CS180 - Hashing (part 2)

Note Title 4/29/2011

Announcements

- Checkpoint today
- Program due Thursday
- Last HW out today, due next Monday

  (**Note:** **Will** include topics you haven't seen yet!)
- Review Session: Friday, Dec. 16, at 10:30am
- Teacher evals later this week - please come!!

Data Storage

Ex:

keys data

Locker #	Name
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) $m \geq n$

BAD → array : O(1) for everything!
Size: O(m)
m >> n

Dictionaries

A data structure which supports the following:

locker#　　　　　　　　　　Name

```
void      insert (keyType &k, dataType &d)
dataType  find   (keyType &k)
void      remove (keyType &k)
```

Note: Everything is based on keys!

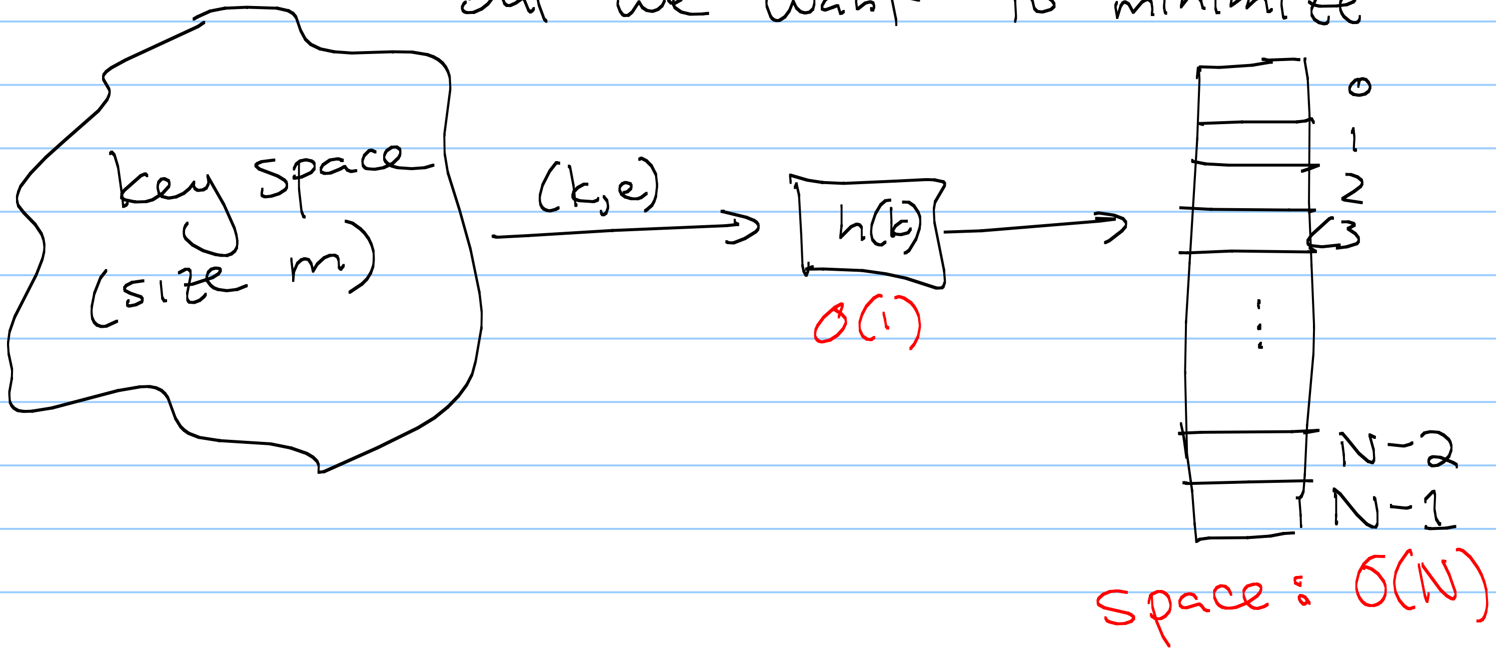
Don't know keyType - might not correspond to an int.

Good hash functions:

- Are fast   goal: $O(1)$
- Don't have collisions.

when $k_1 \neq k_2$ but $h(k_1) = h(k_2)$

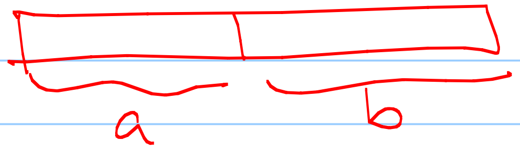
these are unavoidable, but we want to minimize



Step 1: Get a number (& avoid collisions)

char (32-bits) → ASCII

float (64-bits)



a+b = 32-bits

string: Erin

69 + 114 + 105 + 110 = 32-bits

h(Erin) = h(rinE)

But, in some cases, a strategy like this can backfire:

temp01 and temp10 and pm0te1

all hash to same int

We want to avoid collisions between "similar" strings (or other types).

A Better Idea: Polynomial Hash Codes

Pick $a \neq 1$ and split data into k 32-bit parts: $x = (x_0, x_1, x_2, x_3, \ldots, x_{k-1})$

Let $h(x) = \underline{x_0} a^{k-1} + x_1 a^{k-2} + \cdots + x_{k-2} a + x_{k-1}$

Ex: Erin with $a = 37$

$$69 \cdot 37^3 + 114 \cdot 37^2 + 105 \cdot 37 + 110 \cdot 37^0$$

riEn: ?

$$\neq$$

$$114 \cdot 37^3 + 105 \cdot 37^2 + 69 \cdot 37 + 110$$

Polynomial Hashing

This strategy makes it less likely that similar keys will collide.

(Works for floats, strings, etc.)

What about overflow?

☆ → truncate
or
take remainders

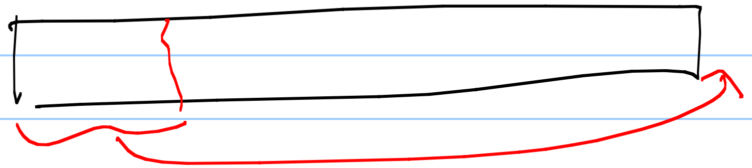
$\underbrace{1011},00111111$ shift by 4

$\rightarrow 0011111\,\underbrace{1011}$

Cyclic shift hash codes

Alternative to polynomial hashing

Instead of multiplying by a, shift each 32-bit piece by some # of bits.

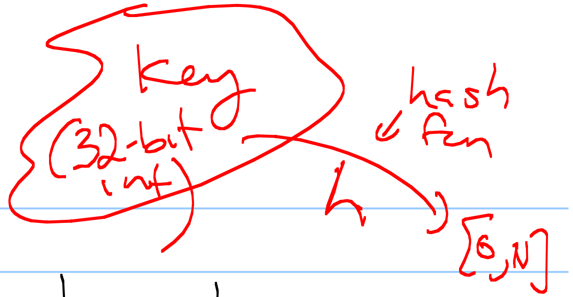


Also works well in practice.

E r i $\underbrace{n}_{\text{shift by 5}}$

n i r $\underbrace{E}_{\text{shift by 5}}$

Step 2: Compression maps



Now we can assume every key k is an integer.

Need to make it between 0 & N−1 (not 0 and 2^{32}).

Goal: Find a "good" map.

"Good": - fast O(1)

- minimize collisions ☆

$$\begin{array}{r} 0 \text{ r. } \boxed{3} \\ 10\overline{)3} \\ \underline{0} \\ 3 \end{array}$$

Modular compression maps

Take $h(k) = k \bmod N$

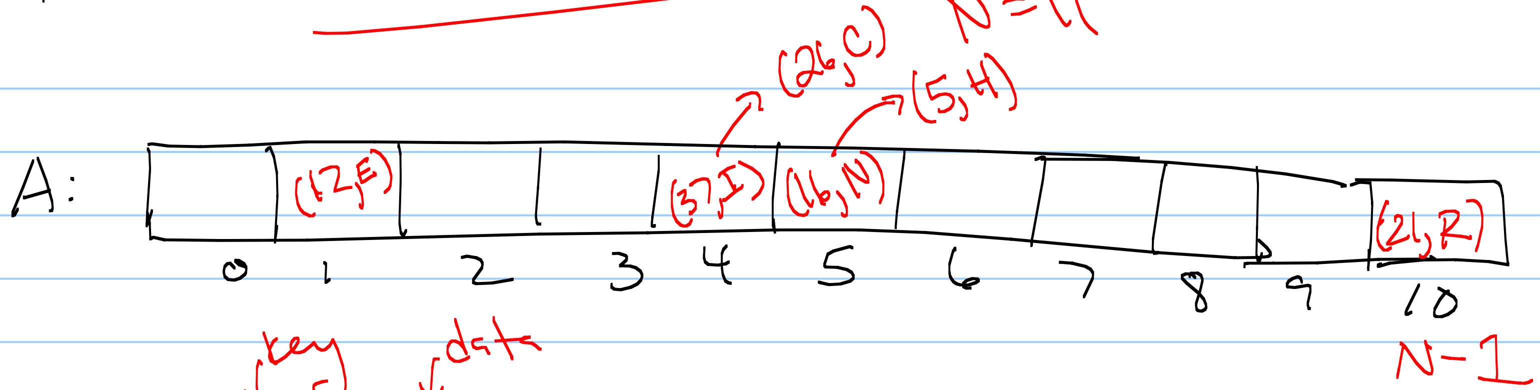
What does mod mean again?
→ remainder

$3 \bmod 10 = 3$

$50 \bmod 10 = 0$ % in C++

$14 \bmod 10 = 4$

Example: $h(k) = k \bmod 11$ $N=11$



Insert:

key data

(12, E)
(21, R)
(37, I)
(16, N)
(26, C)
(5, H)

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

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

$h(37) = 37 \bmod 11 = 4$

$h(16) = 5$

$h(26) = 4$

$h(5) = 5$

find? Compute $h(v)$ & then find in our "list".

remove? $h(v)$ & then delete in list

Some Comments:

This works best if the size of the table is a prime number.

Why?

Go take number theory & Cryptography

Collisions are more common the "less prime" a number is.

$$12 = \underbrace{2\cdot 2\cdot 3}_{}$$

Strategy 2: MAD, Multiply, Add & Divide

First idea: take $\underline{h(k) = k \bmod N}$

Better: $h(k) = |a k \underset{\nwarrow}{+} b| \; \underline{\bmod N}$

where a & b are:

- not equal
- less than N
- relatively prime : no common divisors

$\gcd(a,b) = 1$

$12 = 2 \cdot 2 \cdot 3$
$20 = 2 \cdot 2 \cdot 5$ } not relatively prime

$21 = 3 \cdot 7$
rel. prime w/ 20

(Why? Go take number theory!)

Example: $h(k) = |ak+b| \bmod 11$

$a = 3$

$b = 5$

A:

0	1	2	3	4	5	6	7	8	9	10
		(21, R)						(12, E)		

key data

Insert:

(12, E)
(21, R)
(37, I)
(16, N)
(26, C)
(5, H)

$h(k) = \overbrace{3\cdot 12+5}^{41} \bmod 11 = 8$

$h(21) = 3\cdot 21+5 \bmod 11 = 2$

$h(37)$ ...

Why bother? In practice, fewer collisions.

End Goal: Simple Uniform Hashing Assumption

For any $k \in$ key space,

$$\Pr[h(k) = i] = {}^{1}/_{N}$$

(Essentially, elements are "thrown randomly" into buckets.)

Collisions

Can we ever totally avoid collisions?

No

$$\underset{\overset{\|}{m}}{\text{key space}} \gg N$$

Step 3: Handle collisions (gracefully & quickly)

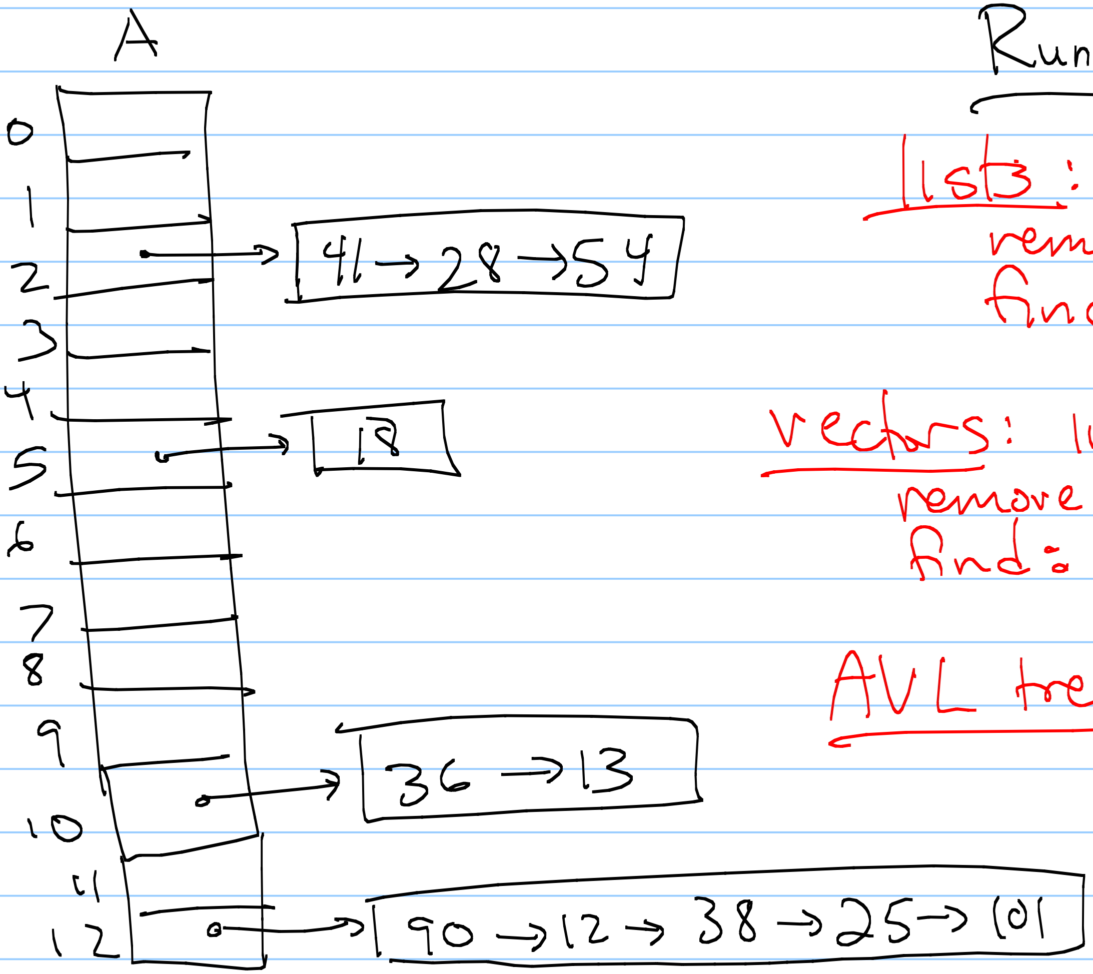
So how can we handle collisions?

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

- lists
- AVL trees
- vectors

Ex:

list → chaining



Running times:

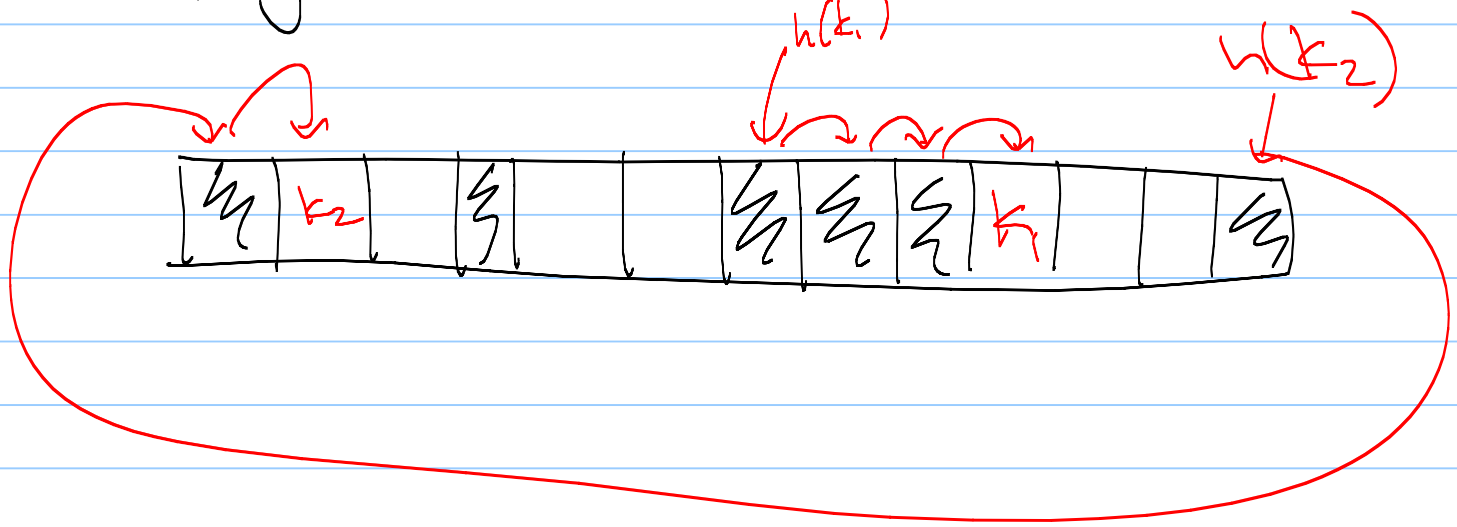
lists: insert: O(1)
remove: O(n)
find: O(n)

vectors: insert: O(1) amortized
O(n) worst case
remove: O(n)
find: O(n)

AVL trees: O(log n)

Linear Probing

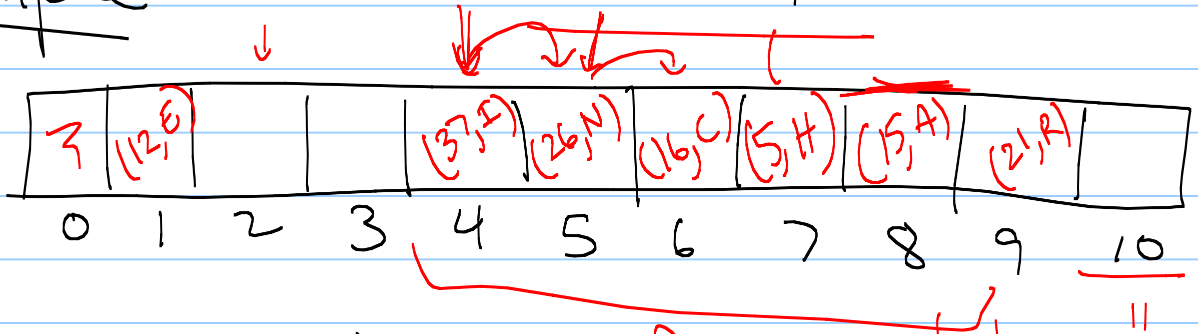
Instead of lists, if we hash to a full spot, just keep checking next spot (as long as the next spot is not empty).



Example

$h(k) = k \bmod 11$

$N > n$



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)	

primary clusters

$10 = N-1$

Insert:

(12, E) = 1

(21, R) = 21 mod 11 = 10

(37, I) = 4

(26, N) = 4

(16, C) = 5

(5, H) = 5

(15, A) = 4

find(15)

h(15) ← start at h(k) & keep looking until a blank

remove (37)

Issue

How can we remove here?

If you remove, create "gap" that linear probing won't know was full at time of insertion.

Solution: "dirty bit": if = 1, then this value has been deleted

If some fraction have dirty bit set, stop + rehash.

Running Time for Linear Probing

Insert: $O(n)$

Remove: $O(n)$ (since remove calls find, then sets a bit)

Find: $O(n)$

rehash: Allocate bigger table. For each entry table, compute $h(k)$

Quadratic Probing

Linear probing checks $A[\overbrace{h(k)+1} \bmod N]$ if $A[h(k) \bmod N]$ is full.

To avoid these "primary clusters", try:

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

where $j = 0, 1, 2, 3, 4, ..$

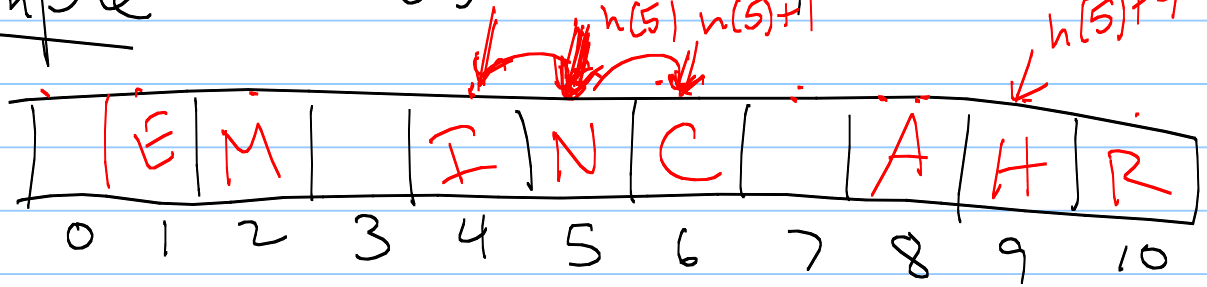
$h(k)$ full, check $h(k) + 1^2$

$h(k)+1$ full, check $h(k) + 2^2 = h(k) + 4$

$h(k)+4$ full → check $h(k) + 3^2 = h(k) + 9$

Example

$h(k) = k \bmod 11$



0	1	2	3	4	5	6	7	8	9	10
	E	M		I	N	C		A	H	R

Insert:

(12, E)	$h(12) = 1$
(21, R)	$h(21) = 10$
(37, I)	$h(37) = 4$
(26, N)	$h(26) = 4$
(16, C)	$h(16) = 5$
(5, H)	$h(5) = 5$, $h(5)+1^2$, $h(5)+2^2$
(15, A)	$h(15) = 4$
(4, M)	$h(4) = 4$

might actually fail

Issues with Quadratic Probing:

- Can still cause "secondary" clustering
- N really must be prime for this to work
- Even with N prime, starts to fail when array gets half full

(Runtimes are essentially the same)