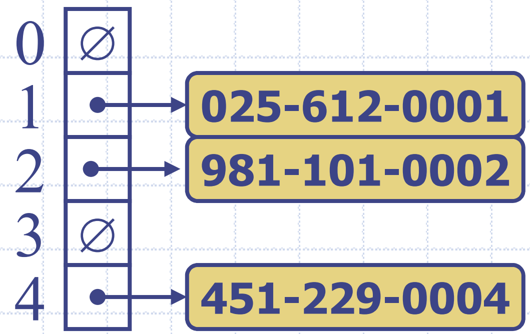
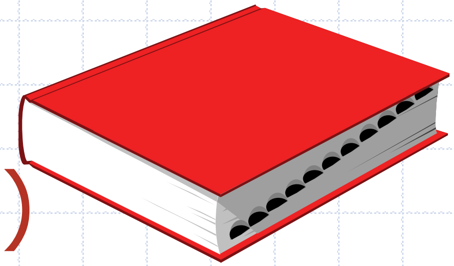
Dictionaries and Hash Tables



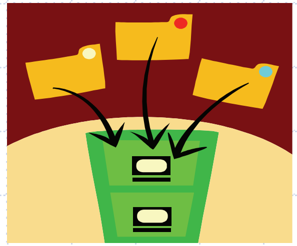
Dictionary ADT (Abstract Data Type)



- The dictionary ADT models a searchable collection of key-element items
- The main operations of a dictionary are searching, inserting, and deleting items
- Multiple items with the same key are allowed (collision)
- Applications:
  - address book
  - credit card authorization
  - mapping host names (e.g., cs16.net) to internet addresses (e.g., 128.148.34.101) DNS sever

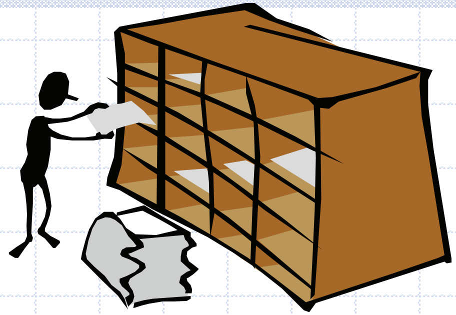
- Dictionary ADT methods:
  - findElement(k): if the dictionary has an item with key k, returns its element, else, returns the special element NO_SUCH_KEY
  - insertItem(k, o): inserts item (k, o) into the dictionary
  - removeElement(k): if the dictionary has an item with key k, removes it from the dictionary and returns its element, else returns the special element NO_SUCH_KEY
  - size(), isEmpty()
  - keys(), Elements()

Log File



- A log file is a dictionary implemented by means of an unsorted sequence
  - We store the items of the dictionary in a sequence (based on a doubly-linked lists or a circular array), in arbitrary order
- Performance:
  - insertItem takes $O(1)$ time since we can insert the new item at the beginning or at the end of the sequence
  - findElement and removeElement take $O(n)$ time since in the worst case (the item is not found) we traverse the entire sequence to look for an item with the given key, average case $O(n)$ time
- The log file is effective only for dictionaries of small size or for dictionaries on which insertions are the most common operations, while searches and removals are rarely performed (e.g., historical record of logins to a workstation)

Hash Functions and Hash Tables



- A hash function h maps keys of a given type to integers in a fixed interval $[0, N-1]$
- Example: (division method)

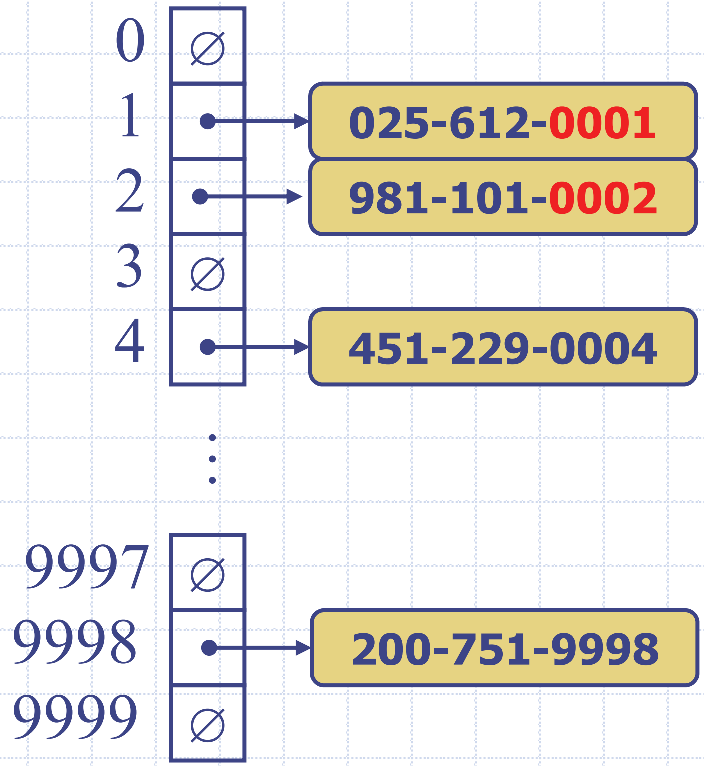
  $$h(x) = x \bmod N$$

  is a hash function for integer keys
- The integer $h(x)$ is called the hash value of key x

- A hash table for a given key type consists of
  - Hash function h
  - Array (called table) of size N
- When implementing a dictionary with a hash table, the goal is to store item (k, o) at index $i = h(k)$

Example

- We design a hash table for a dictionary storing items (SSN, Name), where SSN (social security number) is a nine-digit positive integer
- Our hash table uses an array of size $N = 10{,}000$ and the hash function
  $h(x) =$ last four digits of x



中間平方法（Mid-Square Method）

- 中間平方法是將鍵值乘以自己或某個常數，然後取中間幾位數字做為索引位置。
- 例如：只取中間三個數字來作為雜湊表的索引位置，如下所示：

$(136)^2 = 2\underline{120}4 \rightarrow 120$

$(2642)^2 = 69\underline{801}64 \rightarrow 801$

折疊法（Folding Method）

- 折疊法是將鍵值分成幾個部分，除了最後一個部分外，其餘部分都是相同長度。例如：將一個長整數14237240120933分成5個部分，如下所示：

P1	P2	P3	P4	P5
142	372	401	209	33

邊界折疊法（Folding at the Boundaries）

- 在分成幾個部分後，將左邊邊界折起來後相加，所以P2和P4是反轉資料，如下所示：

```
    1 4 2
    2 7 3   P2反轉
    4 0 1
    9 0 2   P4反轉
+)    3 3
---------
  1 7 5 1
```

在左邊折疊

位移折疊法（Shift Folding）

- 在分成幾個部分後，直接靠左或靠右相加後，就是索引位置，如下所示：

```
    1 4 2        1 4 2
    3 7 2        3 7 2
    4 0 1        4 0 1
    2 0 9        2 0 9
+)    3 3    +)  3 3
---------    ---------
  1 1 5 7      1 4 5 4
   靠右          靠左
```

Hash Functions

- A hash function is usually specified as the composition of two functions:

  Hash code map:

  h_1: keys $\rightarrow$ integers

  Compression map:

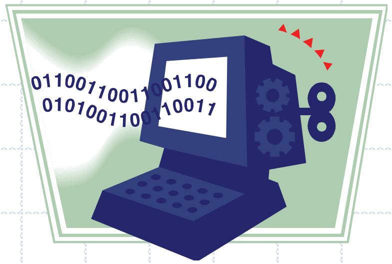
  h_2: integers $\rightarrow [0, N-1]$

- The hash code map is applied first, and the compression map is applied next on the result, i.e.,

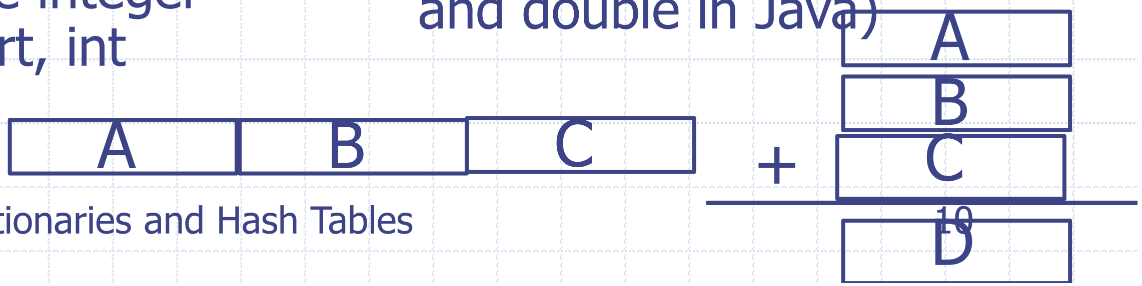
$$h(x) = h_2(h_1(x))$$

- The goal of the hash function is to "disperse" (打散) the keys in an apparently random way

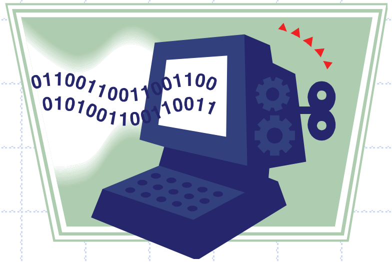
Hash Code Maps cases:



- Memory address:
  - We reinterpret the memory address of the key object as an integer (default hash code of all Java objects)
  - Good in general, except for numeric and string keys
- Integer cast:
  - We reinterpret the bits of the key as an integer
  - Suitable for keys of length less than or equal to the number of bits of the integer type (e.g., byte, short, int and float in Java)
- Component sum:
  - We partition the bits of the key into components of fixed length (e.g., 16 or 32 bits) and we sum the components (ignoring overflows)
  - Suitable for numeric keys of fixed length greater than or equal to the number of bits of the integer type (e.g., long and double in Java)



Hash Code Maps (cont.)



- Polynomial accumulation:
  - We partition the bits of the key into a sequence of components of fixed length (e.g., 8, 16 or 32 bits)

    $$a_0\, a_1 \ldots a_{n-1}$$
  - We evaluate the polynomial

    $$p(z) = a_0 + a_1 z + a_2 z^2 + \ldots \\ \ldots + a_{n-1} z^{n-1}$$

    at a fixed value z, ignoring overflows
  - Especially suitable for strings (e.g., the choice $z = 33$ gives at most 6 collisions on a set of 50,000 English words)
- Polynomial $p(z)$ can be evaluated in $O(n)$ time using Horner's rule:
  - The following polynomials are successively computed, each from the previous one in $O(1)$ time

    $$p_0(z) = a_{n-1}$$

    $$p_i(z) = a_{n-i-1} + z p_{i-1}(z) \\ (i = 1, 2, \ldots, n-1)$$
- We have $p(z) = p_{n-1}(z)$

霍內演算法（Horner's algorithm）

按降冪順序，一次做兩項，提出共同的低次項，則每次計算只有一個乘法和一個加法。

例如，對於二次多項式函數：

$$f(x) = a_2x^2 + a_1x + a_0 = (a_2x + a_1)x + a_0$$

對於三次多項式函數：

$$\begin{aligned} f(x) &= a_3x^3 + a_2x^2 + a_1x + a_0 \\ &= (a_3x + a_2)x^2 + a_1x + a_0 \\ &= ((a_3x + a_2)x + a_1)x + a_0 \end{aligned}$$

對於四次多項式函數：

$$\begin{aligned} f(x) &= a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0 \\ &= (a_4x + a_3)x^3 + a_2x^2 + a_1x + a_0 \\ &= ((a_4x + a_3)x + a_2)x^2 + a_1x + a_0 \\ &= (((a_4x + a_3)x + a_2)x + a_1)x + a_0 \end{aligned}$$

範例

再以 $f(x) = x^4 - 4x^3 + 2x^2 - x + 1$ 的 $f(\frac{1}{2})$ 計算為例，

此時 $n = 4, a_4 = 1, a_3 = -4, a_2 = 2, a_1 = -1$ 和$a_0 = 1$。

1, -4, 3, -1, 1

一開始，令 $p = a_4 = 1$。然後依序執行以下計算：

$$i = 3, p = 1 \times \frac{1}{2} + a_3 = -\frac{7}{2}$$

$$i = 2, p = -\frac{7}{2} \times \frac{1}{2} + a_2 = \frac{1}{4}$$

$$i = 1, p = \frac{1}{4} \times \frac{1}{2} + a_1 = -\frac{7}{8}$$

$$i = 0, p = -\frac{7}{8} \times \frac{1}{2} + a_0 = \frac{9}{16}$$

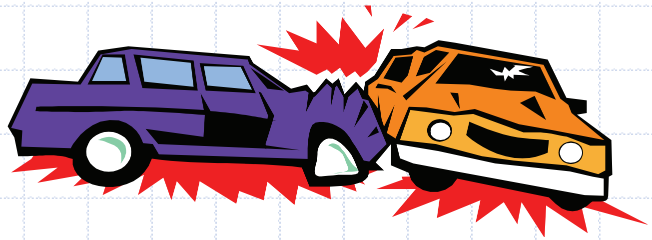
最後的 $p = \frac{9}{16}$ 就是 $f(\frac{1}{2})$ 的值。

Compression Maps

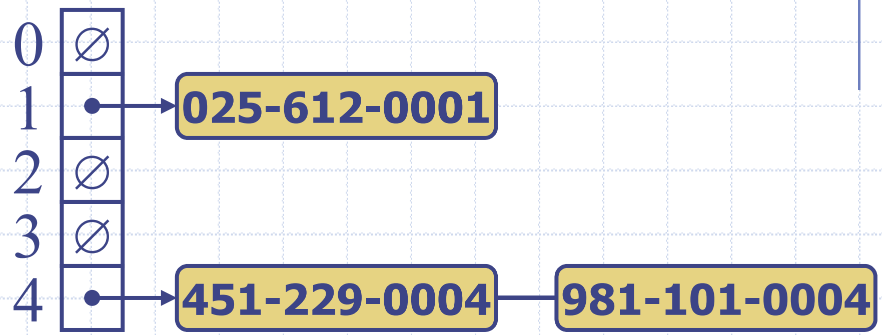
- Division:
  - $h_2(y) = y \bmod N$
  - The size N of the hash table is usually chosen to be a prime
  - The reason has to do with number theory and is beyond the scope of this course

- Multiply, Add and Divide (MAD):
  - $h_2(y) = (ay + b) \bmod N$
  - a and b are nonnegative integers such that
    $a \bmod N \neq 0$
  - Otherwise, every integer would map to the same value b

Collision Handling

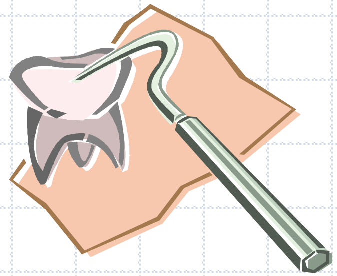


- Collisions occur when different elements are mapped to the same cell
- **Chaining**: let each cell in the table point to a linked list of elements that map there



- Chaining is simple, but requires additional memory outside the table

Collision Handling Linear Probing



- Open addressing: the colliding item is placed in a different cell of the table
- **Linear probing** handles collisions by placing the colliding item in the next (circularly) available table cell
- Each table cell inspected is referred to as a "probe"
- Colliding items lump together, causing future collisions to cause a longer sequence of probes

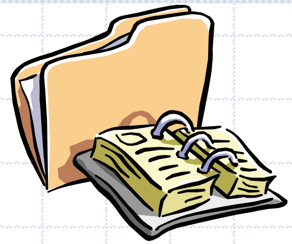
- Example:
  - $h(x) = x \bmod 13$
  - Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order

0	1	2	3	4	5	6	7	8	9	10	11	12

⇩

0	1	2	3	4	5	6	7	8	9	10	11	12
		41			18	44	59	32	22	31	73	

Search with Linear Probing



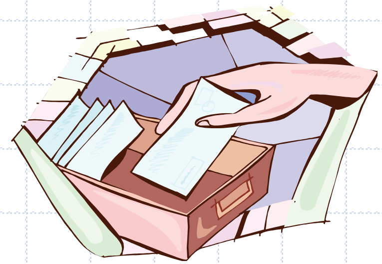
- Consider a hash table A that uses linear probing
- findElement(k)
  - We start at cell $h(k)$
  - We probe consecutive locations until one of the following occurs
    - An item with key k is found, or
    - An empty cell is found, or
    - N cells have been unsuccessfully probed

```
Algorithm findElement(k)
    i ← h(k)
    p ← 0
    repeat
        c ← A[i]
        if c = ∅
            return NO_SUCH_KEY
        else if c.key () = k
            return c.element()
        else
            i ← (i + 1) mod N
            p ← p + 1
    until  p = N
    return NO_SUCH_KEY
```

Updates with Linear Probing

- To handle insertions and deletions, we introduce a special object, called ***AVAILABLE***, which replaces deleted elements
- removeElement(k)
  - We search for an item with key k
  - If such an item (k, o) is found, we replace it with the special item ***AVAILABLE*** and we return element o
  - Else, we return ***NO_SUCH_KEY***
- insert Item(k, o)
  - We throw an exception if the table is full
  - We start at cell $h(k)$
  - We probe consecutive cells until one of the following occurs
    - A cell i is found that is either empty or stores ***AVAILABLE***, or
    - N cells have been unsuccessfully probed
  - We store item (k, o) in cell i

Collision Handling Double Hashing



- Double hashing uses a secondary hash function $d(k)$ and handles collisions by placing an item in the first available cell of the series
  $$(i + jd(k)) \bmod N$$
  for $j = 0, 1, \ldots, N-1$
- The secondary hash function $d(k)$ cannot have zero values
- The table size N must be a prime to allow probing of all the cells
- Common choice of compression map for the secondary hash function:
  $$d_2(k) = q - k \bmod q$$
  where
  - $q < N$
  - q is a prime
- The possible values for $d_2(k)$ are
  $$1, 2, \ldots, q$$

Example of Double Hashing

- Consider a hash table storing integer keys that handles collision with double hashing
  - $N = 13$
  - $h(k) = k \bmod 13$
  - $d(k) = 7 - k \bmod 7$
- Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order

$(i + jd(k)) \bmod N$
for $j = 0, 1, \ldots, N-1$

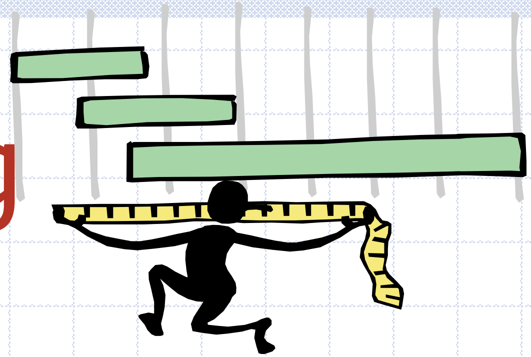
k	$h(k)$	$d(k)$	Probes		
18	5	3	5		
41	2	1	2		
22	9	6	9		
44	5	5	5	10	
59	7	4	7		
32	6	3	6		
31	5	4	5	9	0
73	8	4	8		

0	1	2	3	4	5	6	7	8	9	10	11	12

⇓

0	1	2	3	4	5	6	7	8	9	10	11	12
31		41			18	32	59	73	22	44		

Performance of Hashing

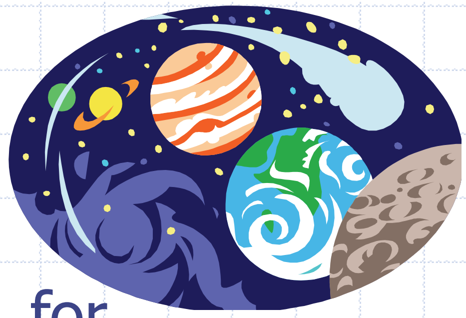


- In the worst case, searches, insertions and removals on a hash table take $O(n)$ time
- The worst case occurs when all the keys inserted into the dictionary collide
- The load factor $\alpha = n/N$ affects the performance of a hash table
- Assuming that the hash values are like random numbers, it can be shown that the expected number of probes for an insertion with open addressing is

  $$1 / (1 - \alpha)$$

- The expected running time of all the dictionary ADT operations in a hash table is $O(1)$
- In practice, hashing is very fast provided the load factor is not close to 100%
- Applications of hash tables:
  - small databases
  - compilers
  - browser caches

Universal Hashing



- A family of hash functions is **universal** if, for any $0 \le i,j \le M-1$, $\Pr(h(j)=h(k)) \le 1/N$.
- Choose p as a prime between M and 2M.
- Randomly select $0<a<p$ and $0 \le b<p$, and define

$$h(k)=(ak+b \bmod p) \bmod N$$

- Theorem: The set of all functions, h, as defined here, is universal.

全域雜湊法（Universal Hashing）

- 在向雜湊表中插入元素時，如果所有的元素全部被雜湊到同一個桶中，此時數據的存儲實際上就是一個鏈表，那麼平均的查找時間為 $\Theta(n)$ 。而實際上，任何一個特定的雜湊函數都有可能出現這種最壞情況，
- 唯一有效的改進方法就是隨機地選擇雜湊函數，使之獨立於要存儲的元素。這種方法稱作全域雜湊（Universal Hashing）。
- 全域雜湊的基本思想是在執行開始時，從一組雜湊希函數中，隨機地抽取一個作為要使用的雜湊函數。
  - 隨機化保證了沒有哪一種輸入會始終導致最壞情況的發生。
  - 同時，隨機化也使得即使是對同一個輸入，算法在每一次執行時的情況也都不一樣。這樣就確保了對於任何輸入，算法都具有較好的平均運行情況。

$$hash_{a,b}(key) = ((a*key + b) \bmod p) \bmod N$$

- 其中，p 為一個足夠大的質數，使得每一個可能的關鍵字 key 都落在 0 到 p - 1 的範圍內。N 為雜湊表中槽位數。
- for $a \in \{1,2,3,\ldots,p-1\}$，$b \in \{0,1,2,\ldots,p-1\}$。

完美雜湊（Perfect Hashing）

- 當關鍵字的集合是一個不變的靜態集合（Static）時，雜湊技術還可以用來獲取出色的最壞情況性能。
- 如果某一種雜湊技術在進行查找時，其最壞情況的內存訪問次數為 O(1) 時，則稱其為完美雜湊（Perfect Hashing）。
- 設計完美雜湊的基本思想是利用兩級的雜湊策略，而每一級上都使用全域雜湊（Universal Hashing）。