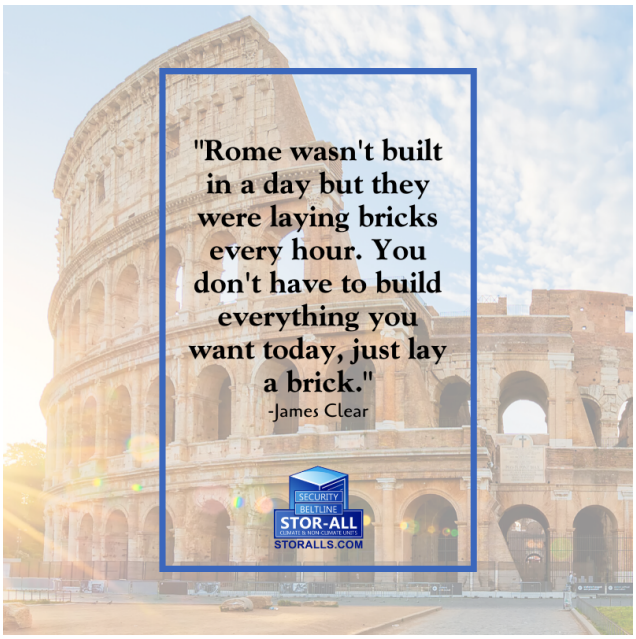
NOTES:

HashMap 3



Good Morning

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 every query, iterate through array & search for the element.

TC : O(Q*N)

SC : O(1)

Optimal Idea

→ Create a Direct Access Table (DAT) where every ele (A[i]) will be mapped with 'i' index.

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

DAT =

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

K = 7 ⟶ if (DAT [7] == 1) return true

DAT creation ↑ ⟶ answer all queries using DAT

TC : O(N + Q)

SC : O(max of A)

Advantages of DAT

01. Insertion → O(1)

02 Search → O(1)

Disadvantages of DAT

01. Wastage of space

A[ ] = { 1 100 }

DAT [101] =

0	1	2	3	. . .	100
1					1

02. Inability to create large arrays

03. storing values of -ve number is complicated.

* Overcome Issues in DAT

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

DAT[10] ⟶ Map all the elements inside this DAT of size 10 by taking modulus with 10

↓ fix size to DAT

modulus with 10 → Hash function

01. int idx = 21 % 10 = 1

DAT[1] = 1;

} Marking presence of 21 at index 1

02 int idx = 42 % 10 = 2

DAT[2] = 1;

} Marking presence of 42 at index 2

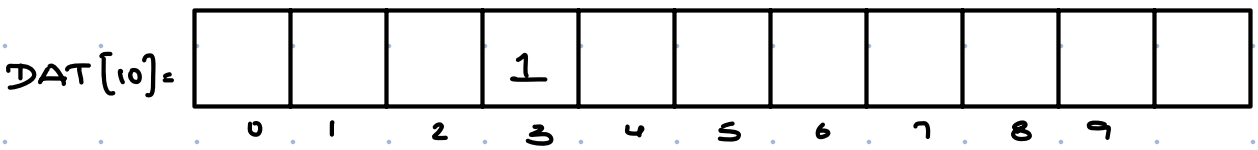
Hashing

* HashMap & HashSet works on Hashing concept

we will provide the element to hash function which will provide us the index where we have to go & mark presence of original ele.

* Issues with hashing

A[ ] = { 23 43 27 }



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

DAT[43 % 10] = DAT[3] // mark presence of 43 at index 3

Issue → Two elements are generating same hash value.

⇓

Collision

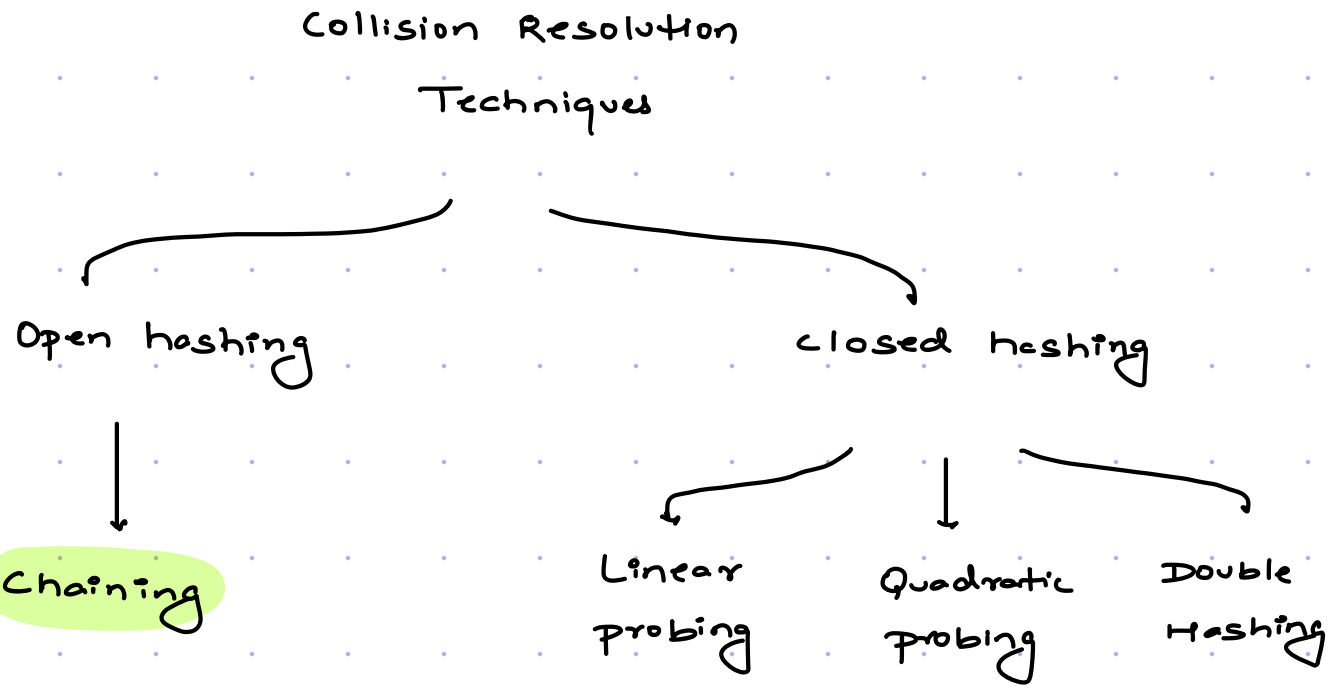
Can we completely avoid collision ? → No

* Pigeon hole principle

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

↓

There will atleast be one hole with more than one pigeon.



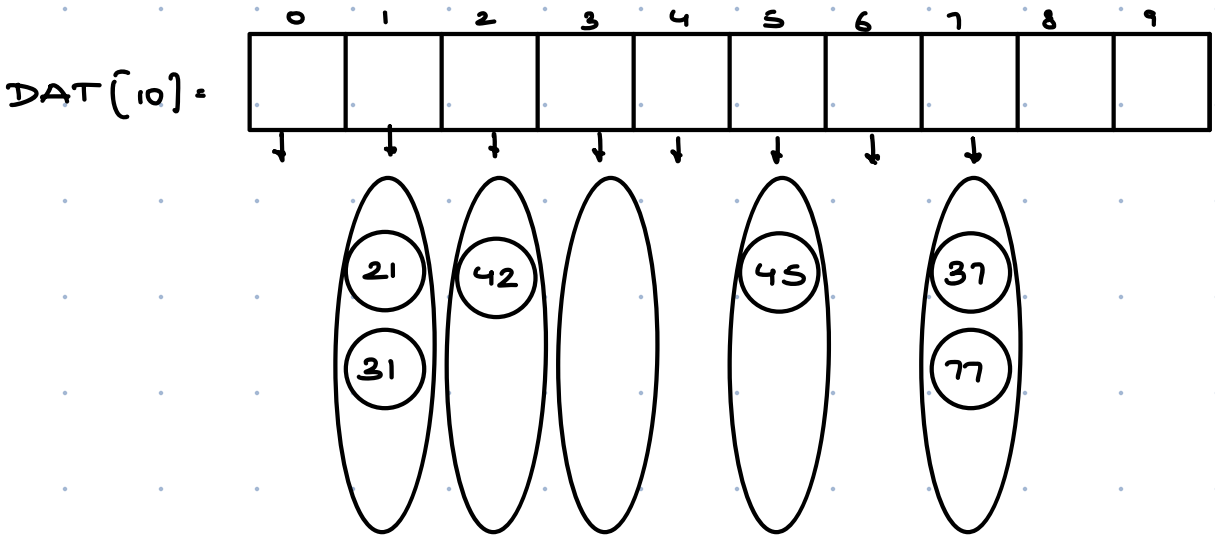
Chaining → For a particular index, if we have more than one element generating same hash value, then we can store the elements in a chain.

↓

Using chain of Linkedlist or Arraylist of Nodes

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

hash value (n%.10) = 1 2 7 5 7 1 2



* Insertion in DAT → O(1)

* Searching & Deletion → O(λ)

lambda (loading factor)

$$\lambda = \frac{\text{No. of elements inserted}}{\text{size of Array(DAT)}}$$

$\lambda \leq K$ (Threshold value) K = 0.7

↓

Loading factor can't exceed this threshold value,

if it exceeds, we will do rehashing

* Rehashing

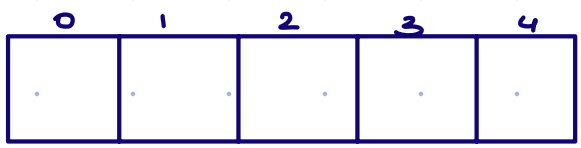
→ Creating a new DAT of size double the old DAT

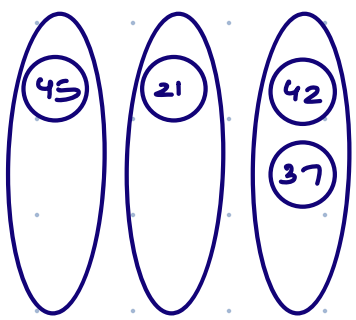
→ Redistribute the existing element in new DAT.

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

hash value (%5)	1	2	2	0	2	1	3	1

DAT [5] =





$k = 0.7$

$$\lambda = \frac{1}{5} = 0.2 \leq 0.7$$

$$\lambda = \frac{2}{5} = 0.4 \leq 0.7$$

$$\lambda = \frac{3}{5} = 0.6 \leq 0.7$$

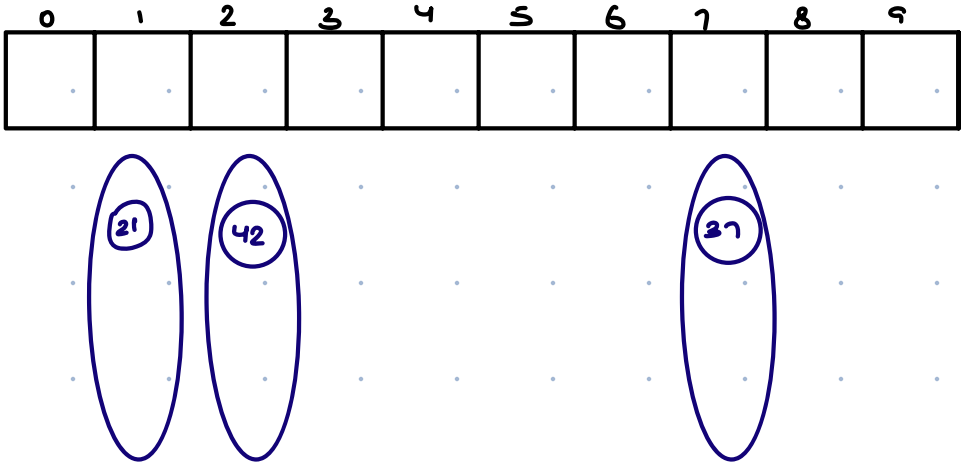
$$\lambda = \frac{4}{5} = 0.8 > 0.7 \longrightarrow \text{Rehashing}$$

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

hash value (%10)

1	2	7	5	7	1	3	1

DAT[10] =



$\lambda = \frac{1}{10} = 0.1 \leq 0.7$

$\lambda = \frac{2}{10} = 0.2 \leq 0.7$

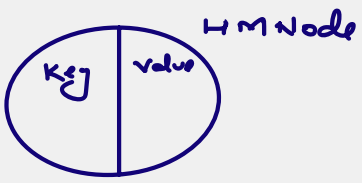
$\lambda = \frac{3}{10} = 0.3 \leq 0.7$

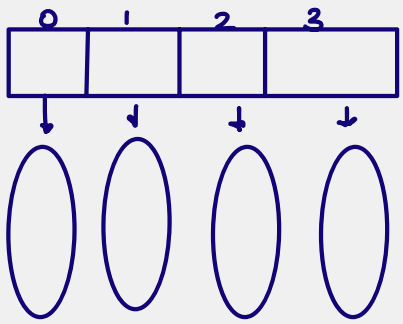
.
.
.

You will be able to store 7 elements before rehashing again

* 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
    private int sz = 4; //initial size of Array
    public HashMap() {
        initbuckets(sz);
        size = 0;
    }

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





sz

```
    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 nn = new HMNode(key, value);
            buckets[bi].add(nn);
            size++;

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

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 n : buckets[bi]) {
         if (n.key.equals(key)) {
             return di; // Key found
         }
         di++;
     }
     return -1; // Key not found
 }
```

```
private void rehash() {
    ArrayList<HMNode>[] oldBuckets = buckets;
    initbuckets(2 * oldBuckets.length);
    size = 0;

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

```
 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);
        } else {
            return null; // Key not found
        }
    }
```

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

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

8:48 → 8:55 AM

Longest Consecutive Sequence

Problem Statement

Given an array of integers, find the length of the longest sequence such that elements in the subsequence are consecutive integers, the consecutive numbers can be in any order.

A[ ] = { 100 3 1 2 4 }

Ans = 100 → 1

1 2 3 4 → 4

Ans = 4

A[ ] = { 1 9 3 10 4 20 2 }

Ans = 4 (1, 2, 3, 4)

A[ ] = { 3 7 2 10 1 11 0 4 8 }

3, 4 → 2

7, 8 → 2

2, 3, 4 → 3

10, 11 → 2

1, 2, 3, 4 → 4

11 → 1

Ans = 5

0, 1, 2, 3, 4 ⟶ 5

8 ⟶ 1

* Code

```
HashSet<I> hs = new HashSet<>();
ans = 0

for (i = 0; i<n; i++) hs.add(A[i]);

for (i = 0; i<n; i++){
    int curr = A[i];

    if (hs.contains(curr-1)) continue;
    else {
        int next = curr
        while (hs.contains(next))
            next = next + 1
        ans = Math.max(ans, next - curr);
    }
}

return ans;
```

TC : O(n)

SC : O(n)