NOTES:

HashMap 3

Good Evening

Today's Agenda

→ HashMap Implementation

→ longest consecutive sequence

01. Given an array of size N & Q queries.

In each query, an element is given. We have to check if that element is present in A[] or not.

A[] = { 10 20 3 4 7 8 }

3 Queries

K = 7 ⟶ True

K = 9 ⟶ False

K = 13 ⟶ False

Brute force → For each query, loop through given array & search element K.

TC : O(N * Q)

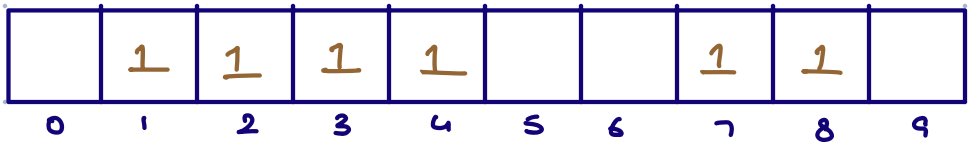
SC : O(1)

* Optimal Approach

→ Create a DAT where we are going to map A[i] with index

A[] = { 1 2 3 4 7 8 }
0 1 2 3 4 5

DAT [ ] =



direct access table

```
for (i=0; i<n; i++){
    freq [A[i]]++;
}
```

3

```
for ( int val : A ){
    freq [val]++;
}
```

3

Advantages of DAT

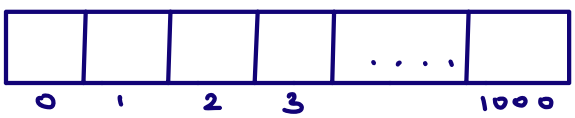
01. Insertion → O(1)
02. Deletion → O(1)
03. Search → O(1)

* Issues with this representation

01. Wastage of space

A[ ] = {1 3 1000}

DAT =



02. Inability to create larger arrays

03. Storing values other than integer is also complicated

* Overcome all these Issues

A[] = {21 42 37 45 99 30}

DAT[10] : Map all these elements in Array of size 10 by taking modulus with 10.

DAT[21 % 10] = DAT[1]
DAT[42 % 10] = DAT[2]
DAT[37 % 10] = DAT[7]
DAT[45 % 10] = DAT[5]

process is known as hashing

Basic hash function

↳ Provides an index at which we will mark the presence of particular index.

* HashMap & Hashset works on Hashing

→ process where we pass data to hash function
→ Hash function will generate a hash value (index) to which we map our data

* Issues with Hashing

Collision

A[ ] = {23 43 37}

DAT[23 % 10] = DAT[3] // marking presence of 23 at 3rd index

DAT[43 % 10] = DAT[3] // marking presence of 43 again at 3rd index

Issues → Two elements are generating the same hash value

Can we avoid collision ⟶ No

Pigeon Hole principle

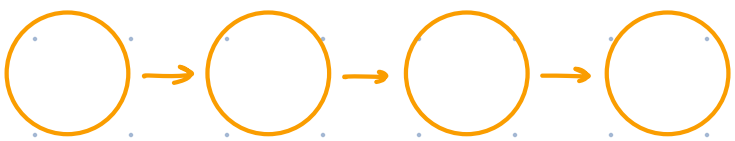
- Say we have 10 pigeon & 7 holes & every pigeon wants to sit in a hole.

There will atleast be 2 pigeon sitting in the same hole.

Collision Resolution Techniques

- Open Hashing
  - Chaining
- Closed Hashing
  - Linear probing
  - Quadratic probing
  - Double Hashing

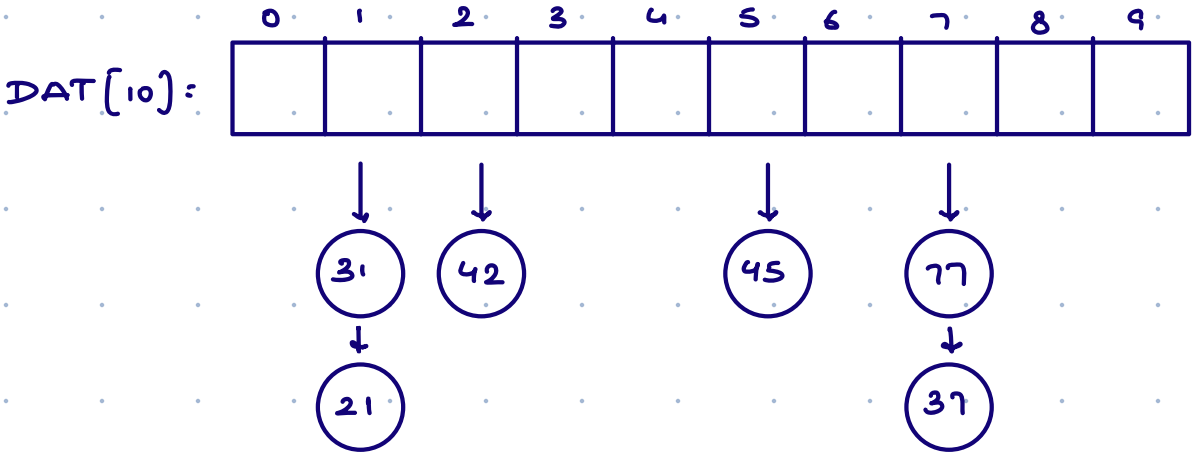
* Linkedlist or Arraylist of Nodes



* **Chaining** → For a particular index, if we have more than one element having same hash value, we are going to create a chain of Nodes / Linkedlist

A[ ] = { 21 42 37 45 77 31 }

hash value	21%10	42%10	37%10	45%10	77%10	31%10



$\lambda = \frac{6}{10} = 0.6$

* Adding a node in DAT → O(1)
* Searching & Deletion → O(λ)

↓

λ (Loading factor) = $\frac{\text{No. of elements inserted}}{\text{size of Array}}$

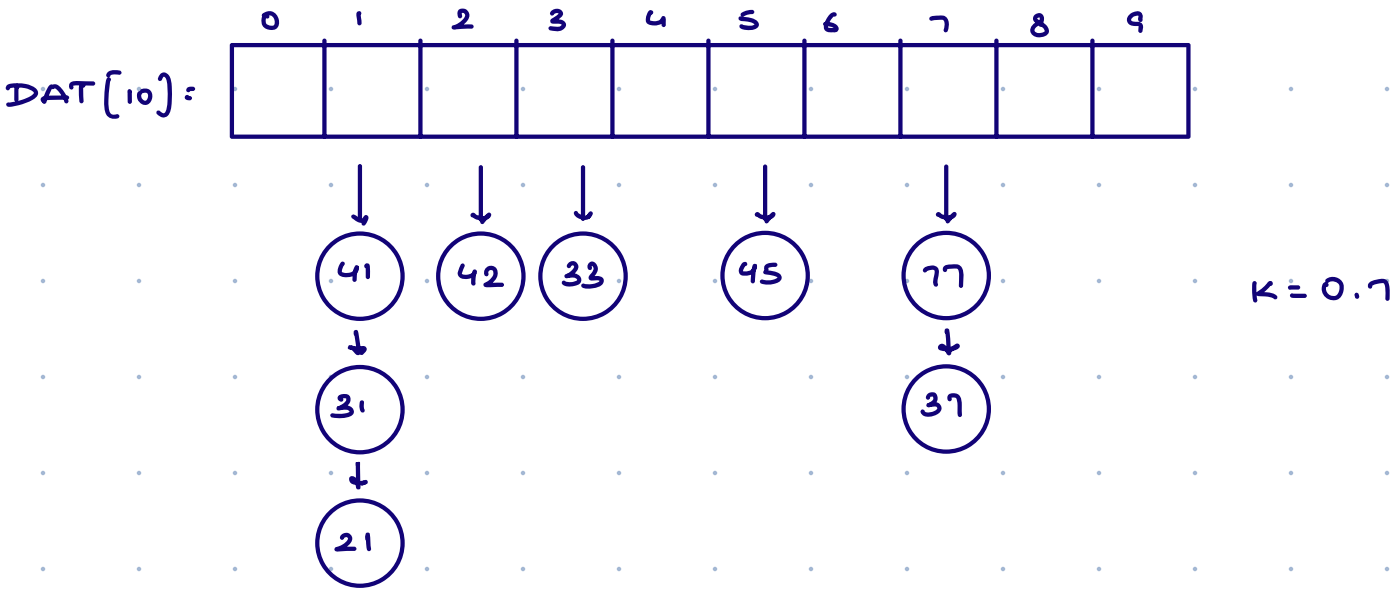
(K) Predefined threshold = 0.7

load factor can't exceed this threshold value. If it does that, then we have to do **rehashing**

Rehashing

01. Create a new DAT of size double to its previous size
02. Redistribute the existing element in new DAT

A[ ] = { 21 42 37 45 77 31 33 41 }

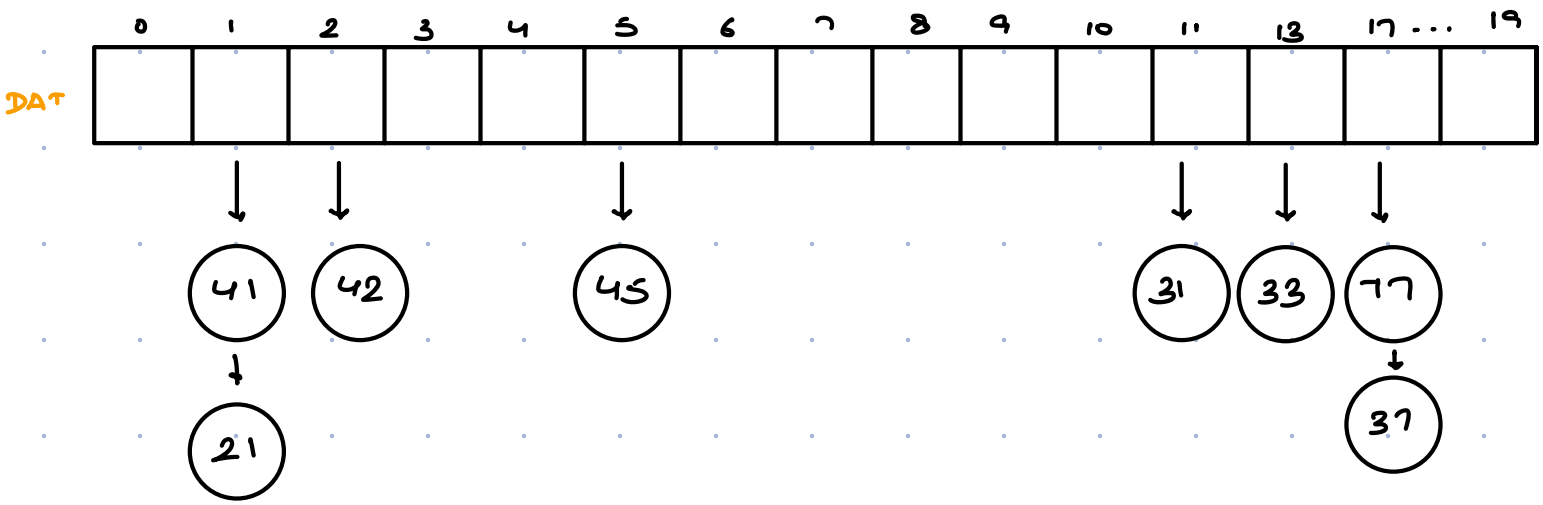


K = 0.7

$\lambda = \frac{6}{10} = 0.6$

$\lambda = \frac{7}{10} = 0.7$

$\lambda = \frac{8}{10} = 0.8 > K$ ⇒ Rehashing



$\lambda = \frac{8}{20} = 0.4 < K$

A[ ] = { 21 42 37 45 77 31 33 41 }

21%20 42%20 37%20 45%20 77%20 31%20 33%20 41%20

mapping ⇒ 1 2 17 5 17 13 13 1

Rehashing makes sure that deletion & searching in worst case senario will take only O(K) time

⇓

Threshold value

10:24 PM → 10:34 PM

```
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;
    }

                            int sz
  private void initbuckets() {
                              sz
      buckets = new ArrayList[4];
      for (int i = 0; i < 4; i++) {
          buckets[i] = new ArrayList<>();
      }
  }
```

```
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();
        }
    }
}
```

(threshold value) k = 2.0

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

```
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
}
```

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

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

initBuckets (2 * oldBuckets.length)

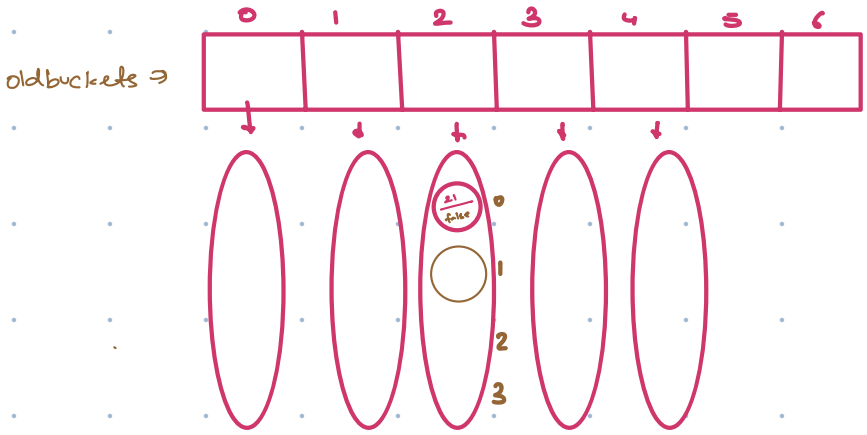
```
 public V get(K key) {
     int bi = hash(key);
     int di = getIndexWithinBucket(key, bi);

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

buckets



value

(21) → true

bi = 2

di = 0

di = 0

0th index e key = 21

buckets[bi].get(di).value

$\lambda = \frac{1}{6} = 0.16$

21.hashcode() = x

(8) % 6 = (2)

8 % 6 = 2

bi = 2