{HASHING}:

↳ searching / insertion / deletion → O(1)

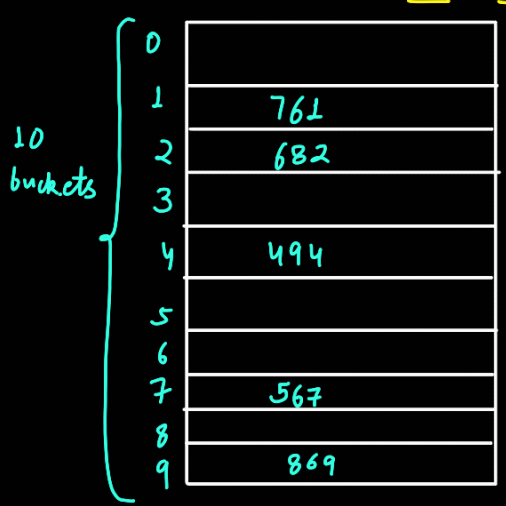
eg → 682, 761, 494, 567, 869

Hashing: finding a unique index for all elements to store them.
↳ hash value

682, 761, 494, 567, 869
=2 =1 =4 =7 =9 ← hash values

h(K) = K mod 10
↳ hash func^n

Hash Table



Types of Hash functions:

1. Division method: h(K) = K mod m
   ↳ buckets
   suppose m = 11
   1276 mod 11 = 0

2. Mid-Square method:
   h(K) = K^2 & extract middle digits

   eg → K = 60
   K^2 = 3600
   ↓
   60 = hash value

   eg. k = 13
   k^2 = 169
   6 = hash value

3. Digit folding method: fold key into equal parts K_1 K_2 K_3 K_4 K_5

   K_1 K_2
   \+ K_3 K_4
   \+ K_5

   Hash value

   eg: K = 1 2 3 4 5
   12
   \+ 34
   \+ 5
   ---
   51 = hash value

4. Multiplication method:
   h(K) = floor(M(KA mod 1))
   ↳ size of hashtable / no. of buckets.
   ↓

eg → K = 12345, A = 0.01, M = 10

h(K) = floor (10 (12345 × 0.01 % 1))

= floor (10 (123.45 % 1))

= floor (10 × 0.45)

= floor (4.5) = 4 ⇒ hash value

Collisions : when two elements have same hash value.

eg → h(K) = K % 10.

	1234,	564
hash val. =	4	4

Handling collisions : → 1. Open hashing (closed addressing)

→ 2. Closed hashing

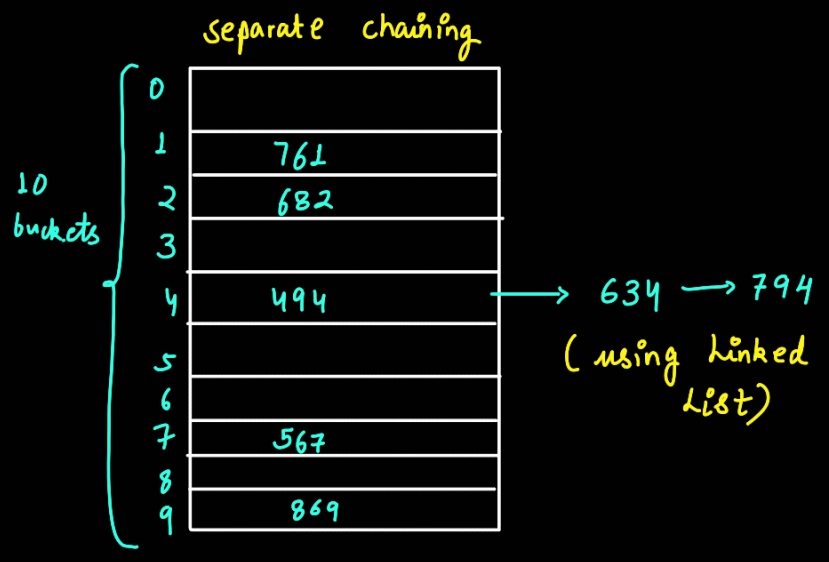
Open hashing (closed addressing) :

↳ separate chaining

eg → 682, 761, 494, 567, 869, 634, 794

hash values :	2	1	4	7	9	4	4

h(K) = K % 10



* searching : O(l)
↳ length of L.L.

Closed Hashing (open addressing / linear Probing) :

↳ kitne chances lage

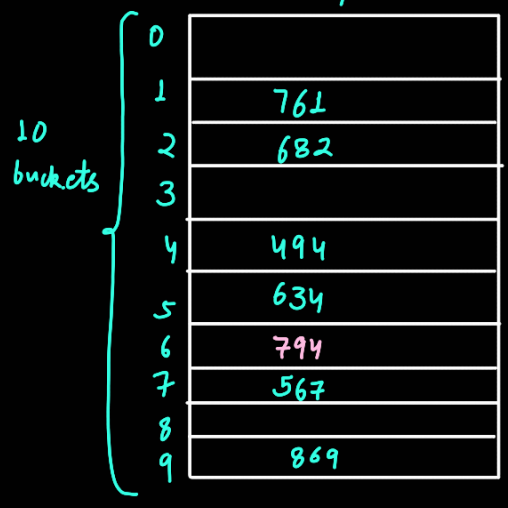
value hash-table mein insert karne ke liye.

eg → 682, 761, 494, 567, 869, 634, 794.
↓ 2 probes ↳ 3 probes

Hash val. = 2 1 4 7 9 4 4

store at $(h(K)+i)\%M \quad \forall \quad 0 \le i \le M$ (M = no. of buckets)

linear probing →



eg → $h(794) = (4+0)\%10 = 4$ } filled
$= (4+1)\%10 = 5$ }
$= (4+2)\%10 = 6 \Rightarrow$ empty] add 794 to 6th bucket

Searching : O(probes)

Quadratic Probing :

↳ to prevent formation of clusters

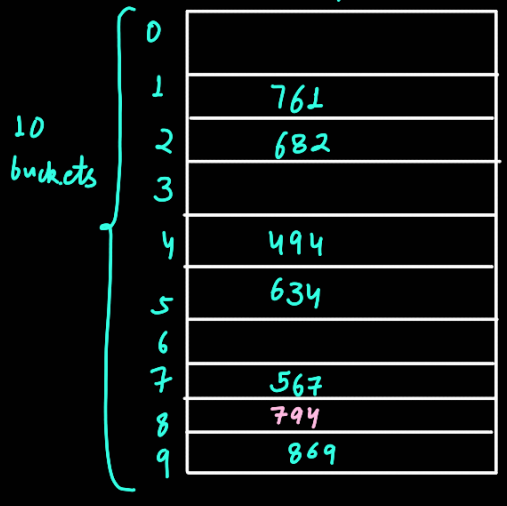
store at $(h(K)+i^2)\%M \quad \forall \quad 0 \le i \le M$

eg → 794

$h(K) = 794\%10 = 4$

at $(4+0^2)\%10 = 4$ ✗ } occupied
$(4+1^2)\%10 = 5$ ✗ }
$(4+2^2)\%10 = 8$ ✓ } vacant

quadratic probing



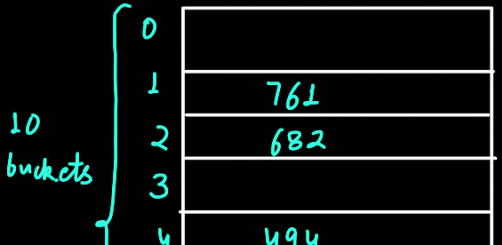
Double Hashing :

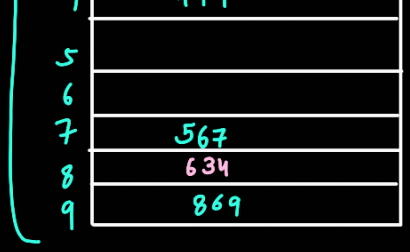
682, 761, 494, 567, 869, 634

$h_1(K) = K\%10$

$h_2(K) = K\%5$

store at $(h_1(K) + i*h_2(K))\%M \quad 0 \le i \le M$





$h(634) = [634 \% 10 + 0\,(634 \% 5)] \% 10$

$= 4 \% 10 = 4$ (✗ occupied)

$i=1 \Rightarrow$ $= [4 + 1(4)] \% 10 = \underline{8}$ ✓ (vacant)

Load factor ⇒ $n \rightarrow$ no. of elements
$m \rightarrow$ no. of buckets

Load factor = n/m ⇒ signifies average entries in one bucket.

↳ should be ideally low

Rehashing : when load factor exceeds a certain limit, we again rehashe the given data.

L.F. > limit

generally **Load factor Limit = 0.75**

* in rehashing, we increase the size of hash table and redistribute the elements again.

↓

very costly (time ↑↑)

Implementing Hash Table (separate chaining) : (VS code)

↳ closed addressing

---

dynamic container

HASHMAPS :

☆ STL container which stores key-value pairs.

☆ The elements are stored in ascending / descending order w.r.t. keys.

↳ by default

A → 12
B → 98
C → 05

eg →

key	→	value
Name	→	Roll no.
Arjun	→	1234
Sneha	→	9928
Kavita	→	9854

☆ Maps cannot have duplicate values.

A → [ ]
A → [ ] ✗

☆ Implemented through BST (binary search trees).

☆ declaration → #include <map>

map < key_datatype , value_datatype > map_name;

eg → map < string , int > phone_directory ; // ascending order by default.

eg → map < datatype 1, datatype 2, greater < datatype 1 > > map_name;

✱ Initialization :

map < key_datatype , value_datatype > map_name = { { key1, val1 }, { key2, val2 }, { key3, val3 } }

eg → map < string, int > directory = { { "Naman", 23456 }, { "Chatur", 12987 } }

✱ Insertion → I. directory.insert (make_pair ("ABC", 386));

O(log N) II. directory [key] = value ; // preffered.

✱ Printing elements : for each loop : → key, value pair

```
for (auto element : map1){
    key = element.first;
    value = element.second;
}
```

✱ Updation → directory ["Chatur"] = 9927;

More Member Func^n :

iterators are part of STL library used to traverse STL containers like lists, stacks, queues etc.

O(log N)
erase ( )

mp.erase (iterator);

mp.erase (key);

mp.erase (start_itr, end_itr);

swap ( )

mp1 , mp2
∨
same type.
↓

* mp1.swap (mp2);

clear ( )

mp.clear ( );

deletes range from start to end (excluding end)

O(n) → range

swap(mp1, mp2);

empty()

mp.empty();

→ returns true if empty else false

size()

mp.size();

max_size()

mp.max_size();

→ max. storage allocated to the map.

find()

→ O(log N)

mp.find(key);

⇒ returns itr. to the element if present, else returns mp.end();

↙ itr after the last element in map

count()

mp.count(key);

⇒ no. of occurrences of key.

returns 1 if element is present else 0.

upper_bound();

returns an itr to next greater element.

{{10, "Name1"}

{20, "Name2"} ← mp.lower_bound(20)

{30, "Name3"}} ← mp.upper_bound(20);

lower_bound();

returns itr to element if present else itr to next greater element.

mp.lower_bound(20);

mp.begin(), mp.end(), mp.rbegin(), mp.rend()

← mp.rend();

[K_1 V_1 ← mp.begin();

K_2 V_2

K_3 V_3

k_4 v_4] ⟵ mp.rbegin();

⟵ mp.end();

Unordered Map : same as map but elements are stored in unordered manner. (→ efficient) (→ implemented using hashing.)

insertion / deletion / search — O(1)

also find(), insert(), erase — O(1) avg. time complexity.
↳ O(n) worst case

MULTIMAP : same as ordered map, the only diff. is that you can also store duplicate keys in a multimap. (→ using BST (implementation))

(# include <map>)

insert / delete / search : O(log N)

* note : in multimaps, we can only insert new elements using insert func^n, we can't use mp[key] = value. ✗

Unordered multimap → duplicate keys allowed and stored in unordered manner.

↳ member functions ⇒ O(1)

☆ equal_range() func^n in Multimap :

☆ Questions → in VS code.

⇒ isomorphic
⇒ longest subarray with zero sum.
⇒ target sum indices.

Ques → Given an array arr[] of length N, find the length of the longest subarray with a sum equal to 0.

input = { 15, -2, 2, -8, 1, 7, 10, 23 }

Output = 5

⇒ using prefix sum and Hashmaps.