Hashing 3 : Internal Implementation & Problems

Question

Given an integer array & multiple queries.

for every query, check if element X is present in array?

A = [2 4 11 6 8 9 1]

Query

x = 10   ans = false

x = 2   ans = true

Bruteforce → ∀ query, traverse the array & check

TC = O(Q * N)   SC = O(1)

Sol^n 2 → ∀ A[i], mark it visited in a seperate array.

Direct Access Table (DAT)

data[i] → true ⇒ element 'i' is present in A

data[i] → false ⇒ element 'i' is not present in A

A = [2 4 11 6 8 9 1]

$$\begin{array}{lcccccccccccc} & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\ \text{data} = & [f & T & T & f & T & f & T & f & T & T & f & T] \end{array}$$

```
∀i, data[i] = false

for ( i=0 to n-1 ) {                    TC=O(N)
    data[A[i]] =true
}
```

Query, X → ans = data[X]

TC = O(1) per query

total TC = O(N+Q)

SC = O(max (A[i]))

Advantage of DAT

TC of insertion, deletion, search in O(1)

Disadvantages of DAT

1. Wastage of space

A = [23, 60, 37, 91]

You have to create 92 size array to store 4 elements.

2. Inability to create large array

if max(A[i]) = 10^9 then we can't create data array (MLE error)

max array size allowed ~ 10^6

How to overcome issues and retain advantages ?

let say we have restriction to create array of size 10 only.

A = [21 42 37 45 99 30]

In array of size 10, index will be from 0 to 9.

=> use mod (% 10)

	0	1	2	3	4	5	6	7	8	9
data = [	T	T	T	f	f	T	f	T	f	T]

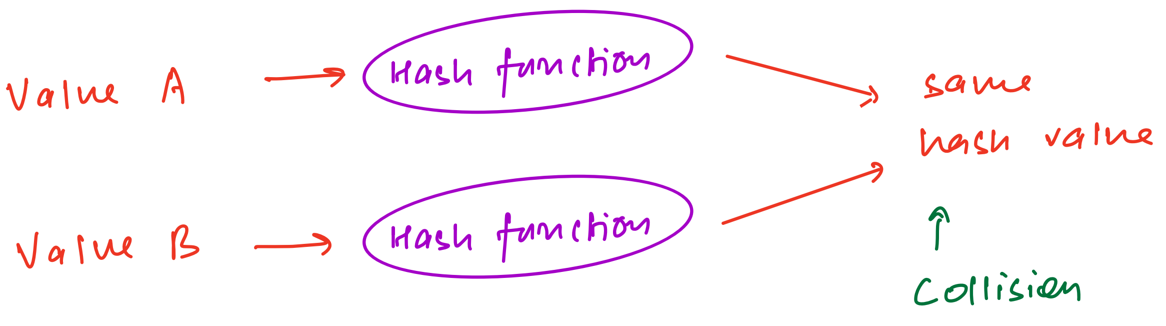
21 % 10 = 1 45 % 10 = 5

42 % 10 = 2 99 % 10 = 9

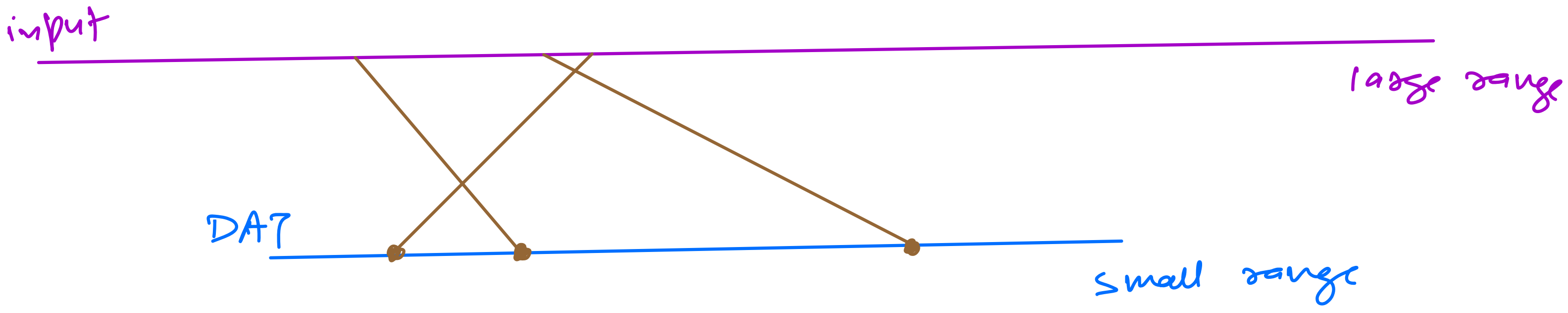
37 % 10 = 7 30 % 10 = 0

Issue with hashing

21 & 31 (mod 10) will map to same index 1.

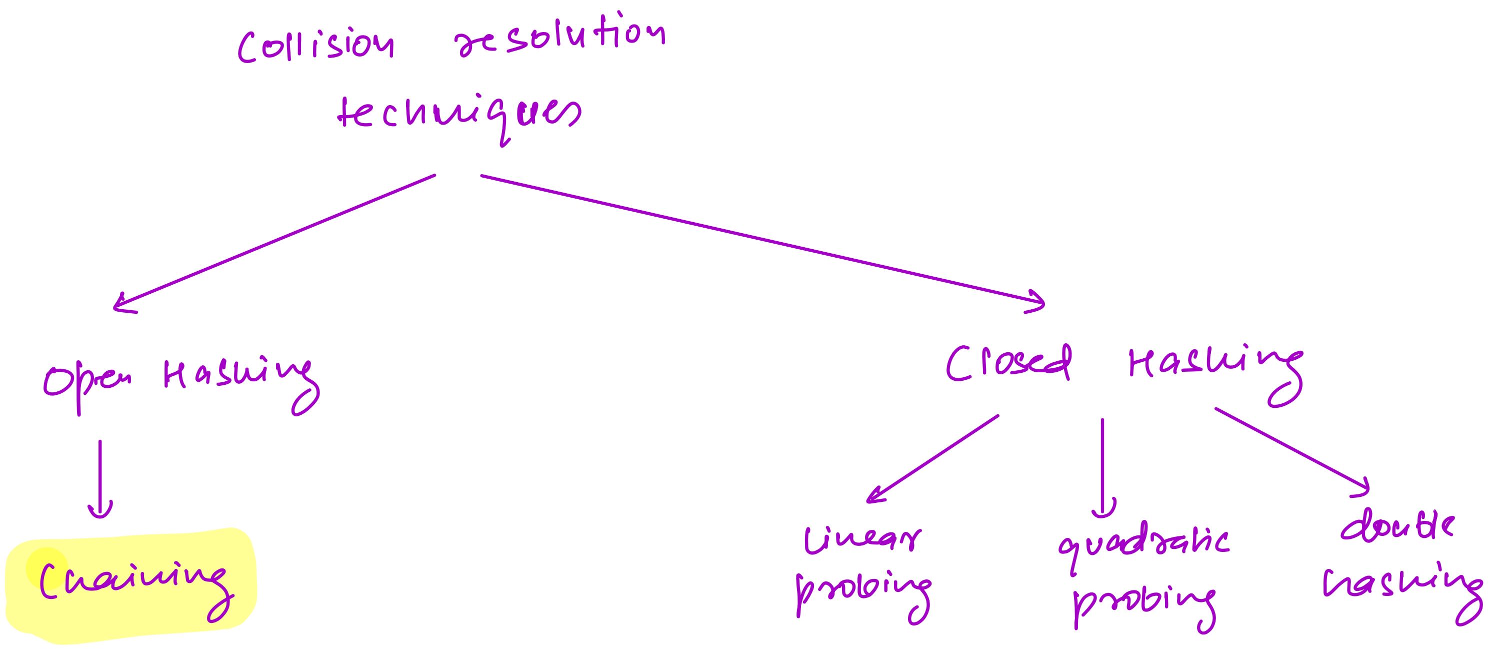


Can we avoid collision ? → NO



Pigeonhole Principle → If there are N pigeons & (N-1) holes. There will be atleast 1 hole with more than 1 pigeon.

Can we handle collision ? → YES


Collision resolution techniques

Open Hashing

Chaining

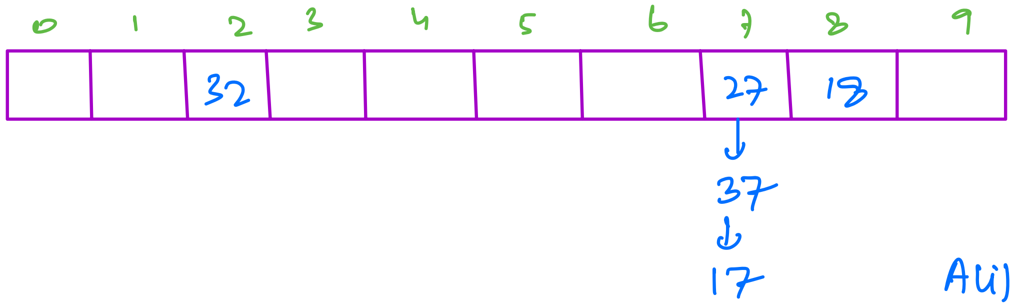
Closed Hashing

linear probing

quadratic probing

double hashing

Chaining



A[i]	h(A[i])
27	27%10 = 7
18	18%10 = 8
32	32%10 = 2
37	37%10 = 7
17	17 %10 = 7

TC of insertion in LL

insert at head → O(1)

TC of search or deletion ?

TC per query → O(1)

(worst case : O(N))

TC on average <= λ (lambda)

$$\downarrow$$

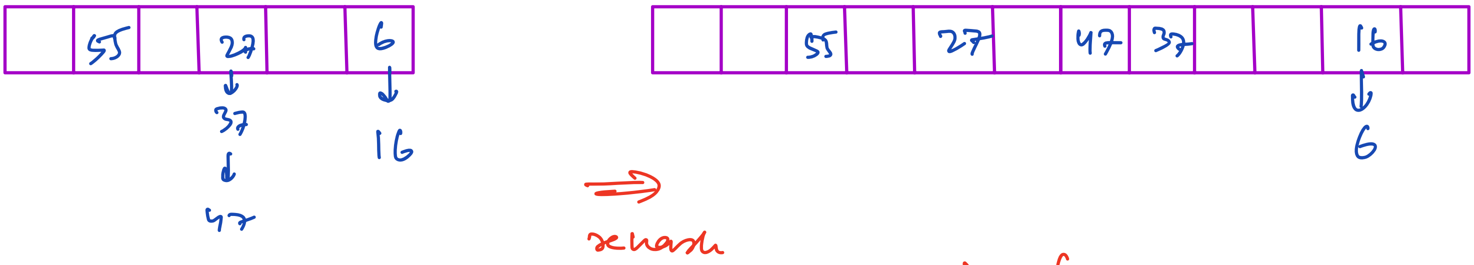
$$\frac{\#\ \text{elements inserted} \rightarrow 5}{\text{size of data array (DAT)} \rightarrow 10}$$

$\lambda = \frac{5}{10} = 0.5$

There is a predefined threshold for $\lambda = 0.7$

If λ becomes greater than threshold ⇒

Redistribute the existing elements on a new DAT array with double size.



$$\lambda = \frac{6}{6} = 1 > 0.7$$

$$\lambda = \frac{6}{12} = 0.5$$

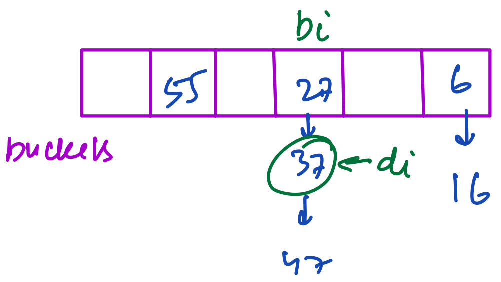
Code Implementation

```
class Hashmap <K,V> {

    private class HMNode {

        K key;
        V value;

        public HMNode (key, val) {
            this.key = key;
            this.value = val;
        }
    }
                                              → array of
    private Arraylist < HMNode > [] buckets;    Arraylist

    private int size; / number of key-val pairs

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

    private void initbuckets () {
        buckets = new Arraylist <> [4]; initial size of
                                         hash table
        for ( i=0 to 3 )
            buckets[i] = new Arraylist <>();
    }
```

Insertion

```
    void put ( K key, V value ) {
        int bi = hash(key); → index of bucket
        int di = get Index Within Bucket( key , bi);
```

```
if ( di != -1) {   → key is present
        buckets[bi].get(di).value = value;
}
else {   → key not present
    HMNode temp = new HMNode( key ,value )
    buckets[bi].add (temp);
    size ++;
                                      for decimal
    double lambda = size * 1.0 / buckets.length ;
    if( lambda > 0.7) {
        rehash();
    }
}
}
```



```
private int hash ( K key) {
    int hc = key.hashcode() ;
    return hc % buckets.length
}
```

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

```
private void rehash () {    → O(size)
    Arraylist<HMNode> [] oldBuckets = buckets;
    buckets = new Arraylist [oldBuckets.length *2]
    // copy old bucket in new bucket
}
```

```
public V get( K key) {

    int bi = hash(key);
    int di = getIndex Within Bucket(key, bi);

    if( di == -1)
        return NULL

    else
        return buckets[bi].get(di). value;

}
```

```
public bool containskey ( K key)
    int bi = hash(key);
    int di = getIndex Within Bucket(key, bi);

    return (di != -1);

}
```

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

    if (di == -1)
        return NULL
    else {
        size--;
        return buckets[bi].remove(di).value;

}
```

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

```
public Arraylist<K> Keyset() {        → O(N)
    Arraylist<K> keys = new Arraylist<>();
    for( Arraylist<HMNode> bucket : buckets) {
        for( HMNode node : bucket) {
            keys.add( node.key)
        }
    }
    return keys
}
```

Question

Given an array, find the length of longest sub-sequence such that elements in subsequence are consecutive (in any order).

A = [1 9 3 10 4 20 2] ans = 4

A = [36 41 56 35 44 33 34 92 43 32 42]

ans = 5

Idea 1 : sort the array

A = [1 2 3 4 9 10 20]

iterate & find the longest consecutive

TC = O(NlogN)

SC = O(1)

Idea 2 :

1. insert in Hashset
2. Iterate over array & check whether current element is starting point or not ?
3. find the length

code

```
hashset<int> hs

ans=0

for(i=0 to n-1){        → O(N)
    hs.put(A[i])
}

for(i=0 to n-1){
    if(!hs. contains( A[i]-1)){   ← A[i] is starting point
        s= A[i]
        while ( hs. contains(s)){
            s++
        }
        ans = max( ans,  s - A[i])
        }

    }

    return ans
```

TC = O(N)

SC = O(N)

```
A = [  1    9    3    10    4    20    2 ]
       ↑    ↑    ×    ×     ×    ↑     ×
       j
```

while
run 4 times
(1,2,3,4)

total iterations of WHILE loop
is O(N)