Recap : Searching

Linear search	O(n) search	O(1) insert an ele
Binary search	O(log n) search	O(n) insert an ele.
Hashing	O(1) search	O(1) insert time. ✓
	O(logn) search	O(logn) insert time ×

n → no. of strings in arr.

arr = ["sagar", "amit", "jatin", "anish"]

n = 10 elements ⇒ capacity of array

"sagar" → 19 + 1 + 7 + 1 + 18 = 46 %n = 6

"amit" → 1 + 13 + 9 + 20 = 43 %n = 3

proxy index

a → 1		k	11	t	20
b → 2		l	12	u	21
c → 3		m	13	v	22
d	4	n	14	w	23
e	5	o	15	x	24
f	6	p	16	y	25
g	7	q	17	z	26
h	8	r	18		
i	9	s	19		
j	10				

Mapping

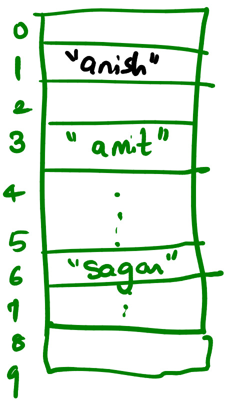
- ASCII
- unicode
- custom

✓ **Insertion logic.**

string → proxy index → go place it. (assume it is possible to insert)

O(1) insert (theoretically)

table



0	
1	"anish"
2	
3	" amit"
4	
5	
6	"sagar"
7	
8	
9	

Searching O(1) search

search ("sagar") → 46%10 ⇒ 6

if table[proxy_index] == key :
return true

anish = 1 + 14 + 9 + 19 + 8 = 51 % 10 = 1

Terminology : "str" → hash table

"str" ↓ proxy index → hash value

hash fn ⇐ method you figure out index with

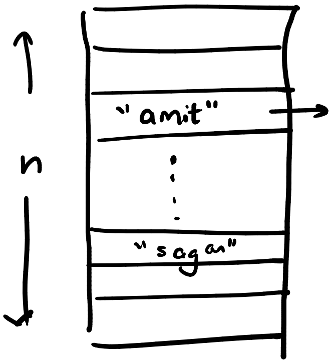
Logic is great : O(1) search O(1) insertion time ⇒ Perfect idea. ×

1. What if 2 strings have value 96 and 46 ? table size = 10.

COLLISION. ⟹ Resolve this.

Optimization are required at every step :

① What should be the size of the table ?

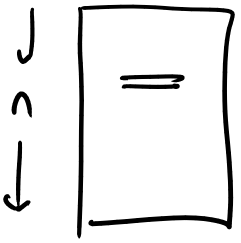


Probability :

$P(\text{collision} \mid \#ele = 0) = 0$ occassional

$P(\text{collision} \mid \#ele = 1) = \frac{1}{n} \leftarrow$ low

$P(\text{collision} \mid \#ele = 2) = \frac{2}{n}$ $P(\text{collision}) \propto \frac{1}{n}$

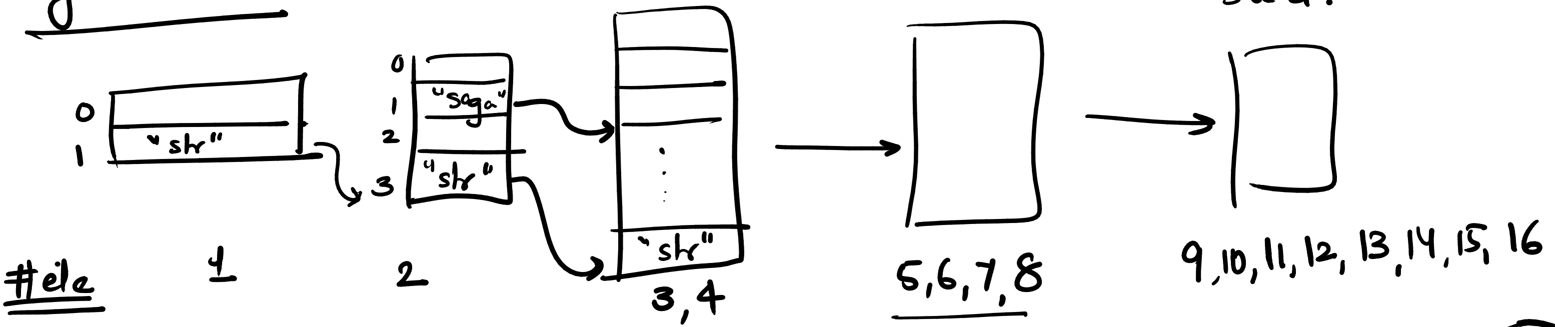


Q. I will handle occassional collisions some other way. After how many elements in array/table of size n, $\rightarrow \frac{n}{2}$ chances of collision are high?

Till $n/2$ elements $\Rightarrow P(\text{collision}) < 50\%$.

most of the times/ atleast 50% of space in array will be wasted.

Logic on size.



capacity $2 \longrightarrow 4 \longrightarrow 8 \longrightarrow 16 \longrightarrow 32 \longrightarrow 64$

$15 \% 2$ ① $\quad$ $15 \% 4 \rightarrow$ ③ $\quad$ $15 \% 8 \rightarrow$ ⑦

#ele. ① ② 3 ④ 5 6 7 ⑧ 9 10 11 12 13 14 15 ⑯ .... ㉜

$\Longrightarrow$ When you double the size, # empty cells increase $\therefore P(\text{collision}) \downarrow$

When you double, you indeed calc all the proxy values of prev de again !

Add $\underline{\underline{N}}$ elements. → no.of elements

n → capacity.

T.C Insertion (assuming doubling logic).

All the instances when you recomputed the hash values :

$\underline{\underline{1}}$	$\underline{\underline{2}}$	3	$\underline{\underline{4}}$	5	6	7	$\underline{8}$	. . . . .	N-1	$\underline{N}^*$	
1	2	×	4	×	×	×	8		$\underline{\#\text{work}}$		# had to recompute hashes

Total work to add N elements in this logic

$$= 1 + \underline{2} + 4 + \dots \underline{\underline{\tfrac{N}{2}}} + \boxed{N}$$

$$= N\left(\frac{1}{N} + \frac{2}{N} + \frac{4}{N} + \dots + 1\right)$$

$$= N\left(\underline{\underline{1}} + \frac{1}{2} + \dots + \frac{4}{N} + \frac{2}{N} + \frac{1}{N}\right)$$ converges.

$$\leq N\left(1 + \frac{1}{2} + \frac{1}{4} + \dots\right)$$

Infinite sum GP

$$S = \frac{a}{1-r}$$

$$< N\left(\frac{1}{1-\frac{1}{2}}\right)$$

$\#$ recomputates $< \underline{\underline{2N}}$ to add N elements.

To add N elements Tc $\Rightarrow O(2N) \sim \underline{\underline{O(N)}}$

To add a single element → $\frac{O(N)}{N} \sim \underline{\underline{O(1)}}$ Insertion per element.

Amoxlization → Vectors in C++ Arraylist in Java list in Py

Statement : If the indices are proxy ⇒ they are bound to change with size of array, I can't return that.

"sagan" → $\underline{\underline{6}}$ × → Next step I am not so sure of this 6.

bool outcome → present or not.

② $x \% \boxed{n}$ there is a uniform distribution.

0	✓✓✓	1/n
1	✓	1/n
2		⋮
2	✓	
4	✓✓	⋮
5		
6		
7		1/n

random "string" → 0 1 2 3 4 ⋮

$\boxed{n \Rightarrow \text{factors}}$

Abstract algebra

↓

Group Theory

↳ Galios theory

n to be prime → no factors

n = 120

factors of n ⇒ 2, 3, 4, 5, 6, 8, 10, 12, 15, ...

Actual

factors ↑ higher is bias towards ↑

n are not exactly double

O(1) insertion

n: 2 5 11 23

P(collision) ↓ if n is prime → n actually blocks the size in RAM.

③ hash f^n ⟹ simple

$O(1)$ time to compute hash value.

No loops No recursion.

sum works okay.

bits ⟹ scramble ⟹ hash value

more randomness than sum can come from scrambling bits.

s _ _

19

| 10110 10001 . . . |

There will be occassional collisions.

Eg. anagrams : sagar ragas

↓ ↓

sum x

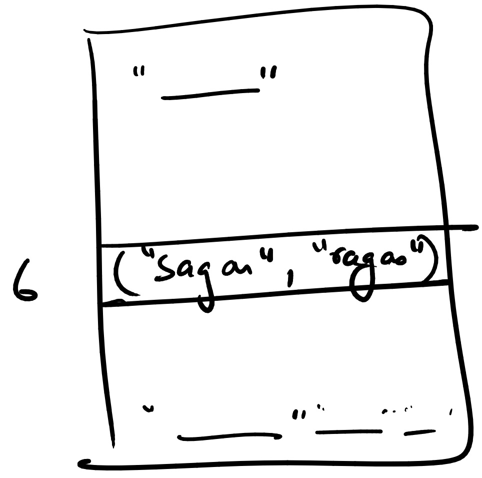
2 collision resolutions : ① Chaining

② Probing

proxy

s_1 = "sagar" → 6

s_2 = "ragas" → 6



search "ragas" → 6

Among multiple value, if anyone matches key ?

Probing.

str → h(str) ⟹ $x = 6$

= "ragas"

n = 10

probe f^n

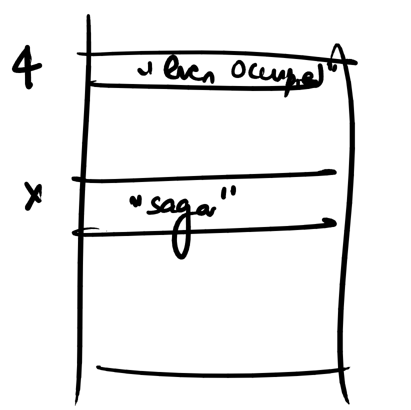
$$g(x) = (ax + bx^2) \% n \quad a, b \in \text{primes}$$

$$(2(6) + 7(6)^2) \% 10$$

$$(12 + 7 \times 36) \% 10$$

$$(12 + 252) \% 10 \Rightarrow \boxed{4.}$$

worst case practical: ③



$$g(x) = a(4) + b(4)^2$$

Only drawback of hashing : Memory wastage *

Best case scenario occupy only 50% of table

worst case scenario occupy only 25% of table

↓

Solve this problem in future. (Trees)

↓

Hashing

Search O(logn) Insertion O(logn) Extra space = 0

NO

n items only n space.

Internal : set. Java collections Py datatype

C++ STL set

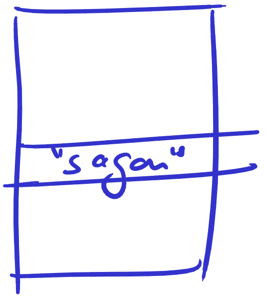
Hashset

set()

SET IS A COLLECTION OF UNIQUE OBJECTS. → uniqueness.

"sagar" → 6

dup Insert "saga" → 6 → 6



key : value

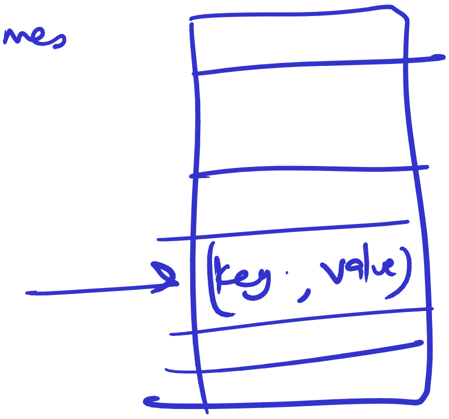
↓

hashvalue (proxy)

string → int | count #times

FREQUENCY.

hashtable



C++: map Java: Hashmap Py : Dict

$$m[\underbrace{\text{"sagan"}}_{\text{proxy}}] \longrightarrow m[\underline{\qquad\qquad}]$$