CSCI 104
Hash Tables & Functions

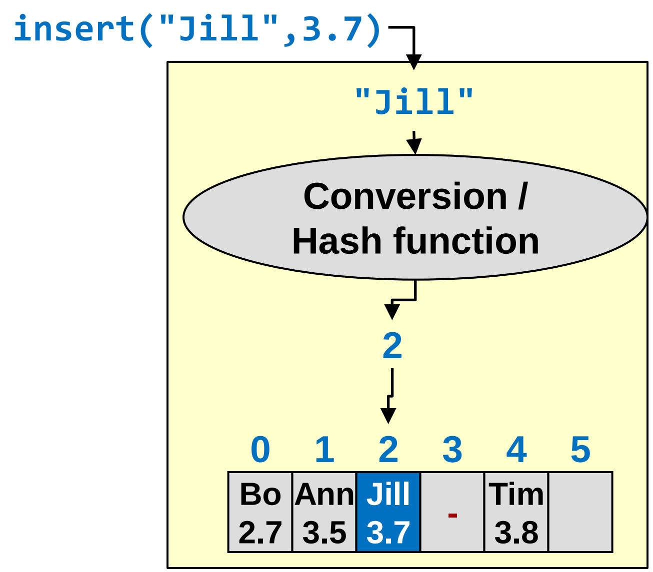
Mark Redekopp
David Kempe

REVIEW

USC Viterbi School of Engineering

Hash Tables - Insert

- To **insert** a key, we hash the (potential non-integer) key to an integer and place the key (and value) at that index in the array
- The conversion function is known as a ***hash function, h(k)***
- A hash table implements a set/map ADT
  - `insert(key)` / `insert(key,value)`
  - `remove(key)`
  - `lookup/find(key) => value`
- ***Question to address***: What should we do if two keys ("Jill" and "Erin") hash to the same location (aka a **COLLISION**)?
- Recall: A "**good**" hash is one where items hash to a given location with probability **1/m**



A map implemented as a hash table (key=name, value = GPA)

<u>Hash table parameter definitions:</u>

n = # of keys entered (=4 above)

m = tableSize (=6 above)

$\alpha = \frac{n}{m}$ **= Loading factor = (4/6 above)**

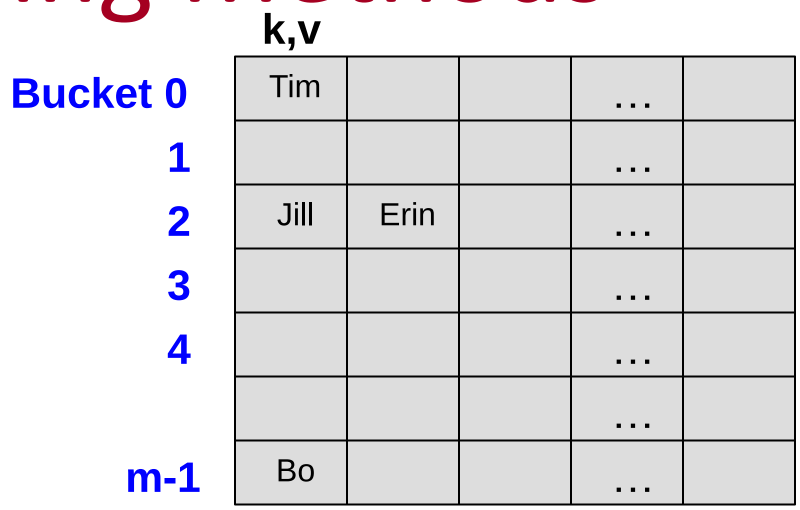
Resolving Collisions

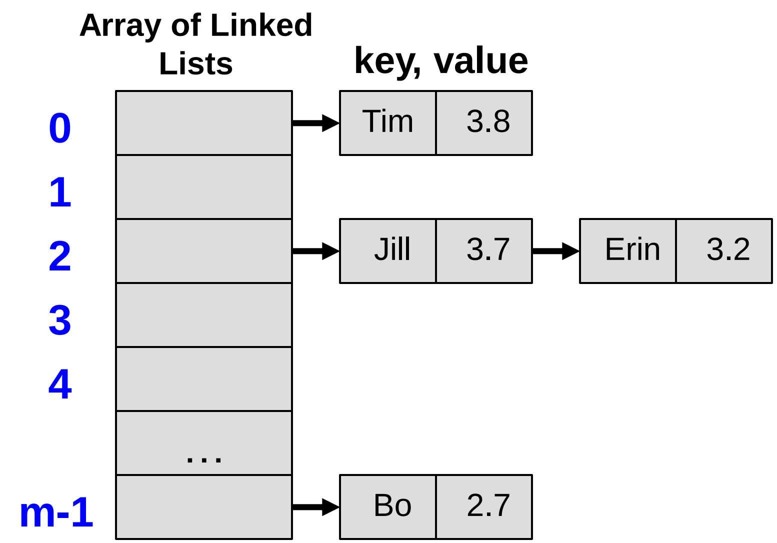
- Collisions occur when two keys, k1 and k2, are not equal, but h(k1) = h(k2)
- Collisions are inevitable so we have to handle them
- Methods
  - **Closed Addressing** (e.g. buckets or **chaining**): Keys MUST live in the location they hash to (thus requiring multiple locations at each hash table index)
  - **Open Addressing (aka probing):** Keys MAY NOT live in the location they hash to (only requiring a single 1D array as the hash table)
    - Methods: 1.) Linear Probing, 2.) Quadratic Probing, 3.) Double-hashing

USC Viterbi School of Engineering

Closed Addressing Methods

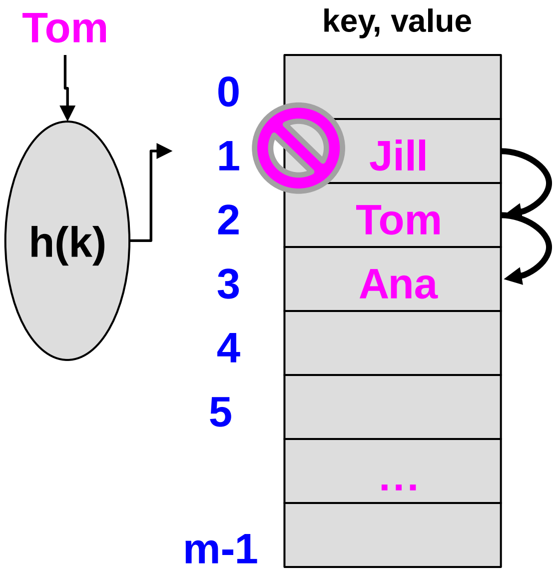
- Make each entry in the table a fixed-size ARRAY (bucket) or LINKED LIST (chain) of items/entries so all keys that hash to a location can reside at that index
  - **Close Addressing** => A key **will reside in the location it hashes to** (it's just that there may be many keys (and values) stored at that location
- **Buckets**
  - How big should you make each array?
  - Too much wasted space
- **Chaining**
  - Each entry is a linked list (or, potentially, vector)

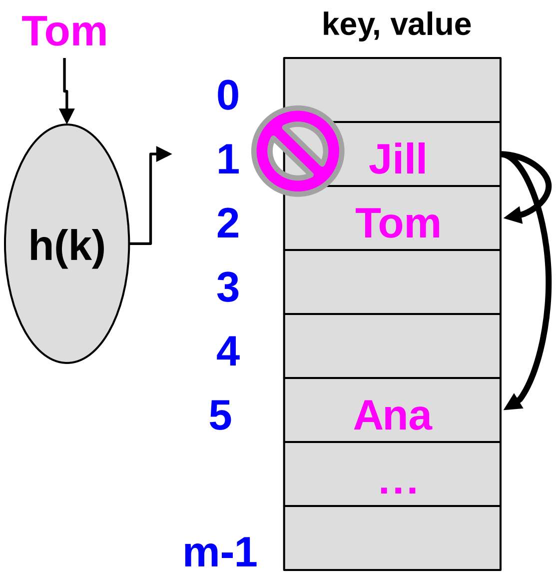




Probing Technique Summary

- If h(k) is occupied with another key, then probe
- Let i be number of **failed** probes
- **Linear Probing**
  - $h(k,i) = (h(k)+i)$ **mod m**
- **Quadratic Probing**
  - $h(k,i) = (h(k)+i^2)$ **mod m**
  - If h(k) occupied, then check $h(k)+1^2$, $h(k)+2^2$, $h(k)+3^2$, …
- **Double Hashing**
  - Pick a second hash function $h_2(k)$ in addition to the primary hash function, $h_1(k)$
  - $h(k,i) = [\ h_1(k) + i*h_2(k)\ ]$ **mod m**





Expected Chain Length

- In a hash table that uses chaining, recall that loading factor, α, defined as:
  - (n=number of items in the table) / (m=tableSize) => **α** = n / m
  - It is just the fraction of locations currently occupied
- For chaining, α, can be greater than 1
  - This is because n > m
  - For given values of n and m, let L = the length of a chain at some location = number of items that hashed to that location
  - **What is E[L]**? (Hint: Consider an item hashing to location x as a Bernoulli trial
    - P(success) = P(1 key hashes to some location x) = 1/**m**
  - **E[L]** = n/m = α
- Best to keep the loading factor, α, below 1
  - Resize and rehash contents if load factor too large (using new hash function)

Hash Efficiency Summary

- Suboperations
  - Compute h(k) should be O(1)
  - Array access of table[h(k)] = O(1)
  - Probing or search of chain = O(??)
- In a hash table using chaining, the runtime of each operation is at most the expected length of the chain (i.e. α) that the item hashes to
  - Find = O(α) = O(1) since α should be kept constant
  - Insert = O(α) = O(1) since α should be kept constant
  - Remove = O(α) = O(1) since α should be kept constant

Review of A Few Things Probability and Number Theory Taught Us

- **Quadratic probing**: If we use a prime table size, **m**, the first **m/2** probes are guaranteed to go to distinct locations.
- **Double hashing**: If we use a prime table size, **m**, and limit our 2nd hash function to a non-multiple of **m**, we will visit ALL **m** **distinct locations** in our probe sequence
  - Theorem: If p is prime and $0 < a < p$, then: $[\mathbf{0} \cdot \boldsymbol{a}], [\mathbf{1} \cdot \boldsymbol{a}], [\mathbf{2} \cdot \boldsymbol{a}], \dots, [(\boldsymbol{p} - \mathbf{1}) \cdot \boldsymbol{a}]$ **are all distinct (i.e. a permutation of 0,1,…,(p-1))**
- What is the expected length, L, of a chain at some location in the hash table?
  - **E[L]** = n/m = α
- What is the expected number of empty buckets?
  - $\mathrm{k} \cdot \left(\frac{k-1}{k}\right)^n$

HASH FUNCTIONS

Possible Hash Functions

- Define **n** = # of entries stored, **m** = Table Size, k is non-negative integer key
- h(k) = 0 ?
- h(k) = k mod **m** ?
- h(k) = rand() mod **m** ?
- Rules of thumb
  - The hash function should examine the entire search key, not just a few digits or a portion of the key
  - When modulo hashing is used, the base should be prime

Hash Function Goals

- A "perfect hash function" should map each of the **n** keys to a unique location in the table
  - Recall that we will size our table to be larger than the expected number of keys…i.e. **n < m**
  - Perfect hash functions are not practically attainable
- A "good" hash function or *Universal Hash Function*
  - Is easy and fast to compute
  - Scatters data uniformly throughout the hash table
    - $P(h(k) = x) = 1/\mathbf{m}$ (i.e. **pseudorandom**)

Universal Hash Example

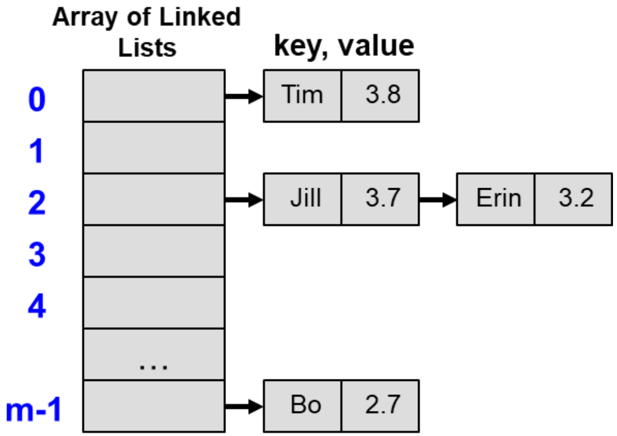
- Suppose we want a universal hash for words in English language
- First, we select a prime table size, **m**
- For any word, w made of the sequence of letters w_1 w_2 ... w_n we translate each letter into its position in the alphabet (0-25).
  - Example: "bad" = 1 0 3
- Suppose the length of the longest word in the English alphabet has length z (or we set the maximum length of a key to z)
- Choose a random number (key), R, of length z, R = r_1 r_2 ... r_z
  - The random key is created once when the hash table is created and kept
  - Example: say z=35 (longest word in English is 35 characters). Pick 35 random numbers: 28 4 15 ... 71
- Hash function: $h(w) = \left(\sum_{i=1}^{len(w)} w_i \cdot r_i\right) mod\ \boldsymbol{m}$
  - **Multiply the number corresponding to each letter times the selected random value and sum them all up**

Universal Hash Example

- First, we select a prime table size, **m**
- For any word, w made of the sequence of letters $w_1 w_2 \dots w_n$ we translate each letter into its position in the alphabet (0-25).
  - Example: "bad" = 1 0 3
- Choose a random number (key), R, of length z, $R = r_1 r_2 \dots r_z$
  - The random key is created once when the hash table is created and kept
  - Example: say z=35 (longest word in English is 35 characters). Pick 35 random numbers: 28 4 15 … 71
- Hash function: $h(w) = \left(\sum_{i=1}^{len(w)} w_i \cdot r_i\right) mod\ \boldsymbol{m}$
  - If w = "hello" then h(w) = (h*28 + e*4 + l*15 + l*18 + o*9) mod **m**
    - **Plug in alphabet position (or ASCII values in reality) for each letter being multiplied above**
  - Notice if w = "olleh" we will get a very different h(w)
    - w = "olleh" then h(w) = (o*28 + l*4 + l*15 + e*18 + h*9) mod **m**

When Collisions Occur

- How early (on which insertion) can a collision occur (if we had an adversary)? **2**
- When is a collision guaranteed to occur (the latest insertion)? **m+1**
- If **n** > **m**, is every entry in the table used?
  - No. Some may be blank?
- If **n** > **m**, is it possible we haven't had a collision?
  - No. Some entries have hashed to the same location according to the **Pigeon Hole Principle**
  - We can only avoid a collision when **n** < **m**
- Collisions are likely even if **n** < **m** *(by the birthday paradox)*
  - Given **n** random values chosen from a range of size **m**, we would expect a duplicate random value in O($\mathbf{m}^{1/2}$) trials
    - For actual birthdays where **m** = 365, we expect a duplicate within the first 23 trials



Taking a Step Back

- In most applications the UNIVERSE of possible keys >> m
  - To store each of the ~40K USC students suppose we choose a table size of **m** ≈ 100K = 10^5 so α ≈ 0.4
  - But because we use 10-digit USC ID's, there are **10^{10}** potential keys
  - That means for each of the 100K table locations there are $10^{10}/10^5$ keys that map to any given location (by the generalized pigeon-hole principle)
  - What if we got REALLY unlucky, or worse, we had an adversary who fed us those $10^{10}/10^5$ keys in an attempt to degrade performance
- How can we try to mitigate the chances of this poor performance?
  - One option: Switch hash functions periodically
  - Second option: choose a hash function that makes engineering a sequence of collisions ***EXTREMELY*** hard (aka 1-way hash function)

One-Way Hash Functions

- Why all this mention of an adversary?
- **Fact of Life: What's hard to accomplish when you actually try is even harder to accomplish when you do not try**

- So if we have a hash function that would make it hard to find keys that collide (i.e. map to a given location, i) when we are ***trying*** to be an adversary…
- …then under normal circumstances (when we are ***NOT trying*** to be adversarial) it would be very rare to accidentally produce a sequence of keys that leads to a lot of collisions

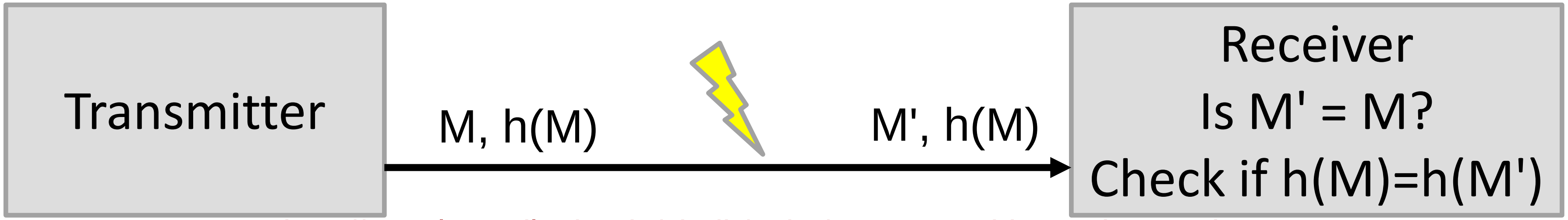
- We call those hash functions, **1-way or cryptographic hash functions**
- **Main Point: If we can find a function where even though our adversary knows our function, they still can't find keys that will collide, then we would expect good performance under general operating conditions**

One-Way Hash Function

- $h(k) = c = k \bmod 11$
  - What would be an adversarial sequence of keys to make my hash table perform poorly?
- It's easy to compute the inverse, $h^{-1}(c) \Rightarrow k$
  - Write an expression to enumerate an adversarial sequence?
  - $11*i + c$ for $i=0,1,2,3,\ldots$
- We want hash function, $h(k)$, where an inverse function, $h^{-1}(c)$ is **hard** to compute
  - Said differently, we want a function where given a location, c, in the table it would be hard to find a key that maps to that location
- We call these functions **one-way hash functions or cryptographic hash functions**
  - Given c, it is hard to find an input, k, such that $h(k) = c$
  - More on other properties and techniques for devising these in a future course
  - Popular examples: MD5, SHA-1, SHA-2

Uses of Cryptographic Hash Functions

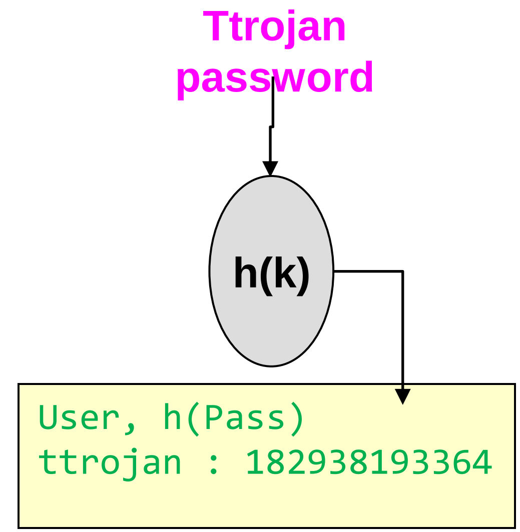
- Hash functions can be used for purposes other than hash tables
- If we no longer use a hash table, the hash code can be in a much larger range
  - We can make the hash code much longer (64-bits => 16E+18 options, 128-bits => 256E+36 options) so that chances of collisions are hopefully miniscule (more chance of a hard drive error than a collision)
- We can use a hash function to produce a "digest" (signature, fingerprint, checksum) of a longer message
  - It acts as a unique "signature" of the original content
- The hash code can be used for purposes of authentication and validation
  - Send a message, m, and h(m) over a network.
  - The receiver gets the message, m', and computes h(m') which should match the value of h(m) that was attached
  - This ensures it wasn't corrupted accidentally or changed on purpose



http://people.csail.mit.edu/shaih/pubs/Cryptographic-Hash-Functions.ppt

USC Viterbi School of Engineering

Another Example: Passwords



- Should a company just store passwords plain text?
  - No
- We could encrypt the passwords but here's an alternative
- **Don't store the passwords!**
- Instead, store the hash codes of the passwords.
  - What's the implication?
  - Some alternative password might just hash to the same location but that probability can be set to be very small by choosing a "good" hash function
    - Remember the idea that if its hard to do when you try, the chance that it naturally happens is likely smaller
  - When someone logs in just hash the password they enter and see if it matches the hashcode.
- If someone gets into your system and gets the hash codes, does that benefit them?
  - No!

SOLUTIONS

When Collisions Occur

- How early (on which insertion) can a collision occur (if we had an adversary)? **2**
- When is a collision guaranteed to occur (the latest insertion)? **m+1**
- If **n** > **m**, is every entry in the table used?
  - No. Some may be blank?
- If **n** > **m**, is it possible we haven't had a collision?
  - No. Some entries have hashed to the same location according to the **pigeon Hole Principle**
  - We can only avoid a collision when **n** < **m**
- Collisions are likely even if **n** < **m** *(**by the birthday paradox**)*
  - Given **n** random values chosen from a range of size **m**, we would expect a duplicate random value in O(**$m^{1/2}$**) trials
    - For actual birthdays where **m** = 365, we expect a duplicate within the first 23 trials