Lecture :- Hashmap Implementation

Agenda

- Hashmap implementation challenges

Qu Given arr[n] and Q queries. In each query, an element is given. Check whether that element exists in an array or not.

2	4	11	15	6	8	14	9

Queries	
k = 4	t
k = 10	f
k = 17	f
k = 14	t

Brute force approach

TC: O(n * Q)

SC: O(1)

Observation

A =

2	4	11	15	6	8	14	9

DAT = Direct access table

dat[] =

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
f	f	f	f	f	f	f	f	f	f	f	f	f	f	f	f

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

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
f	f	t	f	t	f	t	f	t	t	f	t	f	f	t	t

TC: O(a) + O(n)
SC: O(max)

Advantages of DAT

1. T.C of insertion = O(1)
2. T.C of deletion = Point of discussion.
3. T.C. of searching = O(1)

Issue with such representation

1.) Wastage of space.

23	60	37	91

⟶ DAT array size ⇒ 92

2.) Inability to create big arrays.

1	10^{10}	10^9	8

DAT[$10^{10}+1$]

3.) Storing values other than +ve integer.

1	2	-100	26

DAT[27]

Overcoming issues while retaining advantages

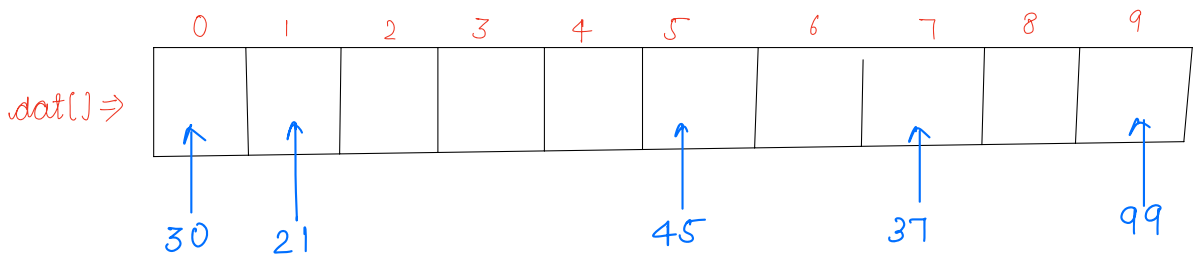
Assumption : DAT array ⟶ Restrict size = 10

21	42	37	45	99	30

How to map all value with indices ?

Hash
Mapping func^n : idx = A[i] % 10

array elements	mapped index A[i] % 10
21	1
42	2
37	7
45	5
99	9
30	0



Issues with hashing

A ⇒

21	42	37	45	77	99	31

Mapped idx of 21 and 31 = 1

" " " 37 and 77 = 7

21 ⟶ idx = 21 % 10 = 1
31 ⟶ idx = 31 % 10 = 1
} collision

Can we completely avoid collision ?

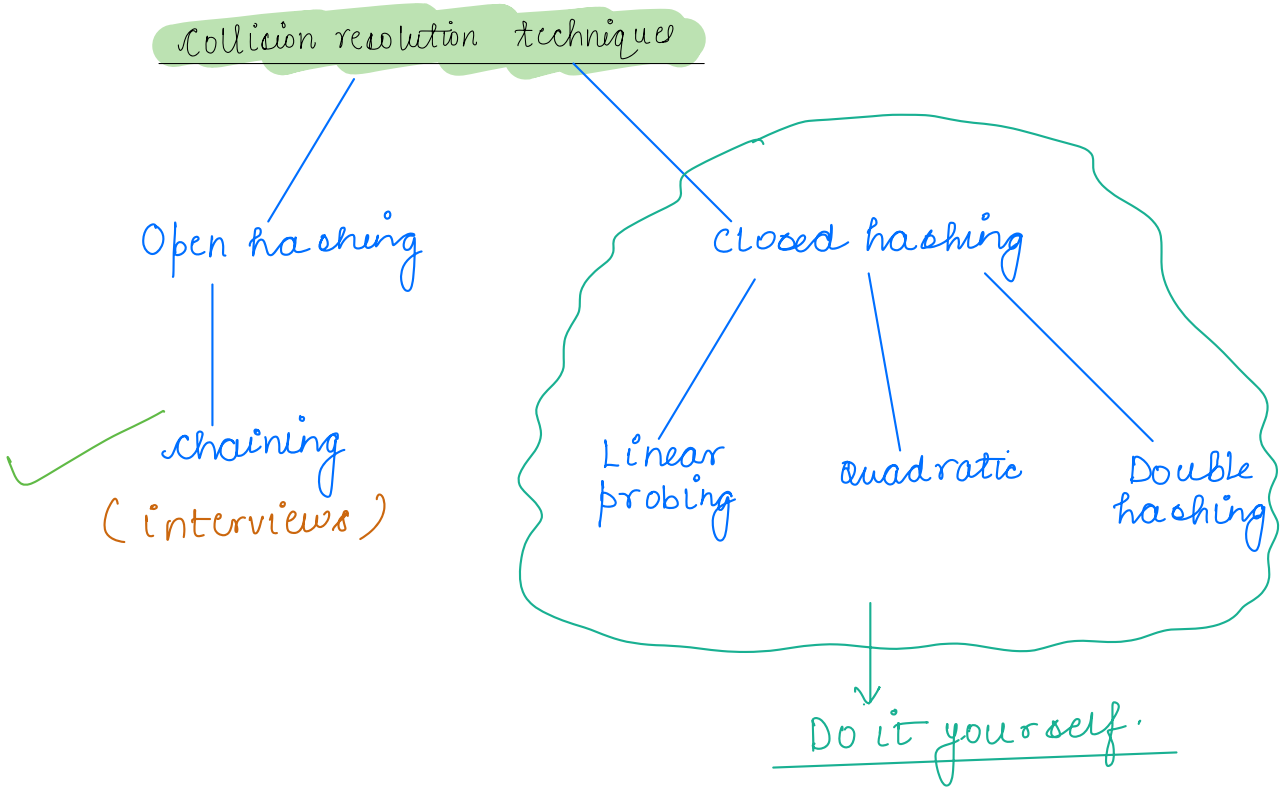
No

Reason: Pigeon hole principle

Pigeon = 11

hole = 8

At least 1 hole ≥ 2 pigeons.


Collision resolution techniques
Open hashing
Closed hashing
chaining
(interviews)
Linear probing
Quadratic
Double hashing
Do it yourself.

Chaining

{ Hash Mapped func = A[i] % 10 }

A ⇒

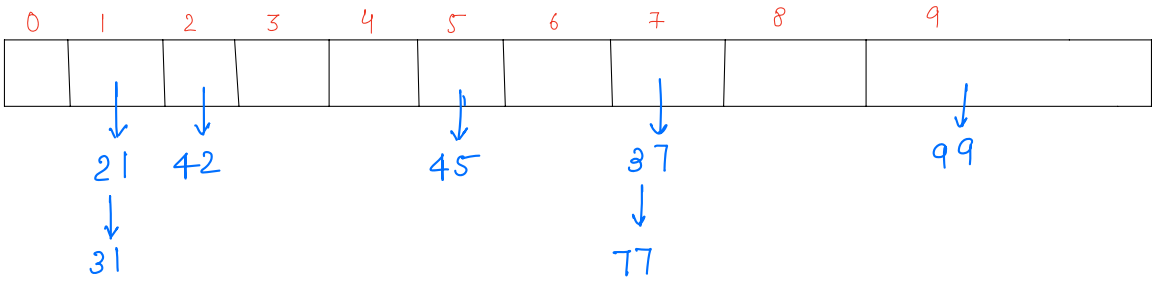
21	42	37	45	77	99	31

Mapped idx of 21 and 31 = 1

" " " 37 and 77 = 7

How can we resolve collision here?

Somehow store 21 & 31
37 & 77 at same idx.



Chaining is a technique used in data structures, particularly hash tables, to resolve collisions. When multiple item hash to same index, chaining stores them in a linked list or another data structures at that index.

Time complexity of insertion

el ⟶ hash func ⟶ idx ⟶ go to that idx

	LL	ArrayList
insert at tail ⇒	O(n)	O(1)
insert at head ⇒	O(1)	O(n)

Time complexity of deletion and searching

- Average
- Worst

Lambda (λ)

$\lambda \Rightarrow \frac{\text{total el inserted}}{\text{size of array.}}$

Random example:

DAT array
Hash table

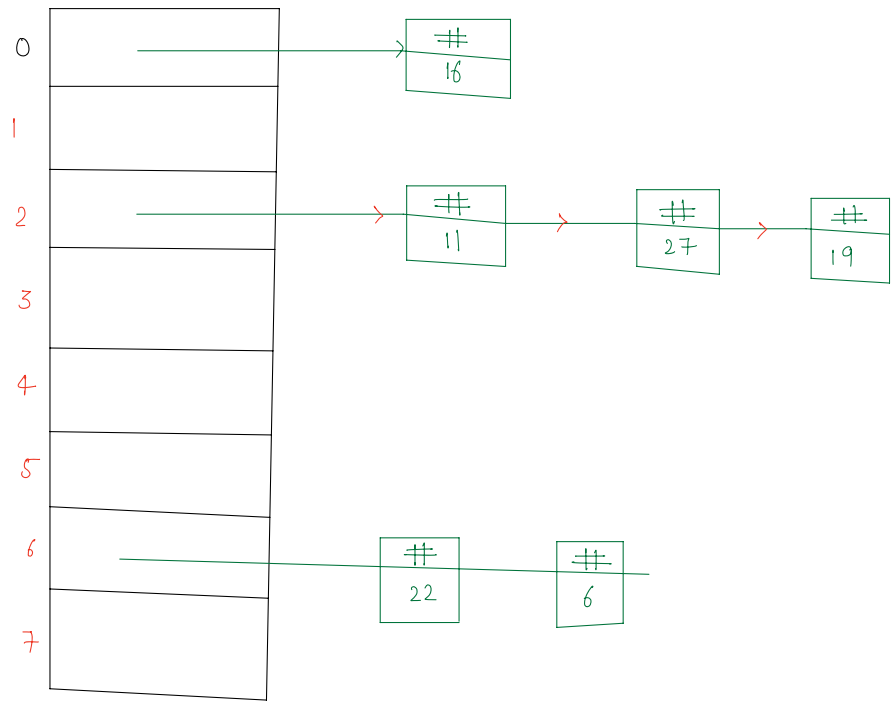


Table size $\Rightarrow$ 8

Inserted elements $\Rightarrow$ 6

$\lambda \Rightarrow \frac{6}{8} = 0.75$ { load factor }

Predefined threshold $\Rightarrow$ 0.7

Rehashing

Hash table

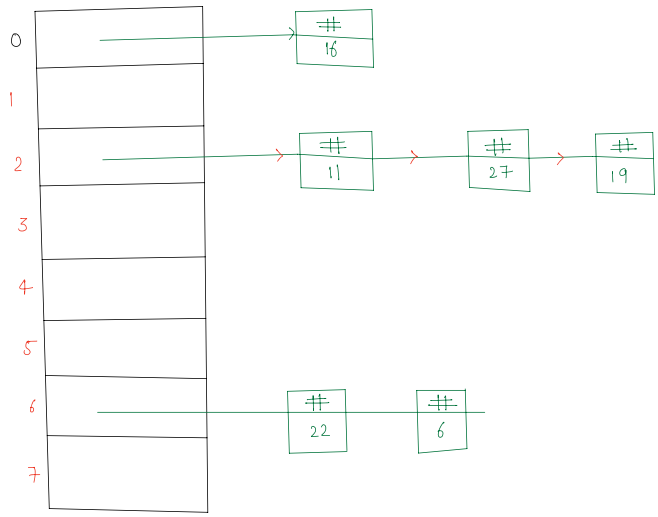


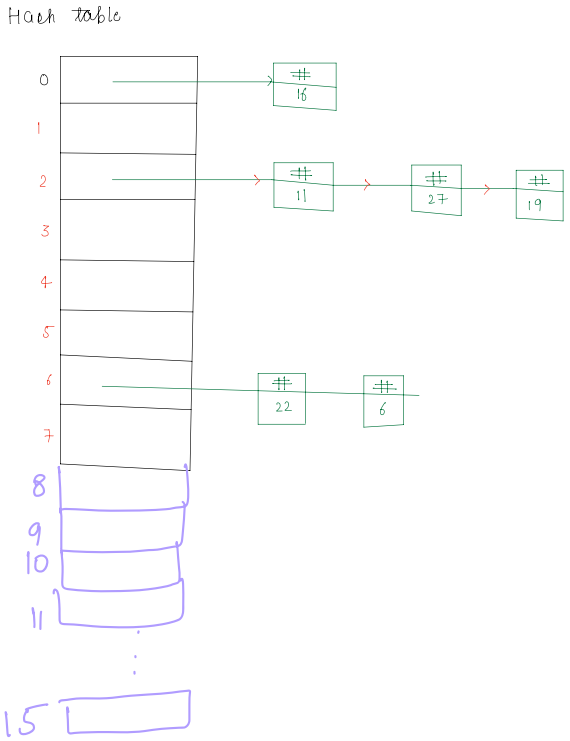
Table size ⇒ 8

Inserted elements ⇒ 6

$\lambda \Rightarrow \frac{6}{8} = 0.75$ [load factor]

Insert a new value : probability ——— x

Hash table



Steps

- Double size of DAT array.
- Restributing all el again

Table size ⇒ 16

Inserted elements ⇒ 6

$\lambda \Rightarrow \frac{6}{16} = 0.375$

Insert a new value : probability = y

$y <<<< x$.

Break: 8:03 - 8:15

Code implementation

Structure

```
class AyushHashMap<K, V> {
    class HMNode {
        K key;
        V value;

        Constructor
    }

    ArrayList<HMNode>[] buckets
    int size;
```

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

Put method

```
public void put(K key, V value) {
    bi = hash(key);                          // O(1)
    di = getIndexWithinBucket(key, bi);      // O(n)
    if (di != -1) {
        key found. update the value.
        buckets[bi].get(di).value = value;
    } else {
        HMnode node = new HMnode(key, value);
        buckets[bi].add(node);
        size++;

        Check for rehashing
        lamda = size * 1.0 / buckets.length;
        if (lambda > 0.7) {
            rehash();                        // O(n)
        }
    }
}
```

Hash Method

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

Rehash method

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

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

Get method

```
V get (K key) {
    bi = hash(key);
    di = getIndexWithinBucket (key, bi);
    if (di == -1) {
        Key not found.
        return null;
    }
    return buckets[bi] . get (di) . value;
}
```

Contains key method

```
boolean containsKey(K key) {
    bi = hash(key);
    di = getIndexWithinBucket(key, bi);
    if(di == -1) {
        return false;
    }
    return true;
}
```

Remove method

```
V remove(K key) {
    bi = hash(key);
    di = getIndexWithinBucket(key, bi);
    if(di==-1) {
        return null;
    }
    size--;
    return buckets[bi].remove(di).value;
}
```

Size method

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

TC

Avg time complexity

└ O(1)

LIS

A[] —— length of longest sub sequence such that el in subsequence are consecutive integers.

1 9 3 10 4 20 2 [1 3 4 2]

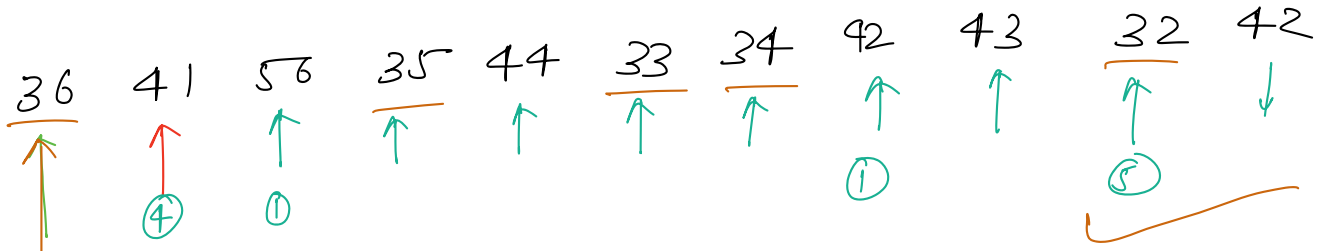
Ans = 4

36 41 56 35 44 33 34 42 43 32 42

Ans: 5

Approach set / Map

→ Insert all el in set



if(A[i]-1 present in set){
sequence cant start from here
} else {

}