lecture 3

hashing, hash tables

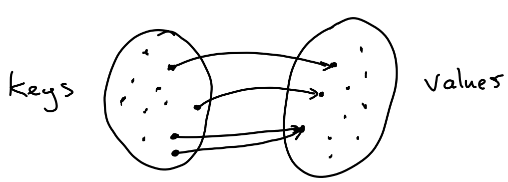
Background Resources

- my COMP 250 lectures 31, 32

Mix of Background + New Material

- coursera
  - Sedgewick Algorithms 1 week 6
    https://class.coursera.org/algs4partI-
    https://class.coursera.org/algs4partI-
    https://class.coursera.org/algs4partI-003/lecture/55
  - Roughgarden Algorithms 1 week 6
    https://class.coursera.org/algo-004/lecture
- Cormen Leiserson Rivest (CLR) Ch. 12

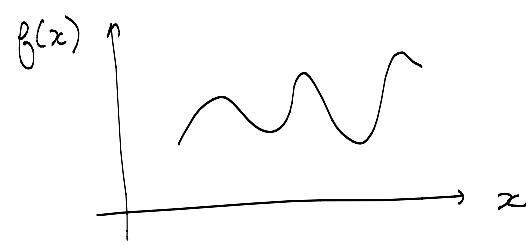
Map



A map is a set of (key, value) pairs.

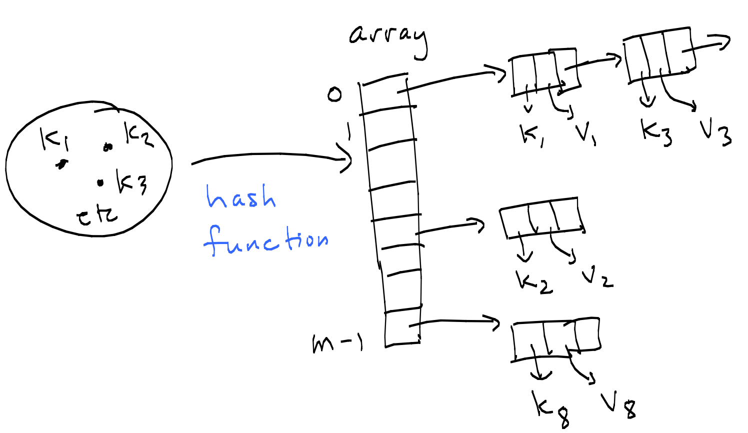
Each key maps to at most one value.

e.g. $\{ (x, f(x)) \}$



This is also a map. ("function" = "map")
But we will NOT be talking about continuous functions / maps.

Hash Map (Hash function + Hash Tables)



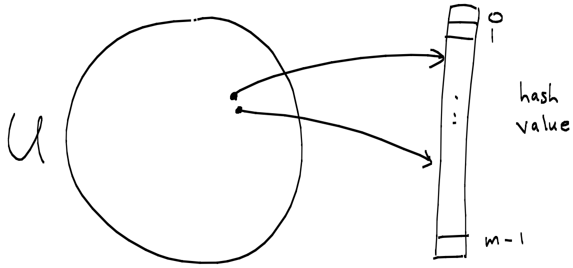
Hash Function

$h : U \to \{0, 1, \ldots\ m-1\}$

U: universe of possible keys

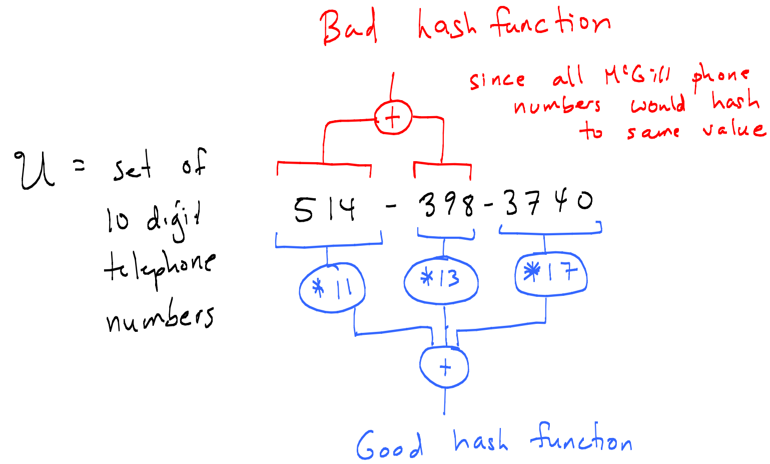
e.g. U might be the set of all finite length strings
(This set has infinite size.)

"Hash" – mixing

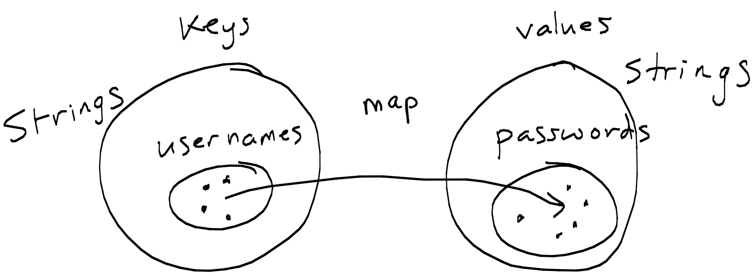


"Nearby / similar" keys in U should map to different values.

Example of good/bad hash function



Example where hashing is used: Password Authentication



{ (usernames, passwords) } map

– stored as a file on some webserver

To login, enter username and password.

Server verifies that password entered matches user's password in file.

Problem: hacker could steal the file

Solution: the webserver hashes password and stores map

{ (username, hash(password)) } instead of { (username, password) }.

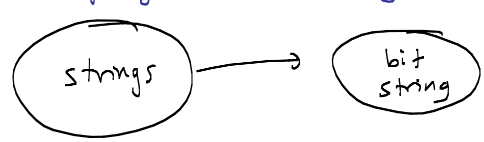
Q: What happens when you log in?

You enter (username, password)

The "system" does what?

A: It computes hash(password), "throws away" your real password, and verifies that the hashed password matches the one in the password file.

There exist "standard" algorithms for hashing passwords and other private text ("cryptographic hashing")



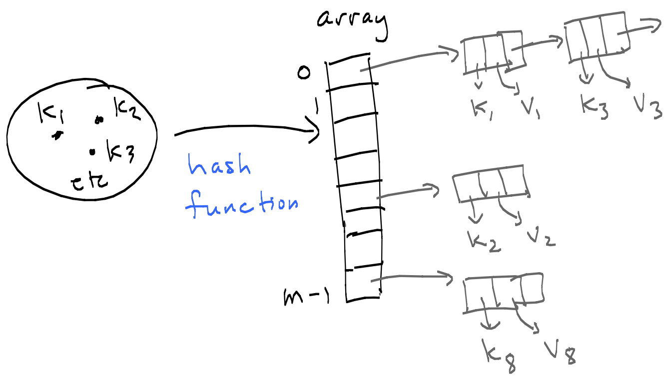
e.g. MD-5 (128 bits)

SHA-1 (256 bits)

http://www.md5.cz/

Details of these algorithms are not worth covering in COMP 251 (complicated "bit mixing" operations).

Hash Functions for Hash Tables

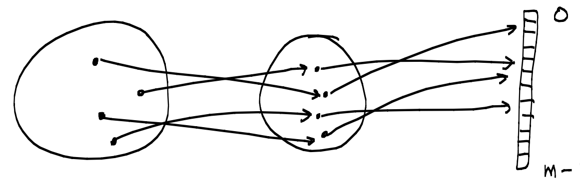


Hash functions for hash tables: two steps

hash coding compression

$$h : \mathcal{U} \rightarrow \{\text{integers}\} \rightarrow \{0, 1, \ldots m-1\}$$

universe of keys "hash codes" "hash values"



Hash function: two steps

$$h : \mathcal{U} \rightarrow \{0, \ldots m-1\}$$

$$h(\text{key}) = \text{compression}(\text{hashCode}(\text{key}))$$

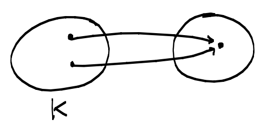
↑

composition of maps

hash coding

$$h : K \rightarrow \{\text{integers}\}$$

In Java, every class has a hashCode() method which returns an "int": 32 bits.



It can happen (but its rare) that two keys e.g. Strings s_1 & s_2 have the same hashCode.

hash Coding Compression

$$h : K \rightarrow \{\text{integers}\} \rightarrow \{0, 1, \ldots m-1\}$$

hash codes hash values

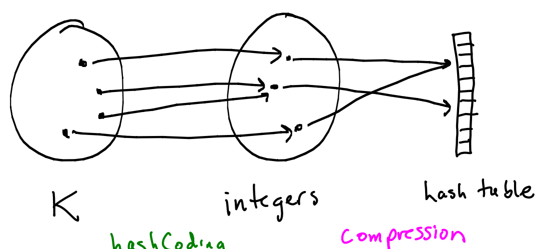
It often happens that compression causes two hash codes to map to the same value.

hash code	hashcode % m ($m = 7$)
41	6
16	2
25	4
21	0
36	1
35	3
53	4

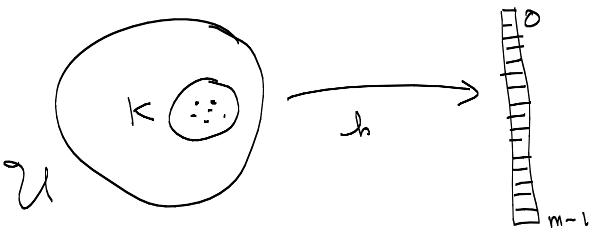
Collisions

Hash function maps two keys to the same index in the hash table

either
- two keys have same hash code
- two keys have different hash codes but are compressed to same hash value



[Exercise: if you have some background in probability]



If there are $|K|$ randomly chosen keys, and m "buckets" in the table, what is the probability of a collision?

"Birthday Problem"

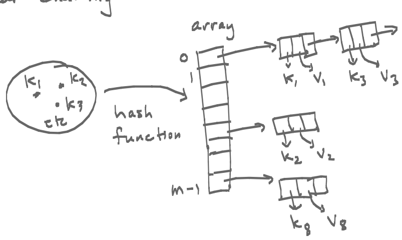
Given a set of (randomly chosen) n people, what is the probability that at least one pair of them has the same birthday?

e.g. for $n = 23$, probability ≈ 0.5 is 50% chance!

The point: collisions happen alot.

Hash Table Data Structures

- open hashing – "linear chaining" (COMP 250)

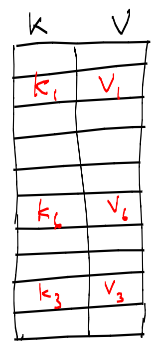


- closed hashing – probing (COMP 251)

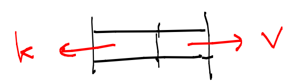
sometimes called "open addressing"

Closed Hashing

At most one (key, value) is stored per array slot. No linked list overhead, no dynamic memory allocation!



use two arrays



or store a pair of references

Linear Probing

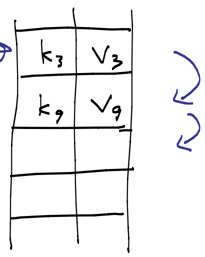
If there is a collision, then check next entry in table:

$h(\text{key})$

$(h(\text{key}) + 1) \% m$

$(h(\text{key}) + 2) \% m$

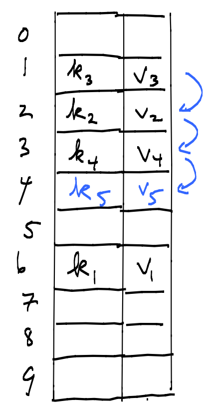
$\vdots$



What is this for?

Example. ($m = 10$)

key	h(key)
k_1	6
k_2	2
k_3	1
k_4	3
k_5	1

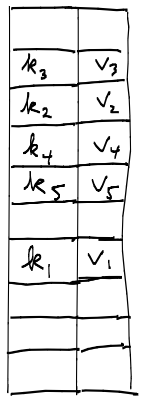


Linear Probing and Clustering

Consider some empty slot in the table.

Q: What is the probability that the next put(key, value) will be to that slot?

A: $\frac{\ell + 1}{m}$ where the ℓ previous slots are occupied.



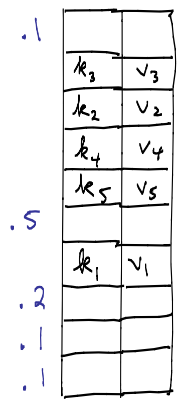
Linear Probing and Clustering

Consider some empty slot in the table.

Q: What is the probability that the next put(key, value) will be to that slot?

A: $\frac{\ell + 1}{m}$ where the ℓ previous slots are occupied.

Bigger clusters grow faster!



More general probing method

$h_0(\text{key})$
$h_1(\text{key}) \ \% \ m$
$h_2(\text{key}) \ \% \ m$
$\vdots$
$h_{m-1}(\text{key}) \ \% \ m$

sequence of hash functions

For linear probing,

$$h_i(\text{key}) = h_0(\text{key}) + i$$

Quadratic Probing

$$\begin{aligned}
h_0(\text{key}) &= h(\text{key}) \\
h_1(\text{key}) &= h(\text{key}) + 1 \quad \% \ m \\
h_2(\text{key}) &= h(\text{key}) + 2^2 \quad \% \ m \\
h_3(\text{key}) &= h(\text{key}) + 3^2 \\
&\vdots \\
h_i(\text{key}) &= h(\text{key}) + i^2
\end{aligned}$$

Idea: by using different step sizes, we avoid clustering.

We also want $h_0(\text{key}), h_1(\text{key}), h_2(\text{key}), \ldots$ to map to different slots! (no "self collisions")

- Linear probing ensures this. but suffers from clustering ☹
- Quadratic probing ensure this up to $h_i()$, where $i = \sqrt{m}$. Why?

What about for larger values of i?

For quadratic probing, under what conditions does

$$h_i(\text{key}) \ \% \ m = h_j(\text{key}) \ \% \ m \ ?$$

Exercise: show the following

$$(h_0(\text{key}) + i^2) \ \% \ m = (h_0(\text{key}) + j^2) \ \% \ m$$

$$\iff \quad (i^2 - j^2) \ \% \ m = 0$$

"if and only if"

$$(i^2 - j^2) \% m = 0$$

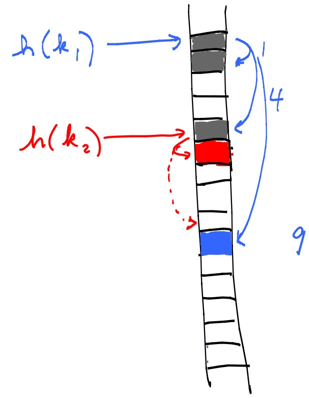
$$\iff (i-j)(i+j) \% m = 0$$

$$\iff (i-j)(i+j) = cm \text{ for some integer } c.$$

<u>Exercise</u> (easy with MATH 240)

If we choose m to be a prime number and we require i and j are less than $m/2$ then the above equations are true if and only if $i = j$.

Quadratic Probing

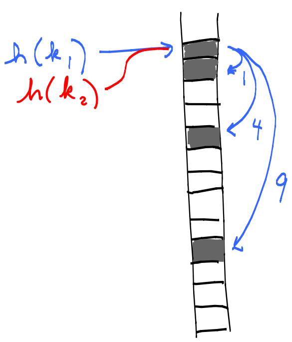


When $h_0(k_2) \neq h_0(k_1)$

the probe paths tend to land in different slots.

(Collisions only rarely occur.)

Quadratic Probing



When $h_0(k_2) = h_0(k_1)$

the probe paths land in exactly the same slots.

(Collisions can easily occur.)

Another common approach: Double hashing

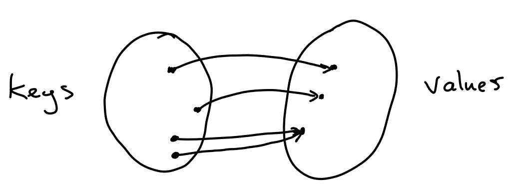
Use two hash functions $h(), g()$.

$$\underbrace{h_i(key)}_{\text{probe sequence}} = h(key) + \overbrace{i\, g(key)}^{\text{step size}}$$

for $i = 0, 1, \ldots, m-1$.

Choose $h(), g()$ such that different keys tend to have different probe sequences. (unlikely to have collision for both)

Map



Map ADT supports several operations

- put (key, value)
- get (key)
- delete (key) ← not always needed

For closed hashing, deletion works poorly. Why? e.g. linear probing

	key	h(key)
put	k_1	6
	k_2	2
	k_3	1
	k_4	3
put	k_5	1
delete	k_3	1
get	k_5	1

0		
1	k_3	v_3 ←
2	k_2	v_2
3	k_4	v_4
4	k_5	v_5
5		
6	k_1	v_1
7		
8		
9		

get(k_5) will fail because slot 1 will be empty even though k_5 is in the table

The previous slide is very important (even though it is only one slide)

Map data structures (lectures 2 & 3)

• balanced search tree $O(\log n)$ access	requires Comparable keys and allows more ops (e.g. find the top k keys)
• hash table $O(1)$ access [assuming good hash functions]	put get delete ← only open hashing

Next lecture

- heaps (review COMP 250)
- A1 should be posted by Thursday evening