Hash Tables

Algorithms & Data Structures
ITCS 6114/8114

Dr. Dewan Tanvir Ahmed
Department of Computer Science
University of North Carolina at Charlotte

Hash Functions and Hash Tables (§2.5.2)

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

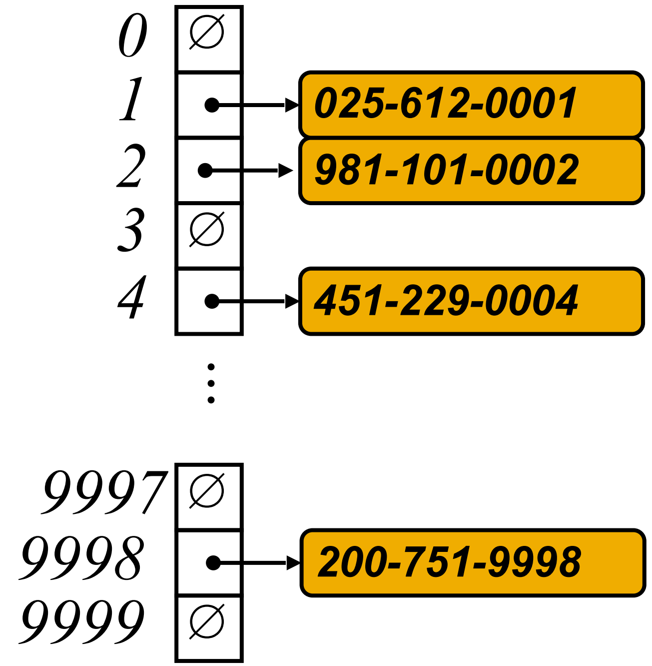
  $$h(x) = x \bmod N$$

  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



- A hash function is good if it maps the keys in dictionary so as to minimize collision as much as possible.
- For practical reason:
  - The evaluation of a hash function to be fast and easy to compute

Hash Functions (§ 2.5.3)

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

  Hash code map:

  $$h_1\text{: keys} \rightarrow \text{integers}$$

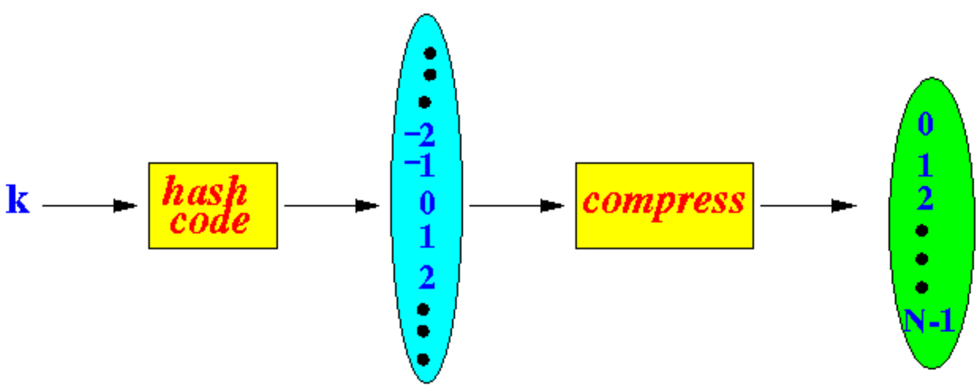
  Compression map:

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

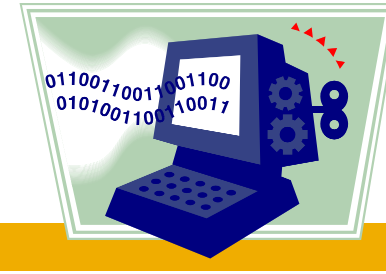
- The hash code map is applied first, and the compression map is applied next on the result, i.e.,

  h(x) = h2(h1(x))

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

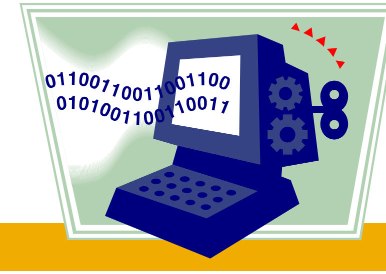


Hash Code Maps (§2.5.3)



- Integer cast:
  - 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, and int)
- Component sum:
  - 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

Hash Code Maps (§2.5.3)



- Problem with Component sum (summation hash codes):
  - Not good for character strings
  - Why?
    - Produces lots of unwanted collisions for common groups of strings
    - Example:
      - “stop”, “tops”, “pots”, “post” and “spot”

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

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

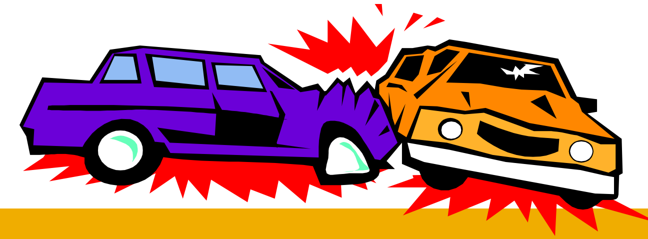
    $p_i(z) = a_{n-i-1} + zp_{i-1}(z)$

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

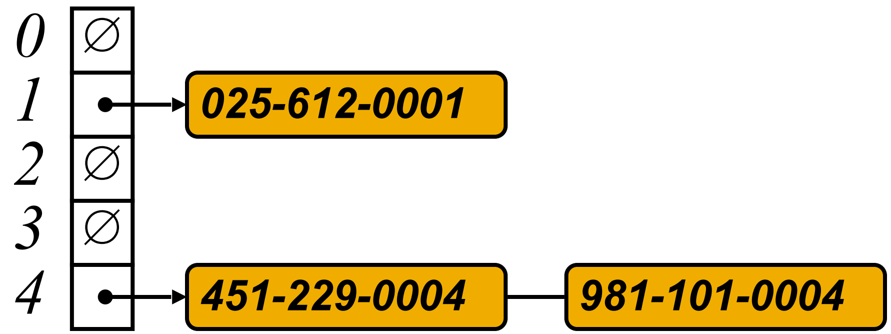
Compression Maps (§2.5.4)

- Division:
  - $h_2(y) = y \bmod N$
  - The size N of the hash table is usually chosen to be a prime
- Multiply, Add and Divide (MAD):
  - $h_2(y) = (ay + b) \bmod N$
  - a and b are nonnegative integers such that
    $a \bmod N \neq 0$
  - WHY?
  - Otherwise, every integer would map to the same value b

Collision Handling (§ 2.5.5)



- 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

Linear Probing (§2.5.5)

- *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

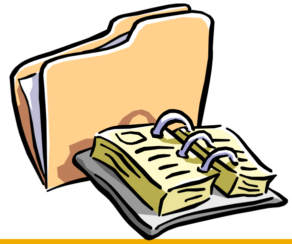
k	$h(k)$
18	5
41	2
22	9
44	5
59	7
32	6
31	5
73	8

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
		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

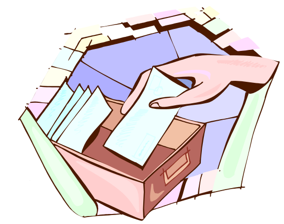
```
Algorithm findElement(k)
  i ← h(k)
  p ← 0
  repeat
    c ← A[i]
    if c = ∅
      return NO_SUCH_KEY
     else if c.key () = k
      return c.element()
    else
      i ← (i + 1) mod N
      p ← p + 1
  until  p = N
  return NO_SUCH_KEY
```

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

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

  $$(i + f(j)) \bmod N$$

  for $j = 1, 2, \ldots,$

  where $f(j) = j . d(k)$
- 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:

  $$d(k) = q - (k \bmod q)$$

  where
  - $q < N$
  - q is a prime
- The possible values for $d(k)$ 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

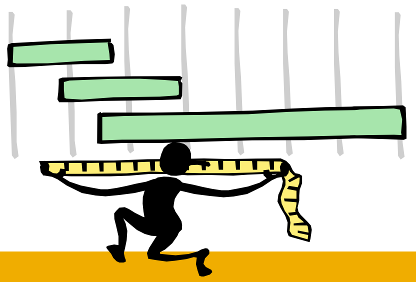
k	$h(k)$	$d(k)$	Probes		
18	5	3	5		
41	2	1	2		
22	9	6	9		
44	5	5	5	10	
59	7	4	7		
32	6	3	6		
31	5	4	5	9	0
73	8	4	8		

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

⇩

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

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 (§ 2.5.6)

- Set of keys are integers in range [0, M-1].
- Hash function h maps integers from the range [0, M-1] to integers [0, N-1]
- A family of hash functions is **universal** if, for any two integers j and k in the range [0, M-1],

$$0 \leq j, k \leq M-1,$$
$$\Pr(h(j) = h(k)) \leq 1/N.$$

- Choose p as a prime between M and 2M.
- Randomly select $0 < a < p$ and $0 \leq b < p$, and define $h(k) = ((ak + b)\ mod\ p)\ mod\ N$

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

Reference

- Algorithm Design: Foundations, Analysis, and Internet Examples. Michael T. Goodrich and Roberto Tamassia. John Wiley & Sons.
- Introduction to Algorithms. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein.