HASHMAP IMPLEMENTATION

Content for Today

01. check if element exist.
02. Hashmap
03. Collision & its resolution
04. Lambda function
05. Code Implementation

01. Given an integer array & some queries.

For every query, you need to check if the elements exists in array or not.

A[] = {	2	4	11	6	8	9	1 }
	0	1	2	3	4	5	6

Query

x = 10 ⟶ false

x = 8 ⟶ True

x = 3 ⟶ false

Brute force → For every query, iterate on array & check if it's present.

TC: O(n*q)

SC: O(1)

* Idea 2 - Freq Array

A[] = { 2, 4, 11, 1, 8, 9, 6 }
0 1 2 3 4 5 6

DAT =

[]	F	T	T	F	T	F	T	F	T	T	F	T
	0	1	2	3	4	5	6	7	8	9	10	11

Direct access Table

dat[i] = True = if i^{th} element exist in A

= false = if i^{th} element doesn't exist

```
boolean [] dat = new boolean [max(A) + 1]

for (int i=0 ; i<n; i++){
    int ele = A[i];
    DAT[ele] = True;
}
```

To answer queries

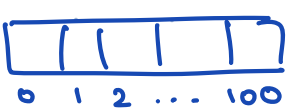
Query = x ⟶ Dat[x] == true ? True : False

TC : O(1)

TC : O(Q+N)

SC : O(Max of Array)

A[] = { 2, 99, 3 }
0 1 2

DAT[] = 

Advantages

01 TC for Insertion → O(1)

02 TC for searching → O(1)

03. TC for deletion → O(1)

Issues

01. lot of space wastage

02 -ve ele ⇒ Mapping

03. Max array size allowed with = 10^6 to 10^7

10^7 integers

⇒ $4 * 10^7$ bytes

⇒ $4 * 10 * \underbrace{10^3 * 10^3}_{1\ MB}$ bytes

⇒ 40 MB → for 1 array

1 kb = 10^3 bytes

1 MB = 10^6 bytes

Q If memory limit of M is given, ∀ A[i], check if it is present or not.

M = 10

DAT =

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

A[i] $\xrightarrow{map}$ [0 to 9]

key → f( ) → valid index

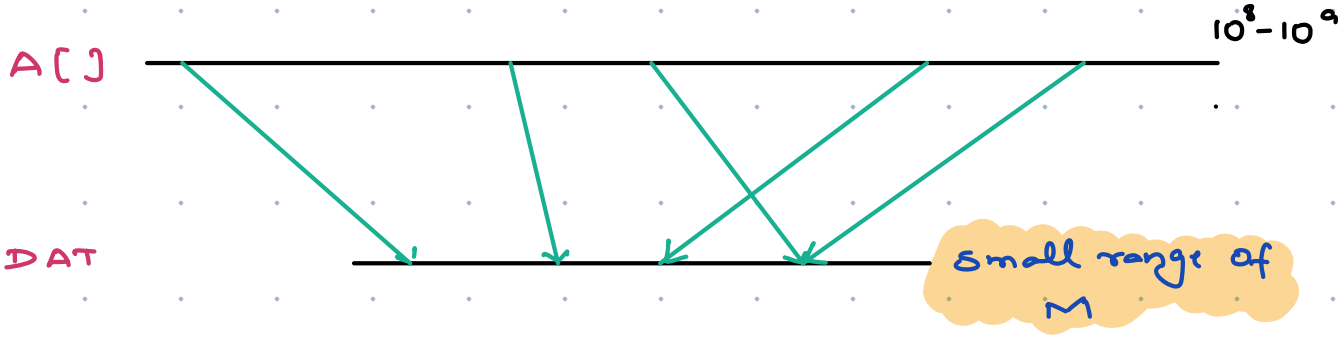
hash function — index value

A[i]	h(A[i])
27	27 % 10 = 7
18	18 % 10 = 8
21	21 % 10 = 1
32	32 % 10 = 2
37	37 % 10 = 7

Key 1
Key 2
} Hash function = generate same value

Collision

Can we avoid collision ? → NO



* Pigeon hole principle

→ If there are N pigeons & (N-1) holes, then there will be atleast one hole with more than one pigeon.

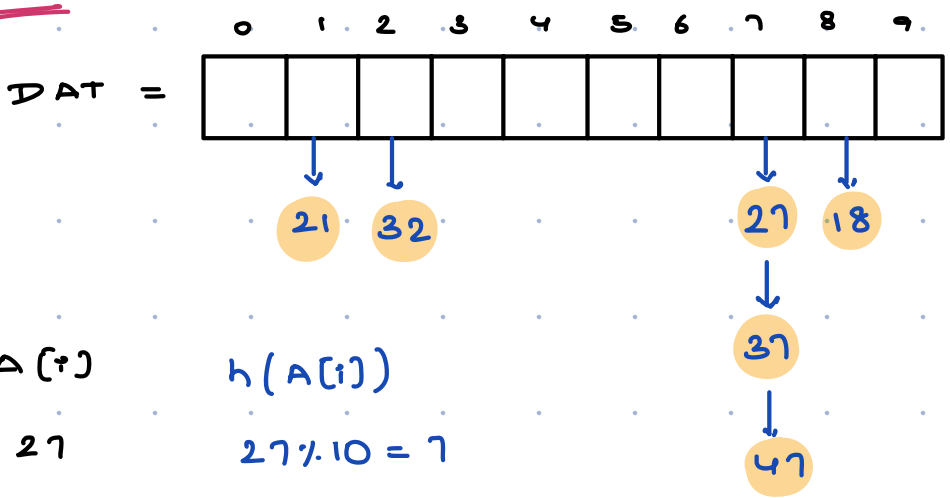
* We can handle collosion ? Yes

Collision Handling / Resolution Technique

- **Open hashing / Chaining**
- **Closed Hashing**
  - → Linear probing
  - → Quadratic probing
  - → Double Hashing

* Chaining

M = 10



A[i]	h(A[i])
27	27 % 10 = 7
18	18 % 10 = 8
21	21 % 10 = 1
32	32 % 10 = 2
37	37 % 10 = 7
47	47 % 10 = 7

for insert - TC : O(1) tail

TC for query > O(1)

Worst case search → O(n)

Average TC for searching = λ

Ratio of No. Of elements inserted / size of DAT

8 elements in array of size 10

$$\lambda = \frac{8}{10} = 0.8$$

80 elements in array of size 10

$$\lambda = \frac{80}{10} = 8$$

800 elements in array of size 10

$$\lambda = \frac{800}{10} = 80$$

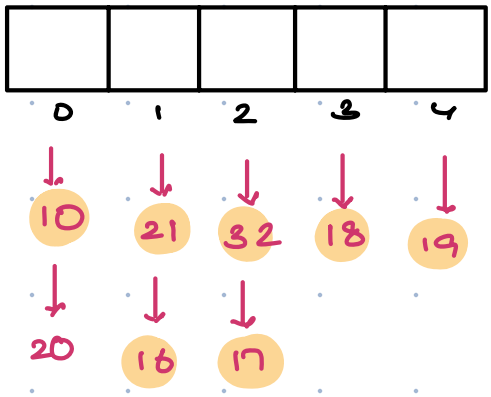
* There is a predefined threshold for λ, let say $\lambda = 2$

If λ becomes greater than threshold,

Rehashing { Redistribute the existing elements on a new DAT array, which will be double the size of old DAT array

Threshold value = 1.5

DAT [5] =



A[i]	h(A[i])
10	10 % 5 = 0
21	21 % 5 = 1
32	32 % 5 = 2
16	16 % 5 = 1
17	17 % 5 = 2
18	18 % 5 = 3
19	19 % 5 = 4
20	20 % 5 = 0

$\lambda = \frac{2}{5} = 0.4$

$\lambda = \frac{5}{5} = 1$

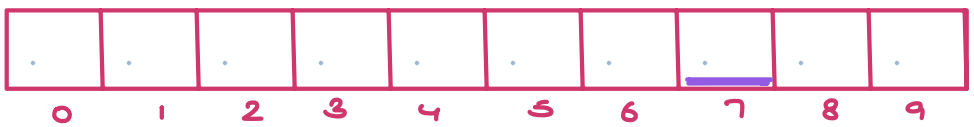
$\lambda = \frac{7}{5} = 1.4$

$\lambda = \frac{8}{5} = 1.6$

* At this position, $\lambda = 1.6$ so rehash

Rehashing

DAT =



Searching in DAT = $O(\lambda)$

Hashmap < K, V >

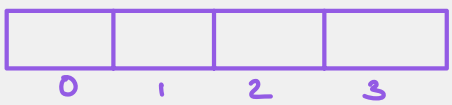
Code

```
import java.util.ArrayList;

class HashMap < K, V > {

    private class HMNode {
        K key;
        V value;

        public HMNode(K key, V value) {
            this.key = key;
            this.value = value;
        }
    }

    private ArrayList < HMNode > [] buckets;
    private int size; // number of key-value pairs

    public HashMap() {
        initbuckets();
        size = 0;
    }

    private void initbuckets() {
        buckets = new ArrayList[32];
        for (int i = 0; i < 32; i++) {
            buckets[i] = new ArrayList<>();
        }
    }
```

HMNode { key, value }

// no. of elements stored in bucket array

HashMap h = new HashMap()



Put

```
    public void put(K key, V value) {
        int bi = hash(key);
        int di = getIndexWithinBucket(key, bi);

        if (di != -1) {
            // Key found, update the value
            buckets[bi].get(di).value = value;
        } else {
            // Key not found, insert new key-value pair
            HMNode newNode = new HMNode(key, value);
            buckets[bi].add(newNode);
            size++;

            // Check for rehashing
            double lambda = size * 1.0 / buckets.length;
            if (lambda > 2.0) {
                rehash();
            }
        }
    }
```

Hash function

```
private int hash(K key) {
    int hc = key.hashCode();
    int bi = Math.abs(hc) % buckets.length;
    return bi;
}
```

for - each loop

```
private int getIndexWithinBucket(K key, int bi) {
    int di = 0;
    for (HMNode node : buckets[bi]) {
        if (node.key.equals(key)) {
            return di; // Key found
        }
        di++;
    }
    return -1; // Key not found
}
```

AL

for (int ⓘ : AL)

```
private void rehash() {
    ArrayList<HMNode>[] oldBuckets = buckets;
    initbuckets();
    size = 0;

    for (ArrayList <HMNode> bucket : oldBuckets) {
        for (HMNode node : bucket) {
            put(node.key, node.value);
        }
    }
}
```

→ 2 * oldbuckets.length

```
public V get(K key) {
    int bi = hash(key);
    int di = getIndexWithinBucket(key, bi);

    if (di != -1) {
        return buckets[bi].get(di).value;
    } else {
        return null;
    }
}
```

```
public boolean containsKey(K key) {
    int bi = hash(key);
    int di = getIndexWithinBucket(key, bi);

    return di != -1;
}
```

```
public V remove(K key) {
    int bi = hash(key);
    int di = getIndexWithinBucket(key, bi);

    if (di != -1) {
        // Key found, remove and return value
        size--;
        return buckets[bi].remove(di).value;
    } else {
        return null; // Key not found
    }
}
```

```
public int size() {
    return size;
}
```

```
public ArrayList<K> keyset() {
    ArrayList<K> keys = new ArrayList<>();
    for (ArrayList<HMNode> bucket : buckets) {
        for (HMNode node : bucket) {
            keys.add(node.key);
        }
    }
    return keys;
}
```

DSA

- → Backtracking + .
- → Rolling hash fn
- → Inverse modulo
- → String pattern match.
- → DP on string
- → Graph Algorithm.

40 - 50 LPA

Contest + Mock

{ DSA 4.2 }

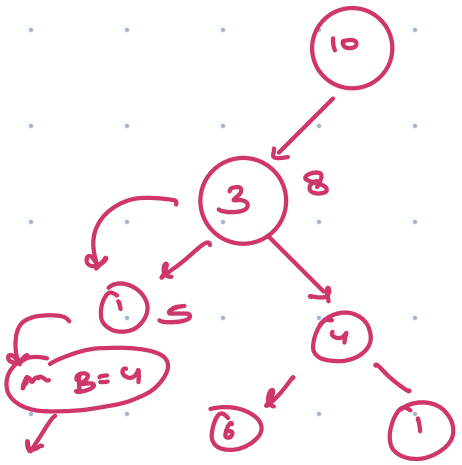
{ SQL } { DSA 4.2 }

Intern Recruiting

At the end of curriculum

Doubts

Root to leaf with sum = B




```
if path ( root, B) {
    if (A==null) return 0;
```

3