Hashing.

A technique used for searching purpose.

Hashing

→ Searching $TC: O(1)$

① Linear Search

$\{3, 10, 7, 8, 14, \dots\}$

→ searching $TC: O(N)$, $SC: O(1)$

② Binary Search

$\{3, 5, 10, 17, 19, 20\}$

→ Searching $TC: O(\log_2 N)$ $SC: O(1)$

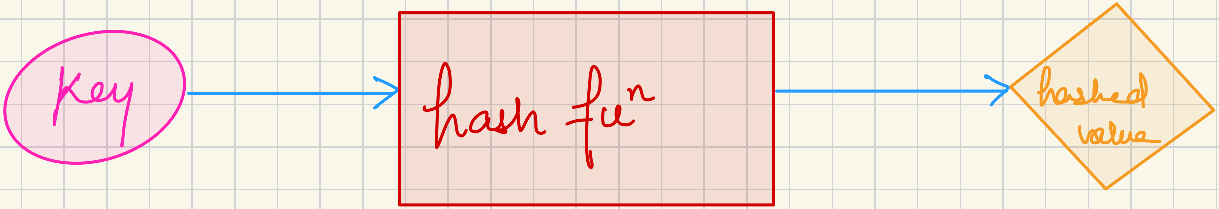
{ 8 ✓, 3 ✓, 13 ✓, 6 ✓, 7 ✓, 4 ✓, 10 ✓ }, 50 → seq. of i/p given by user

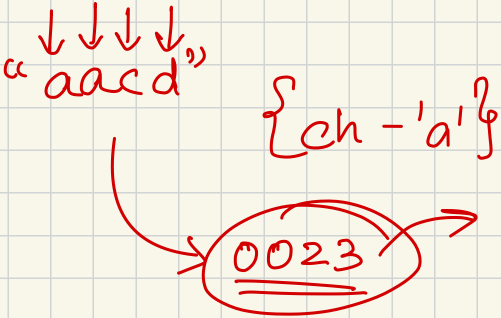
0	1	2	3	4	5	6	7	8	9	10	11	12	13	- - - - - - - -	50
			T	T		T	T	T		T			T		T

```
search(10)
{
    if (arr[10] == True)
        return "found";
    else
        return "Not found";
}
```

TC: O(1)
SC: O(1)

{ high memory is required, hashing was introduced }





hash function

step1 $g(\underset{\text{key}}{x}) = \underset{\text{integer value}}{K}$

step2 $f(K) = \underset{\text{hashed value}}{y}$

key Space

8

3

13

6

4

10

hash function

$f(K) = K$

$f(8) = 8$

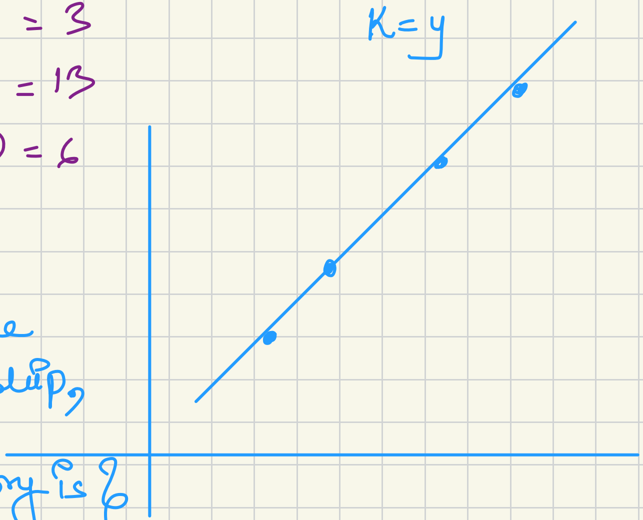
$f(3) = 3$

$f(13) = 13$

$f(6) = 6$

one to one relationship,

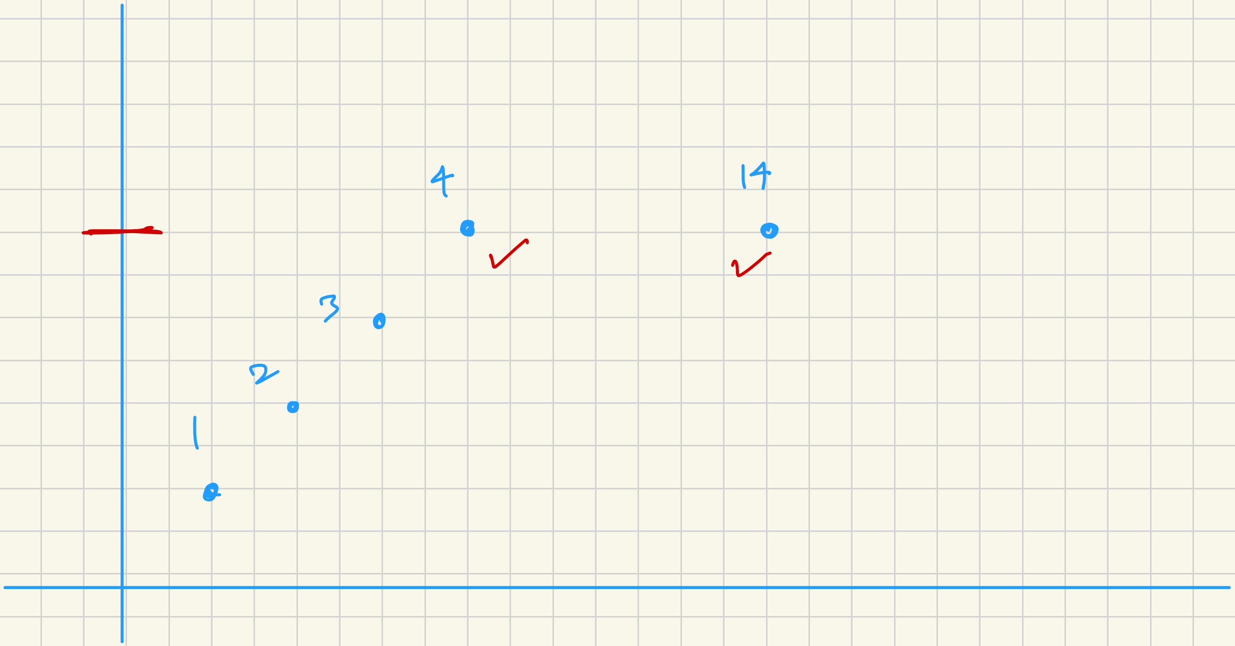
{Extra memory is used}



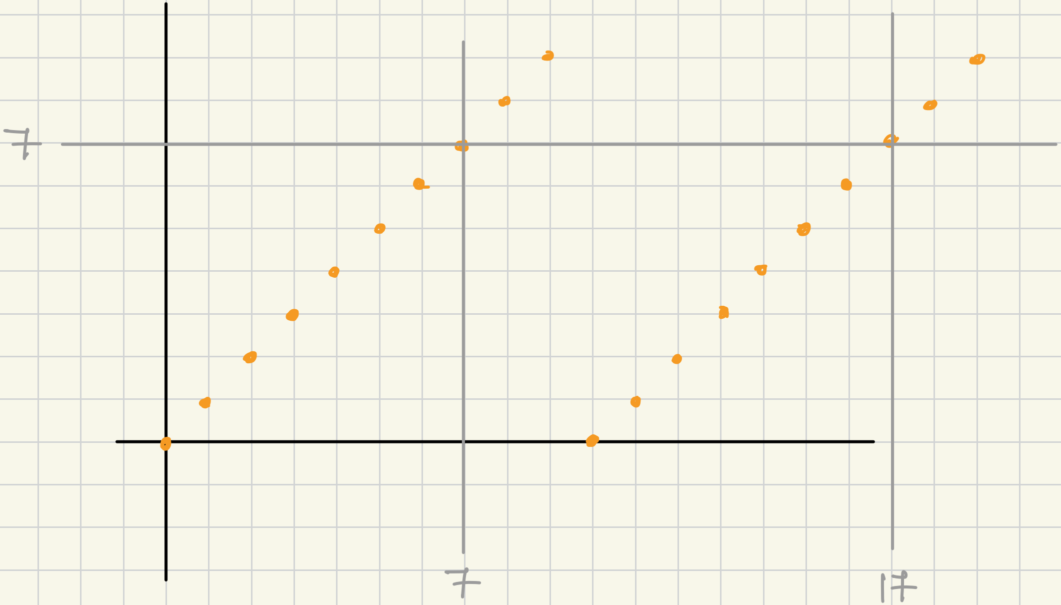
hash table

0	
1	
2	
3	3
4	
6	6
8	8
13	13
∞	

Many to one function



$f(k) = k \% 10$



hash function

$f(k) = k \% 10$

10 → size of hash table

key Space

8

3

6

4

10

13

$f(8) = 8 \% 10 = 8$

$f(3) = 3 \% 10 = 3$

$f(6) = 6 \% 10 = 6$

$f(4) = 4 \% 10 = 4$

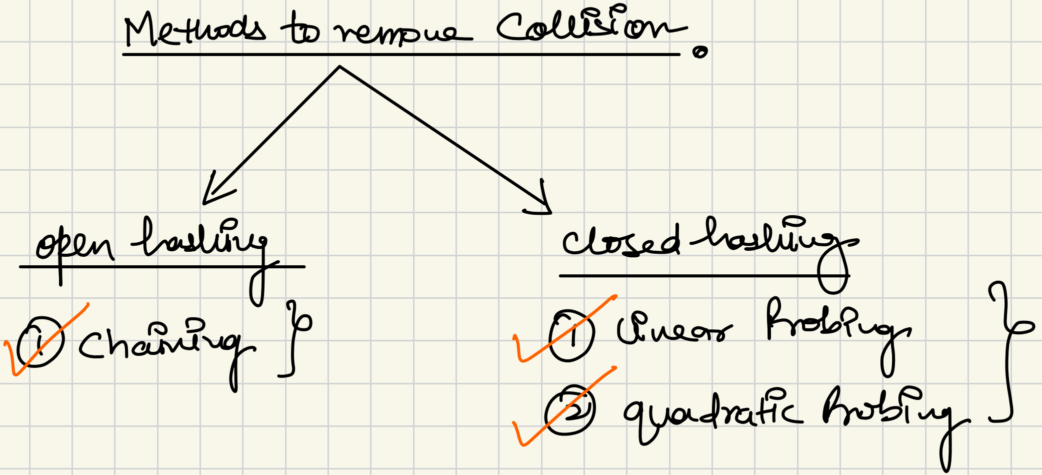
$f(10) = 10 \% 10 = 0$

$f(13) = 13 \% 10 = 3$

hash table

0	10
1	
2	
3	3
4	4
5	
6	6
7	
8	8
9	

Collision



Methods to remove Collision.

open hashing

✓ ① Chaining }

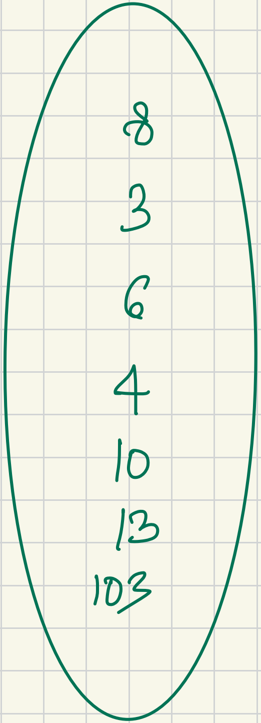
closed hashing

✓ ① linear Probing

✓ ② quadratic Probing }

Chaining

key Space



hash function

$f(K) = K \% 10$ → size of hash table

$f(8) = 8 \% 10 = 8$

$f(3) = 3 \% 10 = 3$

$f(6) = 6 \% 10 = 6$

$f(4) = 4 \% 10 = 4$

$f(10) = 10 \% 10 = 0$

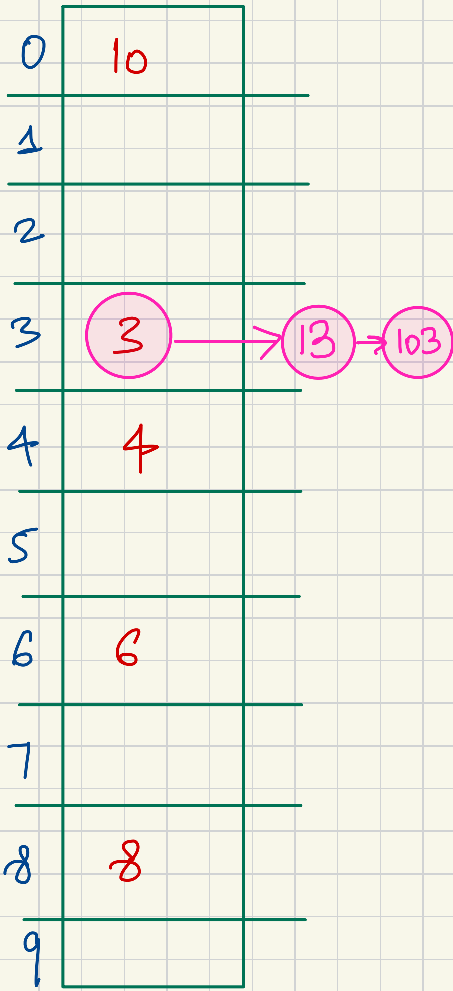
$f(13) = 13 \% 10 = 3$

$f(103) = 103 \% 10 = 3$

search(103)

{avg TC: O(1)}

hash table



Linear Probing.

key Space

8

✓3

✓6

✓4

✓10

13

103

hash function

$$h'(K) = \{h(K) + f(i)\} \% 10$$

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

$$f(i) = 0, 1, 2, 3 \ldots\ldots$$

$$h'(8) = \{h(8) + f(0)\} \% 10$$
$$= (8+0) \% 10 = 8$$

$$h'(3) = \{h(3) + f(0)\} \% 10 = 3$$

$$h'(6) = \{h(6) + f(0)\} \% 10 = 6$$

$$h'(13) = \{h(13) + f(0)\} \% 10 = 3$$

$$h'(13) = \{h(13) + f(1)\} \% 10 = 4$$

$$h'(13) = \{h(13) + f(2)\} \% 10 = 5$$

hash table

0	10
1	
2	
3	3
4	4
5	13
6	6
7	103
8	8
9	

$h(k) = k \% \text{size}$

75% of the Range of the values present

$0 \rightarrow 10{,}000$

7500

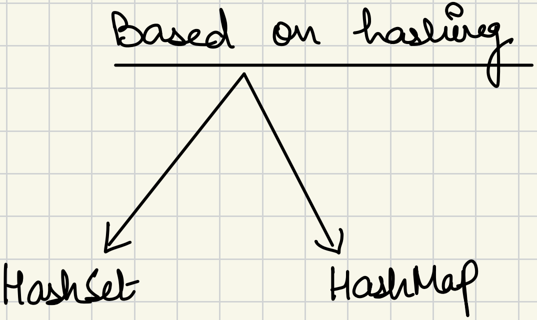
Quadratic Probing

$$h'(k) = \{ h(k) + f(i) \} \% 10$$

where,

$$h(k) = k \% 10$$

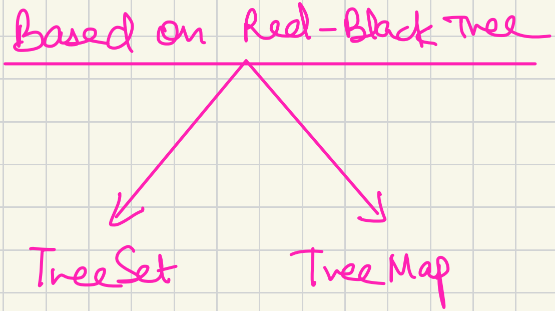
$$f(i) = i^2 \; ; \quad 0, 1, 2 \ldots\ldots$$



Based on hashing

HashSet

HashMap



Based on Red-Black Tree

TreeSet

TreeMap

HashSet

↳ datastructure to store unique entities

3, 13, 13, 4, 5, 6, 4, 6, 3, 3, 3

→ HashSet {3, 13, 4, 5, 6} → Random order

↳ searching TC: O(1)

↳ insertion TC: O(1)

HashMap

↳ ds use to store key-value pairs

String (key)		Integer (value)
lays	→	2
coke	→	3
eggs	→	6
apples	→	3

First Element to occur K times

int[] arr = { 1, 2, 5, 3, 6, 5, 2, 9, 1, 2, 5, 3 }

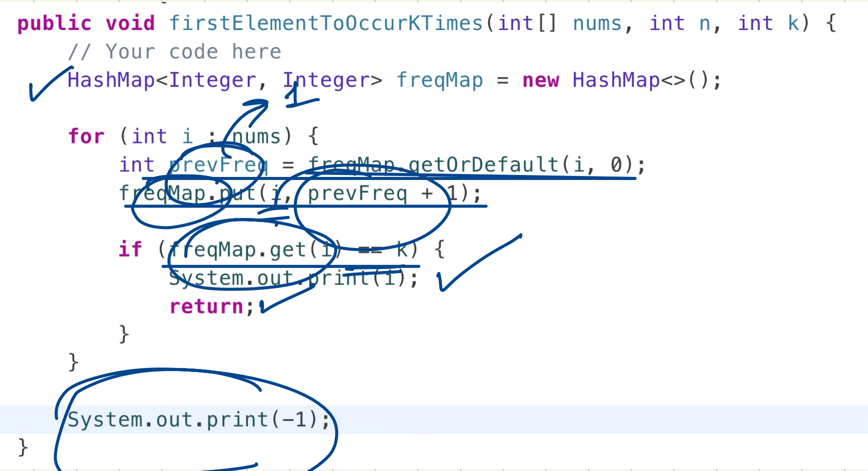
↑ ↑ ↑ ↑ ↑ ↑

freq Map

Key(Ele)	value(freq)
1	1
2	1
5	~~1~~ 2 ✓
3	1
6	1

return

↳ 5 ✓



```
public void firstElementToOccurKTimes(int[] nums, int n, int k) {
    // Your code here
    HashMap<Integer, Integer> freqMap = new HashMap<>();

    for (int i : nums) {
        int prevFreq = freqMap.getOrDefault(i, 0);
        freqMap.put(i, prevFreq + 1);

        if (freqMap.get(i) == k) {
            System.out.print(i);
            return;
        }
    }

    System.out.print(-1);
}
```

1

k=2

{1, 3, 5, 6, 5, 6, 1, 7, 3}

freq Map

Int Key	Int Value
1	1
3	1
5	2
6	1

TC : O(N)

SC : O(N)

o/p

5

Valid Anagram

str1 str2

↳ Anagranic if then can be equal after rearranging

str1 = "accio"

str2 = "ioacc"

Approach :

str1 = " acoci "

str2 = " ocaic "

a → 1
c → 2
o → 1
i → 1 } ?

sort(str1) → accio }
sort(str2) → accio } str.equals ?