CS180 – Hashing

Note Title 4/27/2011

Announcements

- Program due tomorrow
- Next program is posted
- Next HW will be due the last day of class. It will be pass/fail.
- Review in class on Monday (the last day)

Data Storage

Ex:

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

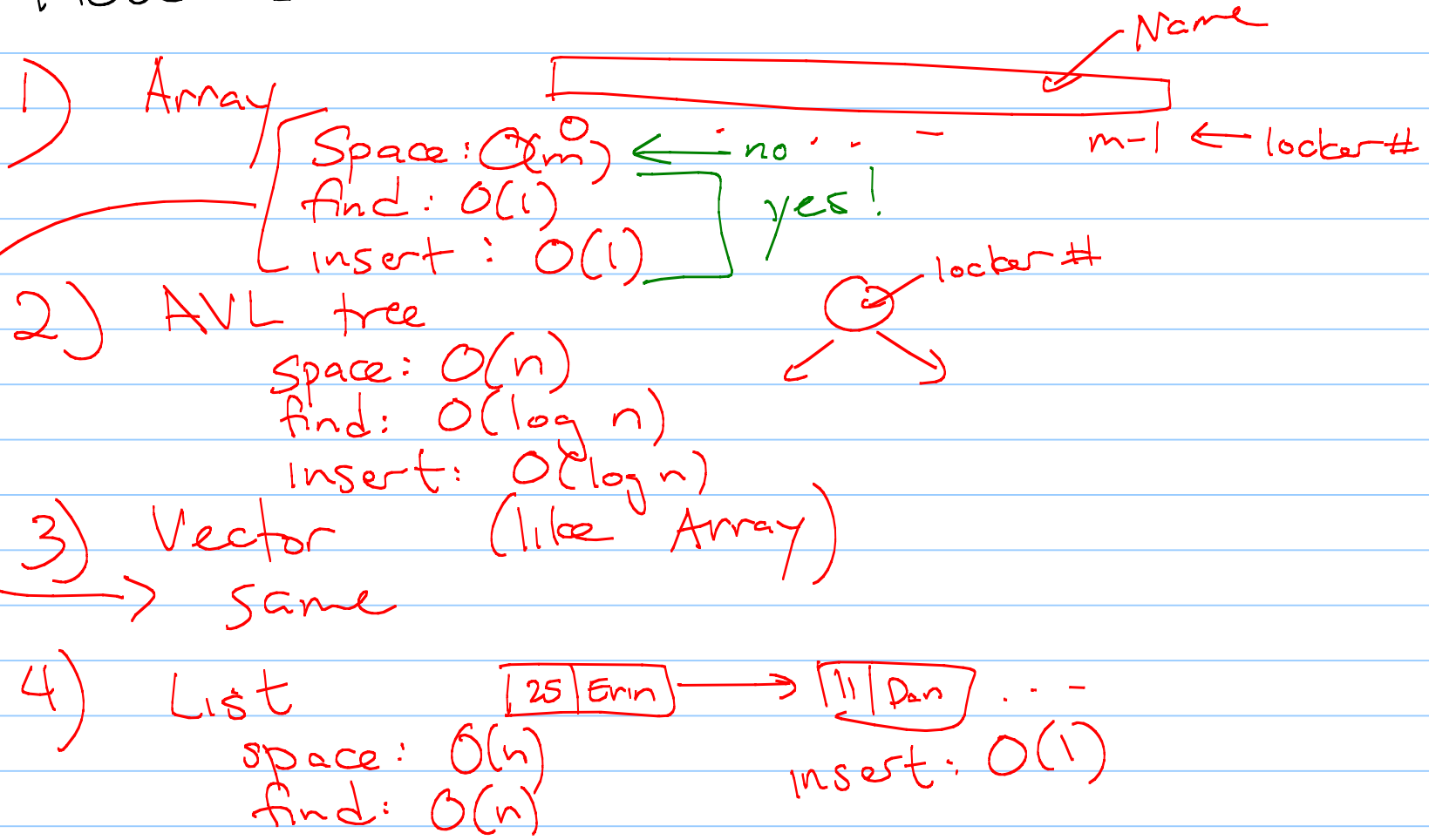
We want to be able to retrieve a name quickly when given a locker number.

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

$n \geq m$

space, find, insert/remove

How could we store this?



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

(k, e)

A data structure which supports the following:

keys — data or element

2 data types

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

Note: Everything is based on keys!

Many possible implementations —

trees, lists, vectors, ...

Data Structures

First thing to note:

An array is a dictionary.

key: array index

data: elements

Other alternatives:

(see prev. slide)

goal: O(n) space

O(1)

↑

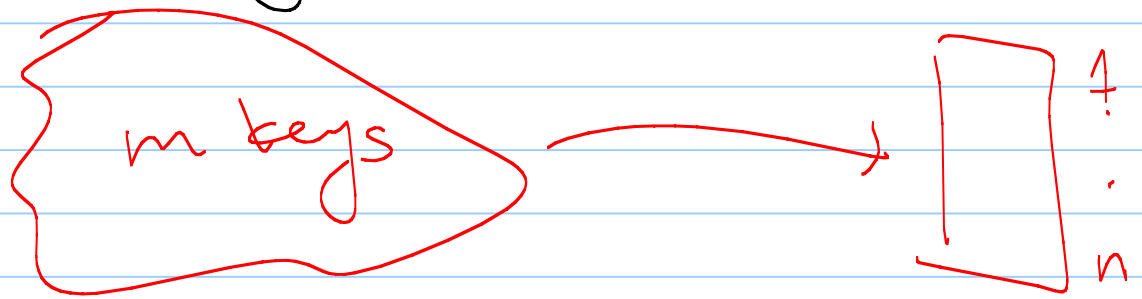
insert, find, remove

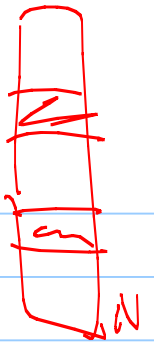
Hashing

Assuming $m \gg n$, 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.





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

usually $n < N < 2 \cdot n$

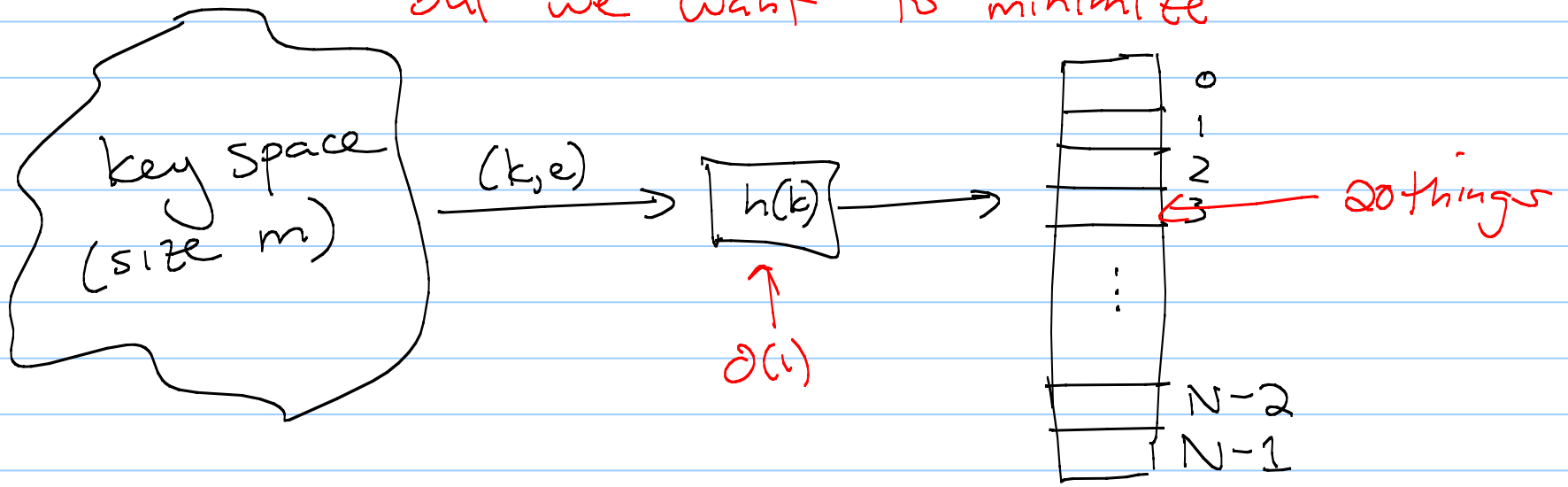
(N should be much smaller than m = # of keys.)

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

Good hash functions:

- Are fast — 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



So we have a few steps.

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

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

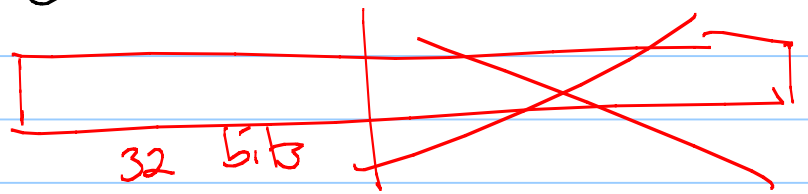
↓ ASCII #

↓ int

↓ Convert (without truncating)

Ex: long or float — 64 bits

(K needs to be 32 bits)



64 bits

← could give lots of collisions

X + Y

or XOR

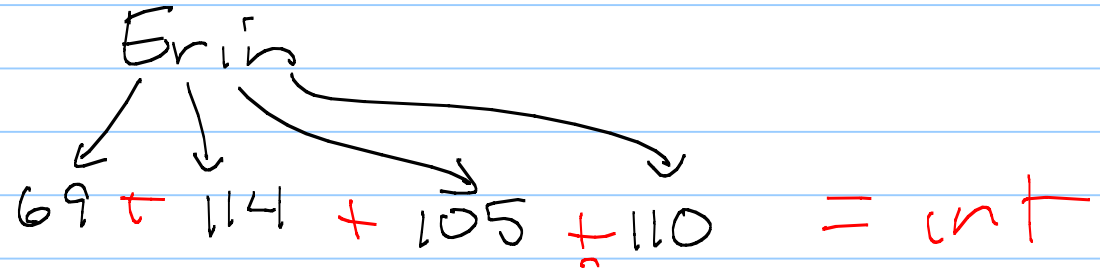
```
int hashCode(long x) {
    return int(unsigned long(x >> 32)
               + int(x));
}
```

keeps least significant bits

keep most sig bits

What about strings?

(Think ASCII.)



Goal: a single int.

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).