費在一個箱子裡面尋找的時間= $O(1)$

概念

問：“菜瓜布”的資料去哪找？

Hash function: $h(k)$

管理員

管理員：“菜瓜布”對應到1028號櫃子

key: 拿來當索引的東西

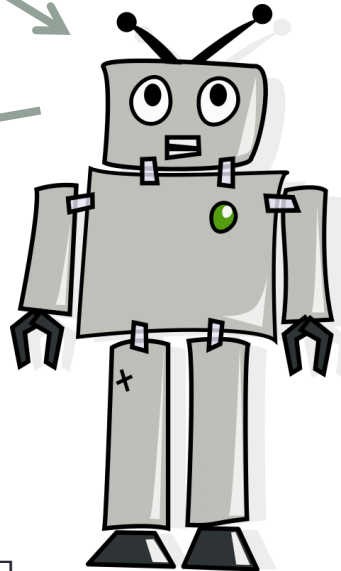
例如：“菜瓜布”

$T=|U|$: 所有可能的key的數目

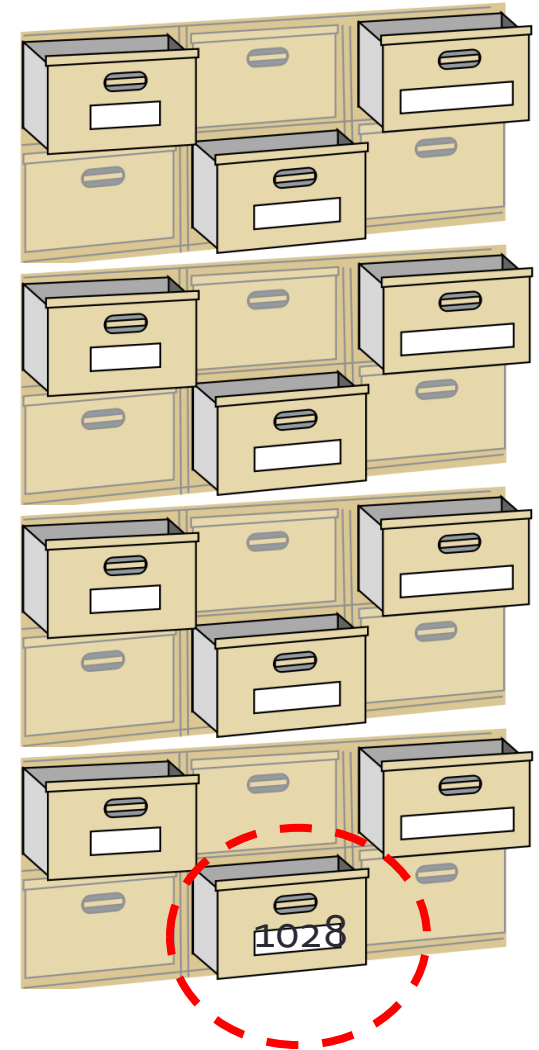
$n=|K|$: 所有要存入的pair的數目

Key density: n/T

Load density (load factor): n/sm



很多很多有編號的櫃子

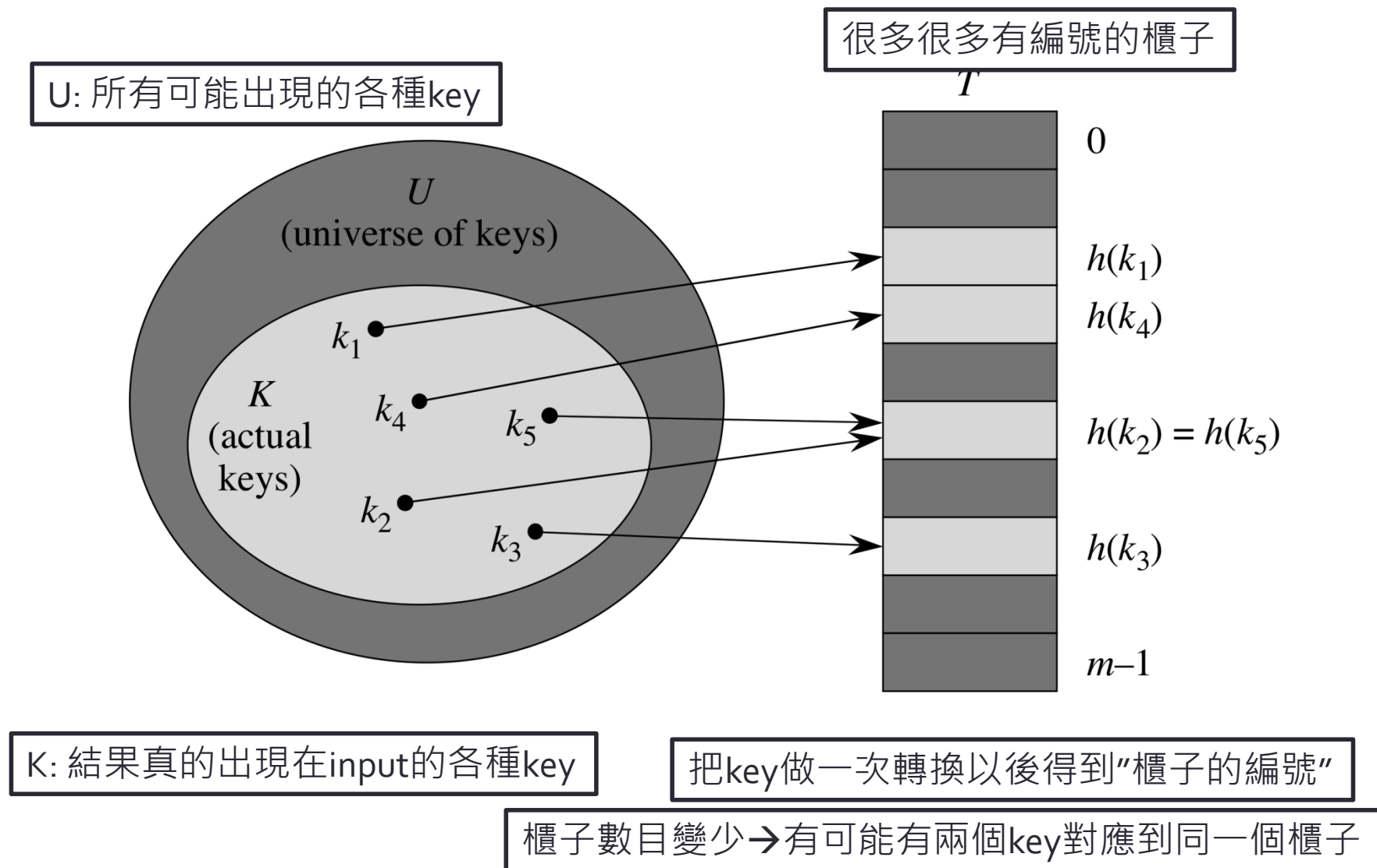


櫃子數目: m

每個位子可以放的資料數: s

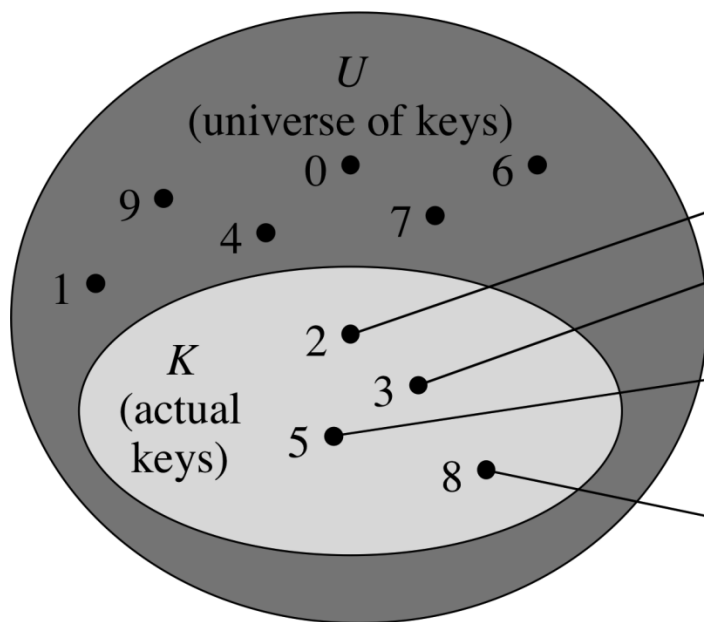
sm : 所有可以放數櫃子資料數目

概念：Hash Table



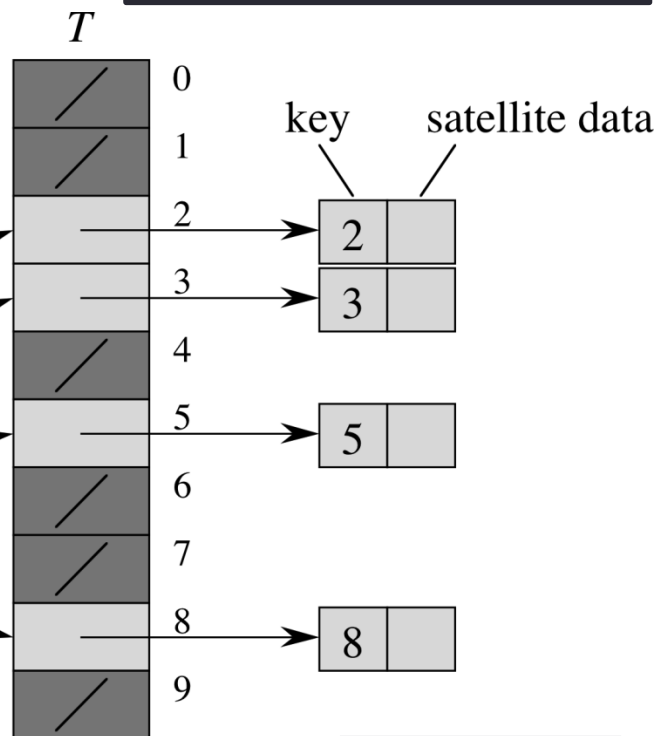
概念：Direct-address Table

U: 所有可能出現的各種key



K: 結果真的出現在input的各種key

很多很多有編號的櫃子



裡面存資料

如果 $|K| \ll |U|$ ，櫃子就浪費很多空間

一些定義

- $h(k)$: hash function
- hash function 把key對應到一個數值(通常為櫃子編號)
- 有可能把不同的key對應到同一個數值
- (但是沒關係)
- 如果 $h(k_1) = h(k_2)$, 則 k_1, k_2 are synonyms with respect to h .
- 最簡單的hash function: $k \% m$ ($k \bmod m$)
- collision: 要把資料存進某櫃子的時候, 該櫃子已經有東西了
- overflow: 要把資料存進某櫃子的時候, 該櫃子已經滿了
- if $s == 1$, 則每次collision都會造成overflow (通常 $s == 1$)

為什麼是 $O(1)$

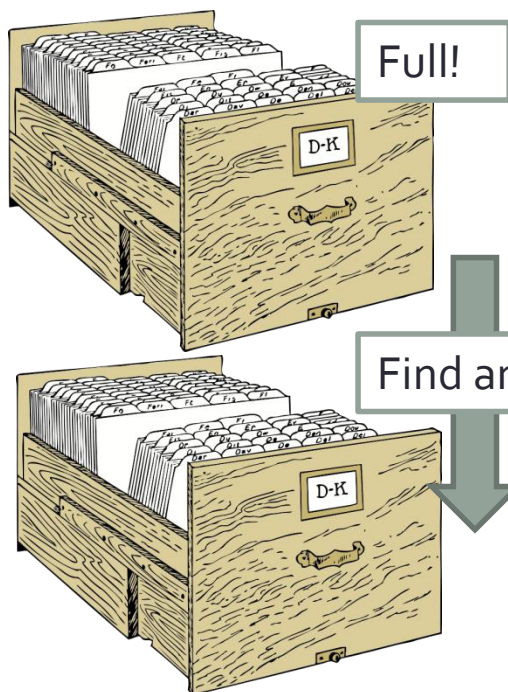
- 當沒有overflow的時候:
- 計算hash function的時間: $O(1)$
- 進到某一個櫃子去insert, delete, search的時間都是 $O(1)$
- worst case為尋找s個空間的時間: 固定
- 所以為 $O(1)$
- 剩下的問題:
 - (1) 當collision發生的時候怎麼處理?
 - (2) 怎麼implement一個好的hash function?

Collision 處理

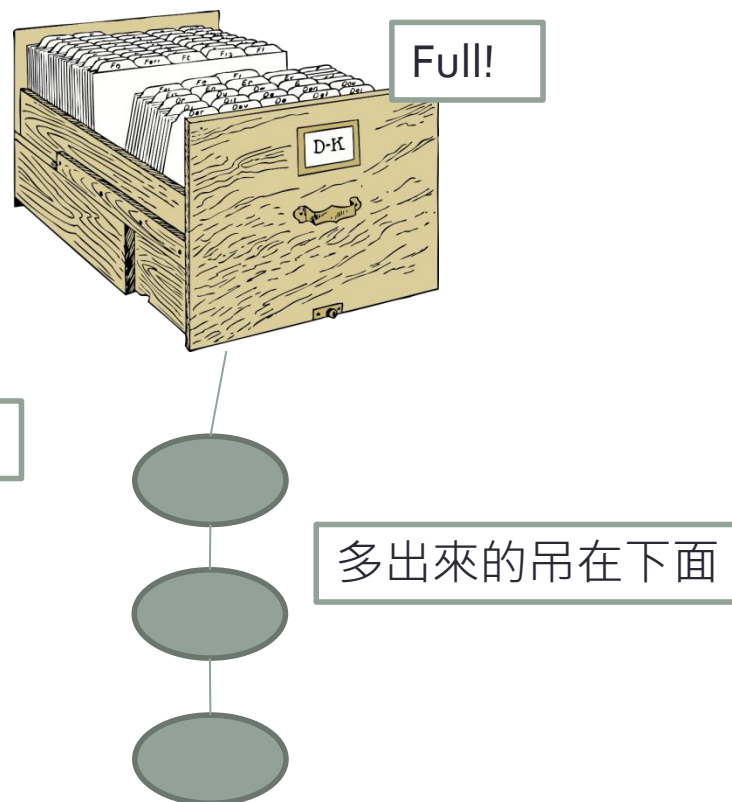
注意: 要確保能夠下次也能找到同一個地方!

- 兩種常用處理collision的方法:

(1) Open addressing



(2) Chaining



Open addressing – Linear probing

- 有好幾種方法:
- (1) Linear probing
- $T[(h(k)+1)\%m]$, $T[(h(k)+2)\%m]$, ...
- Insert的時候順著往下找 (找的動作又叫做probe):
- 一直找到
 1. 有空位 \rightarrow 填入
 2. 回到原來的位置 $h(k)$ 了, 則沒有空位 \rightarrow 可能要擴大.
(load factor 永遠小於1)
- Search的時候, 一樣是從 $T[h(k)]$ 開始往下找, 一直找到
 1. 有空位 $\rightarrow k$ 不在table裡
 2. 找到了, k 在 $T[(h(k)+j)\%m]$ 的位置
 3. 回到原本的位置 $h(k)$ 了, k 不在table裡面

Open addressing

- 好處:
 - 利用hash table裡面沒有儲存東西的空間
 - 不用使用記憶體來存pointer, 省下來的記憶體可以開更大的hash table
- 壞處:
 - 尋找overflow出去的element需要花額外的時間(不是 $O(1)$ 了)
 - 讓在櫃子裏面的key容易集結(clustering)在一起
→ 平均尋找時間更長

Open addressing – General Form

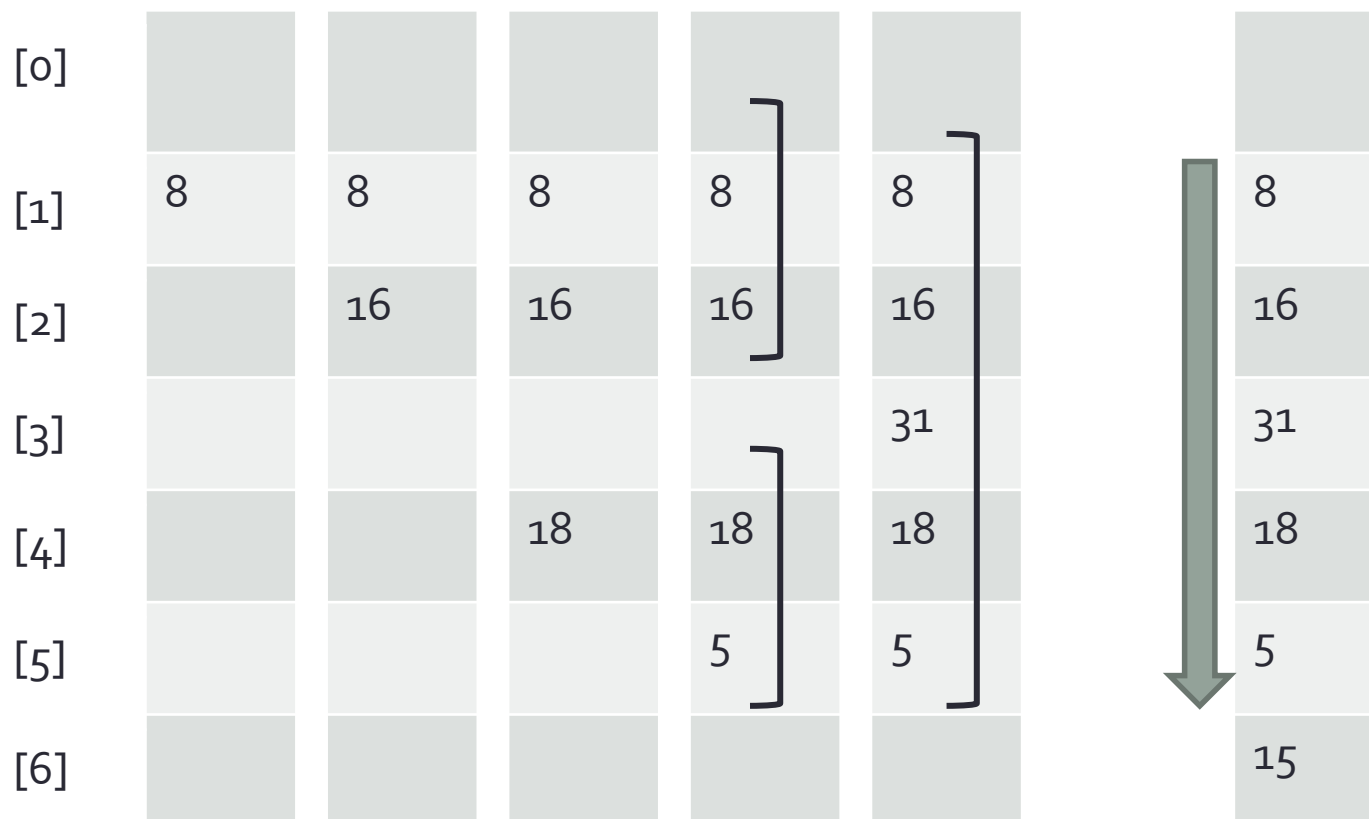
- 我們把hash function h 變成以下的形式:
 - $h: U \times \{0, 1, \dots, m - 1\}$
 - 也就是我們probe的順序為 (probing sequence)
 - $\langle h(k, 0), h(k, 1), h(k, 2), h(k, m - 1) \rangle$
 - 以上為 $\langle 0, 1, 2, \dots, m - 1 \rangle$ 的排列組合
-
- 所以Linear probing的可以寫成:
 - $h(k, i) = (h'(k) + i) \% m$
 - $h'(k)$ 是原本的hash function
 - Linear probing總共只有 m 種probing sequence

Open addressing – Linear probing

Input sequence of keys: {8,16,18,5,31,15}

Primary clustering:

某些open addressing的probing方法會產生一長串填滿的格子



Open addressing – Quadratic probing

- $h(k, i) = (h'(k) + c_1 i + c_2 i^2) \% m$, c_1, c_2 為正常數
- 例1: 我們可以用 $h(k, i) = (h'(k) + i^2) \% m$
- $\langle h'(k) \% m, (h'(k) + 1^2) \% m, (h'(k) + 2^2) \% m, \dots, (h'(k) + (m - 1)^2) \% m \rangle$
- 例2: 如果 $m = 2^n$, 則我們可以用 $h(k, i) = \left(h'(k) + \frac{i}{2} + \frac{i^2}{2} \right) \% m$
- $\langle h'(k) \% m, (h'(k) + 1) \% m, (h'(k) + 3) \% m, (h'(k) + 6) \% m, (h'(k) + 10) \% m, \dots \rangle$
- 用這些方法可以使得clustering的現象較為減輕: Secondary Clustering
- 只有當一開始的hash function產生一樣的位置才會造成一樣的probing sequence
- $h(k_1, 0) == h(k_2, 0) \implies h(k_1, i) == h(k_2, i)$
- 和linear probing一樣, 只有 m 種probing sequence (開始的 $h'(k)$ 決定sequence)

Open addressing – Double hashing

- $h(k, i) = (h_1(k) + i h_2(k)) \% m$
- 為open addressing最好的方法之一
- 例子: $h_1(k) = k \% m, h_2(k) = 1 + (k \% (m - 1))$
- 如果 $k=123456, m=701, h_1(k) = 80, h_2(k) = 257$
- 一開始找 $T[80]$, 後面每隔257格找一次
- 關鍵: 即使 $h_1(k_1) == h_1(k_2), h_2(k_1) == h_2(k_2)$ 應該不成立
- 因此probing sequence有 m^2 種!
- (通常須要求 $m = 2^n$)
- Double hashing是最接近“uniform hashing”的方法
- Uniform hashing: 任何probing sequence出現的機率是一樣的
- 也就是 $\langle 0, 1, 2, \dots, m - 1 \rangle$ 的任一種排列組合出現的機率是一樣的

來做一些分析(沒有推導)

- 在Uniform Hashing的假設下:
- Expected number of probes:
- 尋找一個key時平均所需要找(比較)的key個數
- 因為其他的operation都只需要 $O(1)$, 所以這個動作決定了search的time complexity
- α : load factor= $n/m < 1$
- 失敗(找到空位): $\frac{1}{1-\alpha} = 1 + \alpha + \alpha^2 + \alpha^3 + \dots$
- 成功: $\frac{1}{\alpha} \ln \frac{1}{1-\alpha}$
- (詳細的證明參見 Cormen p.274-276)
- Worst case?
- 全部都連在一起, 全部都填滿了
- $O(n)$

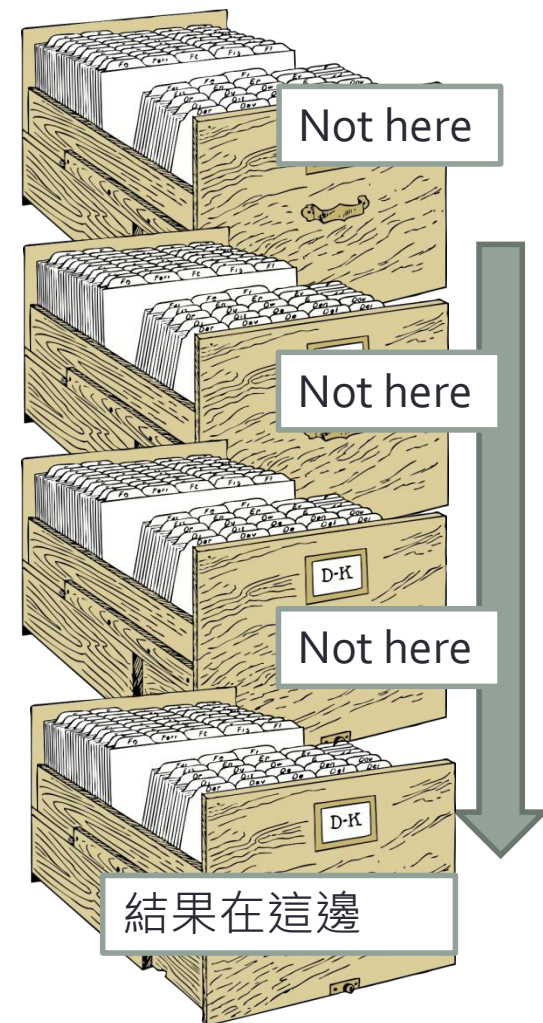
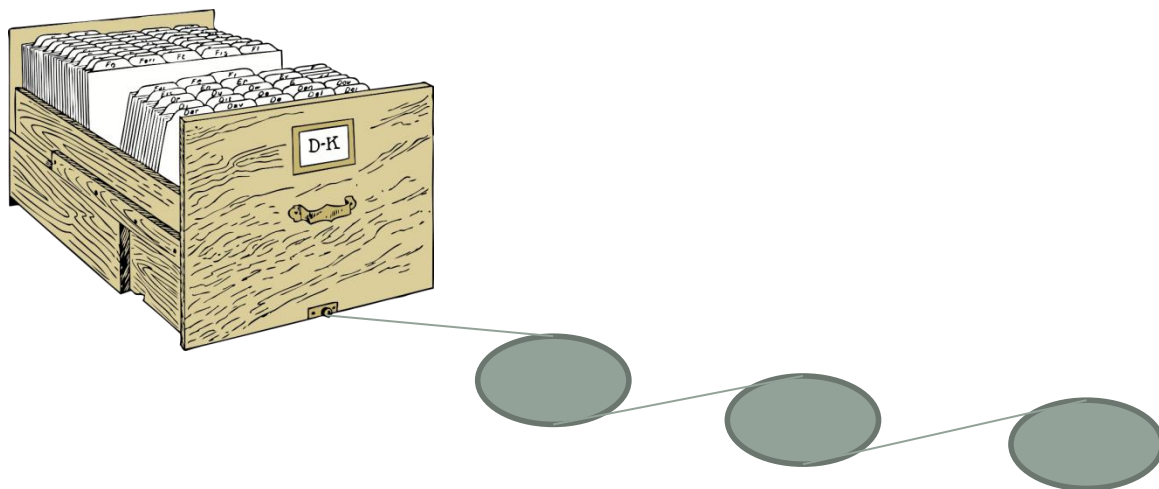
第一次一定要找

第二次有 α 機率要找

第三次有 α^2 機率要找

Chaining

- 之前的方法的缺點?
- 尋找過程中, 需多其他的資料的hash值和現在要找的key k 的hash值根本就不一樣
→ 有點冤枉
- 所以採取“掛勾”的方法
- 每個櫃子是一個linked list
- 搜尋的時候只會找掛在下面的 ($h(k)$ 都一樣)



Chaining – Worst case

- Worst case:
- 全部都塞在同一個櫃子下面的linked list
- time complexity這樣是?
- $O(n)$
- 小小的進步: 底下可以用binary search tree
(之後有balanced 版)
- 可以進步到 $O(\log n)$

Chaining - Expected performance

- 每個櫃子的chain上面平均有幾個pair?
- n : 總共存入的資料pair數目
- m : 櫃子數目
- 所以假設使用simple uniform hashing的話
- 也就是存到每個櫃子的機率相等
- 平均一個chain有 n/m 個pair (α 個pair)
- 這也是如果找不到的話, 平均需要比較的次數
- 加上hash本身要花的時間, 總共為 $\Theta(1 + \alpha)$

- 如果是找得到的話, 平均需要比較的次數為 $1 + \frac{\alpha}{2} - \frac{\alpha}{2n}$
- 加上hash本身要花的時間, 總共仍為 $\Theta(1 + \alpha)$
- (詳細證明可見Cormen p.260)

- 因此總體來說, 只要 $n = O(m)$, $\alpha = \frac{n}{m} = \frac{O(m)}{m} = O(1)$
- n 為 m 的一個比例時, 總時間可為constant time!

Hash function

- 先要知道的事情:
- 不可能讓所有key都map到不同的櫃子
- (因為 $|K|$ 遠大於櫃子數目)
- 目標:
- (1) 希望隨便取一個key, 則平均來說它存到任何一個櫃子的機率都是 $1/m$ (m 為櫃子數目) (都是一樣的)
- (2) 計算hash function的時間為 $O(1)$
- 當(1)符合時, 此hash function稱為simple uniform hashing (hash function)

一些hash function的例子

- 複習: $h(k)$ 把 k 轉成另外一個數字 (櫃子編號)
- (1) Division: $h(k)=k\%D$
- 則結果為 $0 \sim D-1$ 通常我們可以把 D 設為櫃子數目
- (2) Mid-square: $h(k)=bits_{i,i+r-1}(k^2)$
- 則結果為 $0 \sim 2^r - 1$, 所以通常櫃子數目為 2^r

一些hash function的例子

- (3) shift folding
- 用例子解釋:
- $k=12320324111220$
- 每隔幾位數切一份. 例如, 三位數: (櫃子有1000個)
- $\{123, 203, 241, 112, 20\}$
- $h(k)=(123+203+241+112+20)\%1000=699$

- (4)folding at the boundaries
- $\{123, 302, 241, 211, 20\}$
- $h(k)=(123+302+241+211+20)\%1000=897$

一些hash function的例子

- (5) digit analysis
 - 假設先知道所有的key了
 - 此時就可以尋找一個比較好的hash function
 - 假設k有5位數, 我們有100個櫃子
 - 則需要把5位數轉換成2位數
 - 則我們可以每次選某一位數來分類成10組
 - 最不平均的3個位數可以刪掉
 - (記得: 最好可以使得分到某櫃子的機率都相等)
- (6) Multiplication Method: 取 $0 < A < 1$, then $h(k) = \lfloor m (kA \% 1) \rfloor$
- (參看 Cormen p.264)

Key是string怎麼辦?

- 轉成數字! (然後再使用hash function)
- 可不可以把不同字串轉成一樣數字?
- 答: 可以! 反正hash function一樣已經會把不同key轉成一樣的櫃子號碼了
- 方法:
 - (1) 把所有字串的character(數字)加起來, 進位的通通丟掉. (類似checksum)
 - (2) 把所有字串的character (數字)分別往左位移i格, i為該character在字串中的位置, 然後通通加起來.
- 舉例☺

Dynamic hashing

- 觀察: 當 n/m 比較大以後, $O(1)$ 就開始崩壞 (往 $O(n)$ 方向移動)
- 應變: 所以要隨時觀察 n/m , 當它大過某一個threshold時就把 hash table 變大
- 觀察: 把hash table變大的時候,
- 需要把小hash table的東西通通倒出來,
- 算出每一個pair在大hash table的位置
- 然後重新放進大hash table
- 有個可憐鬼做insert正好碰到應該hash table rebuild的時候, 他就會等非常非常久.T_T

Dynamic hashing

- 目標: 重建的時候, 不要一次把所有重建的事情都做完了
- 或許, 留一些之後慢慢做?
- 每個operation的時間都要合理
- 又叫做extendible hashing

例子

k	h(k)
A ₀	100 000
A ₁	100 001
B ₀	101 000
B ₁	101 001
C ₁	110 001
C ₂	110 010
C ₃	110 011
C ₅	110 101

$h(k,i)$ =bits 0-i of $h(k)$

Example:

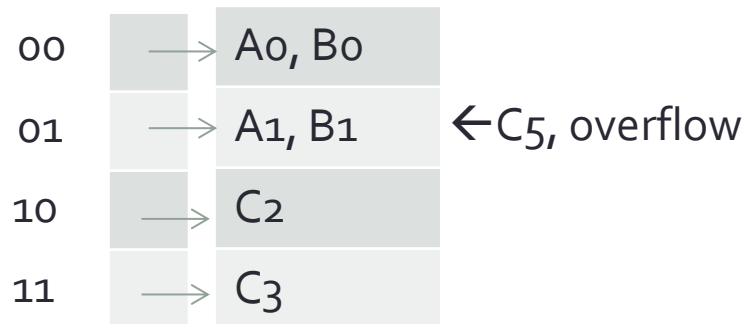
$h(A_0,1)=0$

$h(A_1,3)=001=1$

$h(B_1,4)=1001=9$

Dynamic hashing using directories

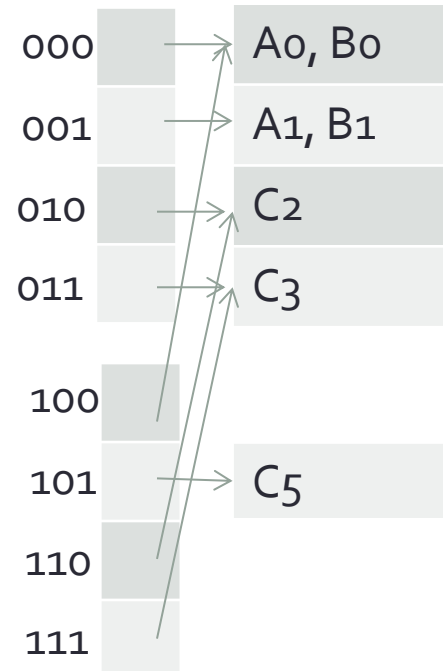
directory depth =
number of bits of the index of the hash table



Insert C₅

$$h(C_5, 2) = 01 = 1$$

we increase d by 1
until not all $h(k, d)$ of the keys in the cell are the same



k	h(k)
A ₀	100 000
A ₁	100 001
B ₀	101 000
B ₁	101 001
C ₁	110 001
C ₂	110 010
C ₃	110 011
C ₅	110 101

動腦時間:

如果原本的要加入C₁呢?

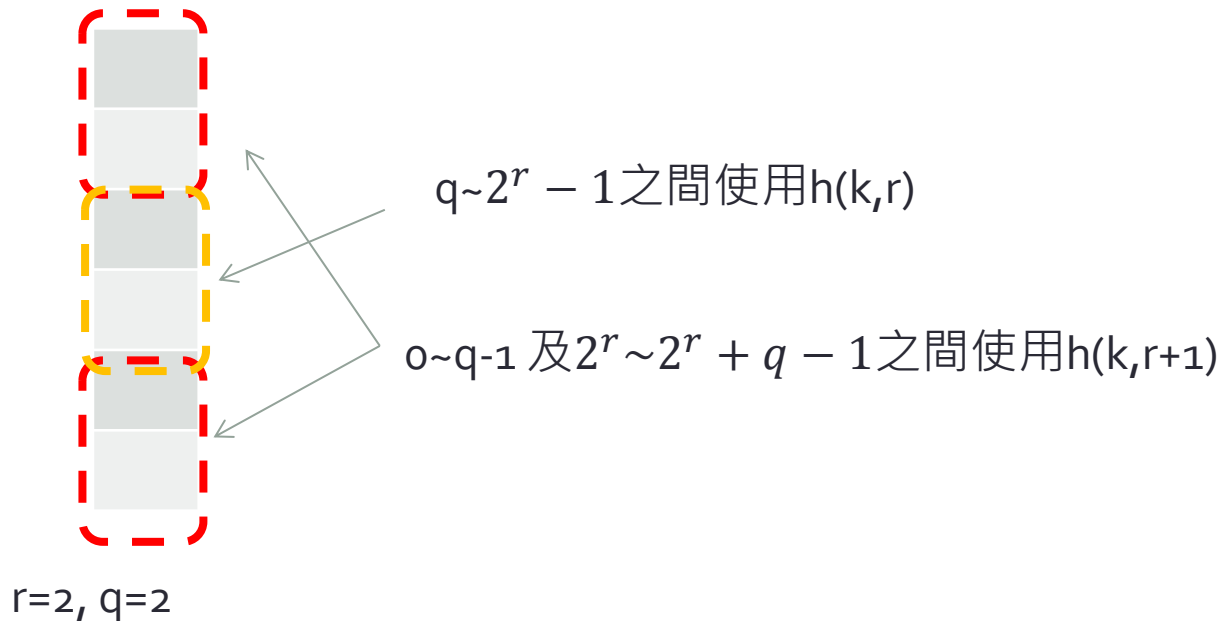
如果第二步驟後加入A₄呢? 答案: Horowitz p. 412-413

Dynamic hashing using directories

- 為什麼比較快?
- 只需要處理overflow的櫃子
- 如果把directory放在記憶體, 而櫃子資料放在硬碟
- 則
- search只需要讀一次硬碟
- insert最多需要讀一次硬碟(讀資料, 發現overflow了), 寫兩次硬碟(寫兩個新的櫃子)
- 當要把hash table變兩倍大時, 不需要碰硬碟(只有改directory)

Directoryless Dynamic hashing

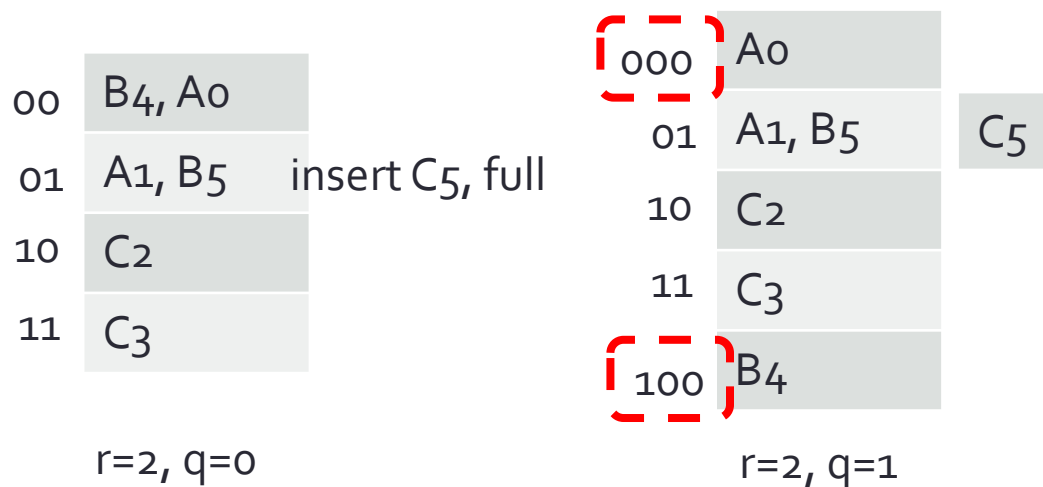
- 假設hash table很大, 但是我們不想一開始就整個開來用 (initialization會花很大)
- 用兩個變數來控制的hash table大小: r, q
- hash table開啟的地方為 $0, 2^r + q - 1$ 之間



Directoryless Dynamic hashing

- 每次輸入的時候, 如果現在這個櫃子滿了
- 則開一個新的櫃子: $2^r + q$
- 原本 q 櫃子裡面的東西用
- $h(k, r+1)$ 分到 q 和 $2^r + q$ 兩櫃子裡
- 注意有可能還是沒有解決問題
- 多出來的暫時用chain掛在櫃子下面

問:再加入C1呢? (Horowitz p.415)



k	h(k)
A ₀	100 000
A ₁	100 001
B ₄	101 100
B ₅	101 101
C ₁	110 001
C ₂	110 010
C ₃	110 011
C ₅	110 101

Today's Reading Assignment

- Cormen ch 11 (11.1, 11.2, 11.3 except 11.3.3, 11.4)
- Horowitz p. 410-416 (posted on the course web site)