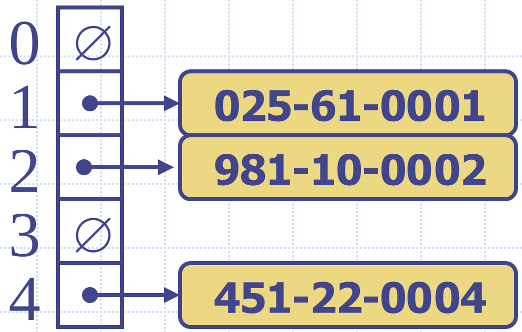
Hash Tables



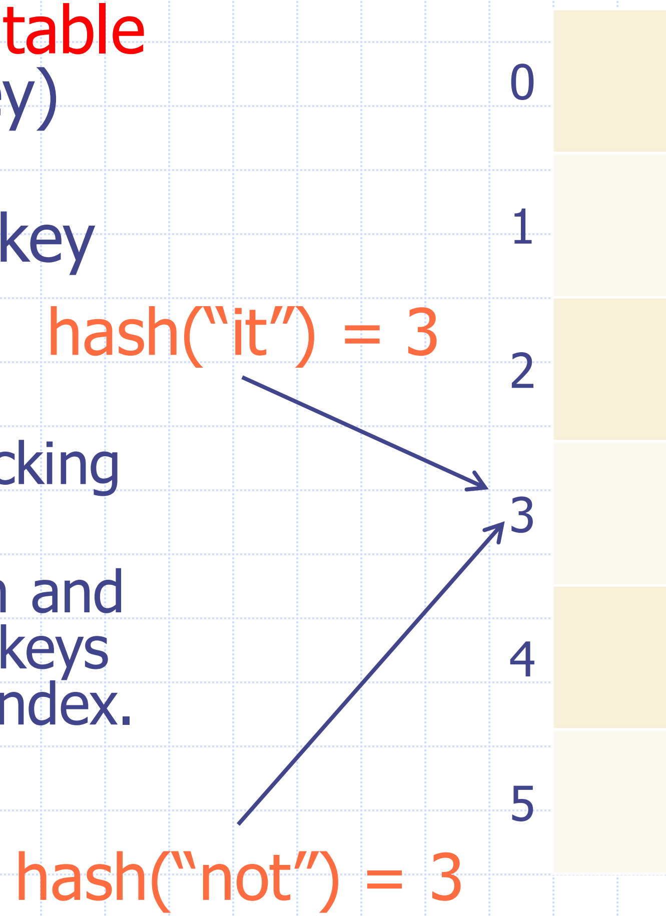
Comparison

implementation	guarantee			average case			ordered iteration?	operations on keys
	search	insert	delete	search hit	insert	delete		
sequential search (linked list)	N	N	N	N/2	N	N/2	no	`equals()`
binary search (ordered array)	lg N	N	N	lg N	N/2	N/2	yes	`compareTo()`
BST	N	N	N	1.38 lg N	1.38 lg N	?	yes	`compareTo()`
red-black tree	2 lg N	2 lg N	2 lg N	1.00 lg N	1.00 lg N	1.00 lg N	yes	`compareTo()`

- Can we do better?

Hashing: Basic Plan

- Save items in a key-indexed table (index is a function of the key)
- Hash function: Method for computing array index from key
- Issues
  - Computing the hash function
  - Equality test: Method for checking whether two keys are equal
  - Collision resolution: Algorithm and data structure to handle two keys that hash to the same array index.



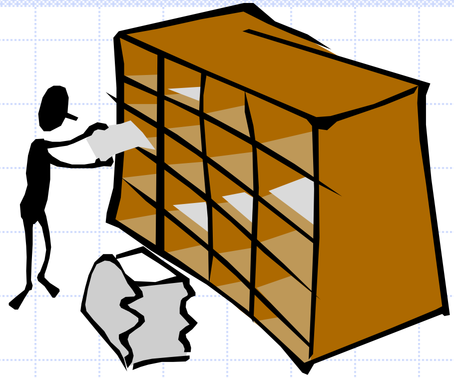
Classic Space-Time Limitation

- No space limitation: trivial hash function with key as index.
- No time limitation: trivial collision resolution with sequential search.
- Space and time limitations: hashing

Computing Hash Functions

- Idealistic goal
  - Scramble the keys uniformly to produce a table index.
  - Efficiently computable.
  - Each table index equally likely for each key.
    - Thoroughly researched problem, still problematic in practical applications
- Example: Phone numbers.
  - Bad: first three digits.
  - Better: last three digits.
- Practical challenge. Need different approach for each key type.

Hash Functions and Hash Tables

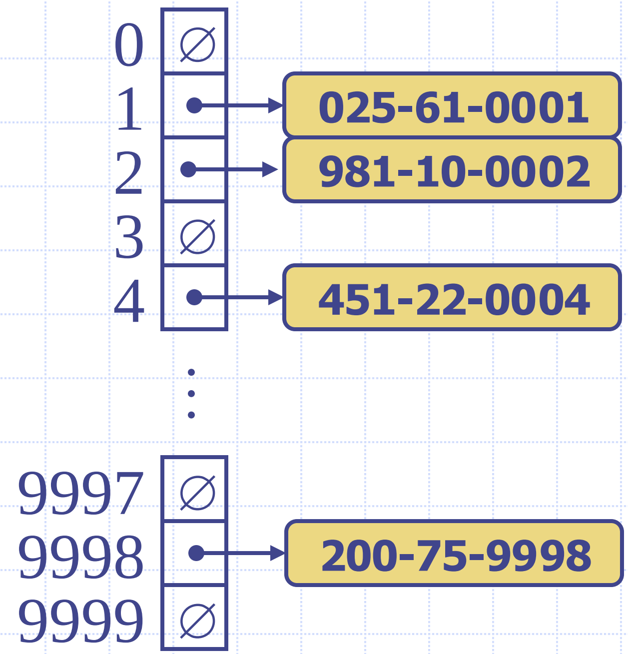


- A hash function h maps keys of a given type to integers in a fixed interval $[0, N-1]$
- Example:
  $$h(x) = x \bmod N$$
  is a hash function for integer keys
- The integer $h(x)$ is called the hash value of key x

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

Example

- We design a hash table for a map storing entries as (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 $\boldsymbol{h}(\boldsymbol{x}) = \text{last four digits of } \boldsymbol{x}$



Hash Functions

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

  Hash code:

  h_1: keys $\rightarrow$ integers

  Compression function:

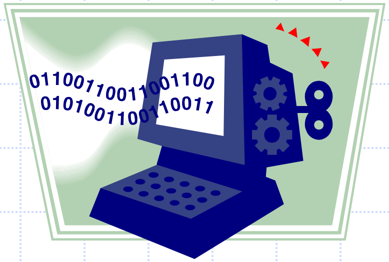
  h_2: integers $\rightarrow [0, N-1]$

- The hash code is applied first, and the compression function 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 a random way

Hash Codes



- **Memory address:**
  - We reinterpret the memory address of the key object as an integer.
  - Doesn't work for numeric and string keys.
  - Also bad if objects can move!
- **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).

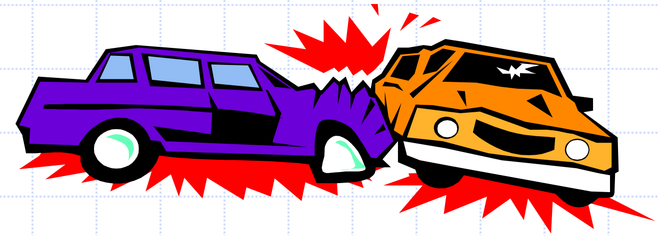
Hash Codes (cont.)

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

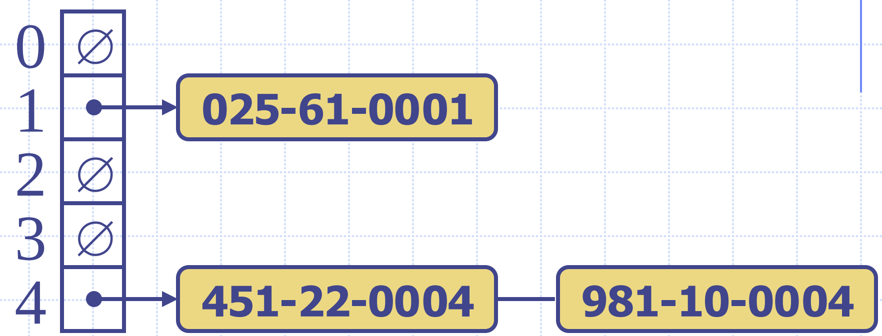
Compression Functions

- 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…
- 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
- **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

Open Addressing

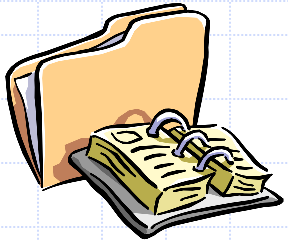
- The colliding item is placed in a different cell of the table.
- **Linear probing** handles collisions by placing the colliding item in the next (circularly) available table cell
- Each table cell inspected is referred to as a "probe"
- Colliding items lump together, causing future collisions to cause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

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

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

Separate chaining vs. linear probing

- Separate chaining.
  - Easier to implement delete.
  - Performance degrades gracefully.
  - Clustering less sensitive to poorly-designed hash function.
- Linear probing.
  - Less wasted space.
  - Better cache performance.

Quadratic Probing

- Iteratively tries buckets
  $A[(i+j^2) \bmod N]$
  for j = 0, 1, 2….. until an empty bucket is found.
- Consider a hash table storing integer keys that handles collision with double hashing
  - $N = 13$
  - $h(k) = (k + j^2) \bmod 13$
- Insert keys 18, 41, 22, 44, 59, 32 in this order

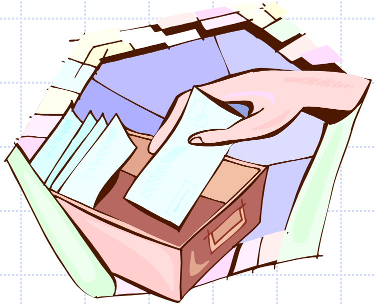
k	$h(k)$	Probes		
18	5	5		
41	2	2		
22	9	9		
44	5	5	6	
59	7	7		
32	6	6	7	10

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

⇩

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

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 function 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

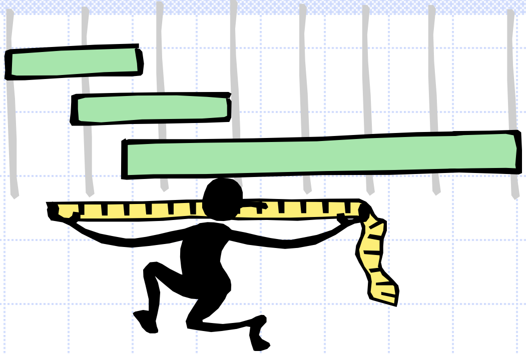
k	$h(k)$	$d(k)$	Probes		
18	5	3	5		
41	2	1	2		
22	9	6	9		
44	5	5	5	10	
59	7	4	7		
32	6	3	6		
31	5	4	5	9	0
73	8	4	8		

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 map 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%
- When the load gets too high, we can rehash….
- Applications: very numerous, e.g. computing frequencies.