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

} using a "key"

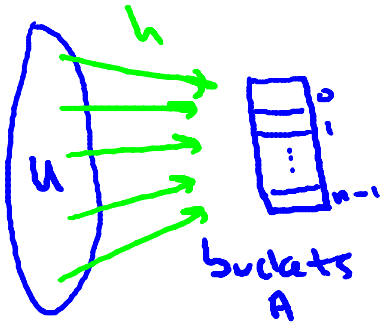
(a "dictionary")

AMAZING GUARANTEE

all operations in O(1) time!*

* ① properly implemented ② non-pathological data

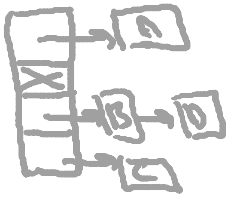
Resolving Collisions



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 $h_1(x), h_2(x), \ldots$ (keep trying til 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) $\alpha = O(1)$ is necessary condition for operations to run in constant time.

(2) with open addressing, need $\alpha << 1$.

Upshot: for good HT performance, need to control load.

Pathological Data Sets

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

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

h: i.e., spreads data evenly across buckets

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

Reason: Fix a hash function $h: U \to \{0, 1, 2, \ldots, n-1\}$.

(assume $|U| >> n$)

=> a la Pigeonhole Principle, $\exists$ bucket i such that $\geq |U|/n$ elements of U hash to i under h.

=> if data set drawn only from these, everything collides!

Pathological Data in the Real World

Reference: 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. → next 2 videos
- design a family $\mathcal{H}$ of hash functions such that, $\forall$ data sets S, "almost all" functions $h \in \mathcal{H}$ spread S out "pretty evenly".

(compare to QuickSort guarantee)

Overview of Universal Hashing

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

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

Part II: Concrete example of simple + practical such functions.

Part III: justification of definition: "good functions" lead to "good performance".