Data Structures

Hash Tables and Applications

Design and Analysis of Algorithms I

Hash Table: Supported Operations

Purpose : maintain a (possibly evolving) set of stuff.
(transactions, people + associated data, IP addresses, etc.)

Insert : add new record

Delete : delete existing record

Lookup : check for a particular record
(a "dictionary")

Using a "key"

AMAZING GUARANTEE
All operations in O(1) time ! *

* 1. properly implemented 2. non-pathological data

Application: De-Duplication

Given : a "stream" of objects.

- Linear scan through a huge file
- Or, objects arriving in real time

Goal : remove duplicates (i.e., keep track of unique objects)

-e.g., report unique visitors to web site

- avoid duplicates in search results

Solution : when new object x arrives

- lookup x in hash table H
- if not found, Insert x into H

Application: The 2-SUM Problem

Input : unsorted array A of n integers. Target sum t.

Goal : determine whether or not there are two numbers x,y in A with

x + y = t

Naïve Solution : $\theta(n^2)$ time via exhaustive search

Better : 1.) sort A ($\theta(n \log n)$ time)

2.) for each x in A, look for t-x in A via binary search

$\theta(n \log n)$

$\theta(n)$ *time*

Amazing : 1.) insert elements of A into hash table H

2.) for each x in A, Lookup t-x

$\theta(n)$ *time*

Tim Roughgarden

Further Immediate Applications

- Historical application : symbol tables in compilers
- Blocking network traffic
- Search algorithms (e.g., game tree exploration)
    - Use hash table to avoid exploring any configuration (e.g., arrangement of chess pieces) more than once
- etc.

Data Structures

Hash Tables: Some Implementation Details

Design and Analysis of Algorithms I

Hash Table: Supported Operations

Purpose : maintain a (possibly evolving) set of stuff.
(transactions, people + associated data, IP addresses, etc.)

Insert : add new record

Delete : delete existing record

Lookup : check for a particular record
(a "dictionary")

Using a "key"

AMAZING GUARANTEE
All operations in O(1) time ! *

* 1. properly implemented 2. non-pathological data

High-Level Idea

Setup : universe U [e.g., all IP addresses, all names, all chessboard configurations, etc.]
[generally, REALLY BIG]

Goal : want to maintain evolving set $S \subseteq U$
[generally, of reasonable size]

Solution : 1.) pick n = # of "buckets" with
(for simplicity assume |S| doesn't vary much)
2.) choose a hash function $h : U \to \{0, 1, 2, ..., n-1\}$
3.) use array A of length n, store x in A[h(x)]

Naïve Solutions

1. Array-based solution
   [indexed by u]
   - O(1) operations but $\theta(|U|)$ space
2. List –based solution
   - $\theta(|S|)$ space but $\theta(|S|)$ Lookup

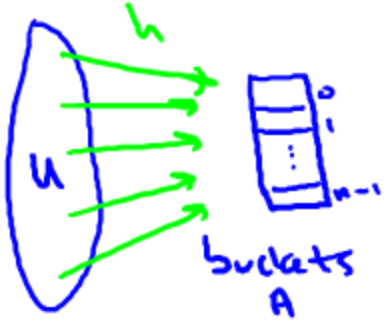
Tim Roughgarden

Consider n people with random birthdays (i.e., with each day of the year equally likely). How large does n need to be before there is at least a 50% chance that two people have the same birthday?

- 23 — 50 %
- 57 — 99 %
- 184 — 99.99….%
- 367 — 100%

BIRTHDAY "PARADOX"

Resolving Collisions



Collision: distinct $x, y \in U$ such that $h(x) = h(y)$

Solution # 1 : (separate) chaining

-keep linked list in each bucket

- given a key/object x, perform Insert/Delete/Lookup in the list in A[h(x)]

Linked list for x

Bucket for x

Solution #2 : open addressing. (only one object per bucket)

-Hash function now specifies probe sequence $h_1(x), h_2(x), ..$

(keep trying till find open slot)

Use 2 hash functions

- Examples : linear probing (look consecutively), double hashing

What Makes a Good Hash Function?

Note : in hash table with chaining, Insert is $\theta(1)$
$\theta(list\ length)$ for Insert/Delete.

Insert new object x at front of list in A[h(x)]

Equal-length lists

could be anywhere from m/n to m for m objects

All objects in same bucket

Point : performance depends on the choice of hash function!
(analogous situation with open addressing)

Properties of a "Good" Hash function

1. Should lead to good performance => i.e., should "spread data out" (gold standard – completely random hashing)
2. Should be easy to store/ very fast to evaluate.

Bad Hash Functions

Example : keys = phone numbers (10-digits). $|u| = 10^{10}$

-Terrible hash function : h(x) = 1^{st} 3 digits of x choose $n = 10^3$

(i.e., area code)

- mediocre hash function : h(x) = last 3 digits of x

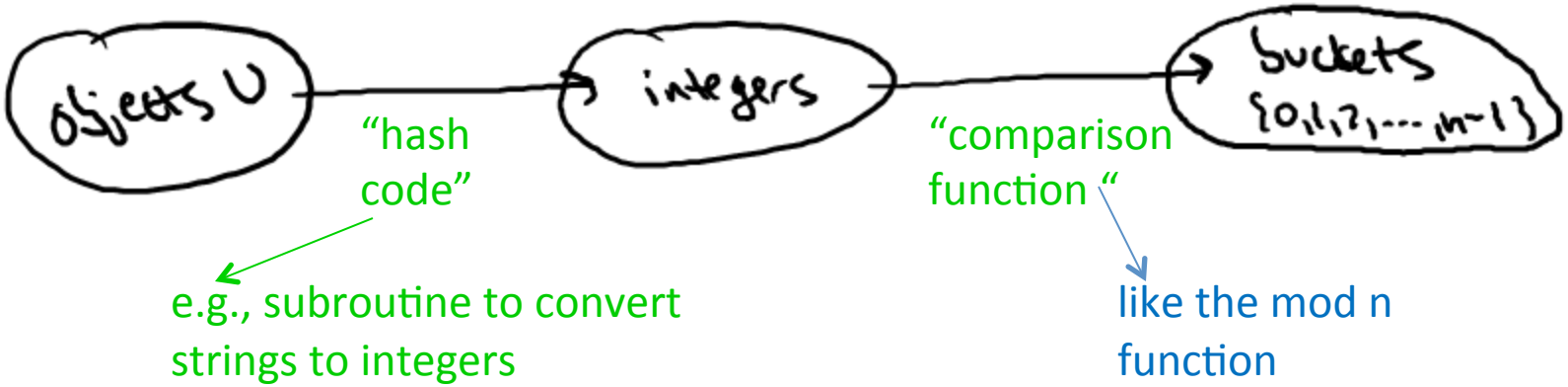
[still vulnerable to patterns in last 3 digits]

Example : keys = memory locations. (will be multiples of a power of 2)

-Bad hash function : h(x) = x mod 1000 (again $n = 10^3$)

=> All odd buckets guaranteed to be empty.

Quick-and-Dirty Hash Functions



How to choose n = # of buckets

1. Choose n to be a prime (within constant factor of # of objects in table)
2. Not too close to a power of 2
3. Not too close to a power of 10

Data Structures

Hash Tables: Some Implementation Details

Design and Analysis of Algorithms I

Hash Table: Supported Operations

Purpose : maintain a (possibly evolving) set of stuff.
(transactions, people + associated data, IP addresses, etc.)

Insert : add new record

Delete : delete existing record

Lookup : check for a particular record
(a "dictionary")

Using a "key"

AMAZING GUARANTEE
All operations in O(1) time ! *

* 1. properly implemented 2. non-pathological data

High-Level Idea

Setup : universe U [e.g., all IP addresses, all names, all chessboard configurations, etc.]
[generally, REALLY BIG]

Goal : want to maintain evolving set $S \subseteq U$
[generally, of reasonable size]

Solution : 1.) pick n = # of "buckets" with
(for simplicity assume |S| doesn't vary much)
2.) choose a hash function $h : U \to \{0, 1, 2, ..., n-1\}$
3.) use array A of length n, store x in A[h(x)]

Naïve Solutions

1. Array-based solution
   [indexed by u]
   - O(1) operations but $\theta(|U|)$ space
2. List –based solution
   - $\theta(|S|)$ space but $\theta(|S|)$ Lookup

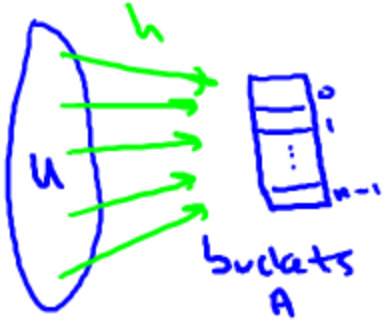
Tim Roughgarden

Consider n people with random birthdays (i.e., with each day of the year equally likely). How large does n need to be before there is at least a 50% chance that two people have the same birthday?

- 23 — 50 %
- 57 — 99 %
- 184 — 99.99….%
- 367 — 100%

BIRTHDAY "PARADOX"

Resolving Collisions



Collision: distinct $x, y \in U$ such that $h(x) = h(y)$

Solution # 1 : (separate) chaining

-keep linked list in each bucket

- given a key/object x, perform Insert/Delete/Lookup in the list in A[h(x)]

Linked list for x

Bucket for x

Solution #2 : open addressing. (only one object per bucket)

-Hash function now specifies probe sequence $h_1(x), h_2(x), ..$

(keep trying till find open slot)

Use 2 hash functions

- Examples : linear probing (look consecutively), double hashing

What Makes a Good Hash Function?

Note : in hash table with chaining, Insert is $\theta(1)$
$\theta(list\ length)$ for Insert/Delete.

Insert new object x at front of list in A[h(x)]

Equal-length lists

could be anywhere from m/n to m for m objects

All objects in same bucket

Point : performance depends on the choice of hash function!
(analogous situation with open addressing)

Properties of a "Good" Hash function

1. Should lead to good performance => i.e., should "spread data out" (gold standard – completely random hashing)
2. Should be easy to store/ very fast to evaluate.

Bad Hash Functions

Example : keys = phone numbers (10-digits). $|u| = 10^{10}$

-Terrible hash function : h(x) = 1^{st} 3 digits of x choose $n = 10^3$

(i.e., area code)

- mediocre hash function : h(x) = last 3 digits of x

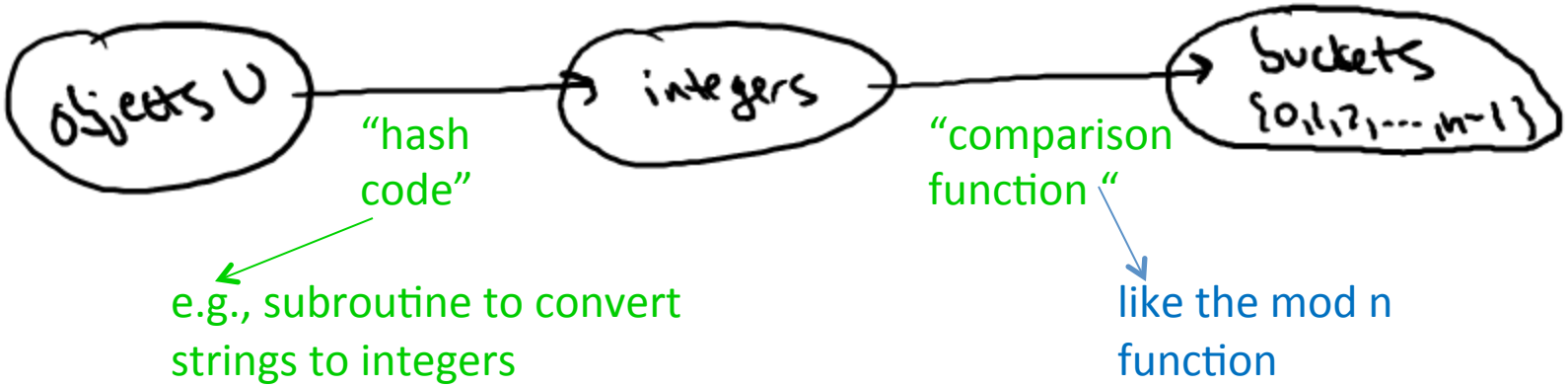
[still vulnerable to patterns in last 3 digits]

Example : keys = memory locations. (will be multiples of a power of 2)

-Bad hash function : h(x) = x mod 1000 (again $n = 10^3$)

=> All odd buckets guaranteed to be empty.

Quick-and-Dirty Hash Functions



How to choose n = # of buckets

1. Choose n to be a prime (within constant factor of # of objects in table)
2. Not too close to a power of 2
3. Not too close to a power of 10

Design and Analysis of Algorithms I

Data Structures

Universal Hash Functions: Motivation

Hash Table: Supported Operations

Purpose : maintain a (possibly evolving) set of stuff.
(transactions, people + associated data, IP addresses, etc.)

Insert : add new record

Delete : delete existing record
easier/more common with chaining than open addressing

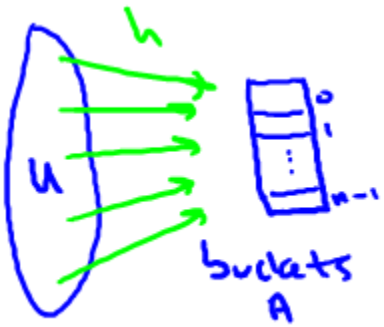
Lookup : check for a particular record
(a "dictionary")

Using a "key"

AMAZING GUARANTEE
All operations in O(1) time ! *

* 1. properly implemented 2. non-pathological data

Resolving Collisions



Collision : distinct x,y in U such that h(x) = h(y).

Solution#1: (separate) chaining.

-- keep linked list in each bucket

-- given a key/object x, perform Insert/Delete/Lookup in the list in A[h(x)]

bucket for x

linked list for x

Solution#2 : open addressing. (only one object per bucket)

-- hash function now specifies probe sequence h1(x), h2(x), …
(keep trying till find open slot)

use 2 hash functions

-- examples : linear probing (look consecutively), double hashing

The Load of a Hash Table

Definition : the load factor of a hash table is

$$\alpha := \frac{\text{\# of objects in hash table}}{\text{\# of buckets of hash table}}$$

Which hash table implementation strategy is feasible for load factors larger than 1?

- ○ Both chaining and open addressing
- ○ Neither chaining nor open addressing
- ○ Only chaining
- ○ Only open addressing

The Load of a Hash Table

Definition : the load factor of a hash table is

$$\alpha := \frac{\text{\# of objects in hash table}}{\text{\# of buckets of hash table}}$$

Note : 1.) α = O(1) is necessary condition for operations to run in constant time.
2.) with open addressing, need $\alpha << 1$.

Upshot#1 : good HT performance, need to control load.

Pathological Data Sets

Upshot#2 : for good HT performance, need a good hash function.

Ideal : user super-clever hash function guaranteed
to spread every data set out evenly.

Problem : DOES NOT EXIST! (for every hash function, there is a pathological data set)

Reason : fix a hash function h : U -> {0,1,2,…,n-1}

$\Rightarrow$a la Pigeonhole Principle, there exist bucket i such that at least |u|/n elements of U hash to I under h.

$\Rightarrow$ if data set drawn only from these, everything collides !

Pathological Data in the Real World

Preference : Crosby and Wallach, USENIX 2003.

Main Point : can paralyze several real-world systems (e.g., network intrusion detection) by exploiting badly designed hash functions.

-- open source

-- overly simplistic hash function

(easy to reverse engineer a pathological data set)

Solutions

1. Use a cryptographic hash function (e.g., SHA-2)
   -- infeasible to reverse engineer a pathological data set

2. Use randomization. ⟵In next 2 videos
   -- design a family H of hash functions such that for all data sets S, "almost all" functions $h \in H$ spread S out "pretty evenly".
   (compare to QuickSort guarantee)

Overview of Universal Hashing

Next : details on randomized solution (in 3 parts).

Part 1 : proposed definition of a "good random hash function".
("universal family of hash functions")

Part 3 : concrete example of simple + practical such functions

Part 4 : justifications of definition : "good functions" lead to "good performance"

Design and Analysis of Algorithms I

Data Structures

Universal Hash Functions: Definition and Example

Overview of Universal Hashing

Next : details on randomized solution (in 3 parts).

Part 1 : proposed definition of a "good random hash function".
("universal family of hash functions")

Part 3 : concrete example of simple + practical such functions

Part 4 : justifications of definition : "good functions" lead to "good performance"

Universal Hash Functions

Definition : Let H be a set of hash functions from U to {0,1,2,…,n-1}

H is universal if and only if :

for all x,y in U (with $x \neq y$)

$$Pr_{h \in H}[x, y \ collide] \leq \frac{1}{n}$$

ie., h(x) = h(y)

(n = # of buckets)

When h is chosen uniformly at random from H.

(i.e., collision probability as small as with “gold standard” of perfectly random hashing

Consider a hash function family H, where each hash function of H maps elements from a universe U to one of n buckets. Suppose H has the following property: for every bucket I and key k, a 1/n fraction of the hash functions in H map k to i. Is H universal ?

- ○ Yes, always.
- ○ No, never.
- ○ Maybe yes, maybe no (depends on the H).
- ○ Only if the hash table is implemented using chaining.

Yes : Take H = all functions from U to {0,1,2,..,n-1}

No : Take H = the set of n different constant functions

Example: Hashing IP Addresses

Let U = IP addresses (of the form (x_1,x_2,x_3,x_4), with each $x_i \in \{0, 1, 2, ..., 255\}$

Let n = a prime (e.g., small multiple of # of objects in HT)

Construction : Define one hash function ha per 4-tuple a = (a_1,a_2,a_3,a_4) with each $a_i \in \{0, 1, 2, 3, ..., n-1\}$

Define : h_a : IP addrs -> buckets by

n^4 such functions

$$h_a(x_1, x_2, x_3, x_4) = \begin{pmatrix} a_1x_1 + a_2x_2 + \\ a_3x_3 + a_4x_4 \end{pmatrix} mod\ n$$

Tim Roughgarden

A Universal Hash Function

Define : $H = \{h_a | a_1, a_2, a_3, a_4 \in \{0, 1, 2, ..., n-1\}\}$

$$h_a(x_1, x_2, x_3, x_4) = \begin{pmatrix} a_1x_1 + a_2x_2 + \\ a_3x_3 + a_4x_4 \end{pmatrix} \ mod \ n$$

Theorem: This family is universal

Proof (Part I)

Consider distinct IP addresses (x_1,x_2,x_3,x_4), (y_1,y_2,y_3,y_4).

Assume : $x_4 \neq y_4$

Question : collision probability ?

$$(i.e., \; Prob_{h_a \in H}[h_a(x_1,..,x_4) = h_a(y_1,..,y_4)])$$

Note : collision <==>

$$a_1x_1 + a_2 + x_2 + a_3 + x_3 + a_4x_4 = a_1y_1 + a_2 + y_2 + a_3 + y_3 + a_4 + y_4 \; (mod \; n)$$

$$<=> a_4(x_4 - y_4) = \sum_{i=1}^{3} a_i(y_i - x_i) \; (mod \; n)$$

Next : condition on random choice of a_1,a_2,a_3. (a_4 still random)

Proof (Part II)

The Story So Far : with a_1, a_2, a_3 fixed arbitrarily, how many choices of a_4 satisfy

$$a_4(x_4 - y_4) = \sum_{i=1}^{3} a_i(y_i - x_i) \ (mod\ n)$$

Still random

<==> x,y collide under h_a

Some fixed number in {0,1,2,..,n-1}

Key Claim : left-hand side equally likely to be any of {0,1,2,…,n-1}

Reason : $x_4 \neq y_4$ $(x_4 - y_4 \neq 0 \ \text{mod}\ n)$

n is prime, a_4 uniform at random

[addendum : make sure n bigger than the maximum value of an ai]

Implies Prob[$h_a(x) = h_a(y)$] = 1/n

"Proof" by example : n = 7, $x_4 - y_4$ = 2 or 3 mod n

Q.E.D.

Design and Analysis of Algorithms I

Data Structures

Universal Hash Functions: Performance Guarantees (Chaining)

Overview of Universal Hashing

Next : details on randomized solution (in 3 parts).

Part 1 : proposed definition of a "good random hash function".
("universal family of hash functions")

Part 3 : concrete example of simple + practical such functions

Part 4 : justifications of definition : "good functions" lead to "good performance"

Universal Hash Functions

Definition : Let H be a set of hash function from U to {0,1,2,..,n-1}

H is universal if and only if :
For all $x, y \in U$ (with $x \neq y$)

$$Pr_{h \in H}[x, y \ collide; \ h(x) = h(y)] \leq 1/n$$ (n = # of buckets)

When h is chosen uniformly at random at random from H.
(i.e., collision probability as small as with "gold standard" of perfectly random hashing)

Chaining: Constant-Time Guarantee

Scenario : hash table implemented with chaining. Hash function h chosen uniformly at random from universal family H.

Theorem : [Carter-Wegman 1979]
All operations run in O(1) time. (for every data set S)

Caveats : 1.) in expectation over the random choice of the hash function h. (h = # of buckets)

2.) assumes |S| = O(n) [i.e., load $\alpha = \frac{|S|}{n} = O(1)$]

3.) assumes takes O(1) time to evaluate hash function

Proof (Part I)

Will analyze an unsuccessful Lookup
(other operations only faster).

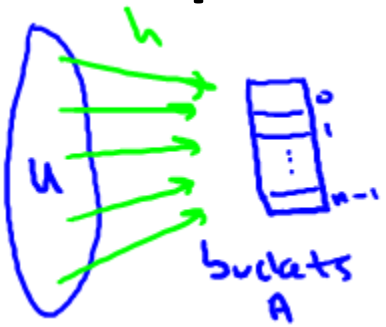
So : Let S = data set with |S| = O(n) (# of buckets)
Consider Lookup for $x \notin S$ (arbitrary data set S)

Running Time : O(1) + O(list length in A[h(x)])

- O(1): Compute h(x)
- O(list length in A[h(x)]): Traverse list
- L: A random variable, depends on hash function h

Tim Roughgarden

A General Decomposition Principle



Collision : distinct x,y in U such that h(x) = h(y).

Solution#1: (separate) chaining.

-- keep linked list in each bucket

-- given a key/object x, perform Insert/Delete/Lookup in the list in A[h(x)]

bucket for x

linked list for x

Solution#2 : open addressing. (only one object per bucket)

-- hash function now specifies probe sequence h1(x), h2(x), …
(keep trying till find open slot)

use 2 hash functions

-- examples : linear probing (look consecutively), double hashing

Tim Roughgarden

Proof (Part II)

Let L = list length in A[h(x)].

For $y \in S$ (so, $y \neq x$) define $z_y = \begin{cases} 1 \text{ if h(y) = h(x)} \\ 0 \text{ otherwise} \end{cases}$

Note : $L = \sum_{y \in S} A_y$

So : $E[L] = \sum_{y \in S} E[Z_y]$

Recall

$$z_y = \begin{cases} 1 \text{ if } h(y) = h(x) \\ 0 \text{ otherwise} \end{cases}$$

What does $\mathrm{E}[Z_y]$ evaluate to?

$$E[z_y] = 0 \cdot Pr[z_y = 0] + 1 \cdot Pr[z_y = 1]$$

- ○ $\Pr[h(y) = 0]$
- ○ $\Pr[h(y) \neq x]$
- ○ $\Pr[h(y) = h(x)]$
- ○ $\Pr[h(y) \neq h(x)]$

$$= Pr[h(y) = h(x)]$$

Proof (Part II)

Let L = list length in A[h(x)].

For $y \in S$ (so, $y \neq x$) define $z_y =$ 1 if h(y) = h(x); 0 otherwise

Note : $L = \sum_{y \in S} A_y$

So : $E[L] = \sum_{y \in S} E[Z_y] = \sum_{y \in S} Pr[h(y) = h(x)]$

Which of the following is the smallest valid upper bound on $\Pr[h(y) = h(x)]$?

- $1/n^2$
- $1/n$
- $1/2$
- $1 - 1/n$

By definition of a universal family of hash functions

Proof (Part II)

Let L = list length in A[h(x)].

For $y \in S$ (so, $y \neq x$) define $z_y = \begin{cases} 1 \text{ if } h(y) = h(x) \\ 0 \text{ otherwise} \end{cases}$

Note : $L = \sum_{y \in S} A_y$

So : $E[L] = \sum_{y \in S} E[Z_y] = \sum_{y \in S} Pr[h(y) = h(x)] \leq \frac{1}{n}$

Since H is universal $\rightarrow$ $\leq \sum_{y \in S} \frac{1}{n}$

$= \frac{|S|}{n} = load\ \alpha = O(1)$ Provided |S| = O(n)

Q.E.D.

Tim Roughgarden

Design and Analysis
of Algorithms I

Data Structures

Performance Guarantees (Open Addressing)

Open Addressing

Recall : one object per slot, hash function produces a probe sequence for each possible key x.

Fact : difficult to analyze rigorously.

Heuristic assumption : (for a quick & dirty idealized analysis only) all n! probe sequences equally.

Heuristic Analysis

Observation : under heuristic assumption, expected Insertion time is ~ $\frac{1}{1-\alpha}$, where α = load

Proof : A random probe finds an empty slot with probability $1-\alpha$

So : Insertion time ~ the number N of coin flips to get "heads", where Pr["heads"] = $1-\alpha$

Let N denote the number of coin flips need to get “heads”, with a coin whose probability of “heads” is $1 - \alpha$. What is E[N]?

- ○ $1/(1-\alpha)$
- ○ $1/\alpha$
- ○ $1-\alpha$
- ○ α

Heuristic Analysis

Observation : under heuristic assumption, expected Insertion time is ~ $\frac{1}{1-\alpha}$, where α = load

Proof : A random probe finds an empty slot with probability $1-\alpha$

So : Insertion time ~ the number N of coin flips to get "heads", where Pr["heads"] = $1-\alpha$

Note : $E[N] = 1 + \alpha \cdot E[N]$

Expected # of further coin flips needed

1st coin flip

Probability of tails

Solution : $E[N] = \frac{1}{1-\alpha}$

Linear Probing

Note : heuristic assumption completely false.

Assume instead : initial probes uniform at random independent for different keys. ("less false")

Theorem : [Knuth 1962] under above assumption, expected Insertion time is

$$= \frac{1}{(1-\alpha)^2}, \ where \ \alpha = \ load$$

The Allure of Algorithms

"I first formulated the following derivation in 1962… Ever since that day, the analysis of algorithms has in fact been one of the major themes in my life."

-D. E. Knuth, *The Art of Computer Programming, Volume 3.* (3rd ed., P. 536)

Data Structures

Bloom Filters

Design and Analysis
of Algorithms I

Bloom Filters: Supported Operations

Raison Dêtre: fast Inserts and Lookups.

Comparison to Hash Tables:

Pros: more space efficient.

Cons:

1) can't store an associated object
2) No deletions
3) Small false positive probability
   (i.e., might say x has been inserted even though it hasn't been)

Bloom Filters: Applications

Original: early spellcheckers.

Canonical: list of forbidden passwords

Modern: network routers.
- Limited memory, need to be super-fast

Bloom Filter: Under the Hood

Ingredients: 1) array of n bits ($So \frac{n}{|S|} = \#\ of\ bits\ per\ object\ in\ data\ set\ S$)

2) k hash functions $h_1,\ldots..,h_k$ (k = small constant)

Insert(x): for i = 1,2,…,k (whether or not bit already set ot 1)

set $A[h_i(x)]=1$

Lookup(x): return TRUE $\Leftrightarrow$ $A[h_i(x)] = 1$ for every I = 1,2,….,k.

Note: no false negatives. (if x was inserted, Lookup (x) guaranteed to succeed)

But: false positive if all k $h_i(x)$'s already set to 1 by other insertions.

Heuristic Analysis

Intuition: should be a trade-off between space and error (false positive) probability.

Assume: [not justified] all $h_i(x)$'s uniformly random and independent (across different i's and x's).

Setup: n bits, insert data set S into bloom filter.

Note: for each bit of A, the probability it's been set to I is (under above assumption):

Under the heuristic assumption, what is the probability that a given bit of the bloom filter (the first bit, say) has been set to 1 after the data set S has been inserted?

- $(1 - 1/n)^{k|S|}$ — prob 1st bit = 0
- $1 - (1 - 1/n)^{k|S|}$ — prob 1st bit = 1
- $(1/n)^{|S|}$
- $(1 - 1/n)^{|S|}$

Heuristic Analysis

Intuition: should be a trade-off between space and error (false positive) probability.

Assume: [not justified] all $h_i(x)$'s uniformly random and independent (across different i's and x's).

Setup: n bits, insert data set S into bloom filter.

Note: for each bit of A, the probability it's been set to I is (under above assumption):

$$1 - (1 - \frac{1}{n})^{k|S|} \leq 1 - e^{-\frac{k|S|}{n}} = 1 - e^{-\frac{k}{b}}$$

b = # of bits per object (n/|S|)

Heuristic Analysis (con'd)

Story so far: probability a given bit is 1 is $\leq 1 - e^{\frac{-k}{b}}$

So: under assumption, for x not in S, false positive probability is $\leq [1 - e^{\frac{-k}{b}}]^k$

where b = # of bits per object.

error rank ϵ

How to set k?: for fixed b, ϵ is minimized by setting

Plugging back in: $\epsilon \approx (\frac{1}{2})^{(ln2)b}$ or $b \approx 1.44 log_2 \frac{1}{\epsilon}$

(exponentially small in b)

$k \approx (ln2).b \approx 0.693$

Ex: with b = 8, choose k = 5 or 6 , error probability only approximately 2%.

Data Structures

Bloom Filters

Design and Analysis
of Algorithms I

Bloom Filters: Supported Operations

Raison Dêtre: fast Inserts and Lookups.

Comparison to Hash Tables:

Pros: more space efficient.

Cons:

1) can't store an associated object
2) No deletions
3) Small false positive probability
   (i.e., might say x has been inserted even though it hasn't been)

Bloom Filters: Applications

Original: early spellcheckers.

Canonical: list of forbidden passwords

Modern: network routers.
- Limited memory, need to be super-fast

Bloom Filter: Under the Hood

Ingredients: 1) array of n bits ($So \frac{n}{|S|} = \#\ of\ bits\ per\ object\ in\ data\ set\ S$)

2) k hash functions $h_1,\ldots..,h_k$ (k = small constant)

Insert(x): for i = 1,2,…,k (whether or not bit already set ot 1)

set $A[h_i(x)]=1$

Lookup(x): return TRUE $\Leftrightarrow$ $A[h_i(x)] = 1$ for every I = 1,2,….,k.

Note: no false negatives. (if x was inserted, Lookup (x) guaranteed to succeed)

But: false positive if all k $h_i(x)$'s already set to 1 by other insertions.

Heuristic Analysis

Intuition: should be a trade-off between space and error (false positive) probability.

Assume: [not justified] all $h_i(x)$'s uniformly random and independent (across different i's and x's).

Setup: n bits, insert data set S into bloom filter.

Note: for each bit of A, the probability it's been set to I is (under above assumption):

Under the heuristic assumption, what is the probability that a given bit of the bloom filter (the first bit, say) has been set to 1 after the data set S has been inserted?

- $(1 - 1/n)^{k|S|}$ — prob 1st bit = 0
- $1 - (1 - 1/n)^{k|S|}$ — prob 1st bit = 1
- $(1/n)^{|S|}$
- $(1 - 1/n)^{|S|}$

Heuristic Analysis

Intuition: should be a trade-off between space and error (false positive) probability.

Assume: [not justified] all $h_i(x)$'s uniformly random and independent (across different i's and x's).

Setup: n bits, insert data set S into bloom filter.

Note: for each bit of A, the probability it's been set to I is (under above assumption):

$$1 - (1 - \frac{1}{n})^{k|S|} \leq 1 - e^{-\frac{k|S|}{n}} = 1 - e^{-\frac{k}{b}}$$

b = # of bits per object (n/|S|)

Heuristic Analysis (con'd)

Story so far: probability a given bit is 1 is $\leq 1 - e^{\frac{-k}{b}}$

So: under assumption, for x not in S, false positive probability is $\leq [1 - e^{\frac{-k}{b}}]^k$

where b = # of bits per object.

error rank ϵ

How to set k?: for fixed b, ϵ is minimized by setting

Plugging back in: $\epsilon \approx (\frac{1}{2})^{(ln2)b}$ or $b \approx 1.44 log_2 \frac{1}{\epsilon}$

(exponentially small in b)

$k \approx (ln2).b \approx 0.693$

Ex: with b = 8, choose k = 5 or 6 , error probability only approximately 2%.