CSCI 104
Hash Tables & Functions

Mark Redekopp
David Kempe

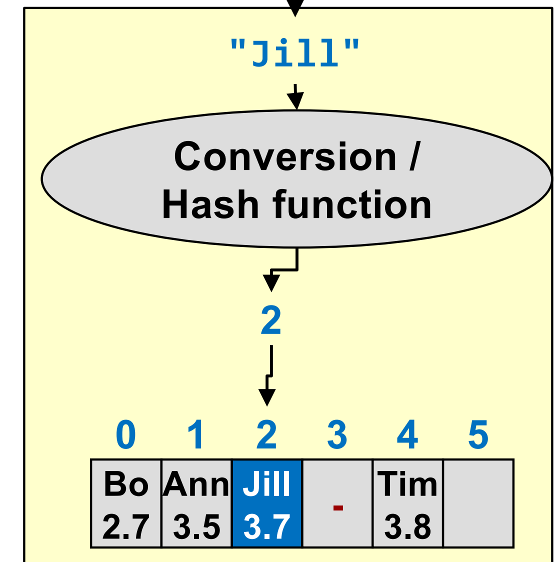
© 2022 by Mark Redekopp. This content is protected and may not be shared, uploaded, or distributed.

REVIEW

© 2022 by Mark Redekopp. This content is protected and may not be shared, uploaded, or distributed.

Hash Tables - Insert

- Can we use non-integer keys to index an array?
- Yes. Let us convert (i.e. "hash") the non-integer key to an integer
- To **insert** a key, we hash it and place the key (and value) at that index in the array
  - For now, make the unrealistic assumption that each unique key hashes to a unique integer
- 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**)?



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

Hash table parameter definitions:

n = # of keys entered (=4 above)

m = tableSize (=6 above)

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

© 2022 by Mark Redekopp. This content is protected and may not be shared, uploaded, or distributed.

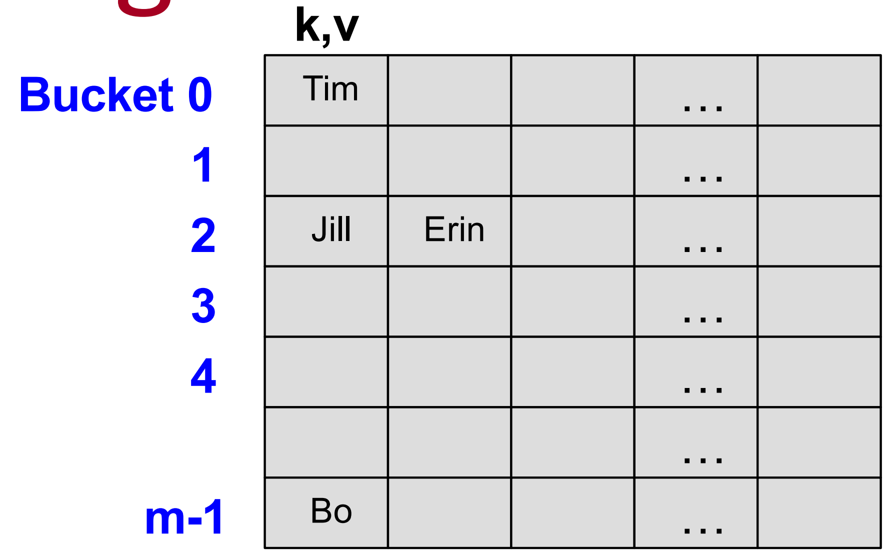
Resolving Collisions

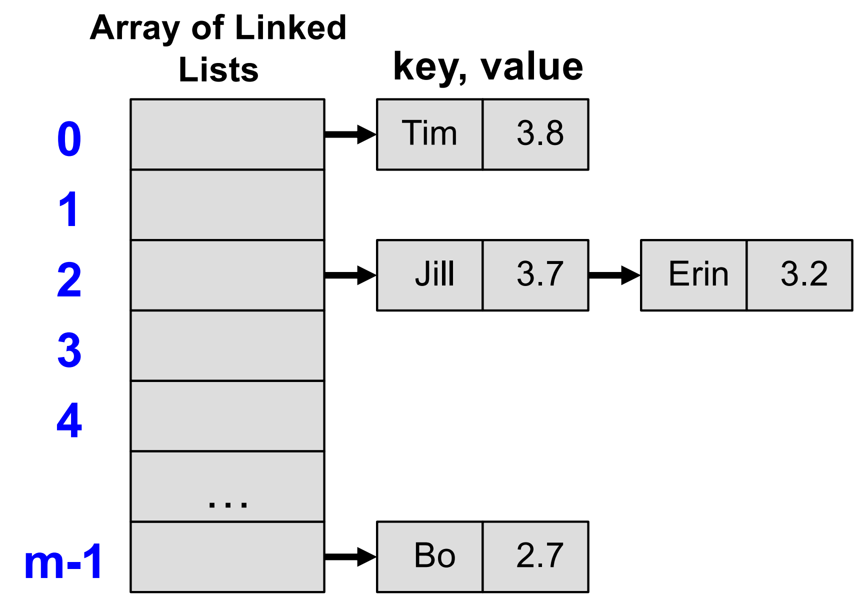
- Collisions occur when two keys, k1 and k2, are not equal, but h(k1) = h(k2).
- Collisions are inevitable if the number of entries, **n**, is greater than table size, **m** (***by pigeonhole principle***) and are likely even if **n** < **m** (***by the birthday paradox…more in our probability unit***)
- 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)
    - Methods: 1.) Buckets, 2.) Chaining
  - **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

© 2022 by Mark Redekopp. This content is protected and may not be shared, uploaded, or distributed.

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)



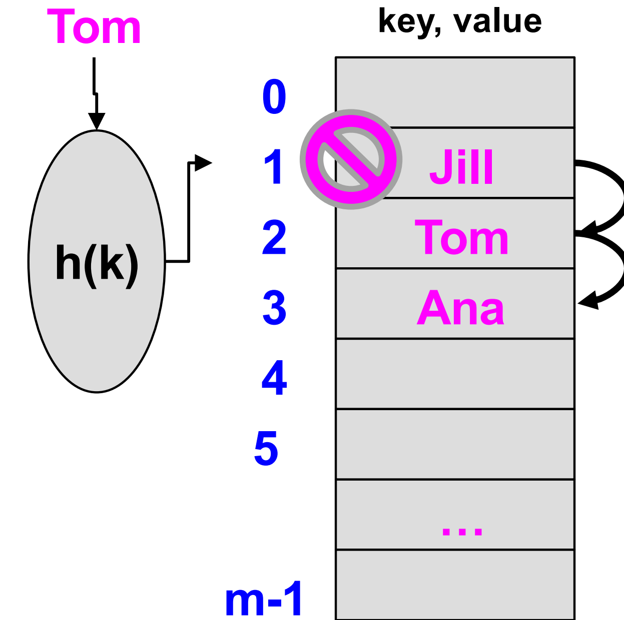


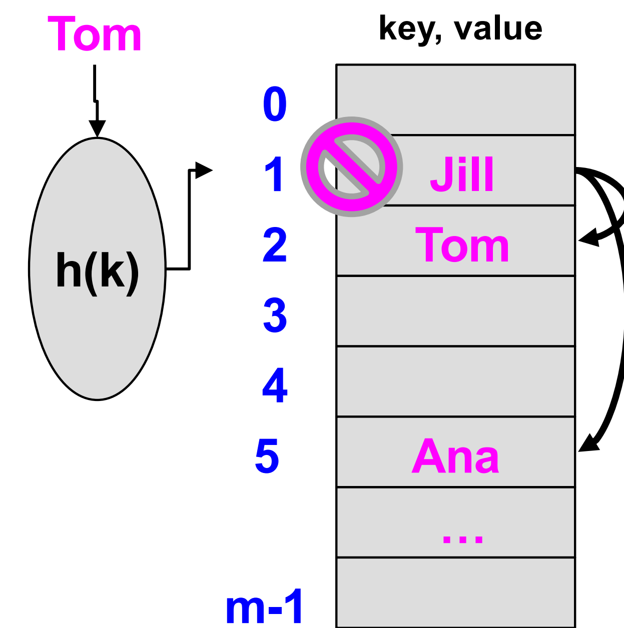
© 2022 by Mark Redekopp. This content is protected and may not be shared, uploaded, or distributed.

USC Viterbi School of Engineering

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**





© 2022 by Mark Redekopp. This content is protected and may not be shared, uploaded, or distributed.

Quadratic Probing Number Theory

- If your hash table has a prime size m, the first **m/2** probes are guaranteed to go to distinct locations.
- Proof by contradiction: Suppose the i-th and j-th probe were to the same locations (where $j < i <= m/2$), then that implies

$$(h(k)+i^2) \% m = (h(k)+j^2) \% m$$

$$(h(k)+i^2) - (h(k)+j^2) = (i^2 - j^2) = mq \quad \text{(for some q) or}$$

$$m \mid (i^2 - j^2)$$

$$m \mid (i+j)*(i-j)$$

Since m is prime it must divide one of (i+j) or (i-j)

But $i,j \leq m/2$, and $i \neq j$, so $0 < i-j$, and $i+j < m$

Since m is prime, you can't divide m over (i+j) and (i-j)

© 2022 by Mark Redekopp. This content is protected and may not be shared, uploaded, or distributed.

Hashing Efficiency

- Loading factor, α, defined as:
  - (n=number of items in the table) / m=tableSize => $\alpha = n / m$
  - Really 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 X = the number of items in some arbitrary hash table location (i.e. chain). What is E[X]?
- Best to keep the loading factor, α, below 1
  - Resize and rehash contents if load factor too large (using new hash function)

© 2022 by Mark Redekopp. This content is protected and may not be shared, uploaded, or distributed.

Hash Efficiency Summary

- Suboperations
  - Compute h(k) should be O(1)
  - Array access of table[h(k)] = O(1)
- In a hash table using chaining, what is the expected efficiency of each operation
  - 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

© 2022 by Mark Redekopp. This content is protected and may not be shared, uploaded, or distributed.

Hash Table Analysis

Assume a universal hash function, and $\alpha=1$

- When finding the item I'm looking for, how many other items, on average, will be in the same bucket?
- (m-1)/m (Linearity of Expectation!)

On average, every item has 1 "roommate".

- Note that this is different than asking what the average bucket size is.

© 2022 by Mark Redekopp. This content is protected and may not be shared, uploaded, or distributed.

HASH FUNCTIONS

© 2022 by Mark Redekopp. This content is protected and may not be shared, uploaded, or distributed.

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

© 2022 by Mark Redekopp. This content is protected and may not be shared, uploaded, or distributed.

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
    - $P(h(k) = x) = 1/m$ (i.e. **pseudorandom**)

© 2022 by Mark Redekopp. This content is protected and may not be shared, uploaded, or distributed.

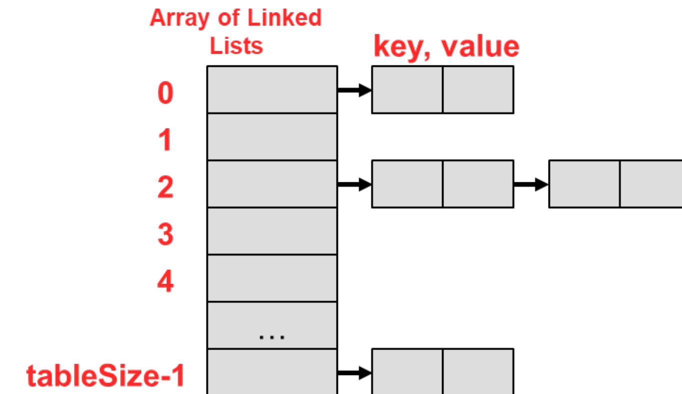
Universal Hash Example

- 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 $w_1 w_2 \ldots w_n$ we translate each letter into its position in the alphabet (0-25).
- Consider the length of the longest word in the English alphabet has length z
- Choose a random number (key), R, of length z, $R = r_1 r_2 \ldots r_z$
  - 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
- 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 ASCII values for each letter being multiplied above**
  - Notice if w = "olleh" we will get a very different h(w)

© 2022 by Mark Redekopp. This content is protected and may not be shared, uploaded, or distributed.

Pigeon Hole Principle

- Recall for hash tables we let…
  - n = # of entries (i.e. keys), m = size of the hash table
- If n > m, is every entry in the table used?
  - No. Some may be blank?
- Is it possible we haven't had a collision?
  - No. Some entries have hashed to the same location
  - Pigeon Hole Principle says given n items to be slotted into m holes and n > m there is at least one hole with more than 1 item
  - So if n > m, we know we've had a collision
- We can only avoid a collision when n < m



© 2022 by Mark Redekopp. This content is protected and may not be shared, uploaded, or distributed.

How Soon Would Collisions Occur

- Even if $\alpha < 1$ (i.e. **n** < **m**), how soon would we expect collisions to occur?
- If we had an adversary…
  - Then maybe after the second insertion
    - The adversary would choose 2 keys that mapped to the same place
- If we had a random assortment of keys…
- 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

© 2022 by Mark Redekopp. This content is protected and may not be shared, uploaded, or distributed.

Taking a Step Back

- In most applications the UNIVERSE of possible keys >> m
  - Around 40,000 USC students each with 10-digit USC ID so suppose we choose a table size of **m** ≈ 100,000 = 10^5 so $\alpha \approx 0.4$
  - How many potential keys (10-digit USC ID) exist? 10^{10}
  - Assuming m ≈ 10^5 and we use a "good" hash function, but get REALLY unlucky, how many keys COULD map to the same location in the table (by the generalized pigeon-hole principle): $10^{10}/10^5$
  - What if an adversary fed those $10^{10}/10^5$ keys to us in an attempt to make performance degrade…
- 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)

© 2022 by Mark Redekopp. This content is protected and may not be shared, uploaded, or distributed.

One-Way Hash Functions

- **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**

© 2022 by Mark Redekopp. This content is protected and may not be shared, uploaded, or distributed.

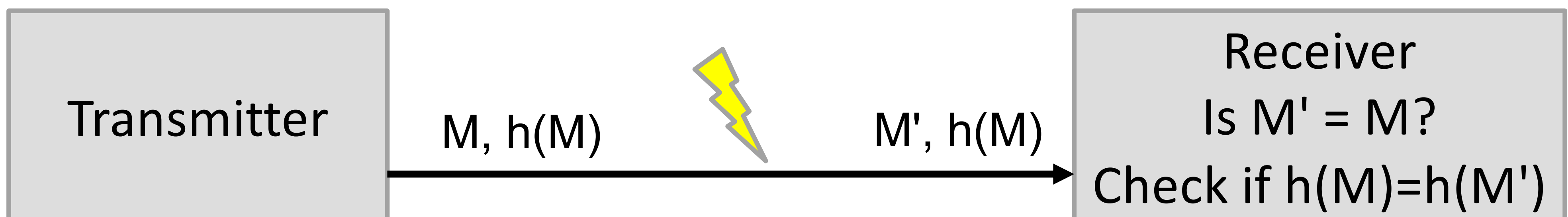
One-Way Hash Function

- $h(k) = c = k \bmod 11$
  - 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,…
- 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

© 2022 by Mark Redekopp. This content is protected and may not be shared, uploaded, or distributed.

Uses of Cryptographic Hash Functions

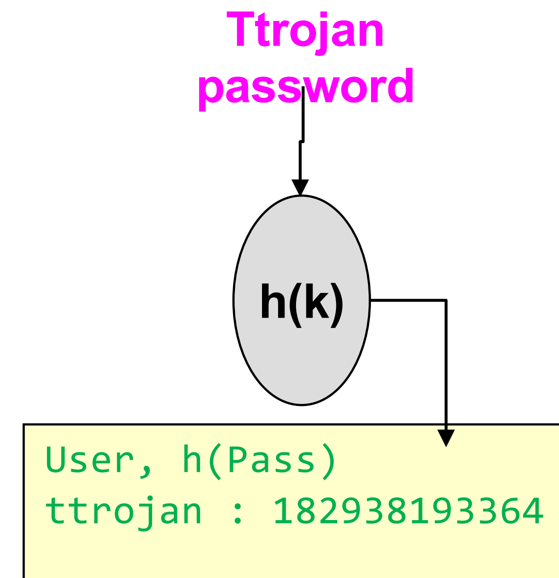
- Hash functions can be used for purposes other than hash tables
- If we no longer use a hash table, the hash code can be in a much larger range
  - We can make the hash code much longer (64-bits => 16E+18 options, 128-bits => 256E+36 options) so that chances of collisions are hopefully miniscule (more 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

© 2022 by Mark Redekopp. This content is protected and may not be shared, uploaded, or distributed.

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!

© 2022 by Mark Redekopp. This content is protected and may not be shared, uploaded, or distributed.

Another example for why we choose prime table size

BACKGROUND

© 2022 by Mark Redekopp. This content is protected and may not be shared, uploaded, or distributed.

Why Prime Table Size (1)?

- Simple hash function is h(k) = k mod **m**
  - If our data is not already an integer, convert it to an integer first
- Recall **m** should be ______________
  - PRIME!!!
- Say we didn't pick **m =** prime number but some power of 10 (i.e. k mod 10^d) or power of 2 (i.e. 2^d)…then any clustering in the lower order digits would cause collisions
  - Suppose h(k) = k mod 100
- Similarly in binary h(k) = k mod 2^d can easily be computed by taking the lower d-bits of the number
  - 19 dec. => 10011 bin. and thus 19 mod 2^2 = 11 bin. = 3 decimal

© 2022 by Mark Redekopp. This content is protected and may not be shared, uploaded, or distributed.

Why Prime Table Size (2)

- Let's suppose we have clustered data when we chose **m**=10^d
  - Assume we have a set of keys, S = {k, k', k"...} (i.e. 99, 199, 299, 2099, etc.) that all have the same value mod 10^d and thus the original clustering (i.e. all mapped to same place when **m**=10^d)
- Say we now switch and choose **m** to be a **prime** number (m=p)
- **What is the chance these numbers hash to the same location (i.e. still cluster) if we now use h(k) = (k mod m) [where m is prime]?**
  - **i.e. what is the chance (k mod 10^d) = (k mod p)**

© 2022 by Mark Redekopp. This content is protected and may not be shared, uploaded, or distributed.

Why Prime Table Size (3)

- Suppose two keys, k* and k', map to same location mod **$m=10^d$** hash table => their remainders when they were divide by **m** would have to be the same => k*-k' would have to be a multiple of **$m=10^d$**
- If k* and k' map to same place also with new prime table size, p, then
  - k*-k' would have to be a multiple of 10^d and p
  - Recall what would the first common multiple of p and 10^d be?
- So for k* and k' to map to the same place k*-k' would have to be some multiple $p*10^d$
  - i.e. $1*p*10^d$, $2*p*10^d$, $3*p*10^d$, …
  - For p = 11 and d=2 => k*-k' would have to be 1100, 2200, 3300, etc.
    - Ex. k* = 1199 and k'=99 would map to the same place mod 11 and mod 10^2
    - Ex. k* = 2299 and k'=99 would also map to the same place in both tables

© 2022 by Mark Redekopp. This content is protected and may not be shared, uploaded, or distributed.

Here's the Point

- Here's the point…
  - For the values that used to ALL map to the same place like 99, 199, 299, 399…
  - Now, only **every m-th** one maps to the same place (99, 1199, 2299, etc.)
  - This means the chance of clustered data mapping to the same location when m is prime is 1/m
  - In fact 99, 199, 299, 399, etc. map to different locations mod 11
- **So by using a prime tableSize (m) and modulo hashing even clustered data in some other base is spread across the range of the table**
  - **Recall a good hashing function scatters even clustered data uniformly**
  - **Each k has a probability 1/m of hashing to a location**

© 2022 by Mark Redekopp. This content is protected and may not be shared, uploaded, or distributed.

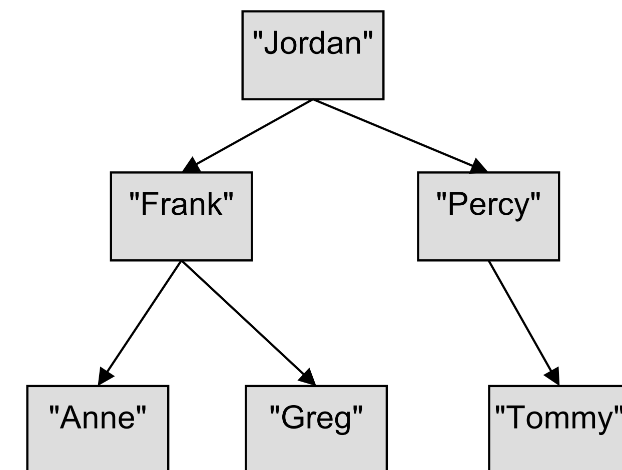
An imperfect set…

BLOOM FILTERS

© 2022 by Mark Redekopp. This content is protected and may not be shared, uploaded, or distributed.

Set Review

- Recall the operations a set performs…
  - Insert(key)
  - Remove(key)
  - Contains(key) : bool (a.k.a. find())
- We can think of a set as just a map without values…just keys
- We can implement a set using
  - List
    - O(n) for some of the three operations (BAD!)
  - (Balanced) Binary Search Tree
    - O(log n) insert/remove/contains
  - Hash table
    - O(1) insert/remove/contains



© 2022 by Mark Redekopp. This content is protected and may not be shared, uploaded, or distributed.

Bloom Filter Idea

- Suppose you are looking to buy the next hot consumer device. You can only get it in stores (not online). Several stores who carry the device are sold out. Would you just start driving from store to store?
- You'd probably call ahead and see if they have any left.
- If the answer is "NO"...
  – There is no point in going...it's not like one will magically appear at the store
  – You save time
- If the answer is "YES"
  – It's worth going...
  – Will they definitely have it when you get there?
  – Not necessarily...they may sell out while you are on your way
- But overall this system would at least help you avoid wasting time

© 2022 by Mark Redekopp. This content is protected and may not be shared, uploaded, or distributed.

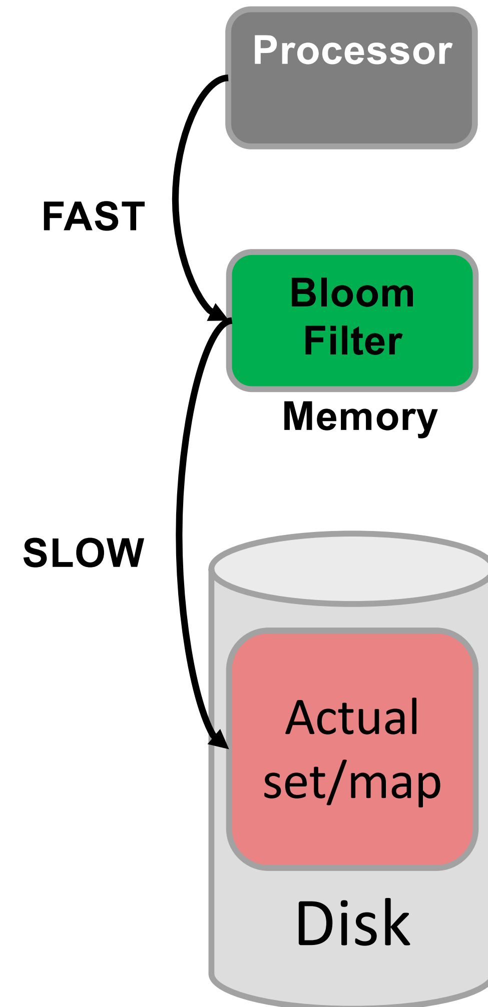
Bloom Filter Idea

- A Bloom filter is a set such that "find()" will *quickly* answer…
  - "No" correctly (i.e. if the key is not present)
  - "Yes" with a chance of being incorrect (i.e. the key may not be present but it might still say "yes")
- Why would we want this?

© 2022 by Mark Redekopp. This content is protected and may not be shared, uploaded, or distributed.

Bloom Filter Motivation

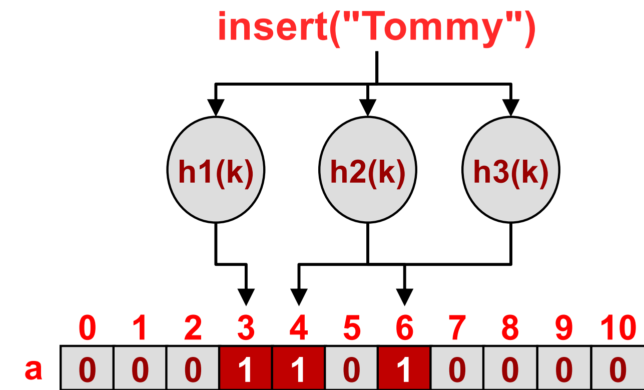
- Why would we want this?
  - A Bloom filter usually sits in front of an actual set/map
  - Suppose that set/map is EXPENSIVE to access
    - Maybe there is so much data that the set/map doesn't fit in memory and sits on a disk drive or another server as is common with most database systems
      - Disk/Network access = ~milliseconds
      - Memory access = ~nanoseconds
  - The Bloom filter holds a "duplicate" of the keys but uses FAR less memory and thus is cheap to access (because it can fit in memory)
  - We ask the Bloom filter if the set contains the key
    - If it answers "No" we don't have to spend time search the EXPENSIVE set
    - If it answers "Yes" we can go search the EXPENSIVE set

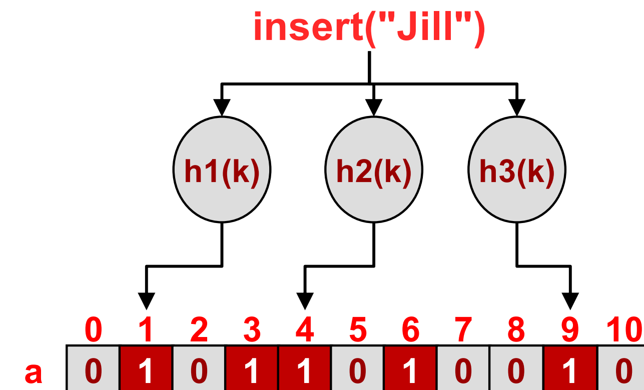


© 2022 by Mark Redekopp. This content is protected and may not be shared, uploaded, or distributed.

Bloom Filter Explanation

- A Bloom filter is…
  - A hash table of individual bits (Booleans: T/F)
  - A set of hash functions, $\{h_1(k), h_2(k), \ldots h_s(k)\}$
- Insert()
  - Apply each $h_i(k)$ to the key
  - Set $a[h_i(k)]$ = True

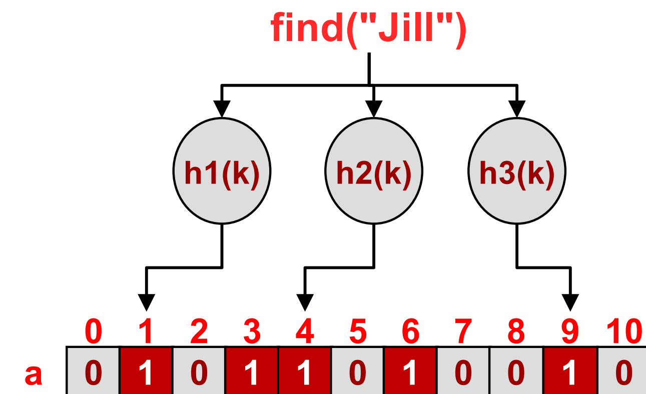


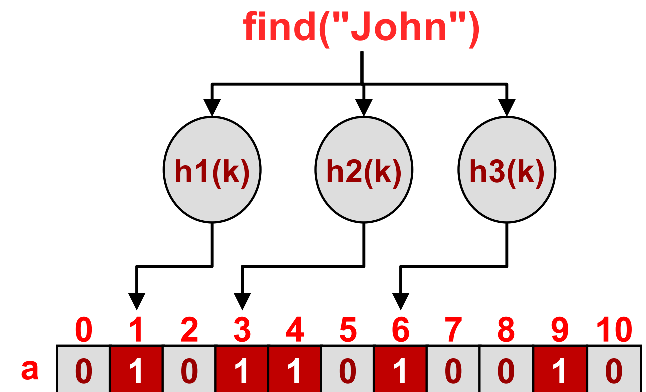


© 2022 by Mark Redekopp. This content is protected and may not be shared, uploaded, or distributed.

Bloom Filter Explanation

- A Bloom filter is…
  - A hash table of individual bits (Booleans: T/F)
  - A set of hash functions, $\{h_1(k), h_2(k), \ldots h_s(k)\}$
- find()
  - Apply each $h_i(k)$ to the key
  - Return True if ***all*** $a[h_i(k)]$ = True
  - Return False otherwise
  - In other words, answer is "Maybe" or "No"
    - May produce "false positives"
    - May NOT produce "false negatives"
- We will ignore removal for now

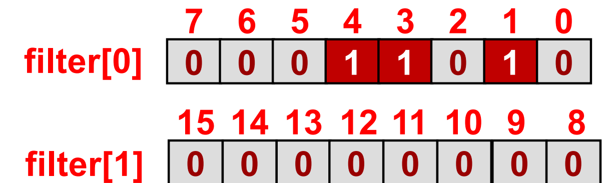




© 2022 by Mark Redekopp. This content is protected and may not be shared, uploaded, or distributed.

Implementation Details

- Bloom filters require only a bit per location, but modern computers read/write a full byte (8-bits) at a time or an int (32-bits) at a time
- To not waste space and use only a bit per entry we'll need to use bitwise operators



- For a Bloom filter with N-bits declare an array of N/8 unsigned char's (or N/32 unsigned ints)
  - `unsigned char filter8[ ceil(N/8) ];`
- To set the k-th entry,
  - `filter[ k/8 ] |= (1 << (k%8) );`
- To check the k-th entry
  - `if ( filter[ k / 8] & (1 << (k%8) ) )`

© 2022 by Mark Redekopp. This content is protected and may not be shared, uploaded, or distributed.

Practice

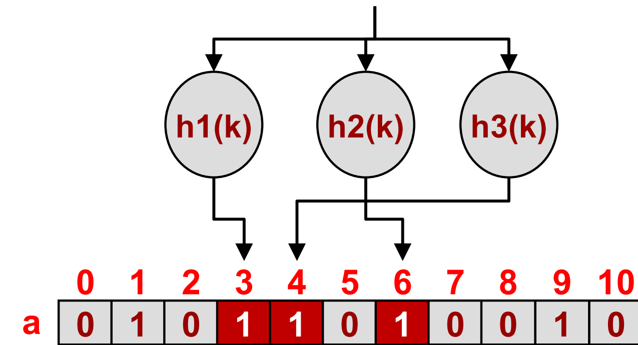
- Trace a Bloom Filter on the following operations:
  - insert(0), insert(1), insert(2), insert(8), find(2), find(3), find(4), find(9)
  - The hash functions are
    - h1(k)=(7k+4)%10
    - h2(k) = (2k+1)%10
    - h3(k) = (5k+3)%10
    - The table size is 10 (m=10).

	0	1	2	3	4	5	6	7	8	9
a	0	0	0	0	0	0	0	0	0	0

© 2022 by Mark Redekopp. This content is protected and may not be shared, uploaded, or distributed.

USC Viterbi School of Engineering

Probability of False Positives (1)



- Let us find the probability of a false positive for find(), once **n** keys are inserted in a table size of **m** using **j** hash functions
- Understand and define the event for which you are solving the probability.
  - P(**at least 1** of the n insertions chose **every** location the j hash functions map to when we call find())
- It might be easier to break this down and compute:
  - P(location **HAS** been chosen after n insertions) = 1 - P(location x **HAS NOT** been chosen after n insertions)
  - Then use that to find the probability that **ALL** j hash functions choose a location that has been set to 1 by the pervious n insertions

Define a game:

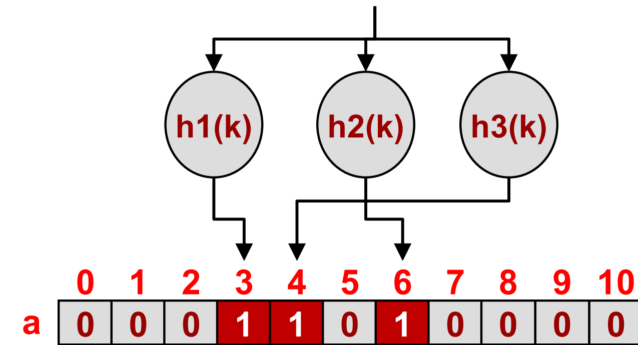
Flip a die 3 times in sequence.

You win if you roll a 6 for any of the rolls, lose otherwise.

P(you win at least once if you play n games)

© 2022 by Mark Redekopp. This content is protected and may not be shared, uploaded, or distributed.

Probability of False Positives (2a)



- What is the probability of a false positive?
- Let's work our way up to the solution
  - Probability that one hash function selects or does not select a location x assuming "good" hash functions
    - $P(h_i(k) = x) =$ ____________
    - $P(h_i(k) \neq x) =$ ____________
  - Probability that all j hash functions don't select a location
    -
  - Probability that all n-insertions in the table have not selected location x
    - $\approx$
  - Probability that a location x HAS been chosen by the previous n insertions
    - $\approx$
  - Probability that all of the j hash functions find a location True once the table has n insertions (keys)
    -

Math factoid

For small y, $1+y \approx$ ____

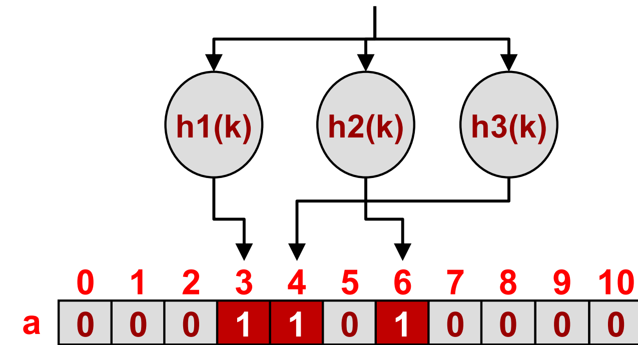
Let's substitute $y = -1/m$

So:
$1 - (1/m)$
$= 1 + (-1/m)$
$\approx$ _____

© 2022 by Mark Redekopp. This content is protected and may not be shared, uploaded, or distributed.

Probability of False Positives (2a Sol)

- What is the probability of a false positive?
- Let's work our way up to the solution
  - Probability that one hash function selects or does not select a location x assuming "good" hash functions
    - $P(h_i(k) = x) = 1/m$
    - $P(h_i(k) \neq x) = [1 - 1/m]$
  - Probability that all j hash functions don't select a location
    - $[1 - 1/m]^j$
  - Probability that all n-insertions in the table have not selected location x
    - $[1 - 1/m]^{nj}$
  - Probability that a location x HAS been chosen by the previous n insertions
    - $1 - [1 - 1/m]^{nj} \approx 1 - e^{-nj/m}$
  - Probability that all of the j hash functions find a location True once the table has n insertions (keys)
    - $(1 - e^{-nj/m})^j$



Math factoid

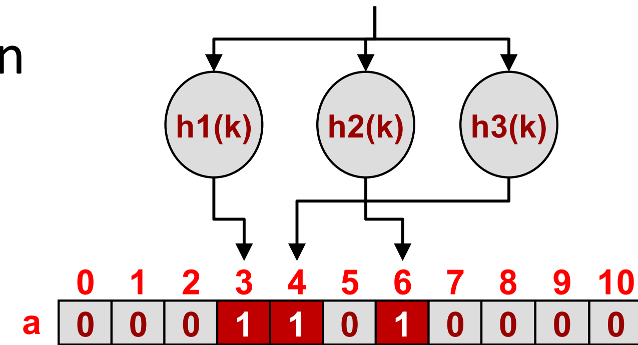
For small y, $1+y \approx e^y$

Let's substitute $y = -1/m$

So:
$1 - (1/m)$
$= 1 + (-1/m)$
$\approx e^{-1/m}$

© 2022 by Mark Redekopp. This content is protected and may not be shared, uploaded, or distributed.

Probability of False Positives (2b)



- Probability that all of the j hash functions find a location True once the table has n entries
  - $(1 - e^{-nj/m})^j$
- Define α = n/m = loading factor
  - $(1 - e^{-\alpha j})^j$
- First "tangent": Is there an optimal number of hash functions (i.e. value of j)
  - Use your calculus to take derivative and set to 0
  - Optimal # of hash functions, $j = \ln(2) / \alpha$
- Substitute that value of j back into our probability above
  - $(1 - e^{-\alpha \ln(2)/\alpha})^{\ln(2)/\alpha} = (1 - e^{-\ln(2)})^{\ln(2)/\alpha} = (1 - 1/2)^{\ln(2)/\alpha} = 2^{-\ln(2)/\alpha}$
- Final result for the probability that all of the j hash functions find a location True once the table has n entries: $\mathbf{2^{-\ln(2)/\alpha}}$
  - Recall $0 \le \alpha \le 1$

© 2022 by Mark Redekopp. This content is protected and may not be shared, uploaded, or distributed.

Sizing Analysis

- Can also use this analysis to answer or a more "useful" question...
- ...To achieve a desired probability of false positive, what should the table size be to accommodate n entries?
  - Example: I want a probability of p=1/1000 for false positives when I store n=100 elements
  - Solve $2^{-m*\ln(2)/n} < p$
    - Flip to $2^{m*\ln(2)/n} \geq 1/p$
    - Take log of both sides and solve for m
    - $m \geq [n*\ln(1/p)] / \ln(2)^2 \approx 2n*\ln(1/p)$ because $\ln(2)^2 = 0.48 \approx ½$
  - So for p=.001 we would need a table of m=14*n since ln(1000) ≈ 7
    - For 100 entries, we'd need 1400 bits in our Bloom filter
  - For p = .01 (1% false positives) need m=9.6*n (9.6 bits per key)
  - Recall: Optimal # of hash functions, j = ln(2) / α
    - So for p=.01 and α = 1/(9.6) would yield j ≈ 7 hash functions

© 2022 by Mark Redekopp. This content is protected and may not be shared, uploaded, or distributed.

Bloom Filter False Positives

- Analysis given by Bloom in 1970
- "Not quite correct" in 2008/2010
  - Makes an independence assumption that isn't correct
  - 2010 paper is correct, but unwieldy:

$$f_{christ} = \frac{m!}{m^{k(n+1)}} \sum_{i=1}^{m} \sum_{j=1}^{i} (-1)^{i-j} \frac{j^{kn} i^k}{(m-i)!j!(i-j)!}$$

  - 2023 paper: "actually Bloom is close enough":
    - "On the Evolutionary of Bloom Filter False Positives - An Information Theoretical Approach to Optimizing Bloom Filter Parameters"

https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=9863640

© 2022 by Mark Redekopp. This content is protected and may not be shared, uploaded, or distributed.

SOLUTIONS

© 2022 by Mark Redekopp. This content is protected and may not be shared, uploaded, or distributed.

Practice

- Trace a Bloom Filter on the following operations:
  - insert(0), insert(1), insert(2), insert(8), find(2), find(3), find(4), find(9)
  - The hash functions are
    - h1(k)=(7k+4)%10
    - h2(k) = (2k+1)%10
    - h3(k) = (5k+3)%10
    - The table size is 10 (m=10).

	H1(k)	H2(k)	H3(k)	Hit?
Insert(0)	4	1	3	N/A
Insert(1)	1	3	8	N/A
Insert(2)	8	5	3	N/A
Insert(8)	0	7	3	N/A
find(2)	8	5	3	Yes
find(3)	5	7	8	Yes
find(4)	2	9	3	No
find(9)	7	9	8	No

© 2022 by Mark Redekopp. This content is protected and may not be shared, uploaded, or distributed.

Practice

- Trace a Bloom Filter on the following operations for a table size (m) = 10:
  - The hash functions are
    - h1(k)=(7k+4)%10
    - h2(k) = (2k+1)%10
    - h3(k) = (5k+3)%10

	0	1	2	3	4	5	6	7	8	9
a	0	0	0	0	0	0	0	0	0	0

insert(0)
insert(1)
insert(2)
insert(8)
find(2)
find(3)
find(4)
find(9)

© 2022 by Mark Redekopp. This content is protected and may not be shared, uploaded, or distributed.