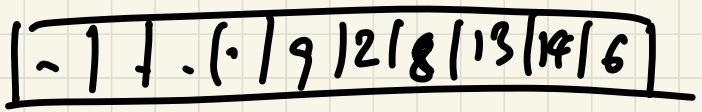
BST → O(logN)

O(1) ?

O(N)



Hash maps

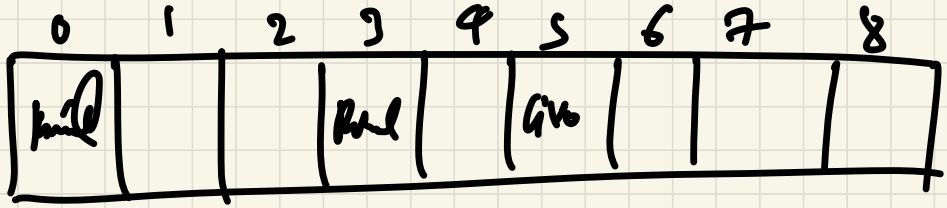
- Why?
- How it works?

Key (name)	value (marks)
✓ Kunal	88
✓ Karan	99
✓ Rahul	95

map.get("Kunal")

→ return the value = 88

in O(1)



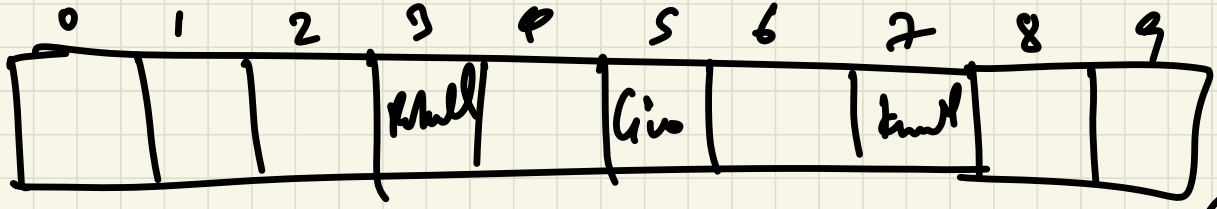
Hashcode

① We need all elements as the nos. → hashcode function

② Hashcode → very large

→ reduce it → Hashing

reduce all elements in table to a size m.



m = 10

"Civo" ⟶ hash("civo") = 739985 % 10 = 5

hash("kumi") = 381947 % 10 = 7

hash("Rahul") = 239873 % 10 = 3

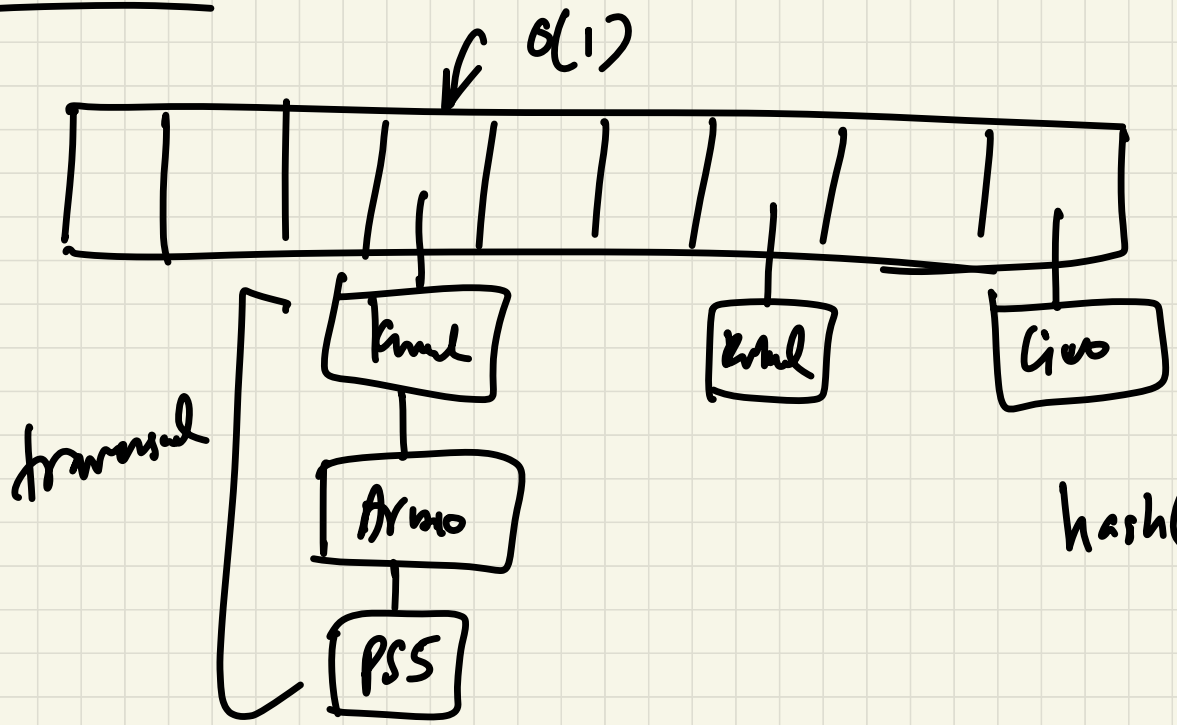
Get the item:

hash("Kumi") = 3 ⟶ O(1)

hash("Arnav") = 394783 % 10 = 3

Collision

2 ways: (1) Chaining (2) open addressing



hash(Gino) = 3

We about to

Simple uniform hashing:

Assumption:

n = no. of keys in table

m = size of table

load factor = $\alpha = \frac{n}{m}$ = expected key per slot

$O(1+\alpha)$

linked list

$\alpha = O(1) \Rightarrow m = \Omega(n)$

$= O(1)$ Time complexity

Hash functions:

① Division method:

$$h(k) = k \,\%\, m$$

m = size of array.

m = prime no. (not too close to power of 2 or 10) → common

② Multiplication method:

$$h(k) = \left[(a \cdot k) \,\%\, 2^w \right] >> (w - r)$$

a = random number
w = no. of bits in k
$m = 2^r$

a is odd
$2^{w-1} < a < 2^w$
a is not too close to 2^{w-1} or 2^w

Universal hashing:

$$h(k) = [(ak+b) \% p] \% m$$

a & b are random $\in [0, 1, \ldots, p-1]$
p is a large prime no.

$$P\left[h(k_1) = h(k_2)\right] = \frac{1}{m}$$

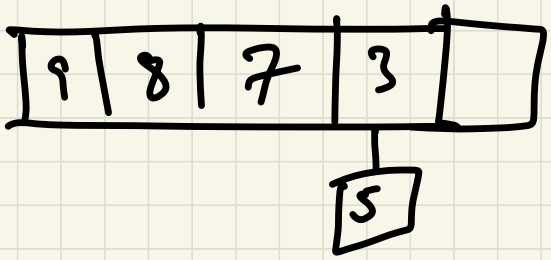
Size of table:

$m = \Theta(n)$ **all the time**

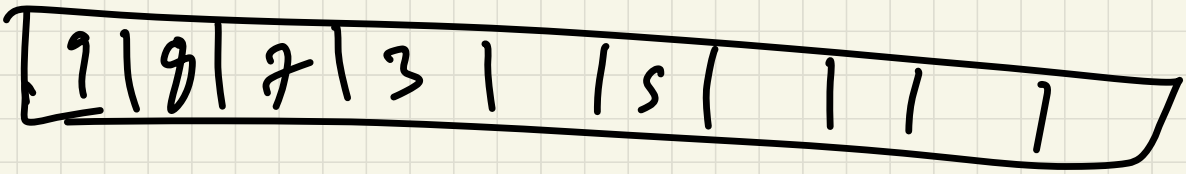
small → slow

big = wasteful

Idea: Start small & then grow



double the size



when, $n = m$,

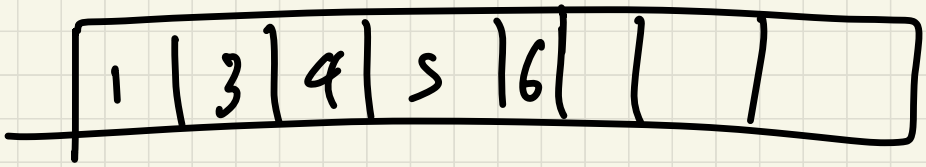
① $m+1$, $n = O(1+2+3+4+5+6+7+\dots n)$

$= O(n^2)$

② multiplying by 2 → double the size

$m*=2$ | $O(1+2+4+\dots+n)$

$= O(N)$



when you double the table, cost to insert n items = $O(N)$ "average"

1 item $\Rightarrow$ $O(1)$ ⟶ amortized constant time

Shrinking

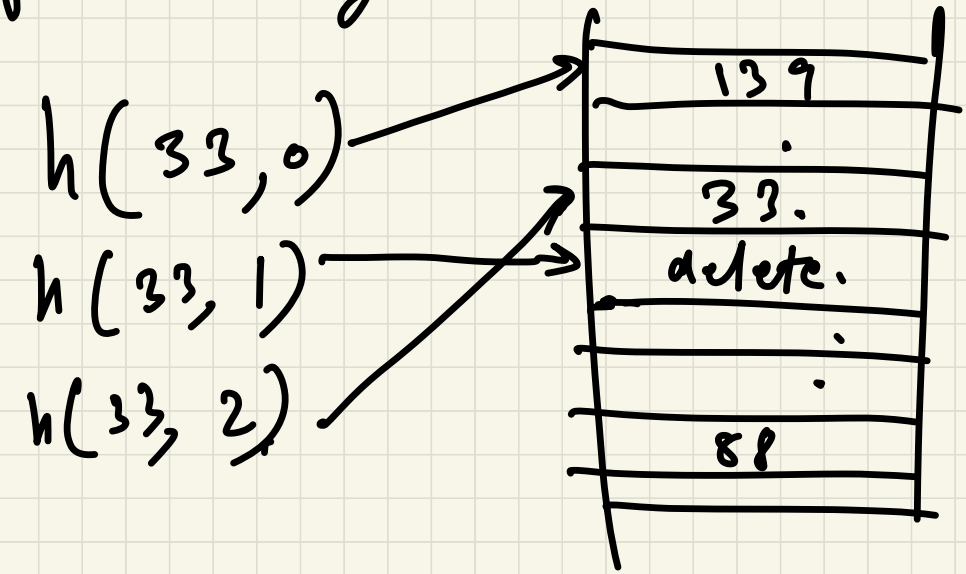
$m = \frac{n}{2}$, shrink by $\frac{m}{2}$ $O(N)$ per operation.

$n = \frac{m}{4}$ ⟶ half the size $O(1)$ amortized time

Open addressing:

One item per slot $\Longrightarrow$ $m \geq n$

probe → try

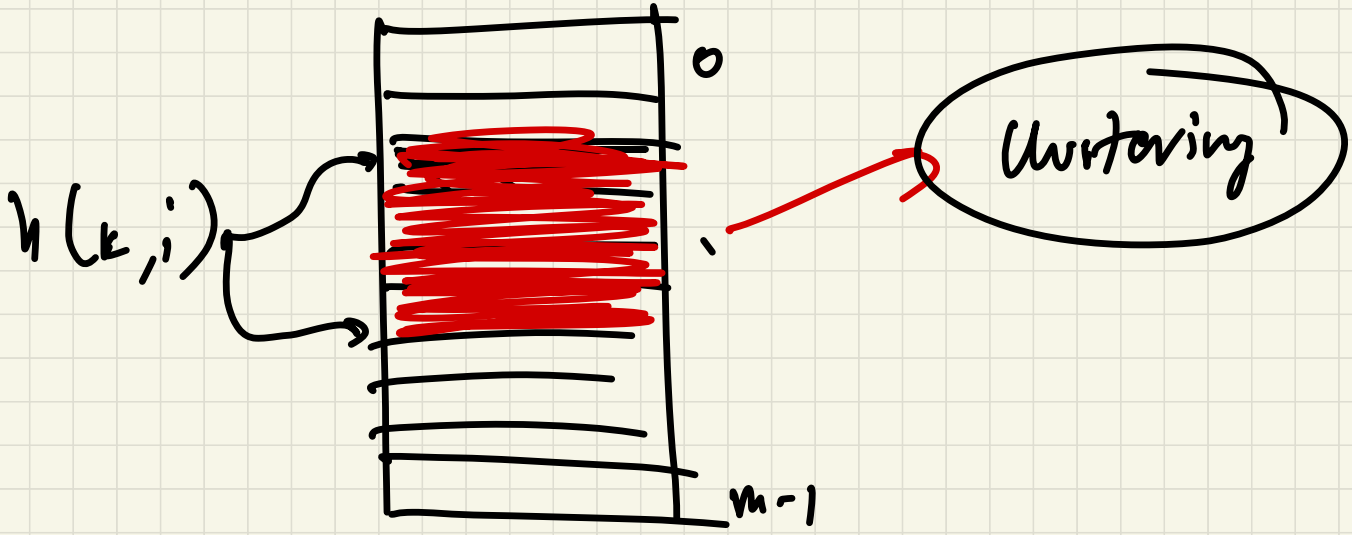


139 → 96 → 33

139 → delete → 33

Probing Strategies:

① Linear probing, $h(k,i) = (h(k) + i) \% m$



⑧ Double hashing:

extra $h(k,i) = (h_1(k) + i * h_2(k)) \,\%\, m$

→ If $h_2(k)$ is relative prime to m for all k.

$h_1(k) + i * h_2(k) \,\%\, m = h_1(k) + j * h_2(k) \,\%\, m$

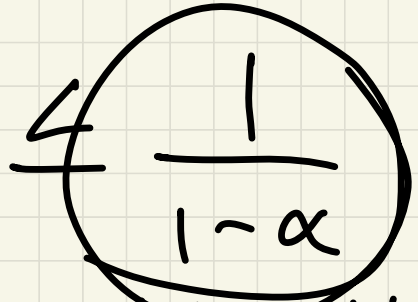
$\Rightarrow$ m divides $(i-j)$

eg: $m = 2^r$, make $h_2(k)$ always odd

$2^n 4 = 16$, $h_2(k) = 5$

Uniform Hashing Assumption:

Every key is equally likely to have $n!$ permutations.

cost of next operation $\leq \frac{1}{1-\alpha}$  , $\alpha = \frac{n}{m} < 1$

$\alpha = 90\% \Rightarrow$ 10 expected probes.

1st success $= \frac{m-n}{m} = p$

2^{n} success $= \dfrac{m-n}{m-1} \geq \dfrac{m-n}{m} = p$

$p = 1 - \dfrac{n}{m} \geq 1-\alpha$

3^{rd} success $= \dfrac{m-n}{m-2} \geq p$

Expected trials $= \dfrac{1}{p} = \dfrac{1}{1-\alpha}$

delete $G\left(\dfrac{1}{1-\alpha}\right)$

When to use which?

OA → better cache performance (ptrs not needed)

chaining → less sensitive to hash functions.