

CS 2100 - Hashing

Note title

4/27/2011

Announcements


- HW due next Tuesday
- The next HW is over Huffman trees
- Final is Sat on Wed (first day of finals)
 - ↳ If you have a conflict, tell me next week!

News problems: Data Storage

Ex:

Locker #	Name
26	Dan
355	Kevin
101	Tracy
53	Nikhil
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 Vector (or array):
create a vector of strings of names
Size $O(m)$

Lookup: $O(1)$
Insert/Remove: $O(m)$

② List

Size: $O(n)$
Lookup: $O(n)$
Insert/Remove: $O(n)$ (find)

③ Balanced BST (AVL):



$O(\log n)$

Other examples

- Course # and Schedule info
- Flight # and arrival info
- VBL and html page
- Color and BMP
- **directors** **movies**

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

Dictionaries

A data structure which supports the following:

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

Note: Everything is based on keys!

Data Structures

First thing to note:

An array is a dictionary.

key: index

value: whatever is in array

Other alternatives:

(go back 3 slides)

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.

~~*~~: $O(1)$ lookup

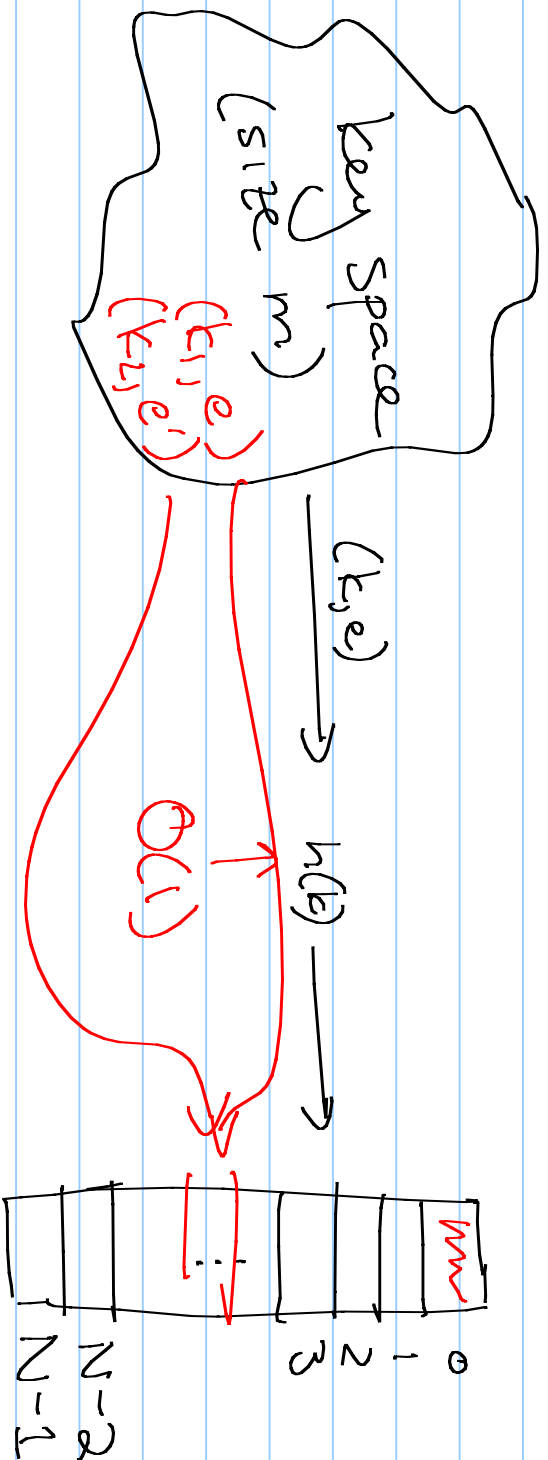
Defn: A hash function h maps each key in our dictionary to an integer in the range $[0, N-1]$.

(N should be much smaller than $m = \#$ of keys.)
or $m < N$

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

Good hash functions:

- Are fast
- Don't have collisions:
 - 2 keys $k_1 \neq k_2$
 - but $h(k_1) = h(k_2)$



So we have a few steps.

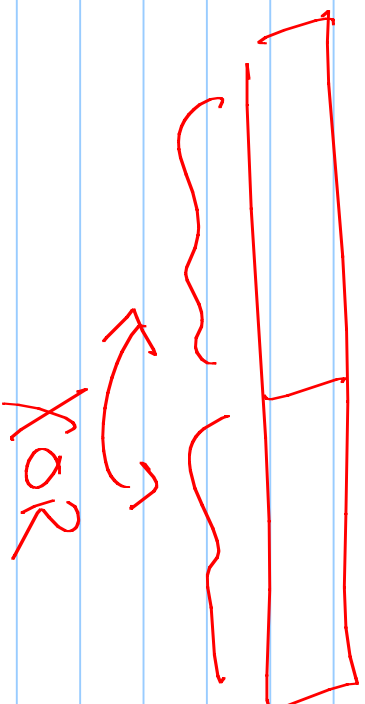
- ① Make key a number
↳ w/ few collisions
- ② Compress that number to $[0, N-1]$
↳ quickly + w/ few collisions
- ③ Since not perfect, handle collisions somehow.

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

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

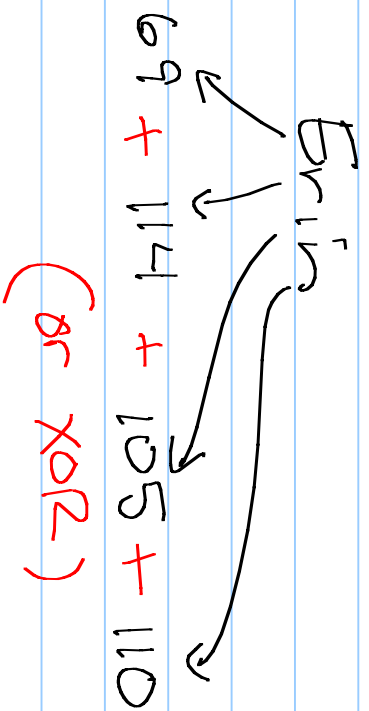
↳ just cast to a #

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



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

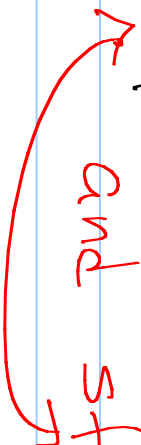
What about strings?
(Think ASCII.)



Goal: a single int.

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

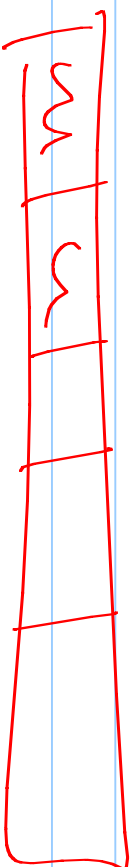
temp01 and temp10 and pm0te1
and sfmp01



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: $x = (x_0, x_1, x_2, x_3, \dots, x_{k-1})$



$$\text{Let } p(x) = x_0 a^{k-1} + x_1 a^{k-2} + \dots + x_{k-2} a + x_{k-1}$$

Ex: $\begin{matrix} 69 & 105 & 114 & 110 \end{matrix}$ with $a = \underline{37}$

$$p(x) = 69 \cdot 37^3 + 105 \cdot 37^2 + 114 \cdot 37 + 110$$

Side Note: How long does this take?

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

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

k additions

$2k$ multiplications

Alternate idea:

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

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?

Chop down & XOR

Cyclic Shift hash codes - faster

Alternative to polynomial hashing

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

Also works well in practice.

