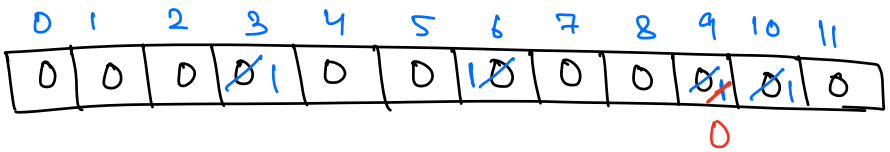
Q: Given an ~~Array~~ of size N = {0}.
Given Q queries.

Type 1: Given an index, toggle the value at this index.

Type 2: Given an index, return index with closest 1 to given index.

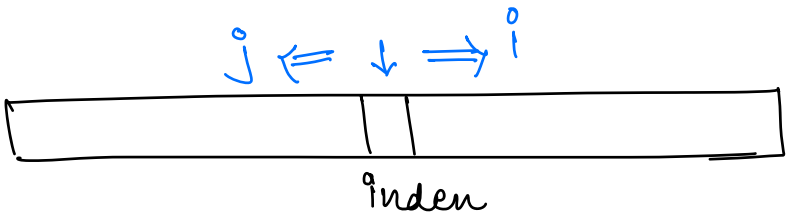
Ex N = 12
Q = 11



[[2, 6] → −1
[1, 3]
[1, 10]
[2, 7] → 10
[2, 2] → 3
[2, 4] → 3
[1, 9]
[1, 6]
[2, 8] → 9
[1, 9]
[2, 10] → 10

⇒ Type (1): index
a[index] = 0|1 ⇒ O(1)

⇒ Type (2): index



TC: O(N)

inden

Type 2: [2, 9]

Indexes at which value is set.

B: {4, 3, 7, 1, 5, 15}

dist: 5 6 2 8 4 6

* Store the indexes at which (1) is present in a separate Array.

TC: O(N)

B: {4, 3, 7, 1, 5, 15}

Sort

C: {1, 3, 4, 5, 7, 15} inden = 9

* Find the closest value to (9) in sorted Array C:
  * Check if (9) is present, if yes (9) is ans.
  * floor(9) ⇒ O(log N) : greatest value < 9.
  * ceil(9) ⇒ O(log N) : smallest value > 9.

TC: $O(\log_2 N)$
↳ to search in sorted Arr.

* If we want to add | delete any index from array C then TC: $O(N)$

TC: $O(N)$

* Here we need some DS that maintains some order.

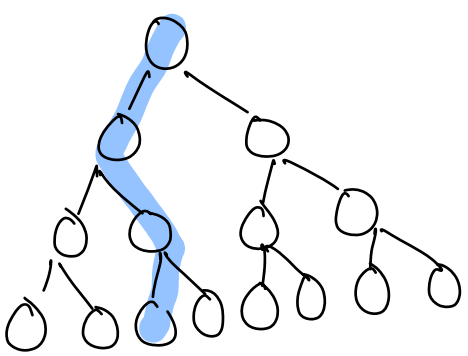
⇒ Ordered Hashmap | HashSet

* Internally uses Balanced BST

(Self Balancing) AVL RB Trees

$|H(LST) - H(RST)| \leq 1$

* TC: $O(\log_2 N)$ Search | Insert in BBST



C++ : map | set.

Java : TreeMap / TreeSet.

Python : OrderedDict.

* Set in C++ :-
  * floor(x) } TC: $O(\log_2 N)$ ⇒ Type 2 Query
  * ceil(x) }

using inbuilt library fun^.

* Type (1) : Index

```
if A[index] == 1 :
    // index is already present
    // in the set, so remove it
    set.remove(index)        <- O(log2 N)
    A[index] = 0;
if A[index] == 0 :
    // insert index to set.
    set.insert(index)        <= O(log2 N)
    A[index] = 1;
```

⇒ TC: $O(\log_2 N)$

1 query ⇒ $O(\log N)$
Q queries ⇒ $O(Q \cdot \log_2 N)$

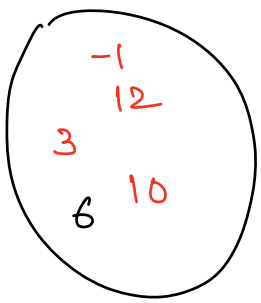
Q: Given N.
Given Q queries.

Type 1: Given an index, toggle the value at this index.

Type 2: Given an index, return index with closest 1 to given index.

N = 12, Q = 11

(-1) [0 11] (12)



Set.
[Ordered]

[[2, 6] → floor(6) = -1, ceil(6) = 12 ⇒ -1
[1, 3]
[1, 10] →
[2, 7] → floor(7) = 3, ceil(7) = 10 ⇒ 10
[2, 2] → floor(2) = -1, ceil(2) = 3 ⇒ 3
[2, 4] ⇒ floor(4) = 3 , ceil(4) = 10 ⇒ 3
[1, 9] ⇒
[1, 6] →
[2, 8] → floor(8) = 6, ceil(8) = 9 ⇒ 9
[1, 9] ⇒ remove from set.
[2, 10] → floor(10) = 10 ceil(10) = 10 ⇒ 10

Q. Implement your own HashMap | HashSet.

add() / get() (K, V)

	add()	get()
HashMap ⇒	add (K, V)	get (K) ⇒ value
HashSet ⇒	add (K)	get (K) < True / False

HashSet Implementation

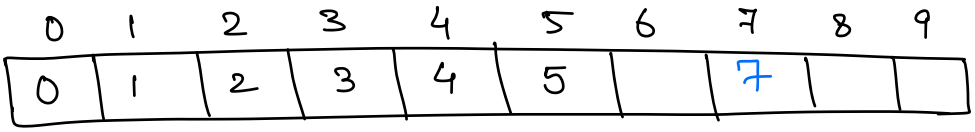
* Array

add (2) ⇒ search for 2
add (5) ⇒ search for 5
add (6) ⇒ search for 6
add (10) ⇒ search for 10
add (5) ⇒ search for (5) ×
get (6) ⇒ linear seach.

add (K) } ⇒ O(N)
get (K) }

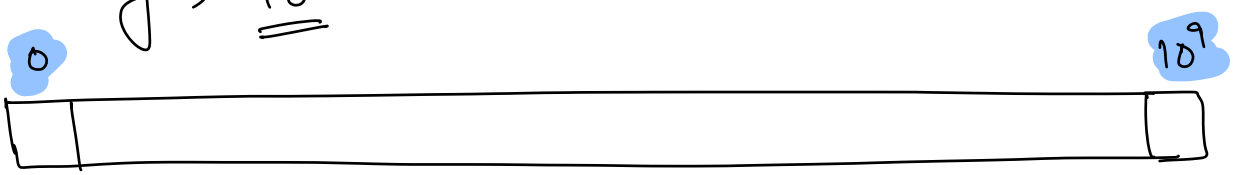
* Array provides us O(1) random access

TC.



add(K) } O(1)
get(K) }

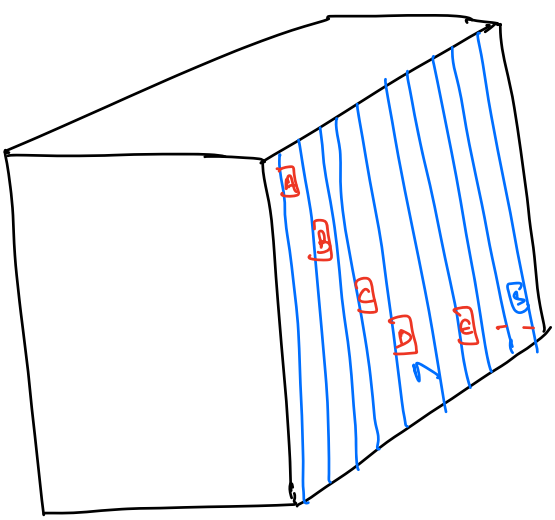
Key ⇒ 10^9



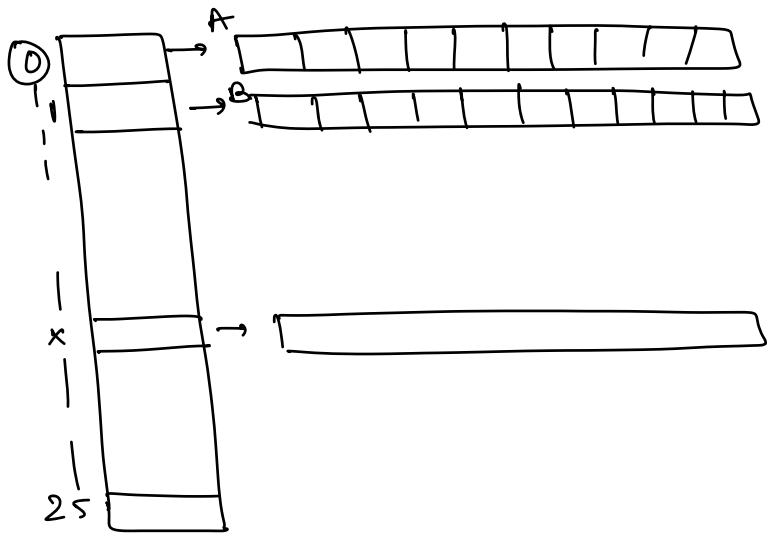
space of Array is the limiting factor.

* NOT a good approach.

Ex



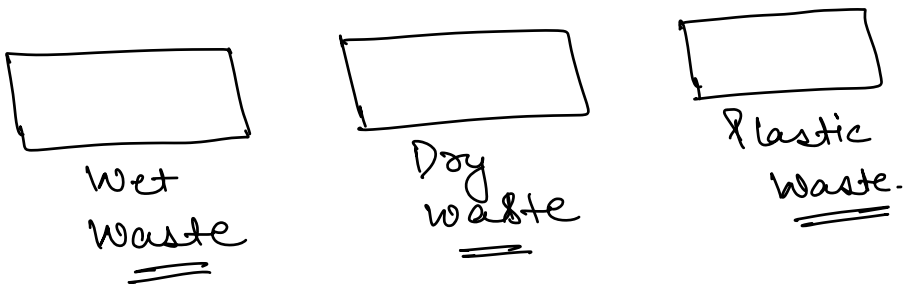
Telephone Directory



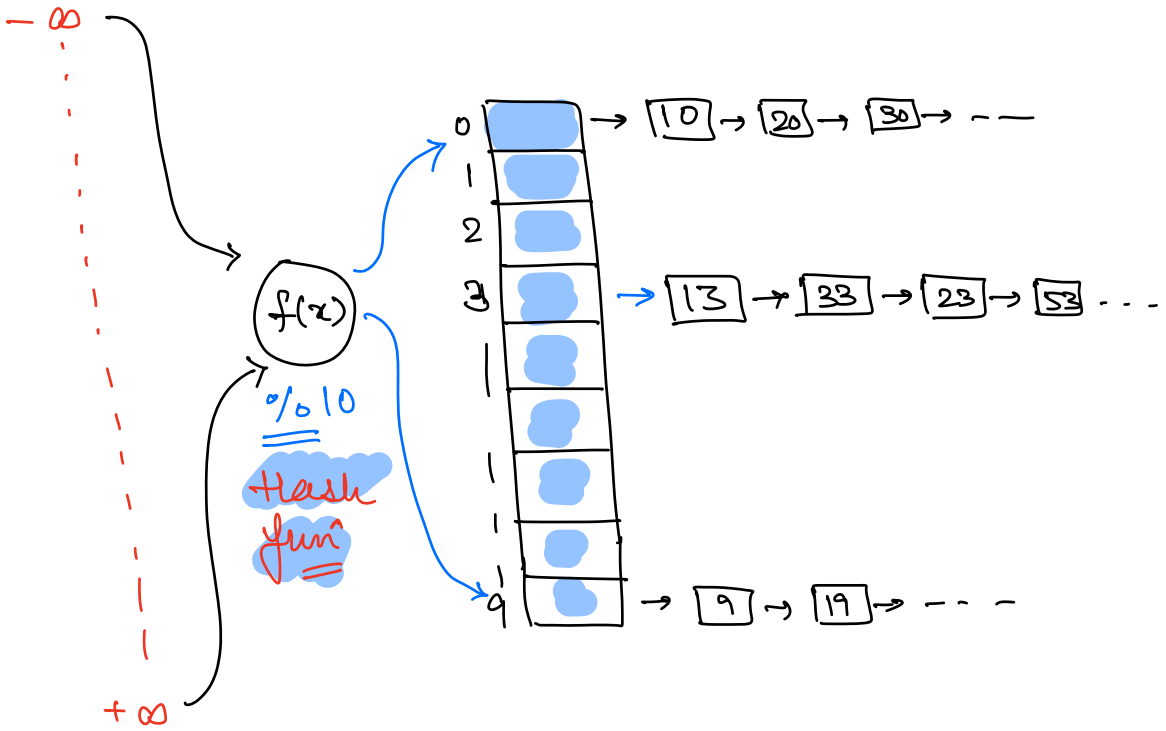
Chemist Shop

Anti Biotics
Pain Killer
Eye
Fever
:

Ex



* Bucketizing | Categorizing helps us to search faster.



* add(13) ⇒ 13 % 10 = 3
* add(33) ⇒ 33 % 10 = 3

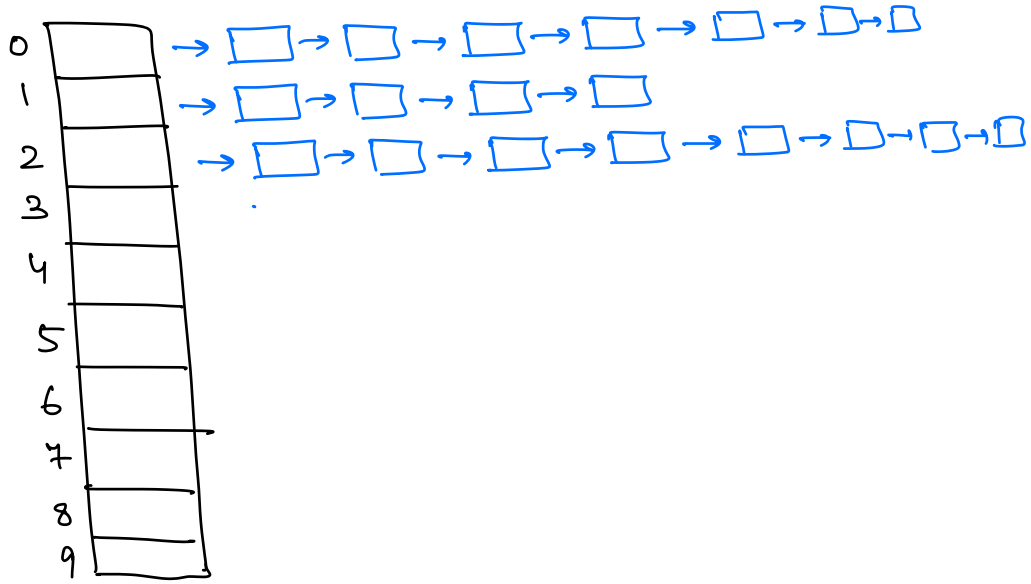
(10 ↑ Table Size)

Collision
- * Chaining
- * Open Addressing
- * Linear Probing.

X % 10 ∈ [0,9]

10 Buckets.

* Getting the same buckets for different objects ⇒ Collision
* We can't avoid collisions, but we can try to reduce the no. of collisions.



$$f(x) = \begin{cases} -1 & , x < 0 \\ 0 & , x = 0 \\ 2 & , x > 0 \end{cases}$$

* TC can go upto O(N) depending on our **Hash fun^{n}**

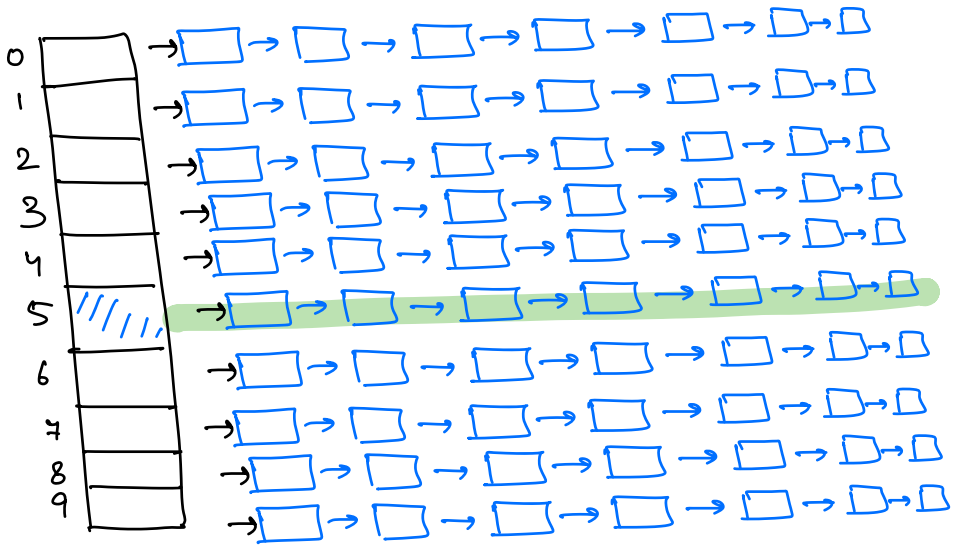
* **Characteristics of a good Hash fun^{n} :-**

1) Should be able to use all the buckets.
2) For any random input, all the buckets should have equal probability.

N ⇒ Input size

B ⇒ No. of Bucket.

Every Bucket will have N/B elements.



TC of search :-

$$\text{Iterations} = \frac{N}{B}$$

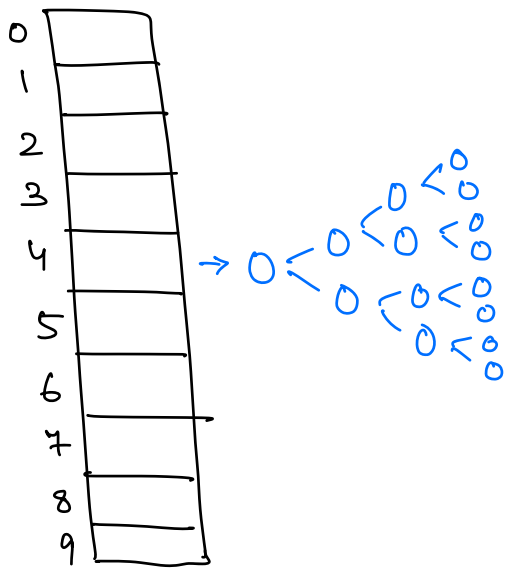
TC : O(N) Worst case

N ⇒ Input size

B ⇒ No. of Bucket.

Every Bucket will have N/B elements.

↳ (Every Balanced BST will have N/B elements)



TC :- $\log_2(N/B)$

Worst Case

$O(\log N)$

Best Case

$O(1)$

Avg case : $N = 10^9$

$B = 10^6$

TC : $O\left(\log_2\left(\frac{10^9}{10^6}\right)\right) = O(\log_2 10^3)$

≈ 10 (Constant)

$\log 2^{10} = \log 1024 = 10$

* $\log_2(10^9) = 30$

$30 < < < < < 10^9$

On an average TC: $O(1)$


```
list<int> arr[N];
void add(K) {
    hash_code = hash_fun(K)
    arr[hash_code].push(K);
}
bool get(K) {
    hash_code = hash_fun(K)
    // Iterate over list (arr[hash_code])
    // and check if (K) is present or not.
}
```

* Ordered / Unordered map => Hashing (2)

HW
* Open Addressing
* Linear Probing
* Quadratic Probing