CSCI 104
Hash Tables & Functions

Mark Redekopp
David Kempe
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 = $O(\_\_\_\_\_\_\_\_)$

```
Map::find("Greg")

"Jordan" Student object

"Frank" Student object

"Greg" Student object

"Percy" Student object

"Anne" Student object

"Tommy" Student object
```

```
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
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. $O(1)$)
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 $[0..\text{tableSize}-1]$ (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
Hash Functions First Look

• Define \( N \) = # of entries stored, \( M \) = Table/Array Size

• A hash function must be able to
  – convert the key data type to an integer
  – That integer must be in the range \([0 \text{ to } M-1]\)
    • Keeping \( h(k) \) in the range of the tableSize (\( M \))
    • Fairly easy method: Use modulo arithmetic (i.e. \( h(k) \mod M \))

• Usually converting key data type to an integer is a user-provided function
  – 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
  – Example function: \( h("hello") = 104 + 101 + 108 + 108 + 111 = 532 \mod M \)
Table Size

• Ideally...
  – Enough entries for all possible keys
  – Example: 3-letter airport codes: LAX, BUR, JFK would require how many table entries?
    • \(26^3 = 17576\)
    • Not all 3-letter codes correspond to airports
  – May be impractical as we will often only use a (small) subset of keys in a real application

• Realistically...
  – 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
Hash Function Desirables

• 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
  – Is easy and fast to compute
  – Scatters data uniformly throughout the hash table
    • $P( h(k) = x ) = 1/M$
Resolving Collisions

• Example:
  – A hash table where keys are phone numbers: (XXX) YYY-ZZZZ
  – Obviously we can't have a table with $10^{10}$ entries
  – Should we define $h(k)$ as the upper 3 or 4 digits: XXX or XXXY
    • Meaning a table of 1000 or 10,000 entries
  – Define $h(k)$ as the lowest 4-digits of the phone number: ZZZZ
    • Meaning a table with 10,000 entries: 0000-9999
  – Now 213-740-4321 and 323-681-4321 both map to location 4321 in the table

• Collisions are hard to avoid so we have to find a way to deal with them

• Methods
  – Open addressing (probing)
    • Linear, quadratic, double-hashing
  – Buckets/Chaining (Closed Addressing)
Open Addressing

• Open addressing means an item with key, k, may not be located at h(k)
• Assume, location 2 is occupied with another item
• If a new item hashes to location 2, we need to find another location to store it
• Linear Probing
  – Just move on to location h(k)+1, h(k)+2, h(k)+3,…
• Quadratic Probing
  – Check location 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

• 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
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 $h_1(k) + 2h_2(k)$
  – Then if it is occupied, look at $h_1(k) + 3h_2(k)$
• TableSize=13, $h_1(k) = k \mod 13$, and $h_2(k) = 5 - (k \mod 5)$
• What sequence would I probe if $k = 31$
  – $h_1(31) = ____$, $h_2(31) = ______________$
  – 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 $h_1(k) + 2h_2(k)$
  – Then if it is occupied, look at $h_1(k) + 3h_2(k)$
• TableSize=13, $h_1(k) = k \mod 13$, and $h_2(k) = 5 – (k \mod 5)$
• What sequence would I probe if $k = 31$
  – $h_1(31) = 5$, $h_2(31) = 5 – (31 \mod 5) = 4$
  – 5, 9, 0, 4, 8, 12, 3, 7, 11, 2, 6, 10, 1
Buckets/Chaining

• Rather than searching for a free entry, make each entry in the table an ARRAY (bucket) or LINKED LIST (chain) of items/entries

• Buckets
  – How big should you make each array?
  – Too much wasted space

• Chaining
  – Each entry is a linked List
Hash Tables

• Suboperations
  – Compute $h(k)$ should be ______
  – Array access of $\text{table}[h(k)] = _____$

• In a hash table, what is the expected efficiency of each operation
  – Find = ______
  – Insert = ______
  – Remove = ______
Hash Tables

• **Suboperations**
  – Compute \( h(k) \) should be \( O(1) \)
  – Array access of \( \text{table}[h(k)] = O(1) \)

• **In a hash table, what is the expected efficiency of each operation**
  – Find = \( O(1) \)
  – Insert = \( O(1) \)
  – Remove = \( O(1) \)
Hashing Efficiency

• Loading factor, $\alpha$, defined as:
  – $(N=\text{number of items in the table}) / M=\text{tableSize} \Rightarrow \alpha = N / M$
  – Really it is just the fraction of locations currently occupied

• For chaining, $\alpha$, can be greater than 1
  – $\alpha = (\text{number of items in the table}) / \text{tableSize}$
  – $(\text{number of items in the table})$ can be greater than tableSize

• What is the average length of a chain in the table?

• # of operations to search
  – Unsuccessful search: _________________

  – Successful search: _________________
Hashing Efficiency

• Loading factor, $\alpha$, defined as:
  - \(\frac{\text{number of items in the table}}{\text{tableSize}}\)
  - Really it is just the % of locations currently occupied For chaining, $\alpha$, can be greater than 1
  - $\alpha = \frac{\text{number of items in the table}}{\text{tableSize}}$
  - (number of items in the table) can be greater than tableSize

• What is the average length of a chain in the table? $\alpha$
  - 10 total items in a hashTable of size 5 => expected chain = 2

• # of operations to search
  - Unsuccessful search: 1 + $\alpha$
  - Successful search: 1 + $\alpha/2$
Rehashing for Open Addressing

• For open addressing/probing time also depends on $\alpha$
• As $\alpha$ approaches 1, expected comparisons will get very large
  – Capped at the tableSize (i.e. $O(n)$)
• Using a dynamic array (vector) we can simply allocate a bigger table
  – Don't just double it because we want the tableSize to be prime
• Can we just copy items over to the corresponding location in the new table?
  – No because the hash function usually depends on tableSize so we need to re-hash each entry to its correct location
  – $h(k) = k \mod 13 \neq h'(k) = k \mod 17$ (e.g. $k = 15$)
• General guideline is to keep $\alpha < 1/2$)
Unordered Maps

- A hash table implements a map ADT
  - Add(key, value)
  - Remove(key)
  - Lookup/Find(key)
    - Returns value
- Given a key, the hash function is interested in producing unique, evenly spread integers, NOT maintaining ordering of the keys
  - That is, just because $k_1 < k_2$, doesn't mean $h(k_1) < h(k_2)$
  - Thus the table holds values in arbitrary order unlike the BST
  - If you iterate through a hash table (and sometimes that is even a challenge), you likely won't see key,value pairs in order
- A hash table implements an **UNORDERED MAP**
- A Binary Search Tree implements an **ORDERED 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)
HASH FUNCTIONS
Pigeon Hole Principle

• Recall for hash tables we let...
  – \( n \) = # of entries (i.e. keys)
  – \( m \) = size of the hash table

• Assuming chaining, 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 \( n < m \)
How Soon Would Collisions Occur

• Even if $n < m$, how soon would we expect collisions to occur?

• If we had an adversary...
  – Then maybe after the first two insertions
  – 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
Hash Functions

• A "perfect hash function" should map each given key to a unique location in the table
  – Perfect hash functions are not practically attainable

• A "good" hash function
  – Is easy and fast to compute
  – Scatters data evenly throughout the hash table
  – Scatters random keys uniformly
    • $M=3$, keys=$[0..4]$, $h(k) = k \mod M$ does not spread data randomly
  – Scatters clustered keys uniformly

• Rules of thumb
  – The hash function should examine the entire search key, not just a few digits or a portion of the key
  – If modulo hashing is used, the base should be prime
Modulo Arithmetic

• 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
  – 17 dec. => 10001 bin. and thus 17 mod \( 2^2 = 01 \) bin.
Relatively Prime Numbers

• Two numbers are relatively prime if they do not share any factors other than 1
• If \( a \) and \( b \) (e.g. 9 and 7) are relatively prime, then their first common multiple is?
  – \( a \times b \)
• If \( m \) (i.e. tableSize) is a prime number (not 2 or 5) what is the first common multiple of \( 10^d \) and \( m \)?
  – \( m \times 10^d \)
  – For \( m = 11 \) and \( d=2 \), common multiples would be 1100, 2200, 3300, etc.
Why Prime Table Size

• 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
  
  \[ h(k) = (k \mod m) \quad \text{[where } m \text{ is prime]}\]
  
  – i.e. what is the chance \( (k \mod 10^d) = (k \mod p) \)
Why Prime Table Size

- Notice if these numbers map to the same location when tableSize was $m=10^d$ then we know:
  - $k \mod 10^d = k' \mod 10^d = r$ (i.e the same remainder for both)
- What can we say about the difference between any two keys that map to the same location? (e.g. $k' - k$)
  - They differ by some multiple of $m=10^d$ (i.e. the table size)
    - $2099-99 = 2000 = 20*10^2$, $2099-199 = 1900 = 19*10^d$, $199-99 = 100 = 1*10^2$
  - To repeat: if $k$ and $k'$ collide, $k'-k$ will yield some multiple of $10^d$
- Proof: Recall two numbers, ($k$ and $k'$) will hash to the same location if
  - $k \mod m = k' \mod m = r$ (i.e the same remainder for both)
  - If that's true then we can write:
    - $k = qm + r$ and $k' = q'm + r$ for some $q$ and $q'$ where $q \neq q'$
  - And thus $k'-k = (q'-q)*m$....the difference is some multiple of $m$
  - So for all these numbers $k'-k$ yielded some multiple of $10^d$
    - Put another way the stepsize = $10^d$
Modulo Hashing

• So to map to the same place in the m=10\(^d\) hash table, k-k' would have to be a multiple of 10\(^d\)

• BUT…if k and k' were to map to the same place using our new tableSize (m=some prime number, p) then k-k' (which we know is a power of 10\(^d\) because they collided when m = 10\(^d\)) would ALSO have to be divisible by p
  – So 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
Another Alternative

• We just said a "good" hashing function provides a uniform probability distribution of hashing to any location
• So given a key, k, can you think of another hash function, h(k) such that k has a uniform probability of ending up in any location?
• How about h(k) = rand() % m
• Pros:
  – Fast
  – Even clustered keys get spread randomly over the hash table
• Cons:
  – How do you do a look up?
  – h(k) is not dependent on k...it is just based on the next random number

Universal Hash Functions

• One alternative to the above is not to make the hash-function random but to make the **selection** of the hash function random
• Anytime we fix on one hash function, an adversary can easily create keys that map to the same place
  – I.e. they could study the function and try to come up with an "bad" sequence
• Instead, suppose...
  – We have a family of "good" hash functions (h1, h2, h3...) where each hash function is independent of the others and has the probability of mapping two keys to the same place = 1/m
  – Certainly an adversary could design a "bad" sequence of keys for each of these "good" hash functions.
  – But now, when we want to create a hash table we simply randomly select which hash function to use at run-time
  – The adversary wouldn't know which one we picked and thus couldn't know which of his bad sequences to feed us...
  – We've essentially made the odds of a "bad" sequence improbable
  – And wasn't that the intention of a hash function anyway?
  – These families of hash functions are known as **universal hash functions**
Taking a Step Back

• How can any deterministic hash function appear to spread items randomly over the hash table?

• The pigeon-hole principle says that given N items for M slots where N>M, at least one hole will have N/M items
  – 1000 items for 100 slots...one slot is guaranteed to have 10 or more

• In reality the UNIVERSE of possible keys >> M
  – E.g. Possible universe of USC ID's (10^{10} options) mapping to a table on the order of 100,000
  – At least 10^{10}/10^{5} could map to the same place no matter how "good" your hash function...it's deterministic
  – What if an adversary fed those in to us...
Inverse 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**

• Key insight: What's hard to accomplish when you actually try is even harder to accomplish when you do not try
One-Way Hash Functions

- Recall: What's hard to accomplish when you actually try is even harder to accomplish when you do not try

- So if we find a hash function where it would be hard to find a key that maps to a given location, i, when we are trying to be an adversary...

- ...then under normal circumstances we would not expect to accidentally (or just in nature) produce a sequence of keys that leads to a lot of collisions

- Main Point: It's obviously hard to predict randomness, but computers aren't random. However, if we can find a one-way hash function, then even though our adversary knows we're not being random, he'll still have a hard time
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!
Cryptographic Hash Functions

• A cryptographic hash function should have the qualities
  • Given d, it is hard to find an input, M, such that h(M) = d
    – (i.e. One-way: given a hash code, hard to find an input that generates it)
  • Collision Resistant
    – It should be hard to generate M1 and M2 such that h(M1) = h(M2)
    – Remember what the birthday paradox says?
      • An n-bit code has $2^n$ options...if we generate random numbers when would we
        expect a duplicate?
  • Given an M1 and h(M1) = d, it should be hard to find an M2 such that
    h(M2) = d
    – Given one input and its hash code it should be hard to tweak M1 to make an
      M2 that yields an identical code
• MD5 and SHA2 are examples of these kind of functions

Avalanche Effect

• A desirable property for a cryptographic hash function is that it exhibit the "avalanche effect"

• Avalanche effect says...
  – Formally: A change of 1-bit in the input should produce on average half of the output-bits to change (i.e. probability of an output bit change due to that input change should be ½)
  – Informally: A small change in input should yield a big change in output

• Along that same line, think about which bit-wise logic operation would be best suited for encryption
  – Ex. If I told you which logic operation I used and you saw the output, would it help you decipher the input that generated that?

• XOR's are often used in cryptographic hash function along with bit shifting and rotations
Example

- Encrypt: "ONE MORE HW" to produce a 5-bit hash by XOR-ing the codes

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>00000</td>
<td>F</td>
<td>00101</td>
<td>K</td>
<td>01010</td>
<td>P</td>
<td>01111</td>
<td>U</td>
<td>10100</td>
</tr>
<tr>
<td>B</td>
<td>00001</td>
<td>G</td>
<td>00110</td>
<td>L</td>
<td>01011</td>
<td>Q</td>
<td>10000</td>
<td>V</td>
<td>10101</td>
</tr>
<tr>
<td>C</td>
<td>00010</td>
<td>H</td>
<td>00111</td>
<td>M</td>
<td>01100</td>
<td>R</td>
<td>10001</td>
<td>W</td>
<td>10110</td>
</tr>
<tr>
<td>D</td>
<td>00011</td>
<td>I</td>
<td>01000</td>
<td>N</td>
<td>01101</td>
<td>S</td>
<td>10010</td>
<td>X</td>
<td>10111</td>
</tr>
<tr>
<td>E</td>
<td>00100</td>
<td>J</td>
<td>01001</td>
<td>O</td>
<td>01110</td>
<td>T</td>
<td>10011</td>
<td>Y</td>
<td>11000</td>
</tr>
<tr>
<td>Sp</td>
<td>11010</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Summary

• Hash tables are LARGE arrays with a function that attempts to compute an index from the key
• In the general case, **chaining** is the best collision resolution approach
• The functions should spread the possible keys evenly over the table
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?
- 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), ... h_s(k)\}
- Insert()
  - Apply each h_i(k) to the key
  - Set a[h_i(k)] = True
- 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) ) )


```
filter[0]  7 6 5 4 3 2 1 0
    0 0 0 1 1 0 1 0

filter[1]  15 14 13 12 11 10 9 8
    0 0 0 0 0 0 0 0
```
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 \( s \)-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
    • 

\[ \begin{array}{cccccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\
\end{array} \]
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
    • \( P(h_i(k) = x) = \frac{1}{m} \)
    • \( P(h_i(k) \neq x) = [1 – \frac{1}{m}] \)
  – Probability that all \( j \) hash functions don't select a location
    • \([1 – \frac{1}{m}]^j\)
  – Probability that all \( s \)-entries in the table have not selected location \( x \)
    • \([1 – \frac{1}{m}]^s\)
  – Probability that a location \( x \) HAS been chosen by the previous \( n \) entries
    • \( 1 – [1 – \frac{1}{m}]^n \)
  – Math factoid: For small \( y \), \( e^y = 1+y \) (substitute \( y = -\frac{1}{m} \))
    • \( 1 – e^{-nj/m} \)
  – 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 s entries: \( 2^{-\ln(2)/\alpha} \)

  – 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 \( 2^{-m \cdot \ln(2)/n} < p \)
    • Flip to \( 2^{m \cdot \ln(2)/n} \geq 1/p \)
    • Take log of both sides and solve for \( m \)
      • \( m \geq \left[ n \cdot \ln(1/p) \right] / \ln(2) \approx 2n \cdot \ln(1/p) \) because \( \ln(2)^2 \approx 0.48 \approx \frac{1}{2} \)
  – So for \( p = .001 \) we would need a table of \( m = 14 \cdot n \) since \( \ln(1000) \approx 7 \)
    • For 100 entries, we'd need 1400 bits in our Bloom filter
  – For \( p = .01 \) (1% false positives) need \( m = 9.2 \cdot n \) (9.2 bits per key)
  – Recall: Optimal # of hash functions, \( j = \ln(2) / \alpha \)
    • So for \( p = .01 \) and \( \alpha = 1/(9.2) \) would yield \( j \approx 7 \) hash functions