HashMap & HashTable

* BST → O(log N)

How to get element in constant amount of time? O(1)?

→ Using HashMaps → why?
→ How it works?

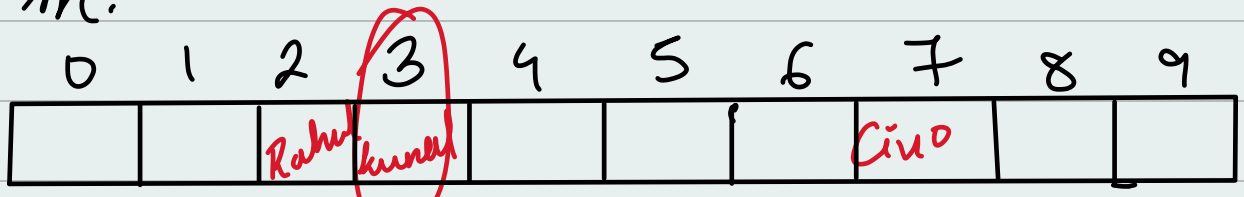
key (Name)	value (Marks)
Kunal	88
Karan	99
Rahul	95

map.get("Kunal")
→ return the value
= 88
in O(1)

→ To convert keys to numbers we use HashCode function

→ ① We need all elements as positive numbers → Hashcode function.

→ ② HashCode can be very large but reduce it by hashing. It reduce all elements in table to a size m.



"Civo" → hash("civo") = 7

m = 10

hash("kunal") = 3

hash("Rahul") = 2

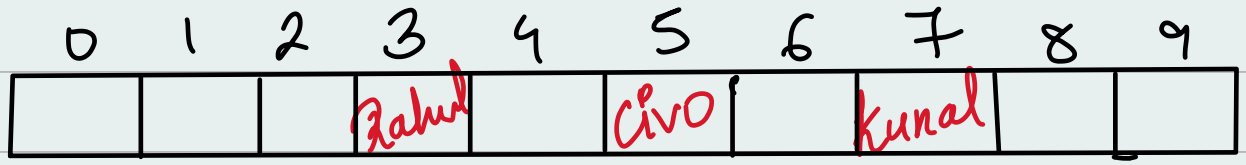
Get the item : hash("kunal") = 3

amortized constant time
O(1)

But in code it does not give us the small integer number it gives us like 7328879 or 7288356 so

how to convert this big numbers into small number? We can do it by hashing (

→ hash ("civo") = 734985 % 10 = 5
hash ("Kunal") = 381347 % 10 = 7
hash ("Rahul") = 239873 % 10 = 3



↳ in constant time

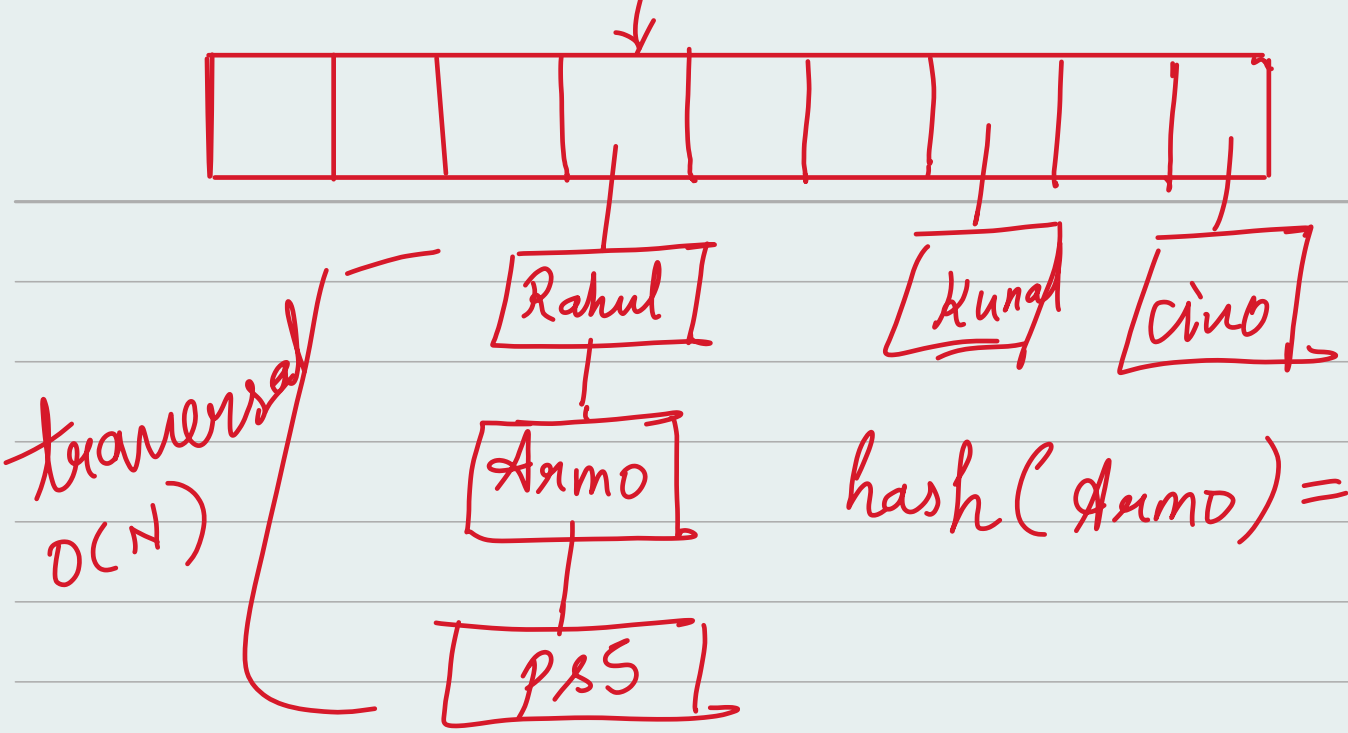
hash ("Armo") = 394783 % 10 = 3

But in index 3 rahul is already there.
This is known as collision

2 ways : ① chaining ② open addressing

→ chaining is basically like linked list

O(1)



hash(Armo) = 3

worst case all the names belongs to same index only with n chained linked list names

How to solve this problem?

We cheat a bit by using simple uniform hashing:

Assumption: n = no. of keys in table
m = size of table

load factor = $\alpha = \frac{n}{m}$ = expected no of keys per slot

→ It means time complexity is $O(1+\alpha)$

assuming that maximum for LL is 2.

→ If α itself is $O(1)$ then above formula would also be $O(1)$

$\alpha = O(1) \Rightarrow$ This happens when the size of the table (m) is $\Omega(n)$

Hash Functions :

① Division Method :

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

↳ size of array

m = prime number (but not too close to the power of 2 or 10)

↳ because these are common in real world

② Multiplication Method :

$$h(k) = \lfloor (a \cdot k) \% 2^w \rfloor >> (w - r)$$

a = random number

w = no of bits in k

$m = 2^r$

Many time's multiplication method is

used because it is faster than division method.

This is practicall when a is odd number and also a is not to close to 2^{w-1} or 2^w.

& $2^{w-1} < a < 2^w$

Universal hashing:

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

a & b are random numbers that belongs $[0, 1, \ldots, p-1]$

p is also a large prime number.

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

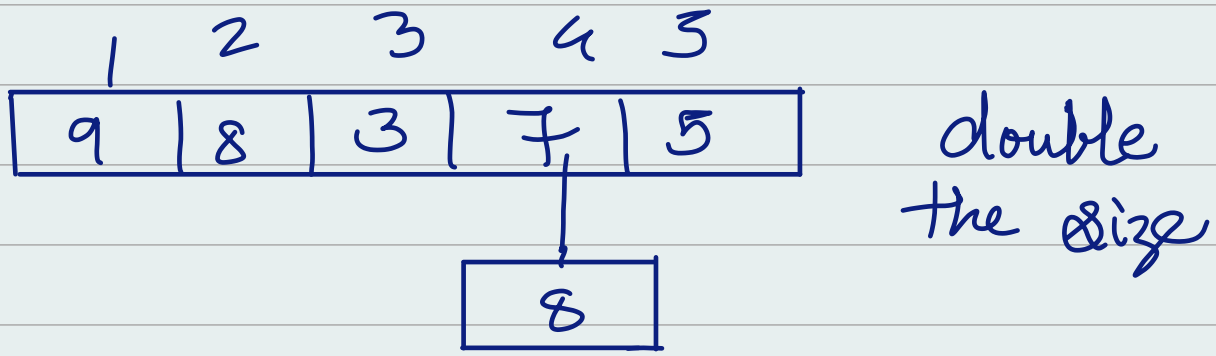
→ How large or size of the table should be?

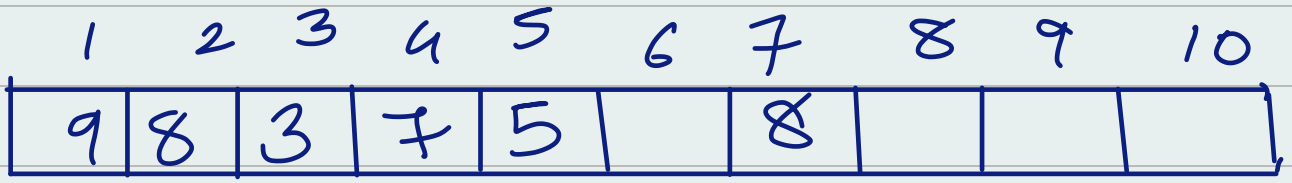
Size of the table: $m = O(n)$ all the time

If m is small → slow

but if m is big → resource wasteful.

Idea: start small & than grow





→ If the size is increase by 1 (m+1) when number of elements = size of the array. [n = m]

(1) So in every step when we add new elements it comes out to be $O(N^2)$

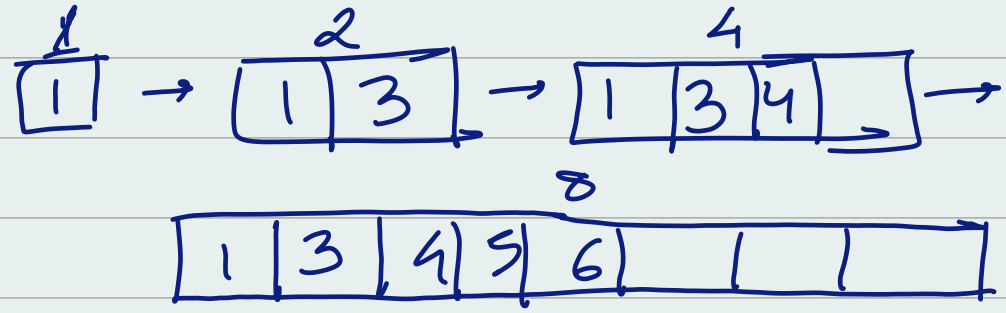
number of elements $(n) = O(1+2+3+4+5+ \ldots n)$

$= O(n^2)$

(2) So better way to do is multiply the size that is doubling the size

do size = 2
↳ m* = 2

O (1 + 2 + 4 + 8 + 16 .. n)



→ O(N)

When you double the table the cost to insert n items comes out to be O(N). "average"

Inserting
1 item ≈ O(1) → amortize constant time.

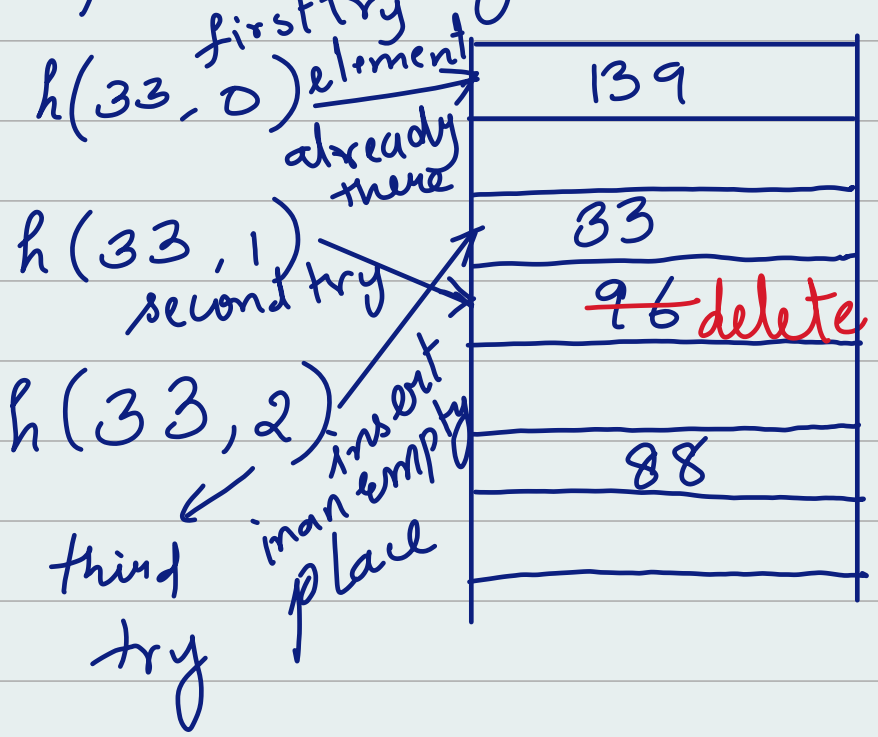
Shrinking
$m = \frac{n}{2}$, shrink by $\frac{m}{2}$
O(N) per operation.

when number of elements (n) $= \frac{m}{4}$ becomes
↳ then half the size
O(1) amortized constant time.

Open Addressing

only 1 item per slot $\Rightarrow m \geq n$

probe → try to insert an item



when again searching for 33 it will go to 139 then, 96 and then 33

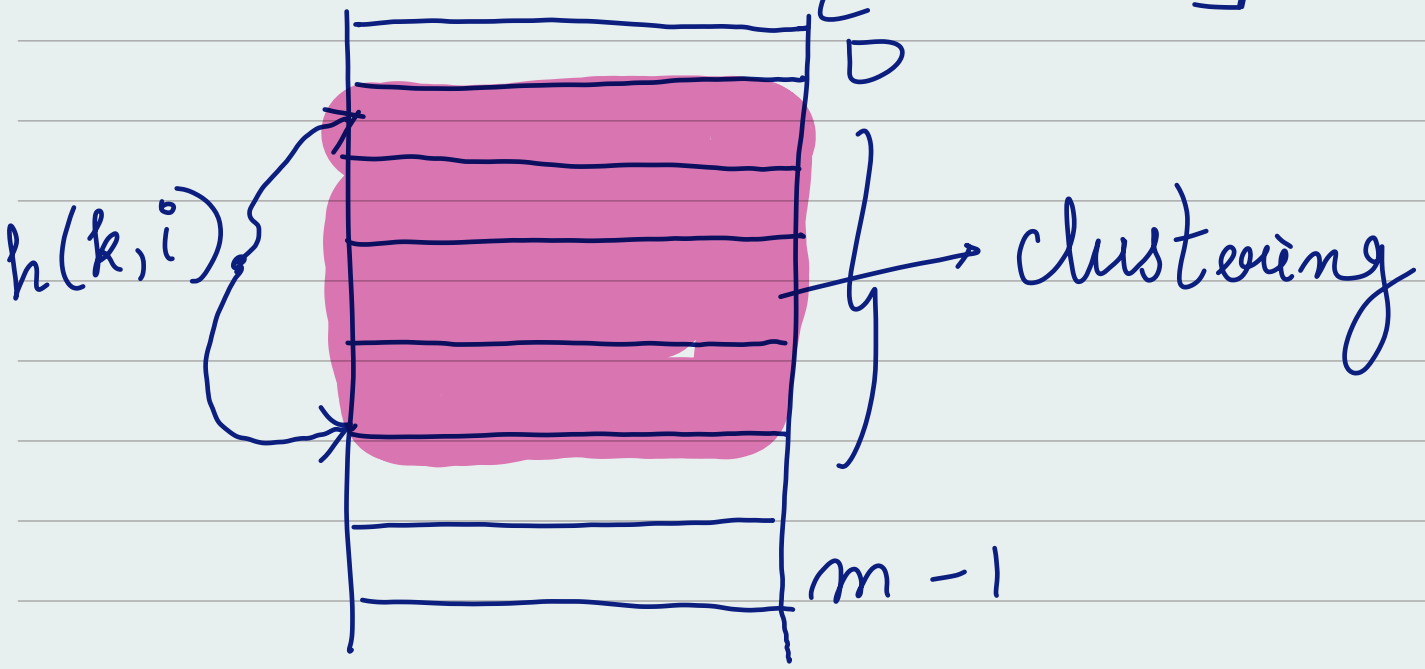
$$139 \longrightarrow 96 \rightarrow 33$$

Suppose we want to delete 96 we have to put a flag or pointer there

$$139 \rightarrow \text{delete} \rightarrow 33$$

Probing strategies :-

i) Linear Probing, $h(k,i) = [h(k) + i] \% m$



② Double hashing :-

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

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

Size of the array $2^4 = 16$, $h_2(k) = 5$

* Uniform Hashing assumptions:-

Every key is equally likely to have $m!$ permutations as the probe sequence

Cost of next operation $\leq \frac{1}{1-\alpha}$,

$$\alpha = \frac{n}{m} < 1$$

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

first succes $= \frac{(m-n)}{m} = p$ (empty slots)

second success $= \frac{(m-n)}{(m-1)}$ (remaining slots) $\geq \frac{m-n}{m} = p$

(Total - 1)

$$p = 1 - \frac{n}{m} = 1 - \alpha$$

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

Expected trials $= \frac{1}{p} \leq \frac{1}{1-\alpha}$

Delete will also take $O\left(\frac{1}{1-\alpha}\right)$

When to use which operations?

Open addressing → better cache performance (better memory usage) pointers not needed.

Chaining → less sensitive to hash functions