Hashmap (hashtable)

A hashmap (also known as a Hashtable) is a data structure that stores key-value pairs.

It allows fast retrieval of values based on the keys by using a process called 'Hashing'.

Each key is transformed into a unique index through a hash function, which can be used to access the value (in constant time).

→ Real world example

1. Dictionary
2. Phone contacts
3. Student ID

→ Practical examples

1. DNS lookup
2. No Sql DBs
3. Caching in Web Browser
4. Symbol table in Compiler

Why Hashing ?

→ hashmaps are pretty awesome.

Which is the highest occuring alphabet ?

1. Sort the string ⟶ O(nlogn)   a | b | c |

2. Count = [0] * 26

   Count [0]

   count ['a'] ✗

a → 1
b → 2
⋮
z → 26

finite   O(n)

list can just have integer as keys.

Q: highest occuring word in the sentence?

words are infinite

count ['in'] ✗

a → 1
aa → 2
ab → 3

In arr/list only integers are allowed as keys, but we want to decide the key

Count ['key'] = value

control both.   (any format)

Operations

→ add/update (key, value)
→ search/get (key)
→ delete (key)

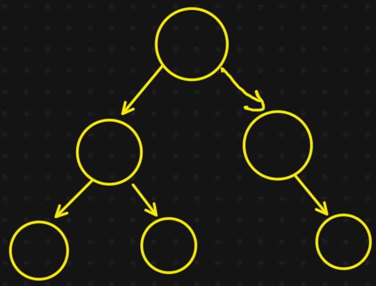
1. Linked List

Key, value
next

1. Insert/update
2. Search
3. Delete

Each operation is $O(n)$

2. BST



balanced

insert
delete
search

$O(\log n)$

3. Hashmaps

$O(1)$

Inbuilt hashmap in Python

dict = { }

we can store things in key - value pair

Please make sure that you try out the questions given

Implementing our own hashmaps : Hashing

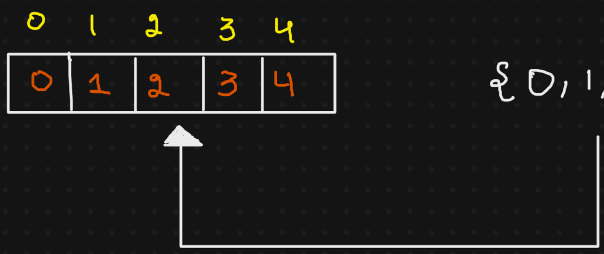
list

access via indexing is O(1) operation

a[5]

independent of len

O(1)



{0, 1, 2, 3, 4}

hash(n) = n

how to store for O(1) retrieval ?

→ store each element at index equal to element

Bucket array

{0, 1, 2, 3, 4, 255873}

A lot of memory wastage if we take a big array/list

inefficient

waste of memory

{ 'Key1', 'Key2', 'Key3' }

Key100

hash( )   s[-1]

Key can be anything & our aim should be to somehow change the given Key into an integer.

Key → $\boxed{\text{hash function}}$ → int

Hash function

1. Hash code → Key converted to integer
2. Compress → key compressed to have an index present in bucket array.

%

$\% n \rightarrow 0 - n-1$

We can have any hash code which converts Key to integer

$0 \rightarrow \boxed{hc} \rightarrow 0$

identity

'abc' → we can use the ascii values 97+98+99

uniform distribution is there

not concentrated

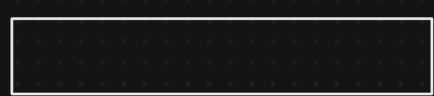
prime number have good distribution

$321 \rightarrow 3 * \underset{p}{\underline{10}}^2 + 2 * \underset{p}{\underline{10}}^1 + 1 * \underset{p}{\underline{10}}^0$ → spread across

$abc \rightarrow 'a' * p^3 + 'b' * p^2 \ldots$ → very large values

What if a node has to be stored as key?

Key → $\boxed{\text{hash code}}$ → integer



bucket array

%5

← 10 numbers

100025 = 0

202075 = 0

% 5

Above is known as Collision

Collision Handling

Key1 $\longrightarrow$ hash fn $\longrightarrow$

50		47	43	

50 43 47 60 80 %5

0 3 2 0 0

1. Open Addressing (Closed hashing)

no. of elements ≤ arraylist size

If initial place is empty save it
else find an alternative.

		47	43	59

59 43 47 69 80 %5

0 3 2 0 0
4 4

$$g(i) = [h(i) + K(i')] \% size$$

$g(i)$ ↓ final index

$K(i')$ ↓ fn = diff f(i)

i' = tries

69 % 5
= 4

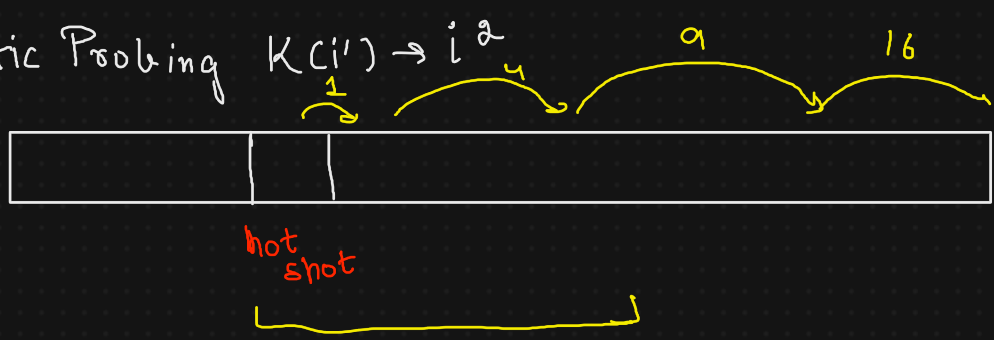
(4+1) % 5
5 % 5 → 0

1. Linear Probing $K(i') \to i$

$g(i) = h(i) + 0$
$= h(i) \% size$

for each increasing try we increase hash fn linearly.

2. Quadratic Probing $K(i') \to i^2$



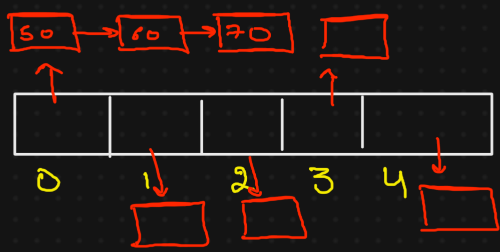
1 → 1
2 → 4
3 → 9

iii) Double hashing

$f(i) \to i * h'(i)$

$g(i) = [h(i) + \underbrace{f(i)}_{\text{Another hash function}}] \% \text{ size}$

2. Closed addressing (Chaining)



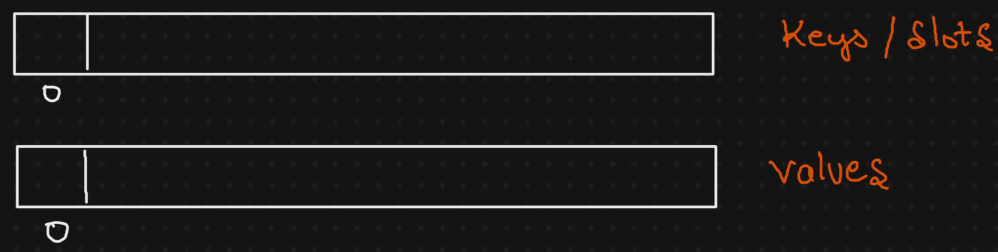
50, 42, 49, 60, 70

%5

0 2 4 0 0

we have each position of bucket array as head of Linked List

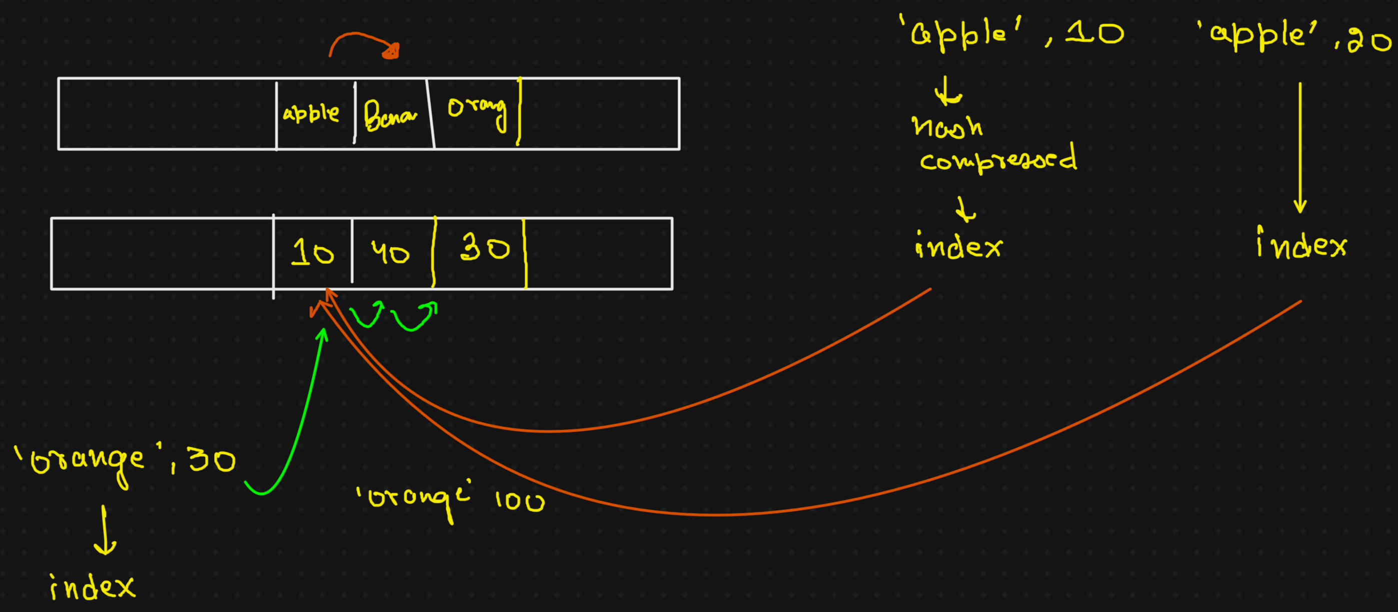
Hashmap Implementation - Open Addressing



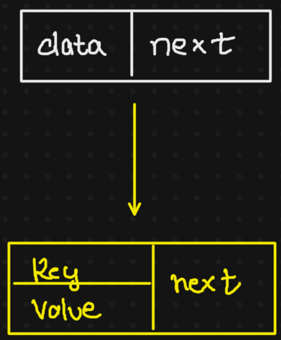
Capacity
size

'apple' , 10
Key value

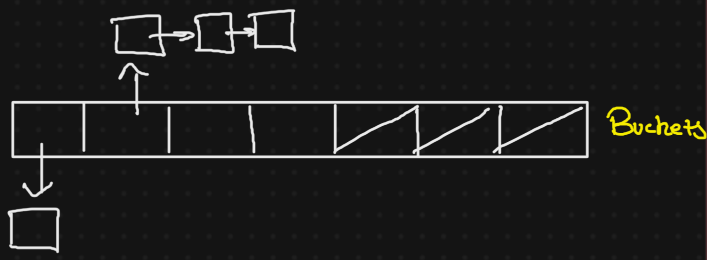
Insert



Hashmap Implementation :- Chaining



we will also have a Key and a value

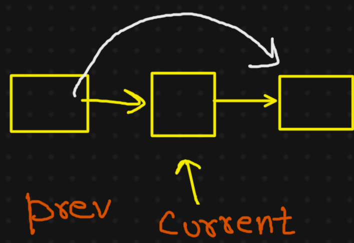


Linked List operation

delete

(i) if node is head

(ii)



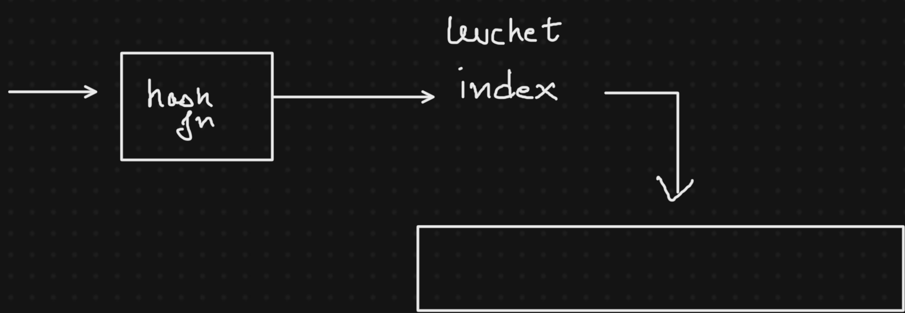
Buckets list

[None] * Capacity

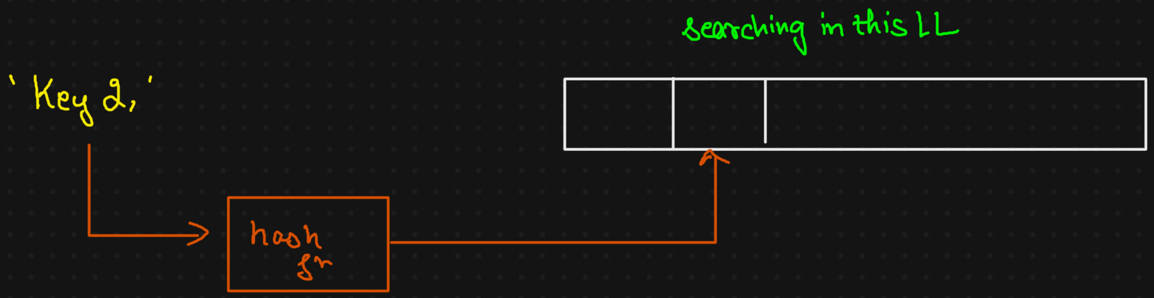
Should be a seperate head of our new linked list class

Insert

'Key1', 100

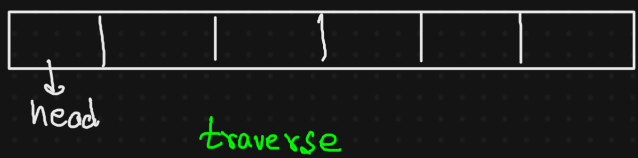


Search



Delete

Print

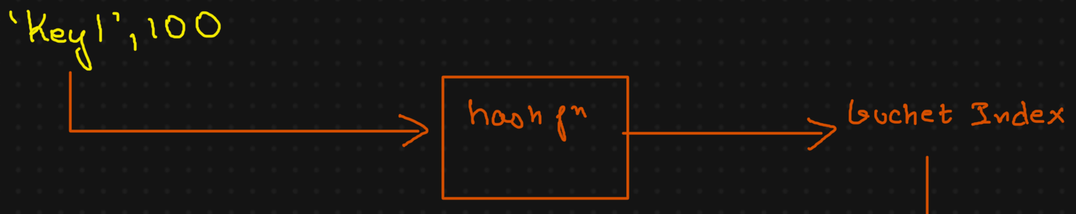


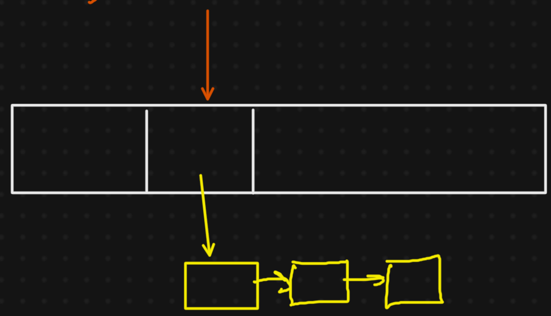
Complexity Analysis of our Hashmap

$n \rightarrow$ no. of entries

$b \rightarrow$ bucket size

1. Insert





1. finding the hash value $\rightarrow O(\text{len})$

2. Insert in LL

$O(n)$

1. In worst case all can come to a same bucket index, search will become $O(n)$
2. $n >>> \text{len}$, so hashing can be $O(1)$ is constant

normally due to good hash codes, we don't get this kind of a scenario and get a uniform distribution

$O(\frac{n}{b})$

$\downarrow$

$O(1)$

$\frac{n}{b}$ = load factor ~ 0.75

we have to increase b to keep load factor in check

old_b = 10

$\frac{\text{Size}}{\text{Capacity}} > 0.75$

'Key1' → 105 % 10 = 5

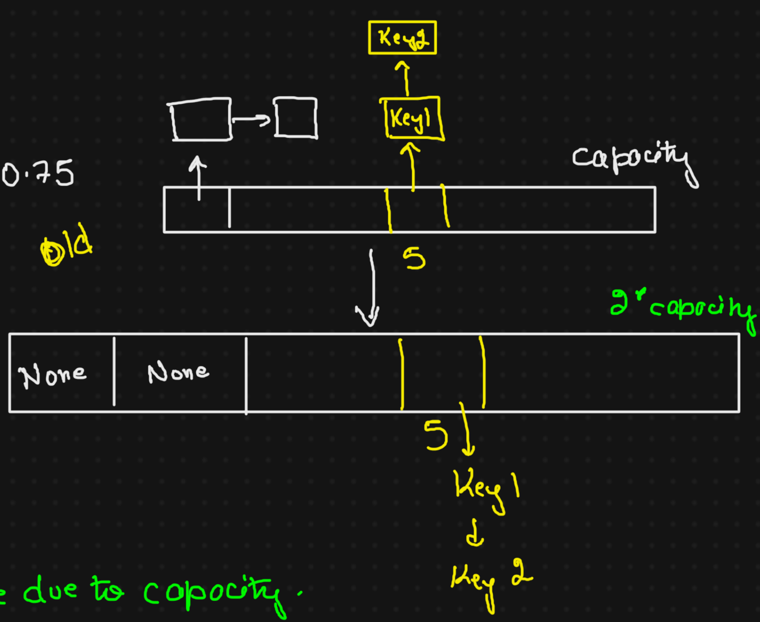
'Key' → 115 % 10 = 5

new_b = 20

new bucket

'Key1' → 105 % 20 = 5

'Key2' → 115 % 20 = 15



✗ The final bucket index will change due to capacity.

Rehashing $\longrightarrow$ to keep load factor $(\frac{n}{b})$ all entry are hashed to a new bucket array.

So, time complexity remain $O(1) \sim O(n/b)$

Rehashing

1. Save oldbuckets
2. double capacity
3. new-buckets
4. self.size = 0
5. for every entry
   just insert