CS 180 — Hashing

Note Title 4/27/2011

Announcements

- Checkpoint on Monday
  Program due Thursday

- HW will be up next week, due last day off class

- Review session: Friday of finals week at 10:30

Recap of trees

BSTs - insert, find & remove in O(n) time

AVL trees:

- insert, find, & remove in O(log n) time

Key idea: height-balance property

Other trees

- Splay trees: After every insert/delete, performs a move-to-root operation called splaying, which gives an <u>amortized</u> $O(\log n)$ behavior

- Red-Black trees: more complex than AVL trees, + give only $O(1)$ "rotations" after each insert or delete

New problem: Data Storage

Ex:

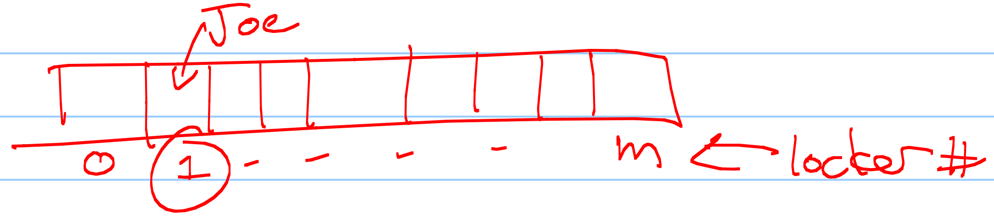
Locker #	Name
26	Dan
355	Kevin
101	Tracy
53	Nitish
201	David
⋮	⋮

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

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

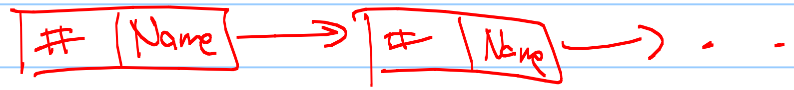
How could we store this?

1. Vectors



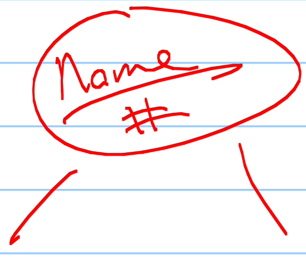
→ size: $O(m)$ find: $O(1)$

2. List



size: $O(n)$ find: $O(n)$

3. AVL tree



size: $O(n)$
find: $O(\log n)$

Other examples

- Course # and schedule info
- Flight # and arrival info
- URL and html page
- Color and BMP

Not always easy to figure out how to store/and look up.

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.

Data Structures

First thing to note:

An array is a dictionary.

key : index
data : value at a position

Other alternatives:

(go back 3 slides)

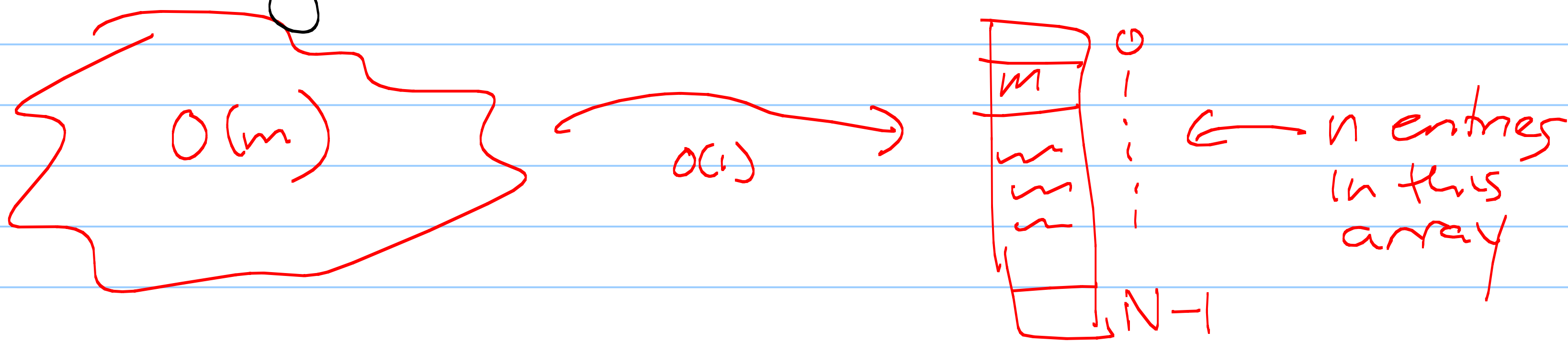
But these aren't good enough.

Hashing

Assuming $m > n$, an array is not very space efficient.

We would like to use $O(n)$ space, not $O(m)$.

But then the key needs to get smaller.



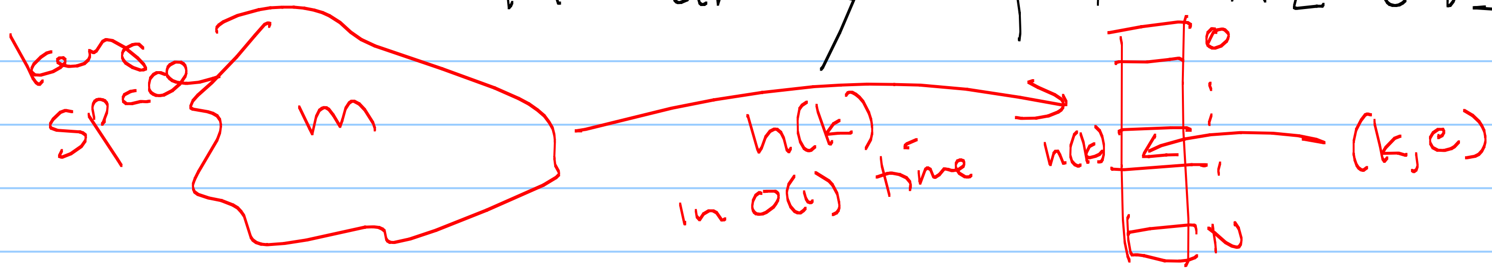
$m > N > n$

Dfn: A hash function h maps each key in our dictionary to an integer in the range $[0, N-1]$.

$N \neq n$

(N should be much smaller than m = # of keys.)

Then given (k,e), we store (k,e) in array spot $A[h(k)]$.

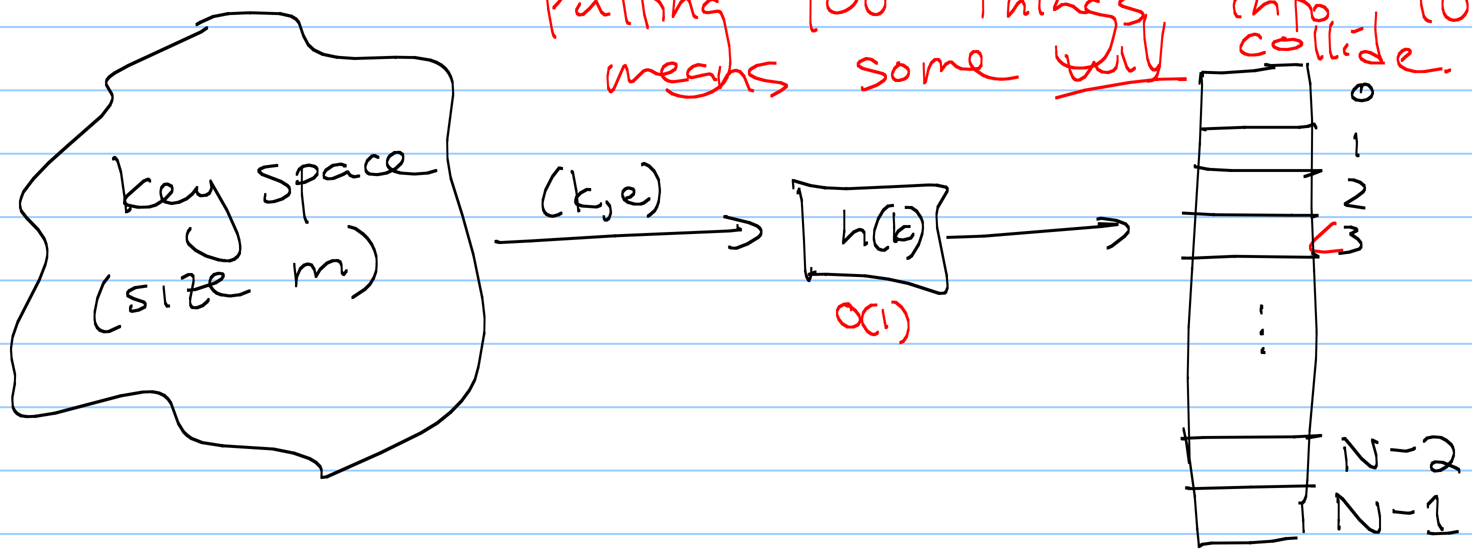


Good hash functions:

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

↳ are unaviodable: Putting 100 things into 10 spots means some will collide.

$(k_1, e_1) \neq (k_2, e_2)$ but $h(k_1) = h(k_2)$



minimize collisions & deal w/ them if they happen

So we have a few steps.

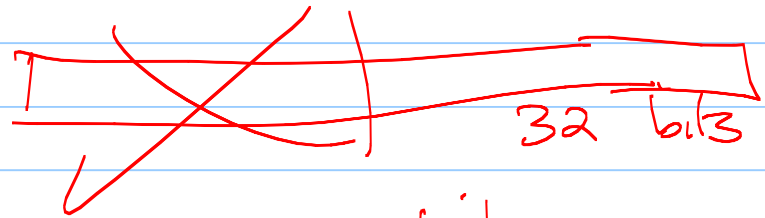
(goal 32-bit key)

1 Take k and make it a number.
(Remember, keys can be anything!)

Ex: char, int, or short (all 32-bits)

ASCII — already there — already there

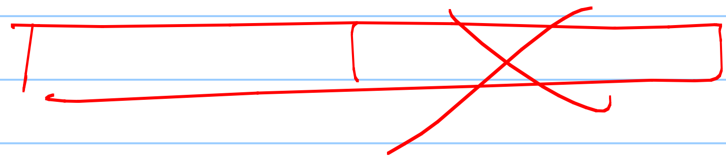
Ex: long or float — 64 bits
(k needs to be 32 bits)



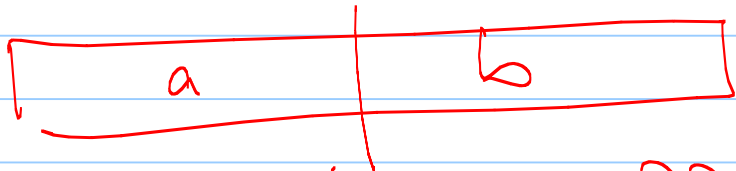
32 bits

potentially lots of collisions

64 bits



3



a+b is 32-bits

3 int hashCode (long x) {

shifts over 32 bits

```
    return int(unsigned long(x >> 32)
                          + int(x)) ;
}
```

cuts off those 32 bits

$O(1)$

What about strings?

(Think ASCII.)

Erin

69 + 114 + 105 + 110 = 32-bits

Goal: a single int.

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

temp01 and temp10 and pm0te1

all hash to same #

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

A Better Idea: Polynomial Hash Codes

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

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

Ex: Erin with $a = 37$

69 114 105 110

nirE

$h(\text{Erin}) \neq h(\text{nirE})$

$$h(k) = 69 \cdot 37^3 + 114 \cdot 37^2 + 105 \cdot 37 + 110 \cdot 1$$

Side Note: How long does this take?

(In terms of $k = \#$ of parts)

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

k additions

$2k$ multiplications $\rightarrow$ k^2 additions
(k^2 mult!)

Horner's rule: $x_{k-1} + a(x_{k-2} + a(x_{k-3} + \cdots))$

$\hookrightarrow$ k additions + k multiplications

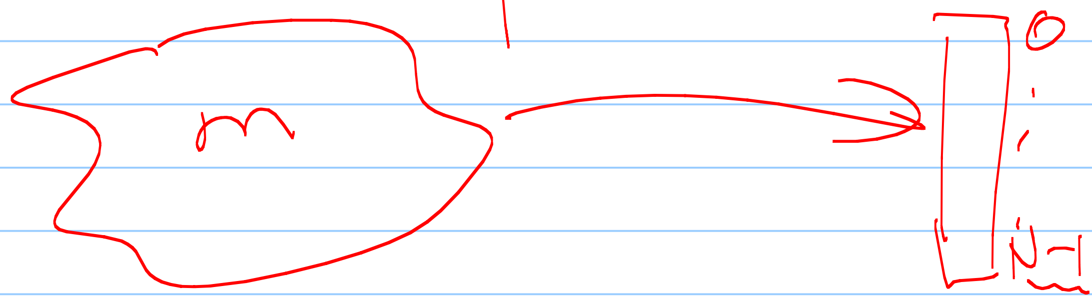
Polynomial Hashing

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

(Works for floats, strings, etc.)

What about overflow? take modulo

all these exponents mean a big #



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.

modulo or cyclic shifts are commonly used

key space
letters, #s,
whatever

#1

int
(32 bits)

h

#2

[0, ..., N-1]

#3 collisions

Step 2: Compression maps

Now we can assume every key k is an integer.

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

Ideas:

- map everything to 0

Why is that bad?

one giant collision

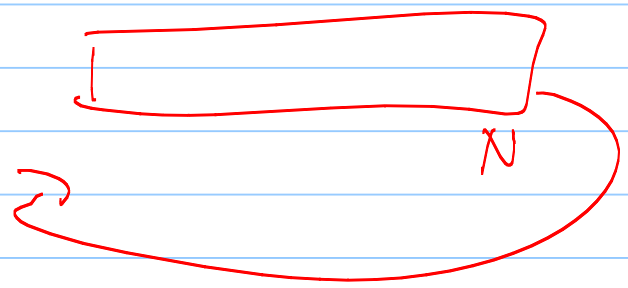
Modular compression maps

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

What does mod mean again?

(like we used in leaky stacks)

remainder



While good, strange behaviors can happen.