CS2100 – Hashing (part 3)

Note Title 5/2/2011

Announcements

- HW due today
- Graded HW will come back from me this
  - ↳ will only be emailed to 1 person
- Next HW – decode – due next Wednesday

Data Storage - Dictionary: insert, find, remove

Ex:

Locker # (key)	Name (data)
26	Dan
355	Kevin
101	Tracy
53	Nitish
201	David
⋮	⋮

We want to be able to retrieve a name quickly when given a locker number.

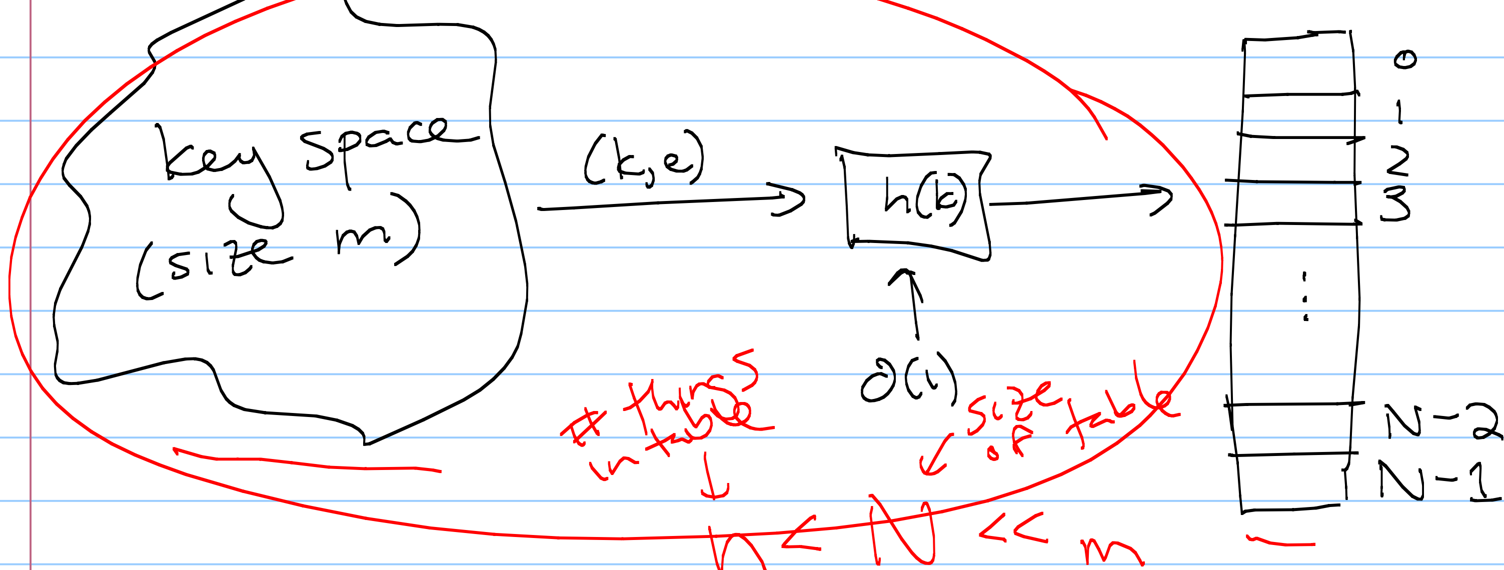
(Let n = # of people, & m = # of lockers)

$n \le m$

Good hash functions:

- Are fast   goal: $O(1)$
- Don't have collisions. ← when $k_1 \neq k_2$ but $h(k_1) = h(k_2)$

  these are unavoidable, but we want to minimize



Step 1: Turn key into an integer

XOR / bit manipulation

polynomial hashing

Step 2: Compression map

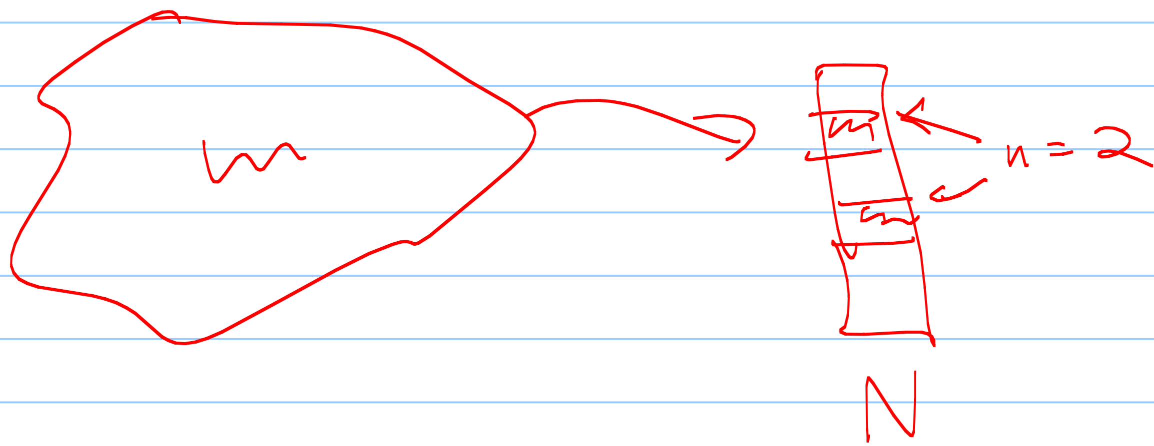
MAD, cyclic permutation, etc.

Collisions

Can we ever totally avoid collisions?

NO!

m is larger than ~~n~~ N



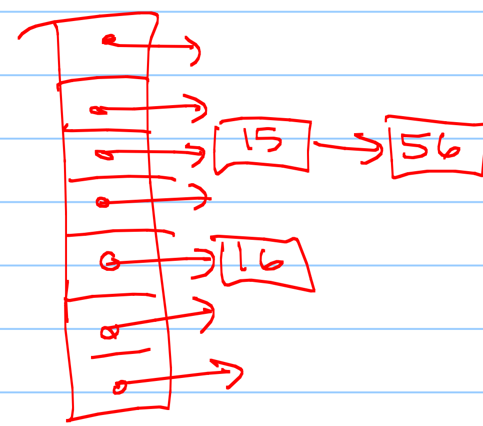
Step 3: Handle collisions (gracefully & quickly)

So how can we handle collisions?

[Hint: Do we have any data structures that can store more than 1 element?]

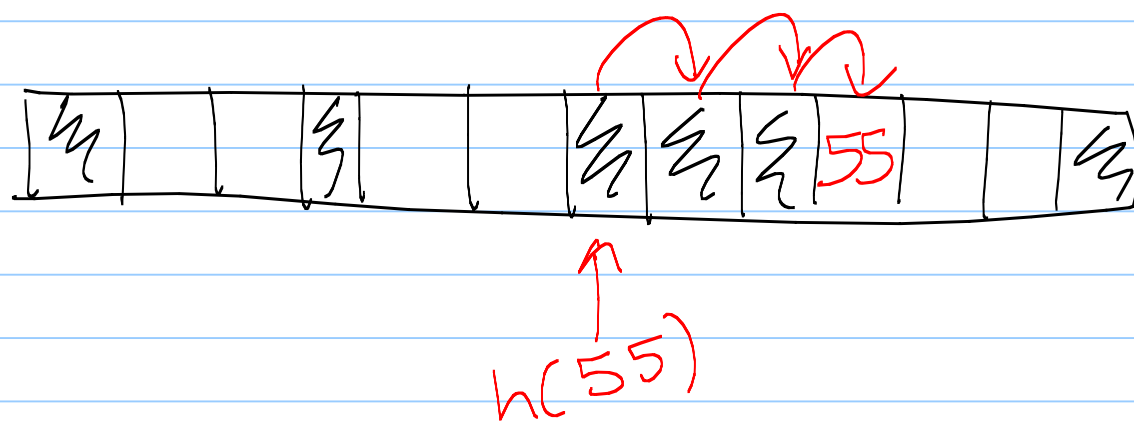
Possibilities:

- Vector
- Linked structure — simple chaining
- Tree structure



Linear Probing

Instead of lists, if we hash to a full spot, just ~~keep~~ checking next spot (as long as the next spot is not empty).



remove(26)

Example

$h(k) = k \bmod 11$

0	1	2	3	4	5	6	7	8	9	10
	(12, E)		(37, I)	~~(26, N)~~	(16, C)	(5, H)	(15, A)			(21, R)

Insert:

(12, E) $h(12) = 12 \bmod 11 = 1$

(21, R) $h(21) = 21 \bmod 11 = 10$

(37, I) $h(37) = 4$

(26, N) $h(26) = 4$, try $h(26)+1$

(16, C) $h(16) = 5$

(5, H) $h(5) = 5$ try $5+1$

(15, A) $h(15) = 4$

Find(15, A) $h(15) = 4$: must walk through array until empty space

Find(3, L) $h(3) = 3$

Issue:

How do we delete? Simply erasing will leave "holes" in array

↳ find would not work!

Solution: "dirty" bit:

when removing, don't remove

↳ set dirty bit

find knows dirty bit means it's been deleted

Running Time for Linear Probing

<u>Insert</u>: $O(n)$ worst case

<u>expected</u>: $O(1)$

<u>Remove</u>: same

<u>Find</u>: same

Issues with linear probing

- "Clusters" form
  - worse if #'s not "good" in hash function
  - terrible when array nears ½ full
- Removing doesn't actually reduce # of elements - just sets the "dirty" bit.
  ↳ frequent re-hashing

S

Quadratic Probing

Linear probing checks $A[h(k)+j \bmod N]$ if previous spot is full (for $j = 1, 2, \ldots$)

To avoid clusters, try

$$A[h(k) + j^2 \bmod N]$$

where $j = 0, 1, 2, 3, 4, \ldots$

so: $h(k)$ first

if full: $h(k)+1$

if full: $h(k)+2^2 = h(k)+4$

$h(k)+3^2 = h(k)+9$

$\vdots$

Example

$h(k) = k \bmod 11$

0	1	2	3	4	5	6	7	8	9	10
	(12, E)	(4, M)		(37, I)	(26, N)	(16, C)		(15, A)	(5, H)	(21, R)

Insert:

(12, E) $= h(12) = 12 \bmod 11 = 1$

(21, R) $h(21) = 10$

(37, I) $h(37) = 4$

(26, N) $h(26) = 4$ full $\rightarrow 4 + 1^2 = 5$

(16, C) $h(16) = 5 \rightarrow$ try $5 + 1^2$

(5, H) $h(5) = 5 \rightarrow$ try $5 + 1^2 \rightarrow 5 + 2^2 = 9$

(15, A) $h(15) = 4 \rightarrow$ try $4 + 1^2 \rightarrow 4 + 2^2$

(4, M) $h(4) = 4 \rightarrow 5 \rightarrow 4 + 2^2 = 8 \rightarrow 4 + 3^2$

Issues with Quadratic Probing:

- Can still cause secondary clustering
- N <u>really</u> must be prime for this to work
- Even with N prime, starts to fail when array gets half full
- Can fail entirely even if array not full.

(Runtimes are essentially the same)

Secondary Hashing

- Try $A[h(k)]$
- If full, try $A[h(k) + f(j) \bmod N]$ for $j = 1, 2, 3, \ldots$

where

$$f(j) = \underbrace{j}_{j^2} \cdot \underbrace{\ell(k)}_{\text{using } \ell(k), \text{ a hash fcn, insted of } j^2}$$

with ℓ a <u>different</u> hash function

Load Factors

Separate chaining actually works as well as most others in practice, although it does use more space.

Most of these methods only work well if $\frac{n}{N} < .5$.

(Even chaining starts to fail if $\frac{n}{N} > .9$)

Rehashing

Because we need $\frac{n}{N} < .5$, ~~most~~ all hash code checks if the array has become more than half full.

If so, it stops & recomputes everything for a larger N, usually at least twice as big.

(Still not too bad in an amortized sense — think vectors.)