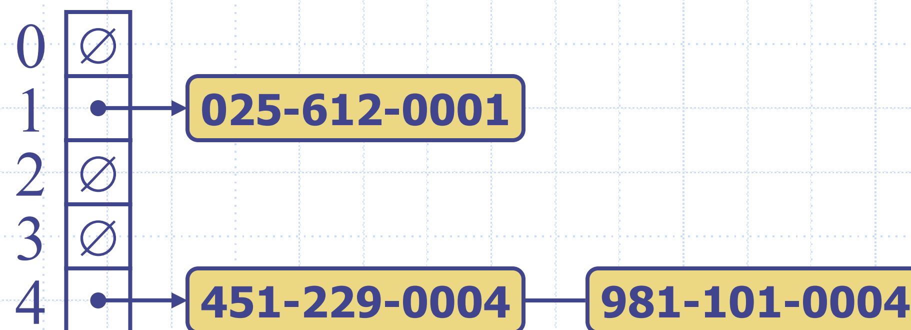
Hash Tables



Outline and Reading

- Map ADT (§9.1)
- Hash functions and hash tables (§9.2.2)
- Hash function details
  - Hash code map (§9.2.3)
  - Compression map (§9.2.4)
- Collision handling (§9.2.5)
  - Chaining
  - Linear probing
  - Double hashing

Map ADT

- The map ADT models the association between keys and their elements in a set of key-element items
- The main operations of a map are searching, inserting, and deleting
- Multiple items with the same key are not allowed
- Applications:
  - address book
  - credit card authorization
  - mapping host names (e.g., cs16.net) to internet addresses (e.g., 128.148.34.101)

- Map ADT methods:
  - get(k): if the map has an item with key k, returns its element, else, returns ***null***
  - put(k, o): if the map has an item with key k, replaces its element with o and returns the old element, else inserts item (k, o) into the map and returns ***null***
  - remove(k): if the map has an item with key k, removes it from the map and returns its element, else returns ***null***
  - size(), isEmpty()
  - keys(), values(), entries()

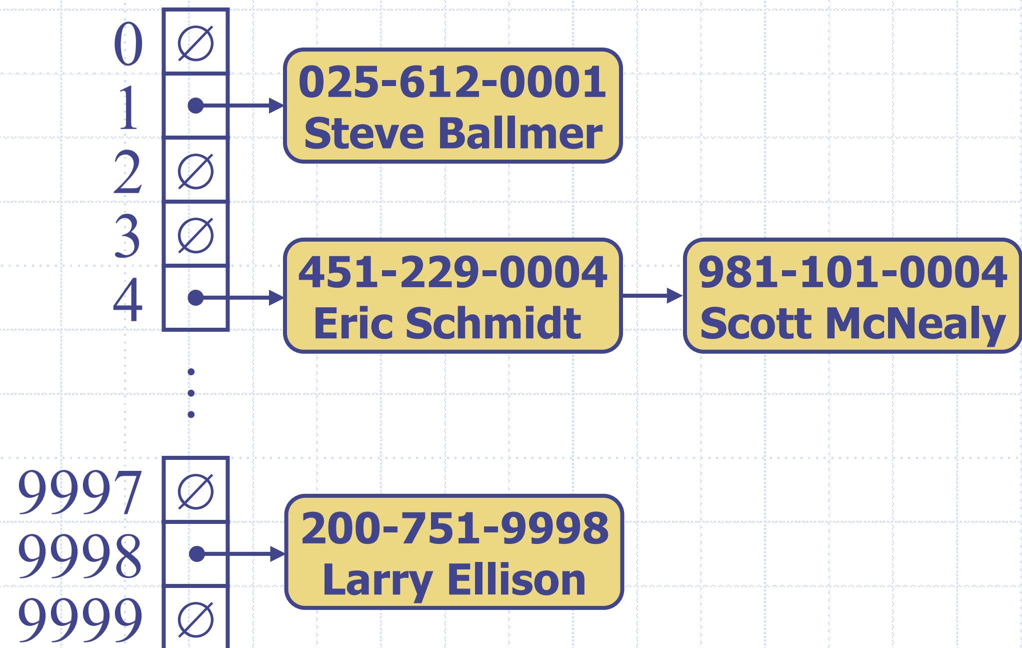
Hash Functions and Hash Tables

- A hash function h maps keys of a given type to integers in a fixed interval $[0, N-1]$
- Example:
  $$h(x) = x \bmod N$$
  is a hash function for integer keys
- The integer $h(x)$ is called the hash value of key x
- The goal of a hash function is to uniformly disperse keys in the range $[0, N-1]$

- A hash table for a given key type consists of
  - Hash function h
  - Array (called table) of size N
- When implementing a map with a hash table, the goal is to store item (k, o) at index $i = h(k)$
- A collision occurs when two keys in the map have the same hash value
- Collision handing schemes:
  - Chaining: colliding items are stored in a sequence
  - Open addressing: colliding items are placed in different cells of the table

Example

- We design a hash table for a map 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 $h(x) = \text{last four digits of } x$
- We use chaining to handle collisions



Hash Functions

- The goal of a hash function is to "disperse" the keys in an apparently random way
- A hash function is usually specified as the composition of two functions:

  Hash code function:

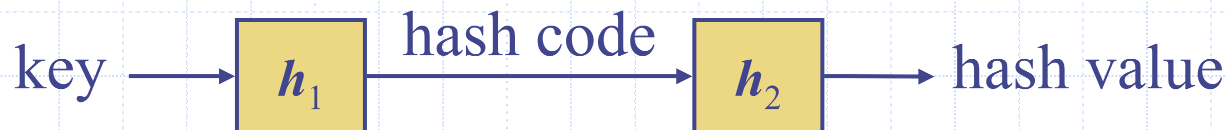
  h_1: {keys} $\rightarrow$ {integers}

  Compression function:

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

- Integer $h_1(x)$ is called the hash code of the key
- The hash code function is applied first, and the compression function is applied next on the result, i.e.,

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



Hash Code Functions

- **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 Functions (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

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

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

- We have $p(z) = p_{n-1}(z)$

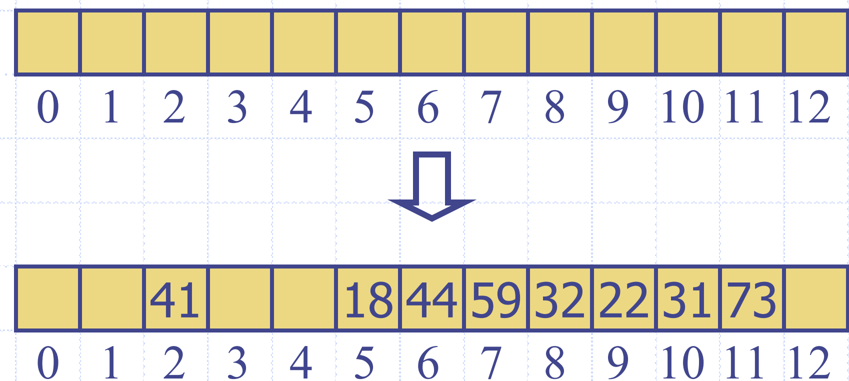
Compression Functions

- 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

Linear Probing

- 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



Search with Linear Probing

- Consider a hash table A that uses linear probing
- get(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 *get*(k)
    $i \leftarrow h(k)$
    $p \leftarrow 0$
    repeat
        $c \leftarrow A[i]$
        if $c = null$
            return *null*
        else if $c.key() = k$
            return *c.element*()
        else
            $i \leftarrow (i + 1) \bmod N$
            $p \leftarrow p + 1$
    until $p = N$
    return *null*

Updates with Linear Probing

- To handle both insertions and deletions, we introduce a special object, called ***AVAILABLE***, which replaces deleted elements
- remove(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 ***null***
- put(k, o)
  - Variation of method get
  - While probing, we remember the index i of the first cell encountered that is ***AVAILABLE*** or ***null***
  - If we find an item with key k, we replace it element with o
  - Else, we store a new item (k, o) in cell i

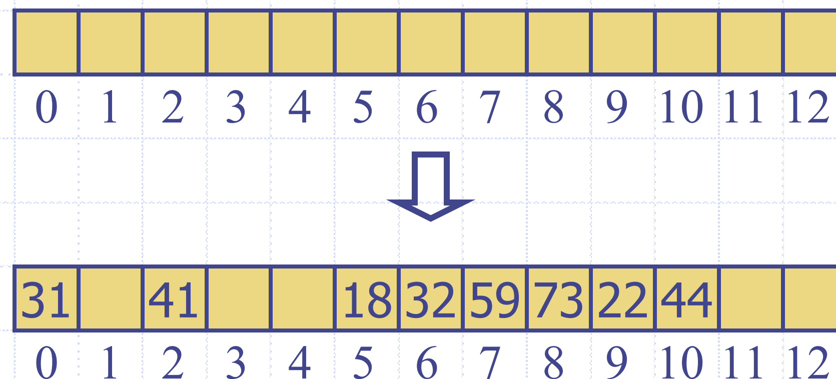
Double Hashing

- Double hashing uses a secondary hash function $d(k)$ and handles collisions by placing an item in the first available cell of the series
  $$(i + jd(k)) \bmod N$$
  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:
  $$d(k) = q - k \bmod q$$
  where
  - $q < N$
  - q is a prime
- The possible values for $d(k)$ are
  $$1, 2, \ldots, q$$

Example of Double Hashing

- How searching works:
  - First try $h(k)$
  - If collision, try $h(k) + d(k)$
  - Continue adding $d(k)$ until key k is found or empty slot is reached
- Using
  - $N = 13$
  - $h(k) = k \bmod 13$
  - $d(k) = 7 - k \bmod 7$,
- 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		



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

Count Word Frequencies

- Application of hash tables to the problem of counting the frequencies of the words in a document
- Create a map for (word, frequency) pairs
- $O(n)$ expected running time for a document with n words

```
Algorith wordCount(D)
Input: document D
Output: map of the words in D
        and their frequencies
   L ← wordList(D)
   M ← new HashTableMap()
   for each w in L
      c ← M.get(w)
      if c = null
         M.put(w, 1)
      else
         M.put(w, c + 1)
   return M
```