WEEK 4 L1

HASH TABLES

Purpose: insertable random access, through keys

Insert: add new record

Delete: delete existing record

Lookup: check for a particular record

} Using a key

ONLY THESE OPERATIONS, BUT GUARANTEED $O(1)$ TIME (if implemented well)
(1) Properly implemented
(2) Non pathological

APPLICATION: DE-DUPLICATION

Input: a "stream" of objects

Goal: Ignore duplicates

hashtable ~~Hash~~ lookup.
If not found, insert x into H

APPLICATION: 2-SUM PROBLEM

Input: Unsorted array of n integers. Target sum t
Goal: Determine whether two numbers in Array x + y = t

Naive solution: $O(n^2)$ $(n(n-1))$

<2SUM PROB CNT'D

Even better: ① Insert elements of A in H. $O(n)$
② for each x, look in H for t-x $O(n)$

FURTHER OBVIOUS APPLICATIONS

- historical application: symbol tables in compilers
- blocking network traffic
- Search algorithms (eg game tree exploration), can use to avoid exploring more than once

WEEK 4 L2

HASH TABLE IMPLEMENTATION

High Level Idea:

Setup: Universe U (generally, really big)

Goal: Maintain evolving set $S \subseteq U$ (of reasonable size)

Naive Solution ①: Array based solution indexed by n ($O(1)$ ops, $O(n)$ space)

Naive Solution ②: List based solution ($O(1)$ pace, $O(S)$ lookups

Solution:

① Pick n = # of "buckets" to store data in with $n \geq |S|$

② Choose a hash function $h: U \to \{0, 1, 2, \ldots, n-1\}$ spits out position in array (for simplicity we'll assume S doesn't vary too much)

③ Use Array A of length n, store x in $A[h(x)]$

What happens when there are collisions? $h(\text{"Bob"}) = h(\text{"Alice"})$

Think of birthdays! 23 people!

RESOLVING COLLISIONS:

Collision: $h(x) = h(y)$ with $x, y \in U$.

Solution #1: Separate chaining. Keep linked list in bucket, perform operations on li

Solution #2: Open Addressing (one object per bucket), h now specifies probe sequence

(Linear probing, or (quadratic probing)

~~Solution #3~~ Double hashing: the buckets are hash lists themselves, takes lots of time!

WEEK 4 L3

HASH TABLES IMPLEMENTATION DETAILS

WHAT IS A GOOD HASH FUNCTION?

Note: in hash table with chaining, Insert $O(1)$ most of the time

Performance depends on the choice of hash function!

Properties of a good hash function:

① Should spread data out well (gold standard: completely random)
② Should be simple enough to remember, not take too much time (must be $O(1)$)

EXAMPLES OF BAD HASH FUNCTIONS

Ex: Keys = phone numbers $|n| = 10^{10}$

terrible: $h(x)$ = 1st 3 digits of x $|n| = 10^3$
mediocre: $h(x)$ = last 3 digits of x $|n| = 10^3$

Example: keys = memory locations (power of 2)

bad hash func = $h(x) = \text{mod } 1000 (x)$ (all odd buckets are empty!)

QUICK AND DIRTY HASH FUNCTIONS

Objects U → ("hash" code) → integers → ("compression function") → buckets $\{0, 1, 2, \ldots, n-1\}$

n should be
- not too close to a power of 2
- prime
- not too close to a power of 10

Very commonly used approach

The hash code eg normally converting strings to integers
compression function eg mod n () n = number of buckets

① n should be prime

WEEK 9 L4

THE LOAD OF A HASHTABLE

Definition: The load factor of a hash table is:

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

Conditions for good performance:

1. $\alpha = O(1)$, else won't run in constant time. (NEED TO BE ABLE TO COMPUTE α in $O(1)$)
2. For open addressing, α must be < 1

For good performance, we need to control load

Also, THE PERFECT HASH FUNCTION DOES NOT EXIST. FOR EVERY $H(+)$ THERE IS A PATHOLOGICAL DATA SET

PATHOLOGICAL DATA IN THE REAL WORLD

Main Point: Can analyze several real-world systems by exploiting badly designed hash functions.

If the hash function is open-source and simple, anyone can read it and reverse engineer it.

SOLUTIONS

1. Use a cryptographic hash function which can't be reverse engineered
2. Use randomization: design a family H of hash functions and pick one at random

WEEK 4 L8

BLOOM FILTERS

Advancement on hashtables, save space, have errors

PROS	CONS
es space	① Can't store associated object ② No deletions ③ Small false positive probability

~~Original~~ APPLICATIONS

Original: early spell checkers

Genomics: List of forbidden passwords

Modern: Network routers
- limited memory, need to be super fast still

WHAT IS A BLOOM FILTER?

Ingredients:

① an array of n bits ; $\frac{n}{|S|}$ = # of bits per object in data set
② K hash functions $h_1, h_2, \ldots, h_K$ K = small constant
③ for $i = 1, 2, \ldots, K$, $A[h_i(x)] = 1$, (whether or not it was 1 already)

Lookup(x) return TRUE if $A[h_i(x)] = 1$ for every $i = 1, 2, \ldots, K$

There are false positives, but no false negatives and it saves

WEEK 4 L9

HEURISTIC ANALYSIS OF BLOOM FILTERS

There is a trade off between % of false positives and space saved.

After inserting dataset S, $1-\left(1-\frac{1}{n}\right)^{K|S|}$ probability for a given bit to have been set to 1

prob $\in 1-e^{\frac{-K}{b}}$ $b=\frac{n}{|S|}$, number of bits per object

This means the false positive prob $\emptyset$ is bounded by $\left(1-e^{-K/b}\right)^K$

how to find K? For fixed b, the false positive probability is minimised by setting $K \simeq (\ln 2)\cdot b$

Example: $b=8$, $K=5$ or 6, error prob $\simeq 2\%$

Problem Sensibility

$h(\underline{n \text{ keys}}) \to T$ $\ell[\bar{0}]$ m

$\frac{n}{m}$ ... like with the n tasks for m servers.

$O(n)$

Ω

$\frac{n(n-1)}{2} \cdot \frac{1}{m} = \frac{n(n-1)}{2m}$

$1 - e^{\frac{n^2}{m}} = < 0,01\%$

$e^{\frac{1}{m}} < 0,199n$