



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”)

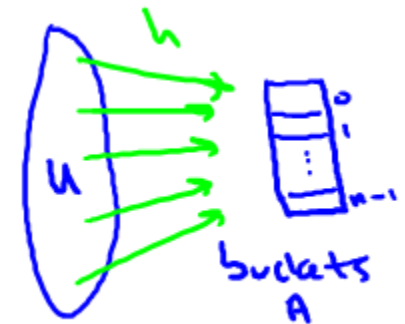
Using a “key”

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 $h_1(x), h_2(x), \dots$
(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

open addressing would collapse if load crosses 1

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 $\alpha \ll 1$.

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

increase or decrease the # of buckets in hash table to control the 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 $h : U \rightarrow \{0,1,2,\dots,n-1\}$

\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”