# Hashing ~ 03

CSE 122 ~ Algorithms & Data Structures

# Hash Functions

- ➔ Division Method (last lecture)
- ➔ Folding Method
- ➔ Hashing by Multiplication
- ➔ Mid-Square Method

# Hash Functions ~ Folding Method

- Sometimes the key is rather long
  - Fold it
- Folding Method
  - Divide the key value into a number of parts. Divide K into parts $k_1$, $k_2$, …, $k_n$, where each part has the same number of digits (chunks) except the last part which may have fewer digits
  - Add the individual parts, $k_1+k_2+...+k_n$. The hash value is found by ignoring the last carry, if any
  - Example:
    - Given m = 100, by folding the key 5678 you get 56 + 78 = 134, or 34 by dropping the 1
    - If K = 321 you would get 32 + 1 = 33
    - If K = 34567 you get 34 + 56 + 7 = 97

# Hashing By Multiplication

- ➔ Sometimes you want *m* to be non-prime
- ➔ Weird, but true math fact
  - If θ is an irrational number (θ cannot be expressed as the ratio of a/b where a and b are integers) and for large enough *n* the fractions {θ}, {2θ}, {3θ}, …, {nθ} are distributed very uniformly from 0 to 1
- ➔ Choosing θ equal to the reciprocal of the golden ratio $\phi^{-1}$ is a particularly excellent choice as it causes the distribution to be particularly excellent choice
  - So choose $\theta = \phi^{-1}$ fix *m* and define the multiplicative hash for key K as $h(K) = \lfloor m\{K\theta\} \rfloor$
- ➔ Example
  - Assume m = 1000 and K = 12345
    - $h(K) = \lfloor 1000\{12345 \times 0.61803399\} \rfloor = 629$

# Hashing ~ Mid-Square Method

- Method
  - Square the value of the key. That is find $K^2$.
  - Extract the middle *r* bits of the result obtained in 1.
- Works because the distribution is not dominated by the bottom or top digit of the original key
- Example:
  - M = 100 and the indexes range from 0 - 99. So need 2 digit numbers or 16 bits
  - If K = 1234, $K^2$ = 1522756, extract the 3rd and 4th digit bits or 27
  - If K = 3287, $K^2$ = 10804369 have to be consistent so extract 43

# Dealing with Collisions

- Collisions - your hash function maps two different keys to the same location
- Collision Resolution Techniques
  - Open Addressing
    - Linear/Quadratic Probing
    - Double Hashing
  - Chaining

# Open Addressing

- Once a collision takes place, open addressing computes a new position using a ***probe sequence*** and the next record is stored in that position
  - All values are stored in the hash table
  - The hash table takes two values:
    - Sentinel value: (-1) a flag that indicates the memory location is open (not occupied)
    - Data values
  - If a location has some value in it other slots are examined systematically to find an open slot
  - The process of examining memory locations is called ***probing***
  - If no free locations are found you have an overflow condition

# Linear Probing

- ➔ Hash with $h(K) = K \bmod m$, where $m$ is prime and equal to the table-size
- ➔ Assume h(K) is already occupied, then use the following to resolve the collision:
  - If key K hashes to index i but that position is occupied by another record, just try positions i+1, i+2, … until an empty slot is found and store the record with K there.
  - If the search continues beyond the end of the table (beyond m−1) then continue from the top
    - *rehash(key) = (h(K)+i) mod m*
  - If search reaches the initial probe position a second time the table is full and no hope to insert the key

# Searching for a Value Using a Linear Probe

- Given a key
  - Calculate h(K)
  - Check the location, if key found. You are done. O(1)
  - If the key does not match begin a search of the array using a linear probe (sequential) until:
    - The value is found
    - The search function encounters a vacant location in the array indicating the value is not present
    - The search terminates because the table is full and the value is not present
    - Worst Case: O(n)

# Cons of Linear Probing

- Linear Probing works if the table is **not too full**
- As hash table fills, you get clusters of consecutive cells which increases the times for insertions and searches ~ **primary clustering**
- Once a block of a few contiguous occupied positions emerges in the table it becomes a *target* for subsequent collisions.
- A collision in any position in the cluster makes the cluster grow larger.
- The larger the cluster, the bigger a target it becomes

# Quadratic Probing

- Similar to linear probing, but now use the following to resolve collisions:
  - If key K hashes to index i, but that position is occupied by another record, just try positions $i+1^2$, $i+2^2$, $i+3^2$ = i+1, i+4, i+9 … until an empty slot is found and store the record with K there
    - H(K,0) = h(K)
    - $H(K,p+1) = H(K,p)+p^2)mod\ m$
    - $rehash(key) = (h(K)+ p^2)\ mod\ m$

# Quadratic Probing: Pros and Cons

- Helps eliminate primary clustering, but you now get ***secondary clustering***
- If there is a collision between two keys the same probe sequence is followed by both keys
- Collisions occur more frequently as the table becomes full
- Search is similar to linear probing

# Double Hashing

- Intervals between probes is defined by another hash function
- Double hashing helps reduce clustering.
- The 2nd hash should be $h_1(K) \neq 0$ and $h_1 \neq h(k)$
  - So $h(k,i) = [h_i(k) + ih_2(k)] \bmod m$
  - M is the table size
  - $h_1(k) = k \bmod m$
  - $h_2(k) = k \bmod m*$
  - $i = 0$ to $m-1$
  - $m* < m$ and can choose $m* = m-1$ or $m-2$
    - If $h(k)$ produces a location that is occupied probe the locations $h(k)+h_1(k) \bmod m$, $(h(k)+2 * h_1(k)) \bmod m$, ...

# Pros and Cons of Double Hashing

- ➔ Minimizes primary and secondary clustering

# Deletions from Open Addressed Hash Table

- Seems trivial ~ just delete the key
  - But this really messes up searching for a key in the hash table
  - Deletion can possibly lead to empty positions which might be on the probe sequence ~ giving false positives
- Usual way to fix false positives ~ people a one bit field to the table entry which indicates the slot has been deleted
  - Thus when searching, you check the deleted flad and skip over if a value has been deleted
  - Searches become longer when deletions of hash table increase and its possible every slot in the table has been deleted
- Another solution: avoid deletions altogether, or rebuild the hash table

# Collision by Chaining

- ➔ In chaining, each location in the hash table stores a pointer to a linked list that contains all the key values that were hashed to the same location
- ➔ Data Structure consists of two levels:
  - The hash table is an index that divides the dictionary into *m* linked lists
  - The linked lists are referred to as ***buckets***

# Load Factor

- Define a ***probe*** as one access to the data structure
  - Thus in chaining one **probe** to get the list header, and a second to retrieve the first record in the linked list
- For a LookUp, the time is proportional to the number of probes.
  - Thus the number of probes is a good indicator of efficiency
- Let *n* be the size of the dictionary to be stored and *m* be the size of the hash table, we define the ***load factor*** to be $\lambda = n/m$
  - A load factor of $\lambda = 0$ indicates an empty table. A load factor of $\lambda = 0.5$ indicates a table that is half full (half-empty)
  - Load factors can never exceed 1 for open addressing
  - Load factors can be > 1 for chaining