Algorithms
Chapter 11 Hash Tables

記錄書卷獎得主:田小陳列處理廣大人口

Associate Professor: Ching-Chi Lin
林清池 副教授
chingchi.lin@gmail.com

Department of Computer Science and Engineering
National Taiwan Ocean University

n: 書卷獎得主　U: 全校人口

dictionary operations: insert, delete, search

其他: min, max, predecessor, successor

	Search 時間	Space	功能
Hash Table	$O(1)$ expected	$O(n)$	dictionary
Array	$O(n)$	$O(n)$	dictionary
Binary Search Tree	$O(\log n)$	$O(n)$	dictionary + 其他
Direct-address-table	$O(1)$	$O(\lvert U \rvert)$	dictionary

Outline

- **Direct-address tables** 將key為k的element直接放在位置k
- Hash tables 儲存空間→是array的擴展→要儲存的數遠小於可能的數目
- Hash functions 用hash function算位置
  h(k)=k的hash value
- Open addressing 當hash value相同時要重新hash
  (兩個的)

Overview $_{1/3}$

許多應用程式會用到 dictionary operation
hash table implement dictionary operation 很有效率

- Many applications require a dynamic set that supports only the **dictionary operations** INSERT, SEARCH, and DELETE.
  dictionary operations = insert, search, delete
- Example: a symbol table in a compiler.
  符号表
- A hash table is effective for implementing a dictionary.
  - The expected time to search for an element in a hash table is $O(1)$, under some reasonable assumptions. 在某些合理的假設下,只要 O(1) 的時間去search
  - Worst-case search time is $\Theta(n)$, however.
    雖然最差需要 $\theta(n)$ 的時間
- A hash table is a generalization of an ordinary array.
  是ordinary array 的擴展
  - With an ordinary array, we store the element whose key is k in position k of the array. 在 ordinary array 時將 key 為 k 的 element 直接放在位置 k
  - Given a key k, we find the element whose key is k by just looking in the kth position of the array. This is called **direct addressing**.
    依照鍵值放位置

Overview$_{2/3}$

- We use a hash table when we do not want to (or can't) allocate an array with one position per possible key.
  當不能或不想為每一個key值保留一個專屬位置時，就用hash table
  - Use a hash table when the number of keys actually stored is small relative to the number of possible keys.
    要儲存的數目遠小於可能的數目. Ex: 2300,000,000中的50 <可能存 v.s. 要存>
  - A typically uses a size proportional to the number of keys to be stored (rather than the number of possible keys).
    hash table size跟要存的數目成一定比例，不是可能要存的數目
  - Given a key *k*, don't just use *k* as the index into the array.
    不会直接用key值k來放它的位置
  - Instead, compute a function of *k*, and use that value to index into the array. We call this function a **hash function**.
    位置是用hash function來計算

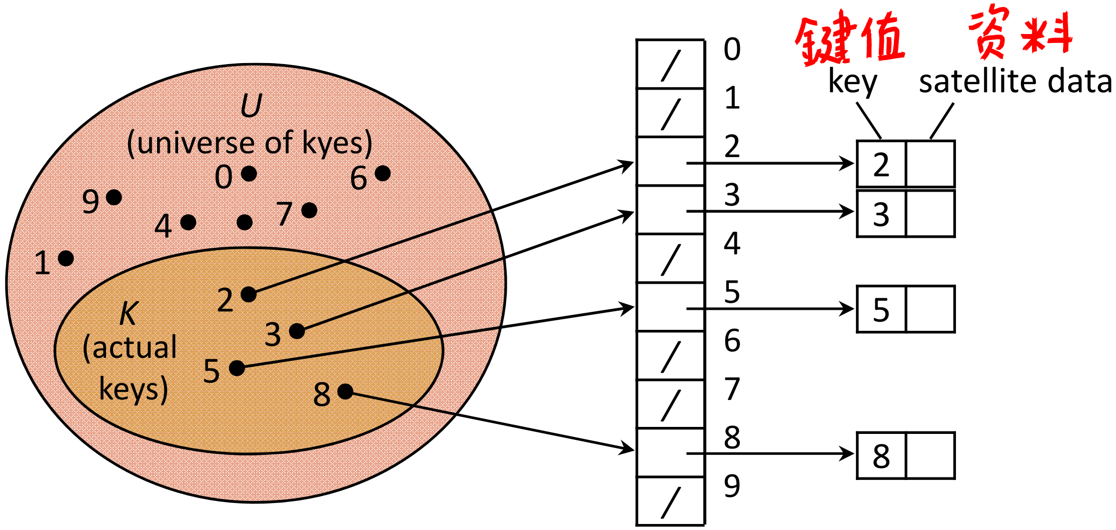
Overview$_{3/3}$

- Issues that we'll explore in hash tables:
  - How to compute hash functions? 如何算位置
    - The multiplication methods. 乘法法則
    - The division methods. 除法法則
  - What to do when the hash function maps multiple keys to the same table entry? 兩個人位置相同怎麼辦
    - Chaining. 鍵結<串起即可>
    - Open addressing.<重新計算結果>

Direct-address tables$_{1/2}$

- Scenario: 集合中的元素会变动 = 动態的集合，可能会 insert，delete
  - Maintain a dynamic set.
  - Each element has a key drawn from a universe $U = \{0, 1, \ldots, m-1\}$ where m isn't too large. key 值為 0 ~ m-1，共 m 個元素
  - No two elements have the same key. 任兩人鍵值皆不同
- Represent by a **direct-address table**, or array, $T[0..m-1]$:
  - Each **slot**, or position, corresponds to a key in U. 每個位置都有一個相对应的鍵值（在 U 中）
  - If there's an element x with key k, then $T[k]$ contains a pointer to x. 若位置 k 有，就用 pointer 指過去
  - Otherwise, $T[k]$ is empty, represented by NIL.
    若 T[k] 沒有 element 就為 NIL

Direct-address tables$_{2/2}$



U: 所有key值種類

k: 真正要儲存的元素

- Dictionary operations are trivial and take $O(1)$ time each:

DIRECT-ADDRESS-SEARCH(T, k)
return $T[k]$

DIRECT-ADDRESS-INSERT(T, x)
$T[key[x]] \leftarrow x$

DIRECT-ADDRESS-DELETE(T, x)
$T[key[x]] \leftarrow$ NIL

Outline

- Direct-address tables
- **Hash tables**
- Hash functions
- Open addressing

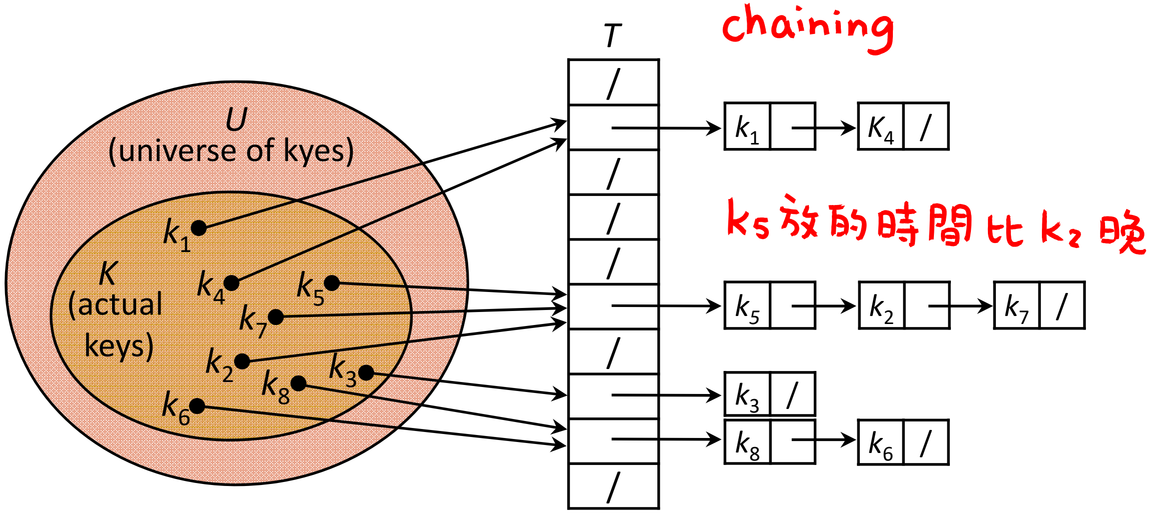
Hash tables$_{1/2}$

- **Problem:**
  - If the universe U is large, storing a table of size $|U|$ may be impractical or impossible. |U|太大⇒不可能儲存
  - The set K of keys actually stored is small, compared to U, so that most of the space allocated for array T is wasted.
    |K|相对於|U|很小⇒浪费空間，k為真正儲存的对象
- **Solution:** Hash tables
  - When K is much smaller than U, a hash table requires much less space than a direct-address table. 當|K|遠小於|U|時，較direct-address省空間
  - Storage requirements can be reduced to $\Theta(|K|)$. 需要Θ(|K|)的空間
  - Searching for an element requires $O(1)$ time, but in the **average case**, not the **worst case**.
    average case = O(1) 時間去search

用hash function算位置
h(k)=k 的 hash value

Hash tables$_{2/2}$

- **Idea:** Instead of storing an element with key *k* in slot *k*, use a function *h* and store the element in slot *h*(*k*).
  - We call *h* a **hash function**.
  - $h : U \to \{0, 1, \ldots, m-1\}$, so that $h(k)$ is a legal slot number in *T*.
    定義域 值域：0~m-1 array的大小為 m
  - We say that *k* **hashes** to slot *h*(*k*).
  - We also say that *h*(*k*) is the **hash value** of key *k*.



Collisions 碰撞

- **Collisions:** When two or more keys hash to the same slot.
  當兩個 key hash 到同一個位置
  - Can happen when there are more possible keys than slots ($|U| > m$).
  - Methods to resolve the collision problem.
    - **Chaining** 鏈結（串起即可）
    - **Open addressing**（重新計算得結果）
  - Chaining is usually better than open addressing.
    chaining 較佳
- **Collision resolution by chaining**
  - Put all elements that hash to the same slot into a linked list.
    放在同一個 linked list
  - Slot j contains a pointer to the head of the list of all stored elements that hash to j. 將最近的放在頭
    頭 後放 ⟷ 先放 chain
  - If there are no such elements, slot j contains NIL.
    沒有東西的話就變成 NULL

Dictionary Operations$_{1/2}$

- How to implement dictionary operations with chaining:
  - CHAINED-HASH-**INSERT**(*T*, *x*):直接放在頭

    Insert *x* at the head of list *T*[*h*(*key*[*x*])]
    - Worst-case running time is *O*(1). 不需 check 是否在 list 中
    - Assumes that the element being inserted isn't already in the list.
    - It would take an additional search to check if it was already inserted.

      若要check 是否已經 insert需要花更多時間 (+ chain長度)
  - CHAINED-HASH-**SEARCH**(*T*, *k*):

    Search for an element with key *k* in list *T*[*h*(*k*)]
    - Running time is proportional to the length of the list of elements in slot *h*(*k*). Time: O(slot h(k)的長度)

      整個 chain 都要看過才能確定是否有在其中

Dictionary Operations$_{2/2}$

- CHAINED-HASH-**DELETE**(*T*,*x*):

  Delete *x* from the list *T*[*h*(*key*[*x*])]

  - Given pointer *x* to the element to delete, so no search is needed to find this element. 因為直接給x的pointer，所以不需search
  - Worst-case running time is *O*(1) time if the lists are doubly linked.
    如果是double linked只需O(1) ∵已知前後為誰
  - If the lists are singly linked, then deletion takes as long as searching, because we must find *x*'s predecessor in its list.
    不是double linked，Time：O(slot h(k)的長度)
    整個chain都要看過才能確定前一個為誰

Analysis of hashing with chaining 用$\alpha = \frac{n}{m}$來描述時間

- Given a key, how long does it take to find an element with that key?
- Analysis is in terms of the **load factor** $\alpha = n / m$:
  - n = # of elements in the table. table中有n個元素
  - m = # of slots in the table = # of (possibly empty) linked lists.
    table slot的個數(table 的 size)
  - Load factor is average number of elements per linked list.
  - Can have $\alpha < 1$, $\alpha = 1$, or $\alpha > 1$. $\frac{n}{m} = \alpha$ = 平均而言每一個 list 的長度
- **Worst case** is when all n keys hash to the same slot
  - get a single list of length n. 全部都到同一個 slot
  - worst-case time to search is $\Theta(n)$, plus time to compute hash function.
    Time = search + hash = $\Theta(n) + O(1) = \Theta(n)$
- **Average case** depends on how well the hash function distributes the keys among the slots.
  average case, 跟 hash function有關

Average-case performance

- Assume **simple uniform hashing**: any given element is equally likely to hash into any of the m slots.
  對於每一個key，hash到任何一個slot的机会都相等
- For $j = 0, 1, \ldots, m-1$, denote the length of the list $T[j]$ by n_j, so that $n = n_0 + n_1 + \ldots + n_{m-1}$. n_j：第j個slot長度
- Average value of n_j is $E[n_j] = \alpha = n/m$. 平均長度 $\alpha = \frac{n}{m}$
- Assume that the hash value $h(k)$ can be computed in $O(1)$ time.
  - Time for the element with key k depends on the length $n_{h(k)}$ of the list $T[h(k)]$. k hash到h(k)這一個slot，長度為 $n_{h(k)}$
- We consider two cases:
  - contains no element with key k ➔ unsuccessful. key k 不在 table中
  - contain an element with key k ➔successful. key k 在 table中

Theorem 11.1 不成功的search：先計算，再search

- An **unsuccessful search** takes expected time $\Theta(1+\alpha)$.
- Proof: 不在table中，期望時間是$\Theta(1+\alpha)$
  - Under the assumption of simple uniform hashing, any key not already in the table is equally likely to hash to any of the *m* slots.
  - To search unsuccessfully for any key *k*, need to search to the end of the list $T[h(k)]$. 因為不在table中，所以將slot h(k)從頭找到尾以確定不在其中
  - This list has expected length $E[n_{h(k)}] = \alpha$.
  - Therefore, the expected number of elements examined in an unsuccessful search is α. slot h(k)的長度是α
  - Adding in the time to compute the hash function.
  - The total time required is $\Theta(1 + \alpha)$.

計算hash function的時間　slot h(k)的長度

Theorem 11.2

$\frac{n-i}{m}$個

□→□→□→□→ x →□→□

- An **successful search** takes expected time $\Theta(1+\alpha)$.

key值在 table 中

- Proof:
  - Assume the element being searched for is equally likely to be any of the *n* elements in the table *T*.

假設是放入的第 i 個　　　　i = 1~n

  - During a successful search for *x*, the # of elements examined = # of elements in the list before *x* + 1.

找的個數是 x 前面的個數加 1

  - The expected length of that list is $(n - i)/m$.

[後放的放前面]

假設 x 是第 i 個，在 x 之後有 n-i 個，所以在 x 之前的平均為 $\frac{n-i}{m}$

  - The expected # of elements examined in a successful search is

$$\frac{1}{n}\sum_{i=1}^{n}\left(\underbrace{1}_{x}+\underbrace{\frac{n-i}{m}}_{\text{在 x 之前的長度}}\right)=1+\frac{1}{nm}\sum_{i=1}^{n}(n-i)=1+\frac{1}{nm}\left(\frac{n(n-1)}{2}\right)=1+\frac{\alpha}{2}-\frac{\alpha}{2n}.$$

  - The total time is $\Theta(2 + \alpha/2 - \alpha/2n) = \Theta(1+\alpha)$.

Outline

- Direct-address tables
- Hash tables
- **Hash functions**
- Open addressing

What makes a good hash function?

- Ideally, the hash function satisfies the assumption of simple uniform hashing. 理想上 hash function 要滿足 simple uniform hashing的假設 [每一個 key hash 到任一個 slot 的机会都等]
- In practice, it's not possible.
  - We don't know in advance the probability distribution.
    不能先知道机率分佈
  - The keys may not be drawn independently.
    取鍵值不是獨立的 例如：电話号碼，喜愛"6 8" 甚過"4"
- Often use heuristics, based on the domain of the keys, to create a hash function that performs well.
  憑藉对特定領域的瞭解去設計 hash function

Interpreting keys as natural numbers

- Most hash functions assume that the universe of keys are natural numbers.
假設鍵值是自然數
- Thus, if the keys are not natural numbers, a way is found to interpret them as natural numbers.
如果不是就轉換它
- **Example:** Interpret a character string as an integer expressed in some radix notation. Suppose the string is CLRS.
將字串看成某一種進位後即為自然數
  - ASCII values: C = 67, L = 76, R = 82, S = 83.
  - There are 128 basic ASCII values. 看成128進位(字母轉換成ASCII)
  - So interpret CLRS as $(67 \cdot 128^3) + (76 \cdot 128^2) + (82 \cdot 128^1) + (83 \cdot 128^0) = 141{,}764{,}947$.

Division method 除法<優点：快 缺点：m不能亂取

- **Method:** $h(k) = k \bmod m$. table size
- **Example:** $m = 20$ and $k = 91$ ➔ $h(k) = 11$.
- **Advantage:** Fast, since requires just one division operation.
- **Disadvantage:** Have to avoid certain values of m:
  - Powers of 2 are bad. If $m = 2^p$ for integer p, then $h(k)$ is just the least significant p bits of k. 若 $m=2^P \Rightarrow$ 10110011 h(k)只和k的後面P個bit相關
    Ex: p=5，和k的後5個有關
  - If k is a character string interpreted in radix 2^p (as in CLRS example), then $m = 2^p - 1$ is bad: permuting characters in a string does not change its hash value. (Exercise 11.3-3).
    字串且以 2^P 進位且 $m=2^P-1 \Rightarrow$ 字串排列不会影響 h(k) 的值
- **Good choice:** Ex："CLRS"="CRLS" 用128進位 p=7，$m=2^7-1$ 則hash出來值相同
  - A prime not too close to an exact power of 2.
    選擇一個质數，且质數不会過於接近 2^P，是一個好的選擇

The multiplication method$_{1/4}$

乘法 缺點：較除法慢
優點：m的選擇不是太重要

- **Method:**
  - Choose constant A in the range $0 < A < 1$. 先選A(0<A<1)
  - Multiply key k by A.
  - Extract the fractional part of kA.
  - Multiply the fractional part by m.
  - Take the floor of the result.
- In short, the hash function is $h(k) = \lfloor m(kA \bmod 1) \rfloor$, where $kA \bmod 1 = kA - \lfloor kA \rfloor$ = fractional part of kA. 取kA小數部份
- **Advantage:** Value of m is not critical.
- **Disadvantage:** Slower than division method.

The multiplication method$_{2/4}$ 一個好的 implement 方法

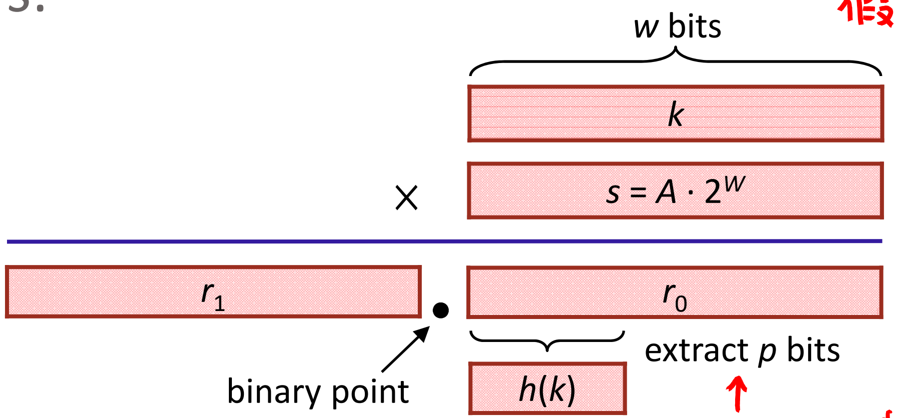
- **Easy implementation:**
  - Choose $m = 2^p$ for some integer p. table 大小為 2^p
  - Let the word size of the machine be w bits.
  - Assume that k fits into a single word. (k takes w bits.)
  - Let s be an integer in the range $0 < s < 2^w$.
  - Restrict A to be of the form $s/2^w$.
  - Multiply k by s.

假設机器有 w bits

$k \cdot A 2^w$

將 A 放大 2^w 倍

整數部份



小數部份

h(k)就是小數最後 P bits

The multiplication method$_{3/4}$

- The result is $2w$ bits, $r_1 2^w + r_0$, where r_1 is the high-order word of the product and r_0 is the low-order word.
- r_1 holds the integer part of kA ($\lfloor kA \rfloor$). r_0 holds the fractional part of kA ($k A \bmod 1 = kA - \lfloor kA \rfloor$).
- The p most significant bits of r_0 holds the value $\lfloor m(kA \bmod 1) \rfloor$.

- **Example:** $m = 8$ (implies $p = 3$), $w = 5$, $k = 21$. $A = \frac{13}{32}$，放大 2^5 倍 $= 13$

  Must have $0 < s < 2^5$; choose $s = 13$, so $A = 13/32$.

- **Formula:** $h(k)$: $kA = 21 \cdot 13/32 = 273/32 = 8\,\frac{17}{32}$

  ➔ $kA \bmod 1 = 17/32 \Rightarrow m(kA \bmod 1) = 8 \cdot 17/32 = 17/4 = 4\,\frac{1}{4}$

  ➔ $\lfloor m(kA \bmod 1) \rfloor = 4$, so that $h(k) = 4$.

The multiplication method$_{4/4}$

- **Easy implementation:** $ks = 21 \cdot 13 = 273 = \underline{8 \cdot 2^5} + \underline{17}$
  第一個word 第二個word

  ➔ $r_1 = 8$, $r_0 = 17$. Written in $w = 5$ bits, $r_0 = 10001$.
  17以5-bit表示(2進位)=10001

  Take the $p = 3$ most significant bits of r_0, get 100 in binary,
  取r的前p個bits ⇒ 10001 (P=3) = $100_{(2)} = 4_{(10)}$
  or 4 in decimal, so that $h(k) = 4$.

How to choose A:

- The multiplication method works with any legal value of A.
  A只要在0~1之間就可以，但某些值比較好
- But it works better with some values than with others, depending on the keys being hashed.
- Knuth suggests using $A \approx (\sqrt{5} - 1)/2$.
  大師說A取$\frac{\sqrt{5}-1}{2}$較好

Outline

- Direct-address tables
- Hash tables
- Hash functions
- **Open addressing**

Open addressing 一種處理 collision 的方法

- An alternative to chaining for handling collisions.
- **Idea:**
  - Store all elements in the hash table itself. 將所有東西都放在 table
  - When searching, we examine table slots until the desired element is found or it is clear that the element is not in the table. search 時，不是找到，就是確定不在 table 中
  - We **compute** the sequence of slots to be examined.
    要計算出 search 的順序
- **Advantage:**
  - Avoid pointers.
    在相同的 memory 下，有較多的 slot 可用
  - Has a larger number of slots for the same amount of memory.
- **Disadvantage:**
  - Deletion is difficult, thus chaining is more common if keys must be deleted. 刪除的時候很麻煩

Insertion & Searching

目標：要使整個 table 都 search 到

- To perform insertion, we successively examine, or **probe**, the hash table until we find an empty slot. 探測

在 insert 時，一個一個看，直到有空的 slot

- The sequence of positions probed **depends upon the key being inserted**. 鍵值不同導致探測的 sequence 不同
- The hash function is $h : U \times \{0,1,\ldots, m-1\} \rightarrow \{0,1,\ldots, m-1\}$.

key 第n次探測

- The **probe sequence** is $h(k,0), h(k, 1),\ldots, h(k, m-1)$.

探測序列 第0次 第 m-1 次

HASH-INSERT(T, k)

```
i ← 0                      第0次探測
repeat j ← h(k, i)
    if T[j] = NIL
        T[j] ← k           若為 empty 則放入
        return j
    else i ← i + 1         找下一個
until i = m
error "hash table overflow"
```

HASH-SEARCH(T, k)

```
i ← 0
repeat j ← h(k,i)          i = search 次數
    if T[j] = k
        return j
    i ← i + 1
until T[j] = NIL or i = m
return NIL                 slot 為空 = 沒有下一個
```

Deletion

- When we delete a key from slot *i*, we can't simply mark that slot as empty by storing NIL in it.

刪除時，不能直接放NULL ∵search至NULL即停止，会使判斷錯誤

- **Solution:** Use a special value DELETED instead of NIL when marking a slot as empty during deletion. 用一個特別的值"DELETED"
  - Search should treat **DELETED** as though the slot holds a key that does not match the one being searched for.

    search遇到"DELETED"⇒繼續找
  - Insertion should treat **DELETED** as though the slot were empty, so that it can be reused.

    Insertion遇到DELETED⇒可以放東西

Three probing methods

- The ideal situation is **uniform hashing**: each key is equally likely to have any of the m! permutations of <0, 1, . . . , $m-1$> as its probe sequence. m個slot的排列机率有m!種
  理想：每一個key对应到m!種排列的任何一個机率相同
- Three commonly used probing methods:
  - Linear probing
  - Quadratic probing
  - Double hashing

  3個常用的probing方法
- None of these techniques fulfills the assumption of uniform hashing. 此3種皆不符合uniform hashing的假設

Linear probing 線性的探測

- Given an ordinary hash function $h' : U \rightarrow \{0, 1, \ldots, m-1\}$, which we refer to as an **auxiliary hash function**, the method of **linear probing** uses the hash function

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

附加的 hashing function

for $i = 0, 1, \ldots, m - 1$.

- Because the initial probe determines the entire probe sequence, there are only m distinct probe sequences.

h'(k) 決定一切，所以共有 m 種 sequence

- Linear probing suffers from **primary clustering**: long runs of occupied sequences build up. 聚在一起（群聚）

最後會形成群聚的效應

Quadratic probing 二次探測

- **Quadratic probing** uses a hash function of the form

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

where h' is an auxiliary hash function, c_1 and $c_2 \neq 0$ are auxiliary constants, and $i = 0, 1, \ldots, m-1$.

比 linear probing 好，但

- This method works much better than linear probing, but to make full use of the hash table, the values of c_1, c_2, and m are constrained. (Problem 11-3) 如果要使整個 table 都 search 到，c_1，c_2 和 m 的選擇会受到限制
- If two keys have the same initial probe position, then their probe sequences are the same. This property leads **secondary clustering.** $h'(k)$ 还是决定一切：$h'(k)$ 相同，之後順序也会相同
- Because the initial probe determines the entire probe sequence, there are only m distinct probe sequences.

只有 m 種 sequence（∵ $h'(k)$ 还是决定一切）

Double Hashing 用2個 hashing function

- **Double hashing** uses a hash function of the form

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

where h_1 and h_2 are auxiliary hash functions.

- The value $h_2(k)$ must be relatively prime to the hash-table size m for the entire hash table to be searched.
  如果要使整個 table 都 search 到，m 和 $h_2(k)$ 要互質
  - Let m be a power of 2 and to design h_2 so that it always produce an odd number >1. $m=2^k$, $h_2(k)$ 產生大於 1 的 odd number 可以滿足規定
  - Let m be prime and have $h_2(k) < m$.
    m 是質數，$h_2(k)$ 小於 m 也可以
- $\Theta(m^2)$ different probe sequences, since each possible combination of $h_1(k)$ and $h_2(k)$ gives a different probe sequence. 產生 m^2 種 sequence → $h_1(k) = m$ 種 $h_2(k) = m$ 種

An example for double Hashing

- Hash table size: 13, key = 14.

m=13

- $h_1(k) = k \bmod 13$; $h_2(k) = 1 + (k \bmod 11)$

i=0, 1 mod 13=1
i=1, 5 mod 13=5
有,看下一個i
i=2, 9 mod 13=9
空,放入14

- $h(14, i) = (h_1(k) + ih_2(k)) \bmod m$
  $= (1 + i(1 + 3)) \bmod 13$
  $= (1 + 4i) \bmod 13.$

- So, the key 14 is inserted into empty slot 9.

插入14

	79			69	98		72		14		50	
0	1	2	3	4	5	6	7	8	9	10	11	12