Design and Analysis of Algorithms I

Data Structures

Hash Tables and Applications

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*

Design and Analysis
of Algorithms I

Data Structures

Hash Tables: Some Implementation Details

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 $h : U \rightarrow \{0, 1, 2, ..., n-1\}$
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

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
- 57
- 184
- 367

50 %

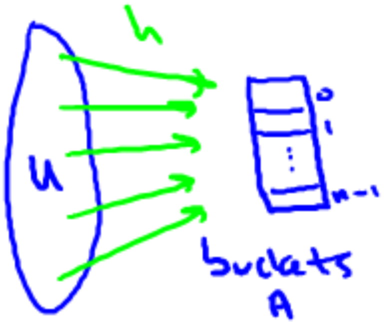
99 %

99.99....%

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)]

Bucket for x

Linked list 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 $\theta(1)$

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

$\theta(list\ length)$ for Insert/Delete.

Equal-length lists

could be anywhere from $\boxed{m/n}$ to m for $\boxed{m}$ 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) = 1st 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

"hash code"

e.g., subroutine to convert strings to integers

"comparison function "

like the mod n function

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: 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)

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

<u>Define</u> : h_a : IP addrs -> buckets by

n^4 such functions

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

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

$$\underbrace{\boxed{\boxed{a_4}(x_4 - y_4)} = \boxed{\sum_{i=1}^{3} a_i(y_i - x_i) \pmod{n}}}_{\Longleftrightarrow\ x,y \text{ collide under } h_a}$$

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 : $x_4 \neq y_4$ $\quad(x_4-y_4 \neq 0 \mod n)$

n is prime, a_4 uniform at random

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

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

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

Q.E.D.

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.