→ Java → C++
HashMap / HashTable / Map

↳ **unordered** collection of **key-value** pairs

↳ Array ✓
Linked list ✓
BT ✓

↳ insertion
search
deletion
} **in avg.** in O(1)

↳ KV ⇒ Mapping b/w two entities

↳ mapping b/w word → meaning
(key) (value)

↳ Food Ordering
mapping b/w food item ↔ price
(key) (value)

1. Search (lookups) should be efficient
2. Order is not imp.

How do we implement a hash table?
↳ fixed size

Array

How?

indices ⇒ key
data ⇒ value

data

key	value
✓ 0	0 ✓
✓ 1	1
✓ 2	4
✓ 3	9
✓ 4	16

0 1 2 3 4 ....
T | 0 | 1 | 4 | 9 | 16 | ... |
← ts →

insertion: T[key] = value ⇒ O(1)
search: value ← T[key] ⇒ O(1)
deletion T[key] = -1 ⇒ O(1)

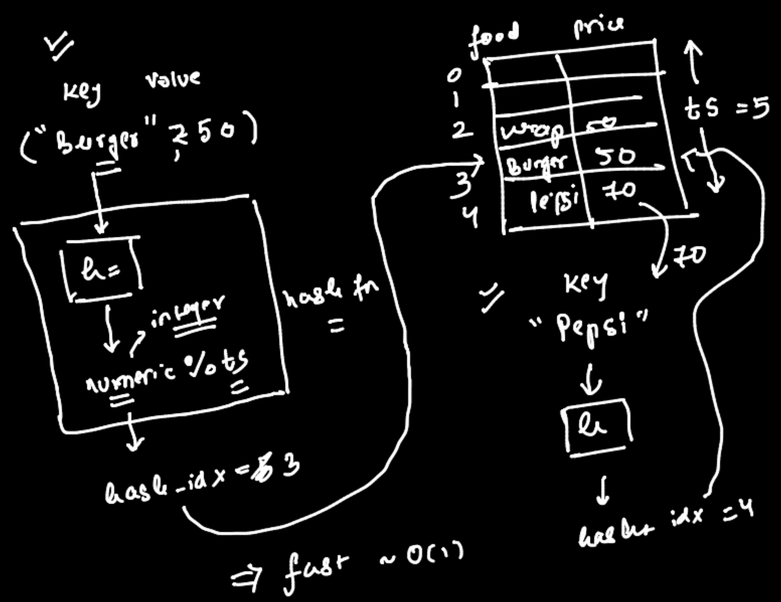
key → **large numeric value** > ts

simple fn → % ts
0 to ts-1

h(key){
return key % ts
}

→ string

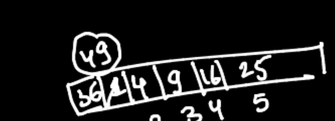
↳ hash fn : key → valid indices
hash idx



⇒ fast ~ O(1)

collision

T



key = 1 % 6
hash = 1

	key	value
⇒	1 =	1 =
⇒	2	4
⇒	3	9
⇒	4	16
⇒	5	25
⇒	6	36
⇒	7 =	49

hash fn ⇒ %

1 % 6 = 1
2 % 6 = 2
3 % 6 = 3

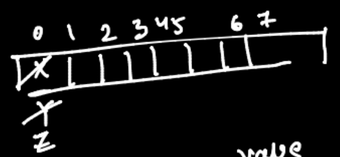
7 % 6 = 1

key ⇒ 1

⇓

two or more <kv> will be mapped to the same idx / bucket ⇒ collision =

Pigeon-hole



key	value
10 =	X
20 =	Y
30 =	Z

hash fn

% 10

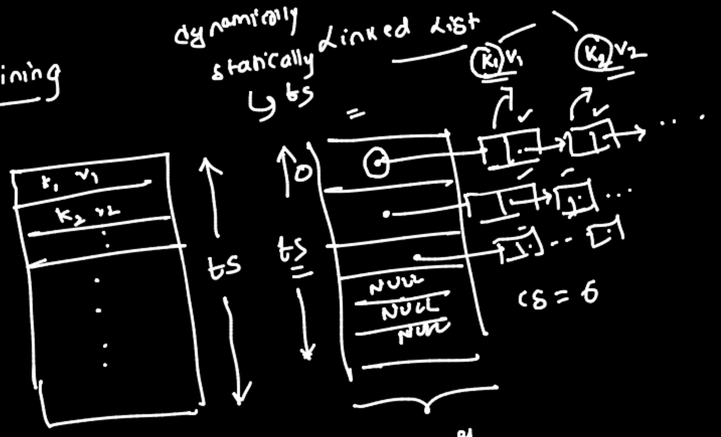
How to resolve collision ⇒

⇒ ↳ Open Hashing or Chaining

⇒ { ↳ Open Addressing or Closed Hashing
  ↳ Linear probing
  ↳ quadratic probing
  ↳ Double Hashing

Chaining

dynamically / statically Linked list
↳ bs



node * T[bs]; ≡
node ** T = new node *[bs];

[<type> * arr = new type [size]]

int arr[10];
int * arr = new int[10];

Static
int * arr[10];
node * T[bs];

Dynamic
int** arr = new int*[10]
node ** T = new node *[bs]

How chaining influence TC?

insertion

(K2, V2) ↑ hash-idx i

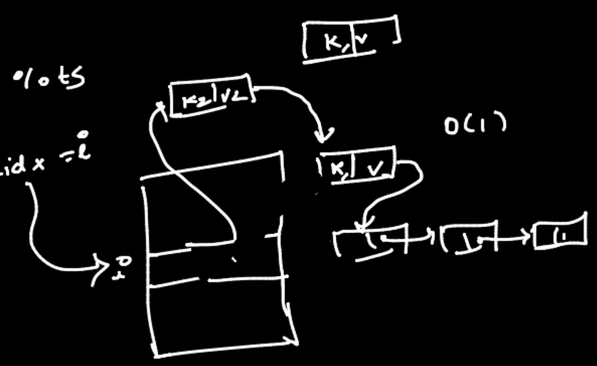
(K, V)

↓

hash fn % bs

↓

hash-idx = i



O(1) + O(1)

⇒ O(1)

search ts cs

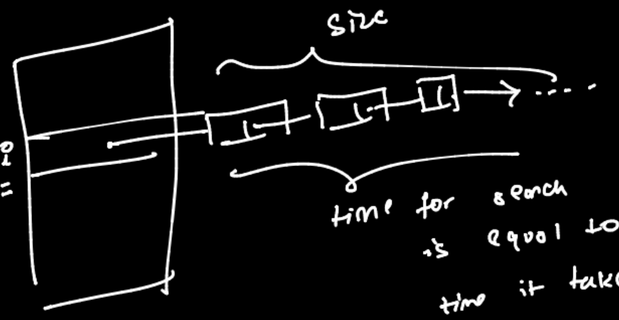
⇒ (K)

↓

(hash fn) ·/· ts

↓

bucket_idx (i)



O(size)

cs > ts O(size)

worst case analysis

↳ best case

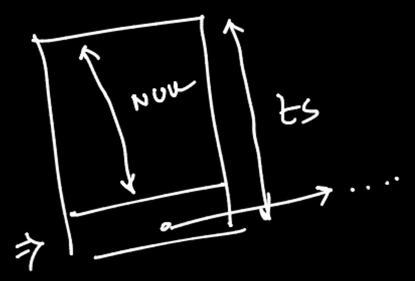
↳ worst case

⇒ if key are dist uniformly across the table

size = cs/ts ← load factor

O(cs/ts)

Dict is not ordered



size = cs

O(cs)

good hash fn

↳ fast

↳ distribute the keys uniformly

⇓

minimize collision

insertion : O(1) ↗ good hash fn

search : O(cs/ts)

deletion : O(cs/ts)

1. Sum of ascii of characters in a string

"abc" → (97 + 98 + 99) %ts

↳ fast

Disadv

↳ Anagrams will collide

"abc"

"bac"

"cab"

↳ No

=

Size of each key 3

=

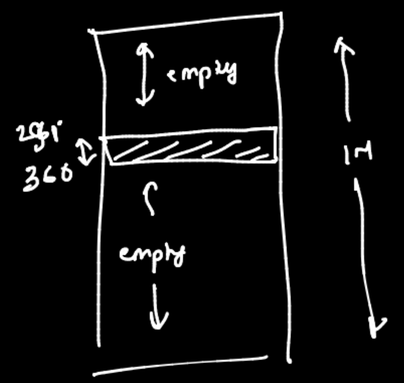
hash value { aaa ⇒ 97 + 97 + 97 = 291 %. 1M

=

zzz ⇒ 122 + 122 + 1222 = 366 %1M

Excess idx

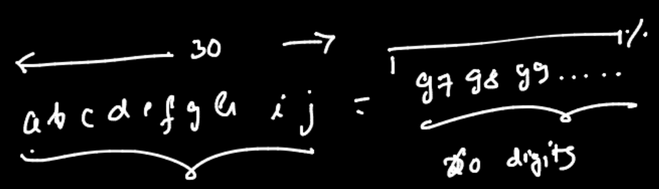
⇒ 291 to 366



(1) Concate (int) characters

abc ⇒ 979899

bac ⇒ 989799



(3.) ⇒ (1 0 1)$_2$ → ( )$_{10}$
0 and 1 ⇒ $2^2\ 2^1\ 2^0$

$2^0 \cdot 1 + 2^1 \cdot 0 + 2^2 \cdot 1 = (5)_{10}$ ⇒ 0 to 9

0 to 2 (2 1 2)$_3$ → ( )$_{10}$
$3^2\ 3^1\ 3^0$

$3^2 \cdot 2 + 3^1 \cdot 1 + 3^0 \cdot 2$ ⇒ ( )$_{10}$

⋮

base-K ⇒ ( )$_{10}$

⇒ ts should be prime
$a^{b^c} \equiv$

base-2 ⇒ 0 and 1
base-3 ⇒ 0, 1, 2
⋮
base-26 ⇒

prime
29, 31, 37, ⋯

$n = 3$

$h(s) = \left(\sum_{i=0}^{n-1} s[i] \cdot K^i\right)$ %ts ⇒ K ≥ 26

"abc" (0 1 2)

$(a * 26^0 + b * 26^1 + c * 26^2)$ %ts

or

$\left(\sum_{i=0}^{n-1} s[n-i-1] \cdot K^i\right)$ %ts

$(c \cdot 26^0 + b \cdot 26^1 + a \cdot 26^2 + \cdots)$ %ts

"26"

K ≥ 26

(a b c) ⇒ ( )$_{10}$

26

$97 * 26^2 + 98 * 26^1 + 99 * 26^0$

%ts

!!

hash idx

"abc" $k=29$

$$idx = (\overset{x_1}{65} * \overset{y_1}{29^0} + \overset{x_2}{66} * \overset{y_2}{29^1} + \overset{x_3}{67} * \overset{y_3}{29^2}) \% ts$$

$$\Rightarrow (a+b) \% n = (a\%n + b\%n) \% n$$

$$\Rightarrow (a*b) \% n = (a\%n * b\%n) \% n$$

$$idx = (x_1 y_1 + x_2 y_2 + \dots) \% ts$$

$$= [(x_1 y_1) \% ts + (x_2 y_2) \% S + \vdots] \% ts$$

Key[i] power

$$= [((x_1 \% ts)(y_1 \% ts)) \% ts + ((x_2 \% ts)(y_2 \% ts)) \% ts + \vdots] \% ts$$

$y_1 =$ power

$$(x_1 y_1 + x_2 y_2 \dots) \% ts$$

29^0, 29^1, 29^2

search $O(CS/ts)$ 1000 factor

$CS \to 1M$, $ts \to 10$

$$O(\frac{1}{10} \cdot 1M)$$

O (1M or CS)

Load factor threshold = 0.7

Load factor i.e $\frac{cs}{ts} > 0.7$

transfer ⇓ rehashing