Week 6

Hash Tables

purpose: maintain a set of stuff with all operations in $O(1)$ time!

Operations:

- insert
- delete
- lookup

} all in $O(1)$ time! *

* when properly implemented and data is not pathological

"dictionary"

but a hash table doesn't support the order!

Applications:

Δ De-duplications

given: a "stream" of objects

goal: ignore duplicates (keep track of unique objects)

solution: when a new object x arrives
- look x up 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

$$x + y = t$$

naive solution: $\Theta(n^2)$ time

better: 1) sort A ($\Theta(n \log n)$ time)
2) for each x in A look for $t - x$ in A via binary search ($\Theta(n \log n)$ time)

more better: 1. insert elements of A into hash table H ($\Theta(n)$ time)
2. for each x in A, lookup $t - x$ ($\Theta(n)$ time)

Also

- historical application: symbol tables in compilers

- Blocking network traffic (blacklist)

- Search algorithms (game tree exploration, etc)

- etc.

high-level idea

Setup: universe U (all IP addresses, all names, etc)
generally, really, really big

Goal: want to maintain set $S \subseteq U$

Solution: pick n - number of buckets

(assume $|S|$ doesn't change much)

1. choose a hash function
$h: U \mapsto \{0, 1, \ldots, n-1\}$

2. use array A of length n, store x in $A[h(x)]$

Birthday paradox

Consider n people with random birthdays. How large does n need to be before there is at least 50% chance that two people have the same birthday?

$\underline{\underline{23}}$ (50%)   57 (99%)

$\Rightarrow$ 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 $\underbrace{A[h(x)]}$

returns an list

Solution #2

open addressing (only one object per bucket)

hash function now specifies probe sequence $h_1(x), h_2(x), \ldots$

- keep trying until we find an open slot

examples: Linear probing
double hashing

- A Good hash function

note: in hash table with chaining,
Θ(list len) for insert/delete

equal-len list

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

all objects are in the same bucket

point: performance depends on the choice of hash function!

So, "good" hash function should:

- spread data out => lead to good performance
- easy to store / be fast to evaluate

example:

keys - phone numbers (10-digits)

- terrible hash: $h(x)$ - 1^{st} 3 digits of x
- mediocre hash: $h(x)$ - last 3 digits of x

example:

keys - memory locations

- bad hash: $h(x) = x \bmod 1000$
  (all odd buckets guaranteed to be empty)

Quick-and-Dirty hash function

Object U $\xrightarrow{\text{hash code}}$ integers $\xrightarrow{\text{comparison function}}$ buckets $\{0 \dots n-1\}$

hash code: eg. subroutine to convert strings to integers

comparison function: like the mod n function

How to choose n = # of buckets

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

The load of a hash table

the load factor

$$\alpha = \frac{\text{number of objects}}{\text{number of buckets}}$$

- for good hash table performance, need to control load
- and a good hash fuction

Pathological data sets

super-clever user-defined hash function does not guarantee to spread evenly

For every hash function there exists a pathological dataset

Solutions:

1. Use a cryptographic hash function (SHA-2, etc)

2. Use randomization

   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"

Universal Hash Function

~~Overview~~ Definition

- let H be a set of hash functions from U to $\{0, 1, \dots, n-1\}$

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

$$\underbrace{\Pr_{h \in H}\Big[\underset{\text{i.e. } h(x) = h(y)}{x, y \text{ collide}}\Big]}_{\text{when } h \text{ is chosen uniformly at random from } H.} \leq \frac{1}{n}$$

(number of buckets)

example:

Hashing IP addresses

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

let n = a prime

construction: define one hash function h_a per 4-point
$$a = (a_1, a_2, a_3, a_4)$$
~~wi~~ with each $a_i \in \{0, 1, \dots, n-1\}$

h_a = IP address $\rightarrow$ buckets by
$$h_a(x_1 x_2 x_3 x_4) = \begin{pmatrix} a_1 x_1 + a_2 x_2 + \\ a_3 x_3 + a_4 x_4 \end{pmatrix} \bmod n$$

n^4 such functions

$$\Downarrow$$

$$H = \{ h_a \mid a_1 a_2 a_3 a_4 \in \{0, \dots, n-1\} \}$$
$$h_a = \sum a_i x_i \bmod n$$

this family is universal

Bloom Filters

reason: fast inserts and lookups

comparison to hash tables:

prons

- more space efficient

cons

- can't store an associated object
- no deletions
- small false positive probability

(i.e. ~~x~~ may say x has been inserted, but it wasn't)

Applications

- original - early spellcheckers
- canonical: list of forbidden passwords
- modern: network routers
  (limited memory, need to be super fast)

Under the hood:

ingredients:

1. array of $\overset{n}{\vee}$ bits ($\frac{n}{|S|}$ - number of bits per object in data set S)

2. k - hash function $h_1 - h_k$ (k - small constant)

Insert (x):

for $i = 1..k$

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

Lookup (x)

True if $A[h_i(x)] == 1$ for every $i = 1,..k$

note: no false ~~positives~~ negatives (if x was inserted, Lookup (x) guaranteed to succeed)

but: false positives if all k $h_i(x)$'s already set to 1 by other insertions