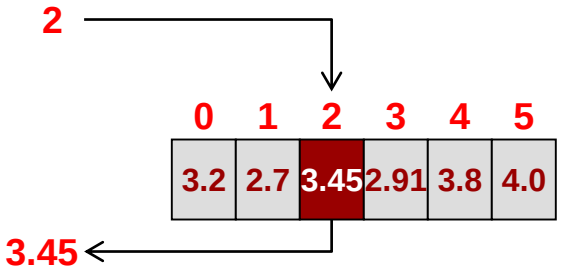
CSCI 104
Hash Tables & Functions

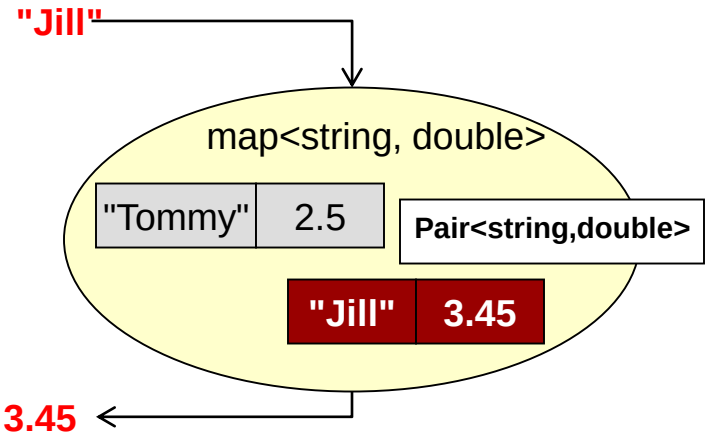
Mark Redekopp
David Kempe
Sandra Batista

Dictionaries/Maps

- An array maps **integers** to values
  - Given i, array[i] returns the value in O(1)
- Dictionaries map **keys** to values
  - Given key, k, map[k] returns the associated value
  - Key can be anything provided…
    - It has a '<' operator defined for it (C++ map) or some other comparator functor
    - Most languages implementation of a dictionary implementation require something similar to operator< for key types



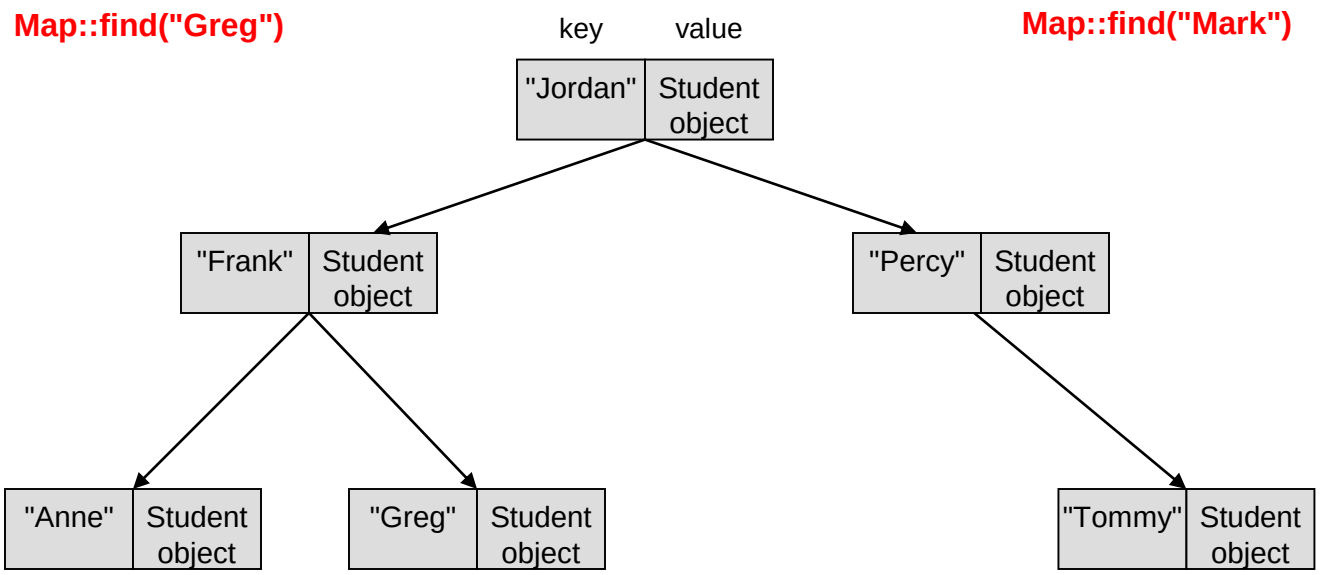
Arrays associate an integer with some arbitrary type as the value (i.e. the key is always an integer)



C++ maps allow any type to be the key

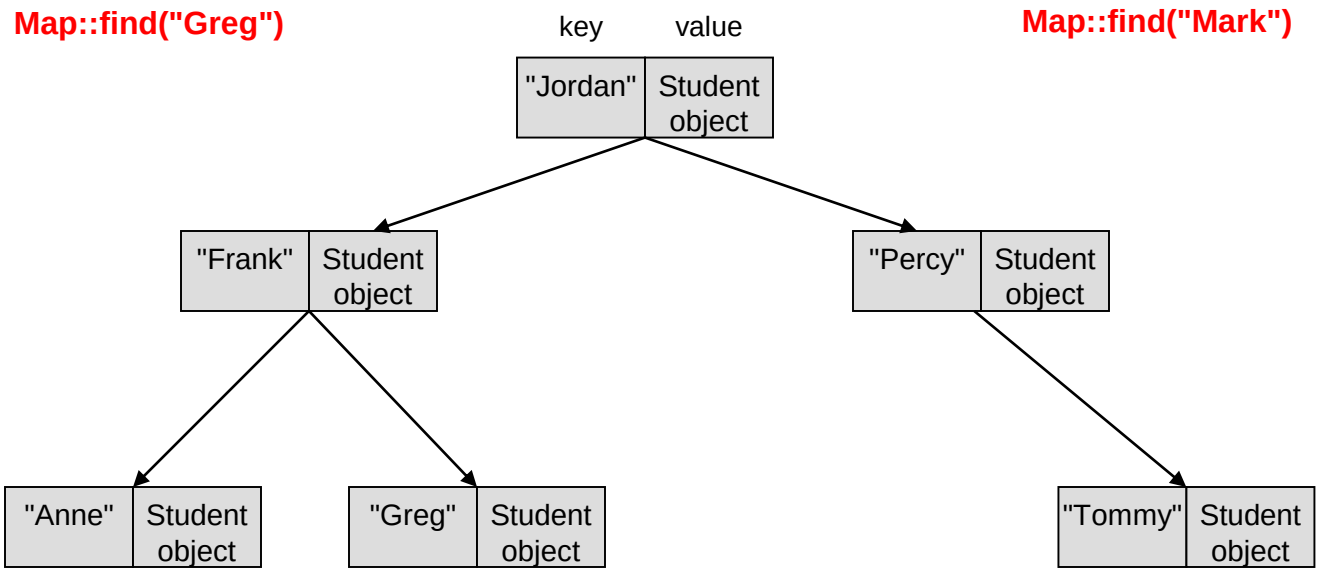
Dictionary Implementation

- A dictionary/map can be implemented with a balanced BST
  - Insert, Find, Remove = O(_____)



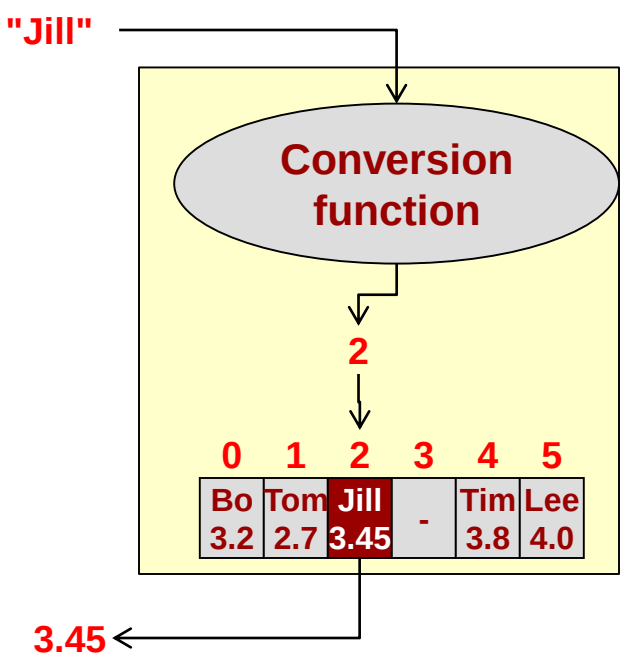
Dictionary Implementation

- A dictionary/map can be implemented with a balanced BST
  - Insert, Find, Remove = $O(\log_2 n)$
- Can we do better?
  - Hash tables (unordered maps) offer the promise of O(1) access time



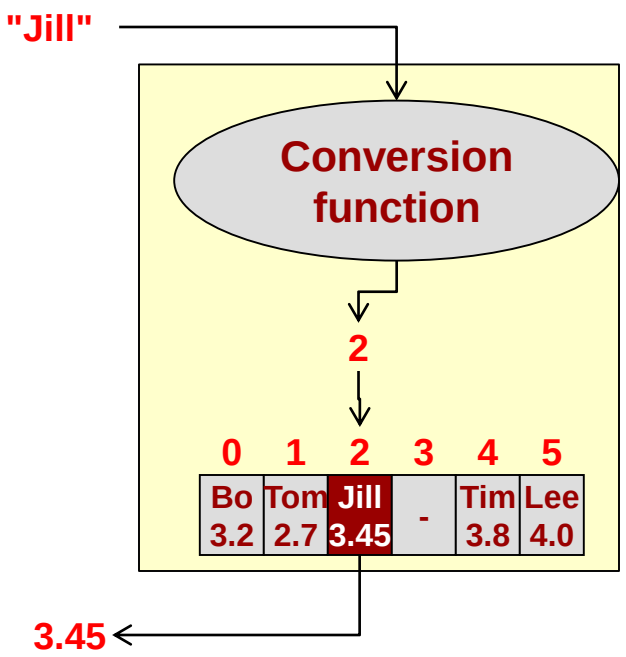
Unordered_Maps / Hash Tables

- Can we use non-integer keys but still use an array?
- What if we just convert the non-integer key to an integer.
  - For now, make the unrealistic assumption that each unique key converts to a unique integer
- This is the idea behind a hash table
- The conversion function is known as a ***hash function, h(k)***
  - It should be fast/easy to compute (i.e. O(1))



Unordered_Maps / Hash Tables

- A hash table implements a map ADT
  - Add(key,value)
  - Remove(key)
  - Lookup/Find(key) : returns value
- In a BST the keys are kept in order
  - A Binary Search Tree implements an **ORDERED MAP**
- In a hash table keys are evenly distributed throughout the table (unordered)
  - A hash table implements an **UNORDERED MAP**

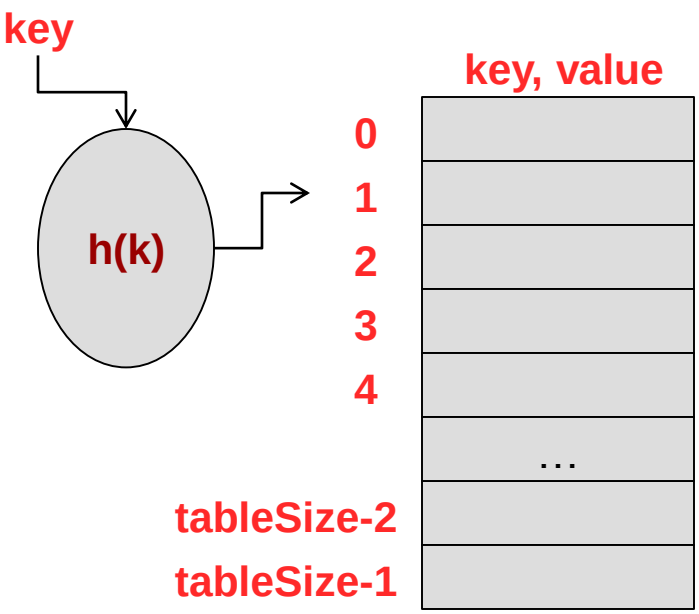


C++11 Implementation

- C++11 added new container classes:
  - `unordered_map`
  - `unordered_set`
- Each uses a hash table for average complexity to insert , erase, and find in O(1)
- Must compile with the `-std=c++11` option in g++

Hash Tables

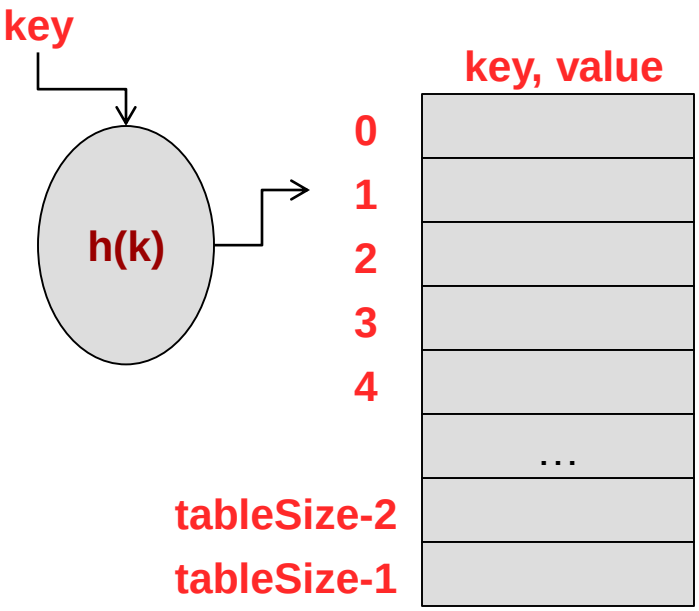
- A hash table is an array that stores key,value pairs
  - Usually smaller than the size of possible set of keys, |S|
    - USC ID's = 10^{10} options
  - But larger than the expected number of keys to be entered (defined as **n**)
- The table is coupled with a function, *h(k)*, that maps keys to an integer in the range [0..tableSize-1] (i.e. [0 to **m**-1])
- What are the considerations…
  - How big should the table be?
  - How to select a hash function?
  - What if two keys map to the same array location? (i.e. h(k1) == h(k2))
    - Known as a collision



m = tableSize
n = # of keys entered

Table Size

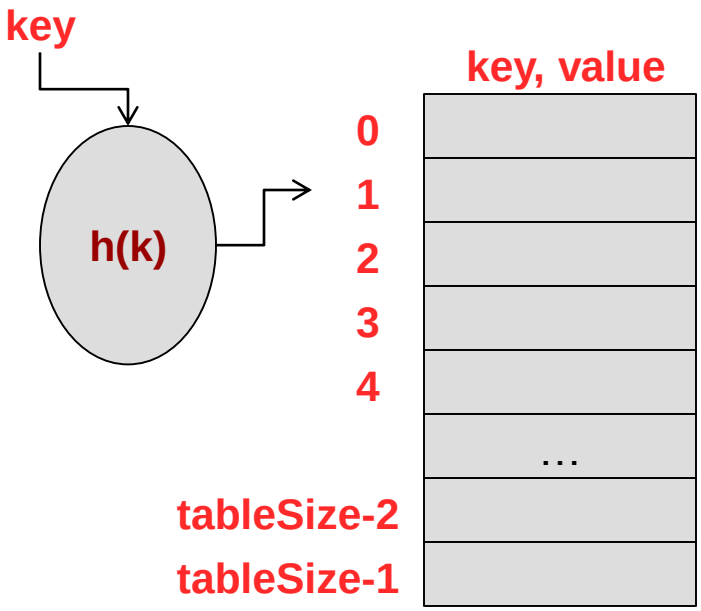
- How big should our table be?
- **Example 1**: We have 1000 employees with 3 digit IDs and want to store record for each
  - What is |S|? Expected n?
- **Solution 1**: Keep array a[1000]. Let key be ID and location, so a[ID] holds employee record (no collisions)
- **Example 2**: Using 5 letter ('a'-'z') nicknames for each student in a class.
  - $|S| = 5^{26}$ for classes of around n=100 students
  - Pick a hash table of some size much smaller (how many students do we have at any particular time)



m = tableSize
n = # of keys entered

General Table Size Guidelines

- The table size should be bigger than the amount of expected entries (**m > n**)
  - Don't pick a table size that is smaller than your expected number of entries
- But anything smaller than the size of **all possible keys** admits the chance that two keys map to the same location in the table (a.k.a. ***COLLISION***)
- You will see that tableSize should usually be a **prime number**



m = tableSize
n = # of keys entered

Hash Functions First Look

- Challenge: Distribute keys to locations in hash table such that
- Easy to compute and retrieve values given key
- Keys evenly spread throughout the table
- Distribution is consistent/repeatable for retrieval
- If necessary key data type is converted to integer before hash is applied
  - Akin to the operator<() needed to use a data type as a key for the C++ map
- Example: Strings
  - Use ASCII codes for each character and add them or group them
  - "hello" => 'h' = 104, 'e'=101, 'l' = 108, 'l' = 108, 'o' = 111 = 532
  - Hash function is then applied to the integer value 532 such that it maps to a value between 0 to M-1 where M is the table size

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/**m** (i.e. **pseudorandom**)

Universal Hash Example

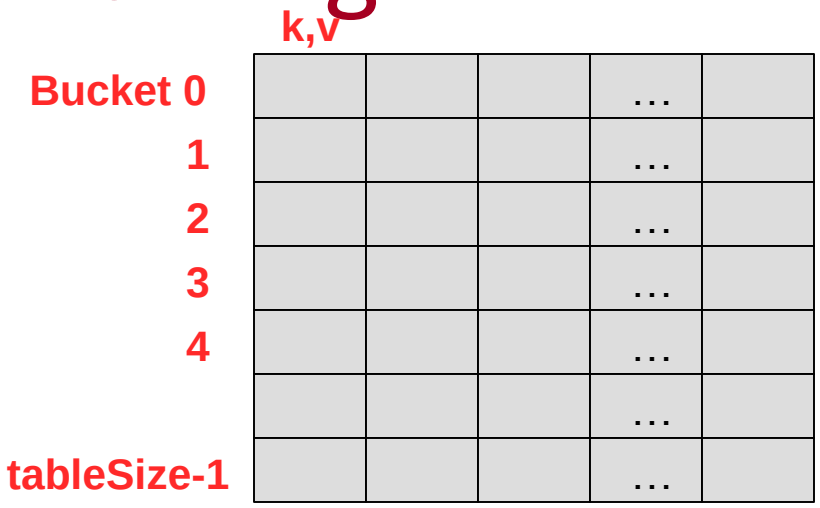
- Suppose we want a universal hash for words in English language
- First, we select a prime table size, **m**
- For any word, w made of the sequence of letters w_1 w_2 … w_n we translate each letter into its position in the alphabet (0-25).
- Consider the length of the longest word in the English alphabet has length z
- Choose a random key word, K, of length z, K = k_1 k_2 … k_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 characters: uebghakdjthzndpwjsqisndaoclcdiwevza
- Hash function: $h(w) = \left(\sum_{i=1}^{len(w)} w_i \cdot k_i\right) mod\ \boldsymbol{m}$
  - If w = "hello" then h(w) = (h*u + e*e + l*b + l*g + o*h) mod **m**
    - **Plug in ASCII values for each letter being multiplied above**
  - Notice if w = "olleh" we will get a very different h(w)

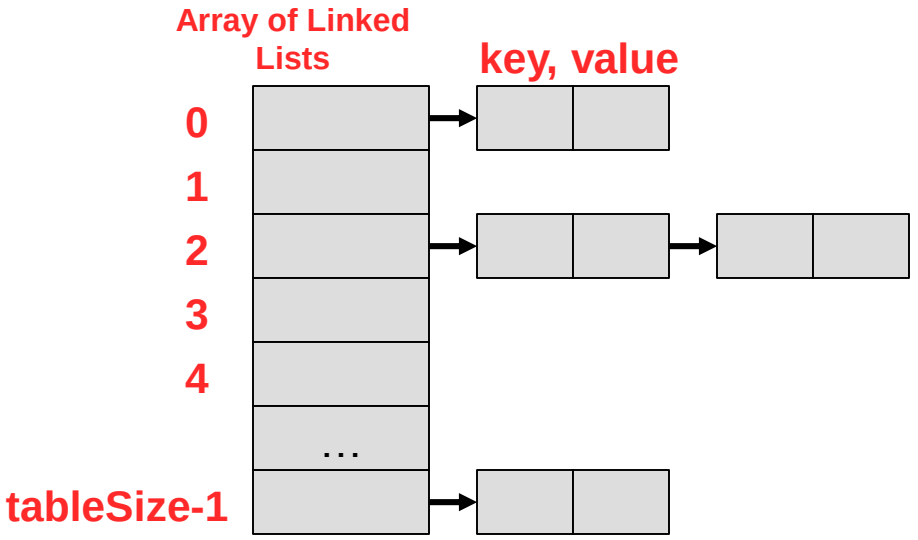
Resolving Collisions

- 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 before ***(by the birthday paradox)***
- Methods
  - Closed Addressing (e.g. buckets or **chaining**)
  - Open addressing (aka probing)
    - Linear Probing
    - Quadratic Probing
    - Double-hashing

Buckets/Chaining

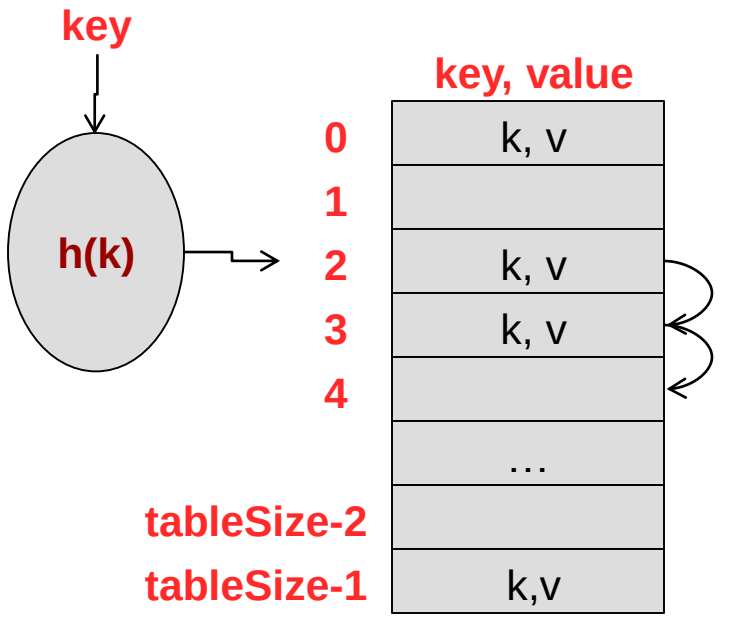
- Simply allow collisions to all occupy the location they hash to by making each entry in the table an ARRAY (bucket) or LINKED LIST (chain) of items/entries
  - Close Addressing => You will live in the location you hash to (it's just that there may be many places at that location)
- Buckets
  - How big should you make each array?
  - Too much wasted space
- Chaining
  - Each entry is a linked list





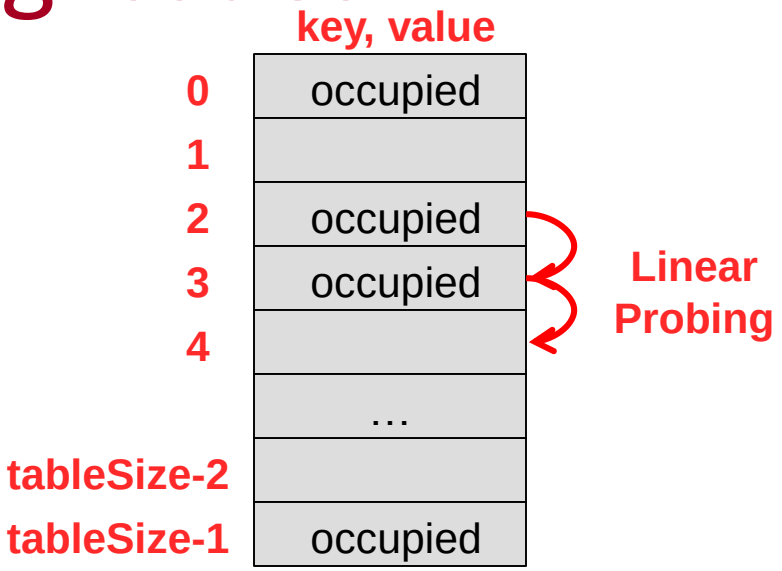
Open Addressing

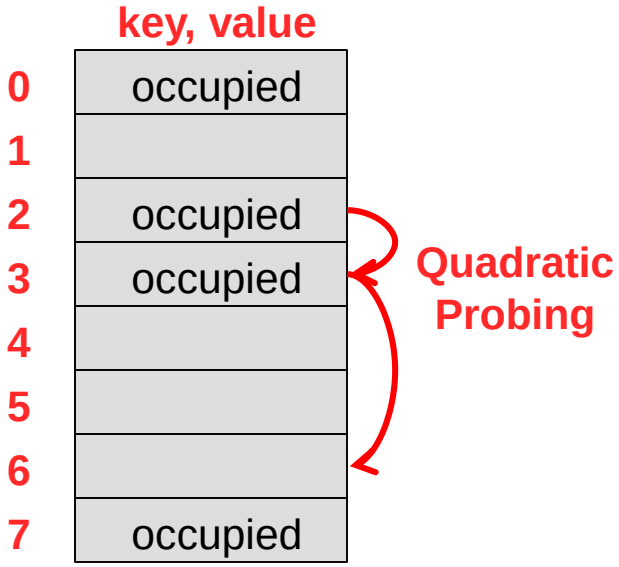
- Open addressing means an item with key, k, may not be located at h(k)
- If location 2 is occupied and a new item hashes to location 2, we need to find another location to store it.
- Let i be number of failed inserts
- Linear Probing
  - h(k,i) = (h(k)+i) mod m
  - Example: If h(k) occupied (i.e. collision) then check h(k)+1, h(k)+2, h(k)+3, …
- Quadratic Probing
  - h(k,i) = (h(k)+i^2) mod m
  - If h(k) occupied, then check $h(k)+1^2$, $h(k)+2^2$, $h(k)+3^2$, …



Linear Probing Issues

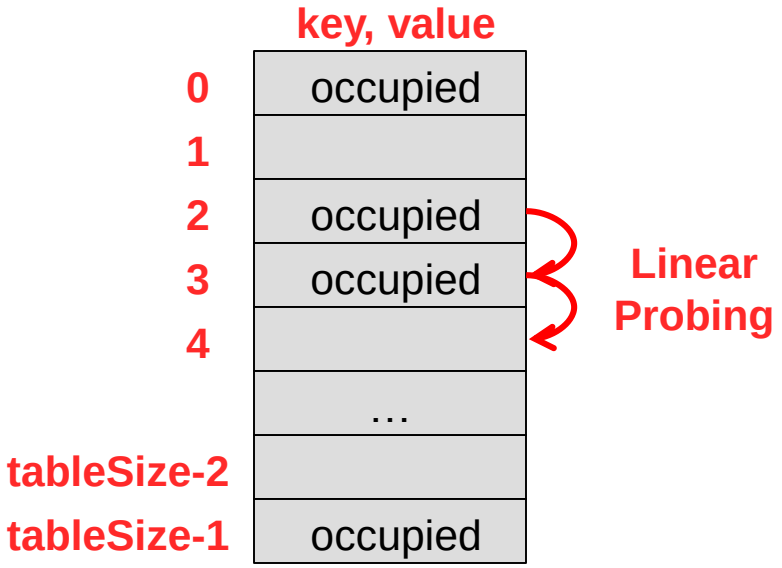
- If certain data patterns lead to many collisions, linear probing leads to clusters of occupied areas in the table called ***primary clustering***
- How would quadratic probing help fight primary clustering?
  - Quadratic probing tends to spread out data across the table by taking larger and larger steps until it finds an empty location

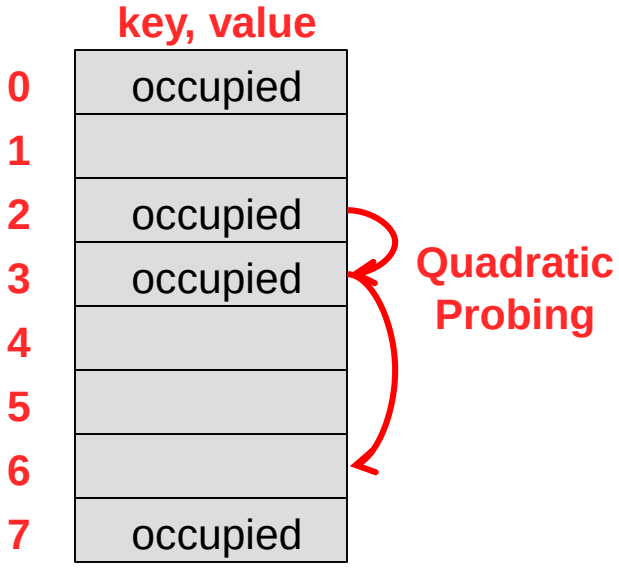




Find & Removal Considerations

- Given linear or quadratic clustering how would you find a given key, value pair
  - First hash it
  - If it is not at h(k), then move on to the next items in the linear or quadratic sequence of locations until
    - you find it or
    - an empty location or
    - search the whole table
- What if items get removed
  - Now the find algorithm might terminate too early
  - Mark a location as "removed"=unoccupied but part of a cluster





Practice

- Use the hash function h(k)=k%10 to find the contents of a hash table (m=10) after inserting keys 1, 11, 2, 21, 12, 31, 41 using linear probing

0	1	2	3	4	5	6	7	8	9

- Use the hash function h(k)=k%9 to find the contents of a hash table (m=9) after inserting keys 36, 27, 18, 9, 0 using quadratic probing

0	1	2	3	4	5	6	7	8

Double Hashing

- Define $h_1(k)$ to map keys to a table location
- But also define $h_2(k)$ to produce a linear probing step size
  - First look at $h_1(k)$
  - Then if it is occupied, look at $h_1(k) + h_2(k)$
  - Then if it is occupied, look at $h_1(k) + 2*h_2(k)$
  - Then if it is occupied, look at $h_1(k) + 3*h_2(k)$
- TableSize=13, h1(k) = k mod 13, and h2(k) = 5 – (k mod 5)
- What sequence would I probe if k = 31
  - h1(31) = ___, h2(31) = _____
  - Seq: _____

Double Hashing

- Define $h_1(k)$ to map keys to a table location
- But also define $h_2(k)$ to produce a linear probing step size
  - First look at $h_1(k)$
  - Then if it is occupied, look at $h_1(k) + h_2(k)$
  - Then if it is occupied, look at $h_1(k) + 2*h_2(k)$
  - Then if it is occupied, look at $h_1(k) + 3*h_2(k)$
- TableSize=13, h1(k) = k mod 13, and h2(k) = 5 – (k mod 5)
- What sequence would I probe if k = 31
  - h1(31) = 5, h2(31) = 5-(31 mod 5) = 4
  - 5, 9, 0, 4, 8, 12, 3, 7, 11, 2, 6, 10, 1

Hashing Efficiency

- Loading factor, α, defined as:
  - (n=number of items in the table) / m=tableSize => α = 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
  - What is the average length of a chain in the table (e.g. 10 total items in a hash table with table size of 5)?

- Best to keep the loading factor, α, below 1
  - Resize and rehash contents if load factor too large (using new hash function)

Rehashing for Open Addressing

- For probing (open-addressing), as α approaches 1 the expected number of probles/comparisons will get very large
  - Capped at the tableSize, **m** (i.e. O(**m**))
- Similar to resizing a vector, we can allocate a larger prime size table/array
  - Must rehash items to location in new table size.
  - Cannot just items to corresponding location in the new array
  - Example: h(k) = k % 13 != h'(k) = k %17 (e.g. k = 15)
  - For quadratic probing if table size m is prime, then first m/2 probes will go to unique locations
- General guideline for probing: keep α < 0.5

Hash Tables

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

Hash Tables

- 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

Summary

- Hash tables are LARGE arrays with a function that attempts to compute an index from the key
- Open addressing keeps uses a fixed amount of memory but insertion/find times can grow large as α approaches 1
- Closed addressing provides good insertion/find times as the number of entries increases at the cost of additional memory
- The functions should spread the possible keys evenly over the table [i.e. p(h(k) = x) = 1/m]

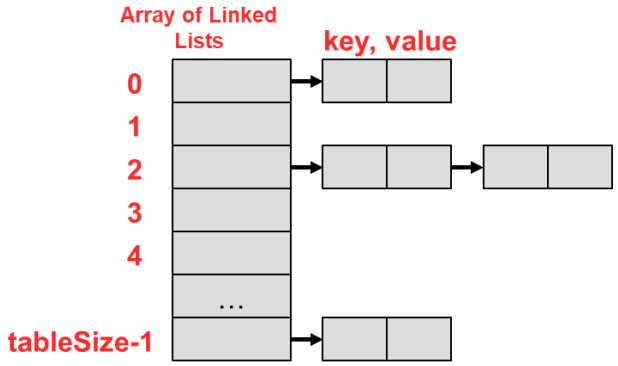
HASH FUNCTIONS

Recall: 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
  - Scatters data uniformly throughout the hash table
    - P(h(k) = x) = 1/**m** (i.e. **pseudorandom**)

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**



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 a 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
  - Suppose we hash your birth years
  - We'd have a lot of collisions around _____
- 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

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)**

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

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**

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

Taking a Step Back

- In most applications the UNIVERSE of possible keys >> M
  - Around 40,000 USC students each with 10-digit USC ID
  - **n** = 40000 and so we might choose **m** = 100,000 so $\alpha = 0.4$
  - But there are 10^{10} potential keys (10-digit USC ID) hashing to a table of size 100,000
  - That means at least $10^{10}/10^5$ could map to the same place no matter how "good" your hash function spreads data
  - What if an adversary fed those in to us and 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)

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) we would not expect to accidentally (or just in nature) produce a sequence of keys that leads to a lot of collisions

- **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 mod 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

Uses of Cryptographic Hash Functions

- Hash functions can be used for purposes other than hash tables
- 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
- We no longer need h(m) to be in the range of tableSize since we don't have a table anymore
  - The hash code is all we care about now
  - 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)

Another Example: Passwords

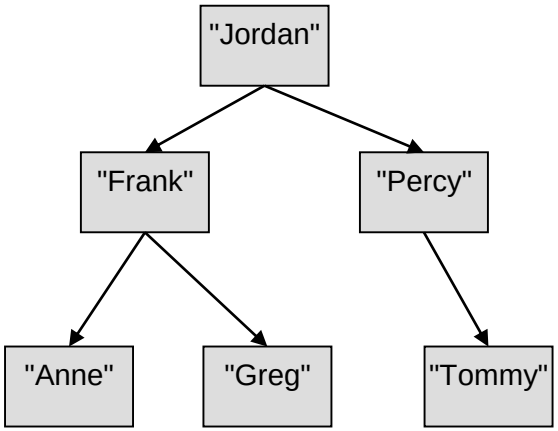
- Should a company just store passwords plain text?
  - No
- We could encrypt the passwords but here's an alternative
- Just don't store the passwords!
- Instead, store the hash codes of the passwords.
  - What's the implication?
  - Some alternative password might just hash to the same location but 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!

An imperfect set…

BLOOM FILTERS

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
  - (Balanced) Binary Search Tree
    - O(log n) insert/remove/contains
  - Hash table
    - O(1) insert/remove/contains



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

Bloom Filter Idea

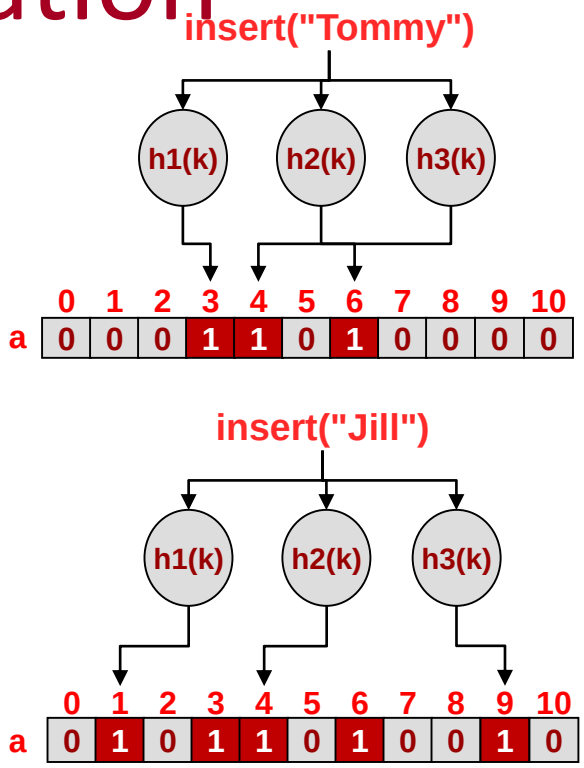
- A Bloom filter is a set such that "contains()" will *quickly* answer…
  - "No" correctly (i.e. if the key is not present)
  - "Yes" with a chance of being incorrect (i.e. the key may not be present but it might still say "yes")
- Why would we want this?

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

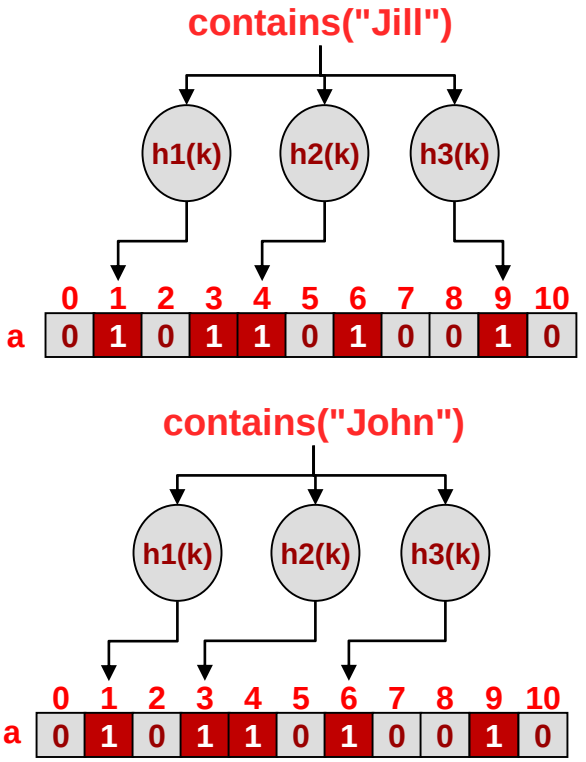
Bloom Filter Explanation

- 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



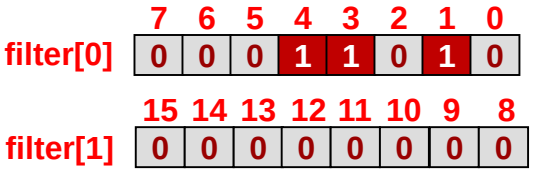
Bloom Filter Explanation

- A Bloom filter is…
  - A hash table of individual bits (Booleans: T/F)
  - A set of hash functions, $\{h_1(k), h_2(k), \ldots h_s(k)\}$
- Contains()
  - 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



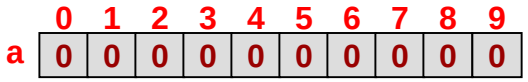
Implementation Details

- Bloom filter's 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) ) )`



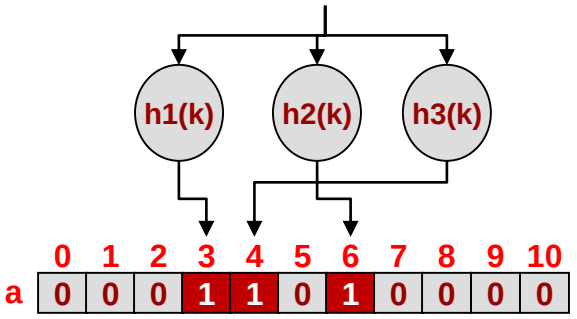
Practice

- Trace a Bloom Filter on the following operations:
  - insert(0), insert(1), insert(2), insert(8), contains(2), contains(3), contains(4), contains(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).



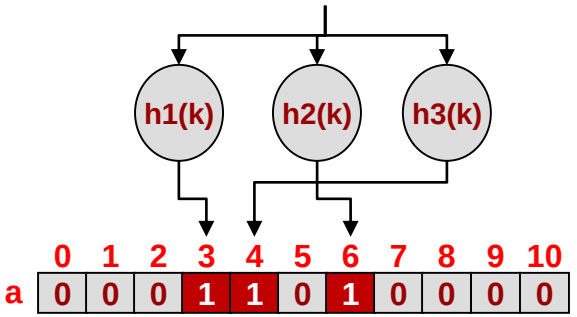
Probability of False Positives

- 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-entries in the table have not selected location x
    - _____
  - Probability that a location x HAS been chosen by the previous n entries
    - _____
  - Math factoid: For small y, $e^y = 1+y$ (substitute y = -1/m)
    - _____
  - Probability that all of the j hash functions find a location True once the table has n entries
    - _____



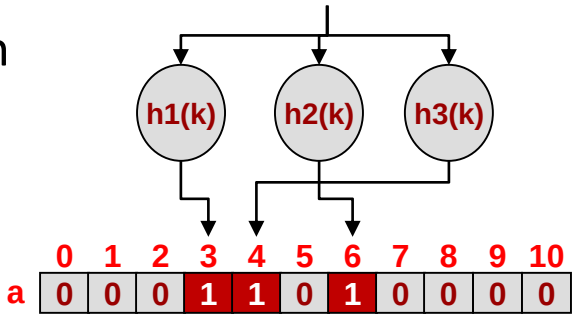
Probability of False Positives

- 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-entries in the table have not selected location x
    - $[1 - 1/m]^{nj}$
  - Probability that a location x HAS been chosen by the previous n entries
    - $1 - [1 - 1/m]^{nj}$
  - Math factoid: For small y, $e^{y} = 1+y$ (substitute y = -1/m)
    - $1 - e^{-nj/m}$
  - Probability that all of the j hash functions find a location True once the table has n entries
    - $(1 - e^{-nj/m})^{j}$



Probability of False Positives

- Probability that all of the j hash functions find a location True once the table has s entries
  - $(1 - e^{-nj/m})^j$
- Define $\alpha = n/m$ = 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 $0 \leq \alpha \leq 1$



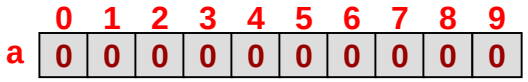
Sizing Analysis

- 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/(9.2) would yield j ≈ 7 hash functions

SOLUTIONS

Practice

- Trace a Bloom Filter on the following operations:
  - insert(0), insert(1), insert(2), insert(8), contains(2), contains(3), contains(4), contains(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
Contains(2)	8	5	3	Yes
Contains(3)	5	7	8	Yes
Contains(4)	2	9	3	No
Contains(9)	7	9	8	No