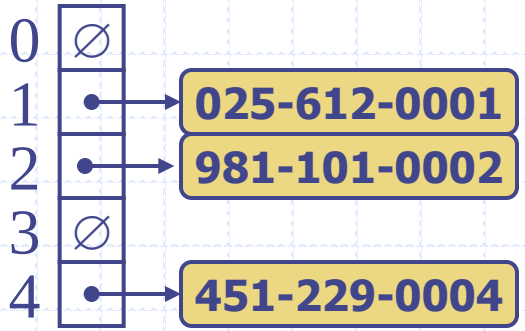
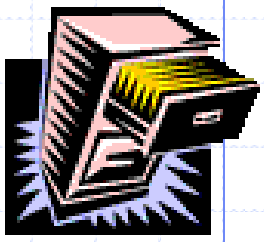
Presentation for use with the textbook Data Structures and Algorithms in Java, 6th edition, by M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, Wiley, 2014

Hash Tables

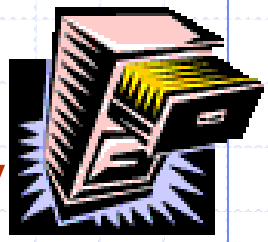


Recall the Dictionary or Map ADT

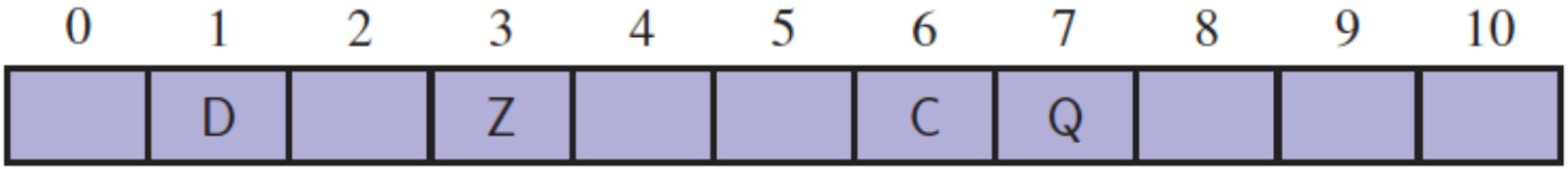


- get(k): if the dictionary M has an entry with key k, return its associated value; else, return null
- put(k, v): insert entry (k, v) into M; if key k is not already in M; else ERROR
- remove(k): if M has an entry with key k, remove it from M else ERROR
- size(), isEmpty()
- entrySet(): return an iterable collection of the entries in M
- keySet(): return an iterable collection of the keys in M
- values(): return an iterator of the values in M

Intuitive Notion of a Dictionary

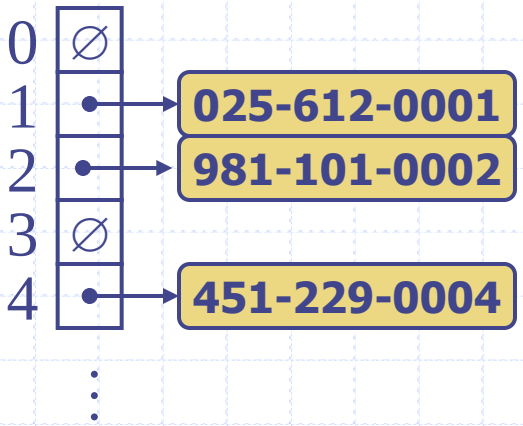


- A dictionary T supports the abstraction of using keys as indices with a syntax such as T[k].
- As a mental warm-up, consider a restricted case where a dictionary with n items uses keys that are known to be integers in a range from 0 to N − 1, for some N ≥ n.

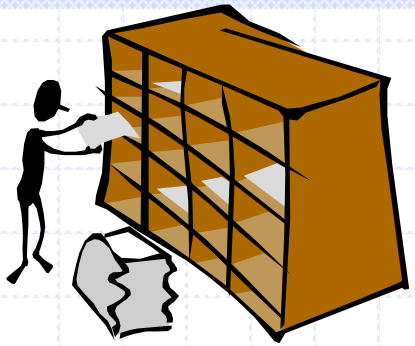


More General Kinds of Keys

- But what should we do if our keys are not integers in the range from 0 to N – 1?
  - Use a **hash function** to map general keys to corresponding indices in a table.
  - For instance, the last four digits of a Social Security number.



Hash Functions and Hash Tables



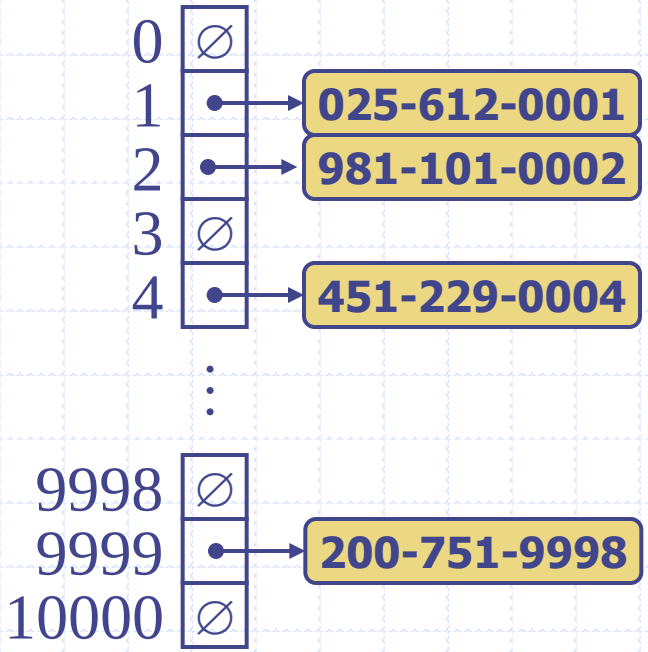
- A hash function h maps keys of a given type to integers in a fixed interval $[0, M-1]$
- Example:

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

  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 M
- When implementing a map 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 entries as (SSN, Name), where SSN (social security number) is a nine-digit positive integer
- Our hash table uses an array of size $M = 10{,}001$ and the hash function $h(x) = \text{last four digits of } x$



Hash Functions

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

  Hash code:

  h_1: keys $\rightarrow$ integers

  Compression map:

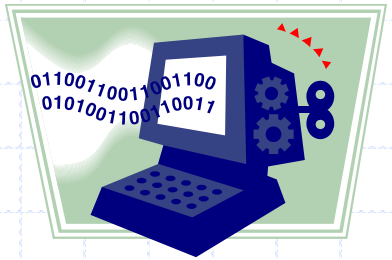
  h_2: integers $\rightarrow [0, M-1]$

- The hash code 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" the keys in an apparently random way

Hash Codes



- 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)
  - Not very good as it might produce only small values

Hash Codes

- Polynomial hash function:
  - We partition the bits of the key into a sequence of components of fixed length (e.g., 8, 16 or 32 bits)
    $$a_0 a_1 \ldots a_{k-1}$$
  - We evaluate the polynomial
    $$p(x) = a_0 + a_1 x + a_2 x^2 + \ldots + a_{k-1} x^{k-1}$$
    at a fixed value x, ignoring overflows
  - Especially suitable for strings (e.g., the choice $x = 33$ gives at most 6 collisions on a set of 50,000 English words)
- Polynomial $p(x)$ can be evaluated in $O(k)$ time using Horner's rule:
  - The following polynomials are successively computed, each from the previous one in $O(1)$ time
    $$p_0(x) = a_{k-1}$$
    $$p_i(x) = a_{k-i-1} + x p_{i-1}(x) \quad (i = 1, 2, \ldots, k-1)$$
- We have $p(x) = p_{k-1}(x)$

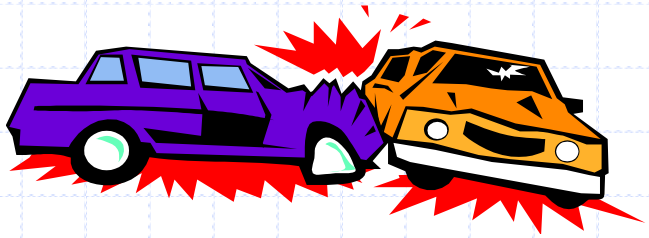
Compression Functions

- Division:
  - $h_2(y) = y \bmod M$
  - The size M 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):
  - $h_2(y) = (ay + b) \bmod M$
  - a and b are nonnegative integers such that
    $a \bmod M \neq 0$
  - Otherwise, every integer would map to the same value b

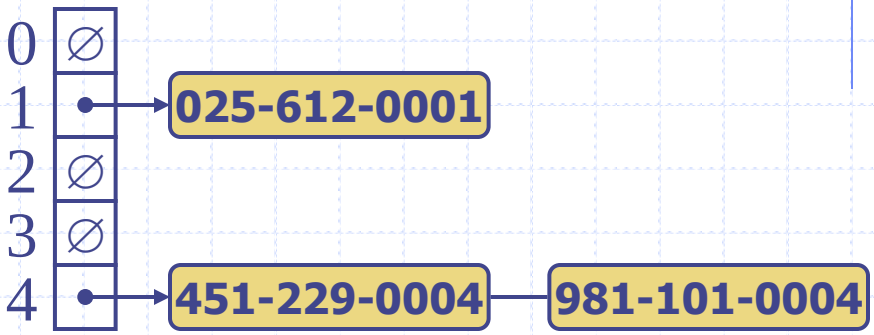
Polynomial Hash Function

Algorithm PolynomialHash(S = "$S_{k-1}S_{k-2} \ldots S_1S_0$", M, x)
Input: String S of length *k*, size M of hash table,and value x
Out: value of hash function for S
val $\leftarrow$ (int) S_{k-1}
for i $\leftarrow$ *k*-2 **downto** 0 **do**
    val $\leftarrow$ (val * x + (int)S_i) **mod** M
return val

Collision Handling



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



- Separate chaining is simple, but requires additional memory outside the table

Map with Separate Chaining

Algorithm get(T,k):
Input: Hash table T with hash function h and key k
Out: Data for key k, or NULL if no record in T hash key k

```
pos ← h(k)
p ← T[pos]
while (p != NULL) and (p.getKey() != k) do
    p ← p.getNext()
if p = NULL then return NULL
else return p.getData()
```

Time Complexity of the *get* operation

Let c be the constant number of operations performed outside the while loop and c' be the constant number of operations in one iteration of the while loop. Then the total number of operations performed by the algorithm is $c + c'$x length of list in entry $T[pos]$.

In the worst case, the hash function will map all n data items to the same position of the hash table, so the maximum number of operations performed by the algorithm is

$f(n) = c + c'n$ is $O(n)$

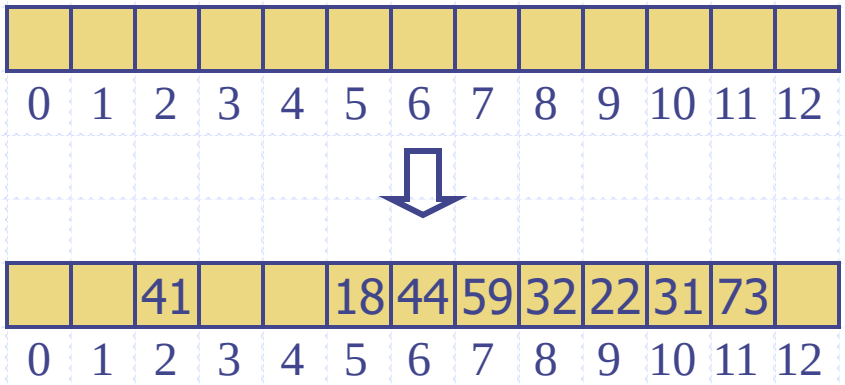
If we choose properly the size of the table and a hash function that maps the keys uniformly across the entire table, then each one of the lists in the tablewill have average size n/M. Selecting $M > n$, the average time complexity of the get operation is

$f(n) = c + c'n/M$ is $O(1)$.

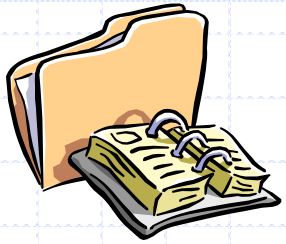
Linear Probing

- 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; this phenomenon is called clustering

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



Search with Linear Probing



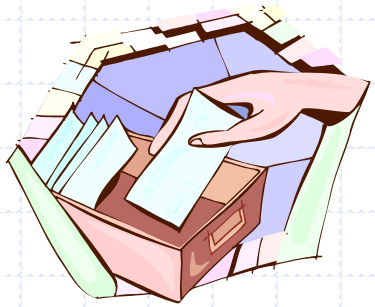
- Consider a hash table T that uses linear probing
- get(k)
  - We start at cell $h(k)$
  - We probe consecutive locations until one of the following occurs
    - An item with key k is found, or
    - An empty cell is found, or
    - M cells have been unsuccessfully probed

```
Algorithm get(k)
    i ← h(k)
    p ← 0
    repeat
        c ← T[i]
        if c = null
            return null
        else if c.getKey () = k
            return c.getData()
        else
            i ← (i + 1) mod M
            p ← p + 1
    until p = M
    return null
```

Updates with Linear Probing

- To handle insertions and deletions, we introduce a special object, called $DELETED$, which replaces deleted elements
- remove(k)
  - We search for an entry with key k
  - If such an entry (k, d) is found, we replace it with the special item $DELETED$
- put(k, d)
  - We throw an exception if the table is full or if k is in the table
  - 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 $DELETTED$, or
    - M cells have been unsuccessfully probed
  - We store (k, d) in cell i

Double Hashing



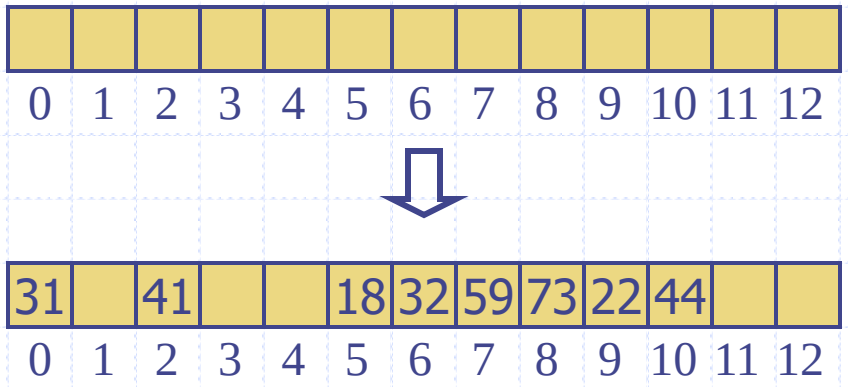
- Double hashing uses a secondary hash function $h'(k)$ or $d(k)$ and handles collisions by placing an item in the first available cell of the series
$$(i + jh'(k)) \bmod M$$
for $j = 0, 1, \ldots, M-1$
- The secondary hash function $h'(k)$ cannot have zero values
- The tablesize M must be a prime to allow probing of all the cells

- Common choice of compression function for the secondary hash function:
$$h'(k) = q - k \bmod q$$
where
  - $q < M$
  - q is a prime
- The possible values for $h'(k)$ are
$$1, 2, \ldots, q$$

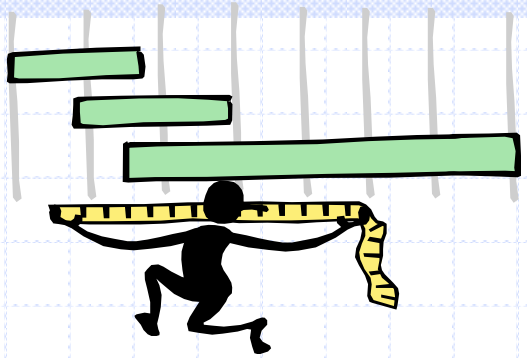
Example of Double Hashing

- Consider a hash table storing integer keys that handles collision with double hashing
  - $M = 13$
  - $h(k) = k \bmod 13$
  - $h'(k) = 7 - k \bmod 7$
- 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 map collide
- The load factor $\alpha = n/M$ 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