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	true
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

Advantages of DAT

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

Issue with such representation

1.) Wastage of space.

23	60	37	90

⟶ DAT array size ⇒ 91

2.) Inability to create big arrays.

1	10^{15}	10^{9}	8

3.) Storing values other than +ve integer.

1	2	-100	26

Overcoming issues while retaining advantages

Assumption : DAT array ⟶ Restrict size = 10

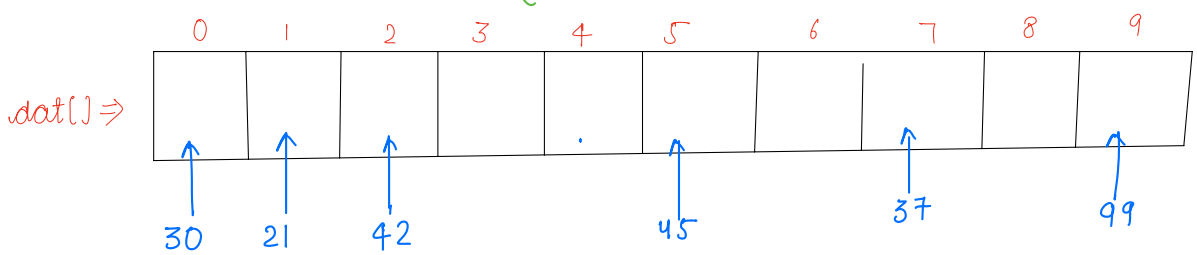
21	42	37	45	99	30

How to map all value with indices ?

Mapping func^n : idx = arr[i] % 10 [hash function]

array elements	mapped index
21	21 % 10 = 1
42	2
37	7
45	5
99	9
30	0

(Hash table)



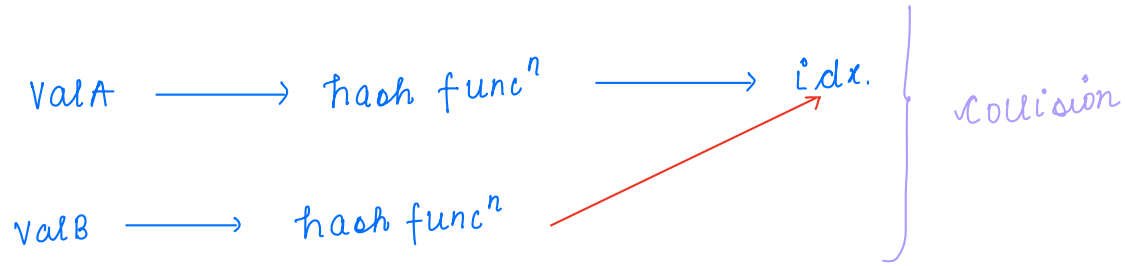
Issues with hashing

A ⇒

21	42	37	45	77	99	31

Mapped idx of 21 and 31 = 1

" " " 37 and 77 = 7



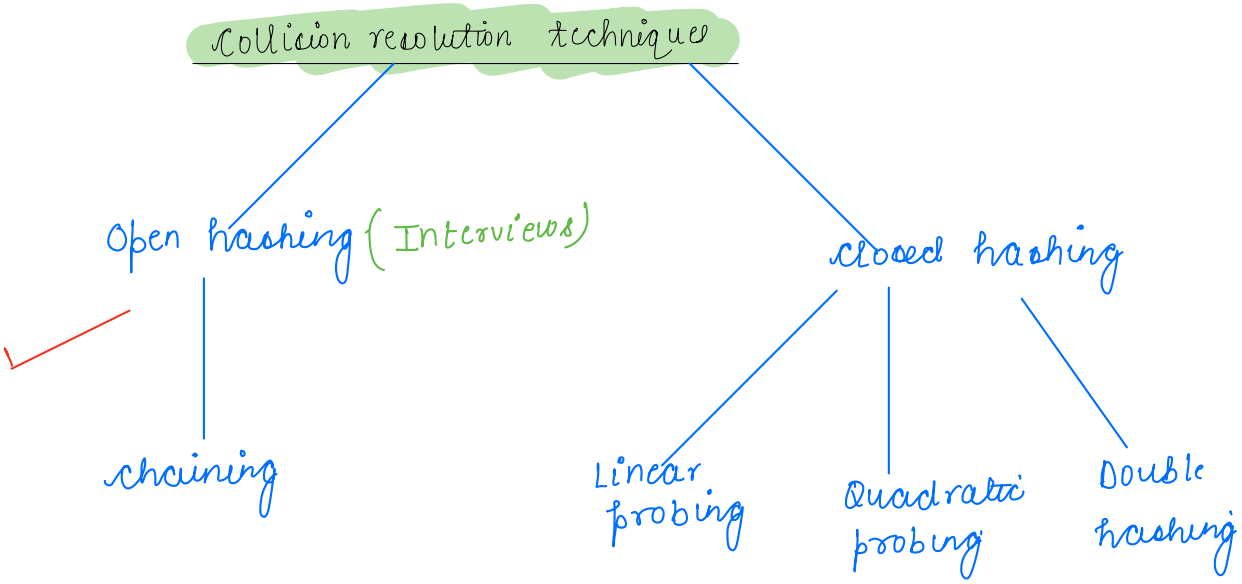
Can we completely avoid collision ? [No]

Reason: Pigeon hole principle.

Pigeon = 11

Holes = 8

At least 1 hole having >= 2 pigeons.


Collision resolution techniques
Open hashing (Interviews)
closed hashing
chaining
Linear probing
Quadratic probing
Double hashing

Chaining

A ⇒

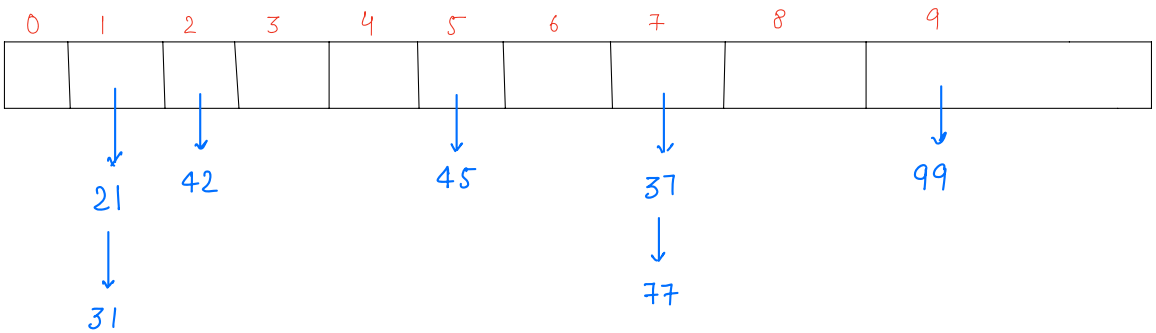
21	42	37	45	77	99	31

Mapped idx of 21 and 31 = 1

" " " 37 and 77 = 7

How can we resolve collision here?

Somehow store 21 & 31 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

insert at tail ⇒ O(n)

insert at head ⇒ O(1)

Time complexity of deletion and searching

- Average ⟶ 1
- Worst O(n)

Lambda (λ)

$\lambda \Rightarrow \frac{\text{Total elements inserted}}{\text{size of array}}$

Random example:

Hash table

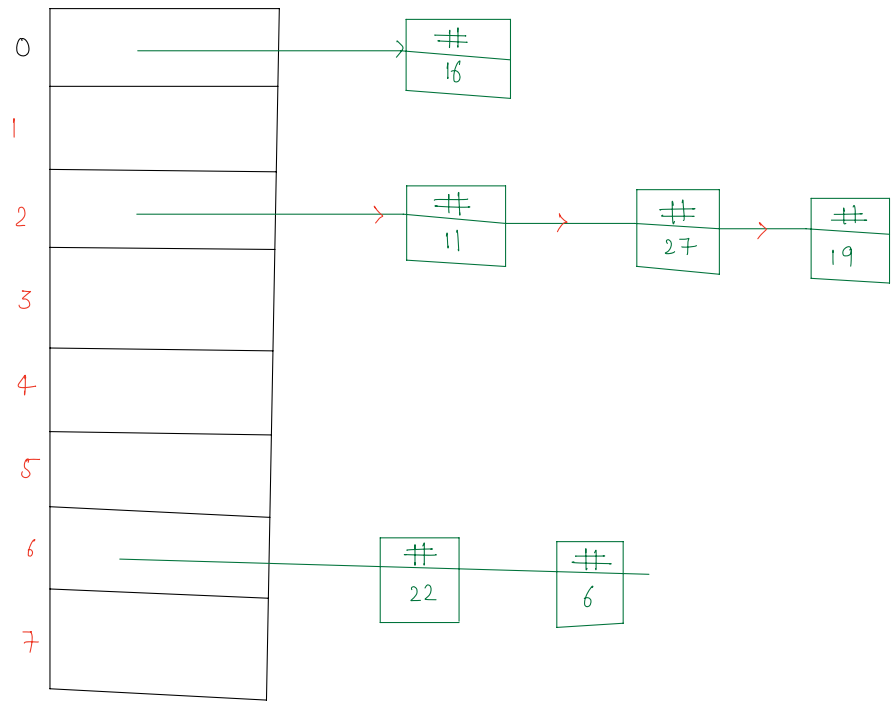


Table size $\Rightarrow$ 8

Inserted elements $\Rightarrow$ 6

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

Predefined threshold $\Rightarrow$ 0.7

Rehashing

Hash table

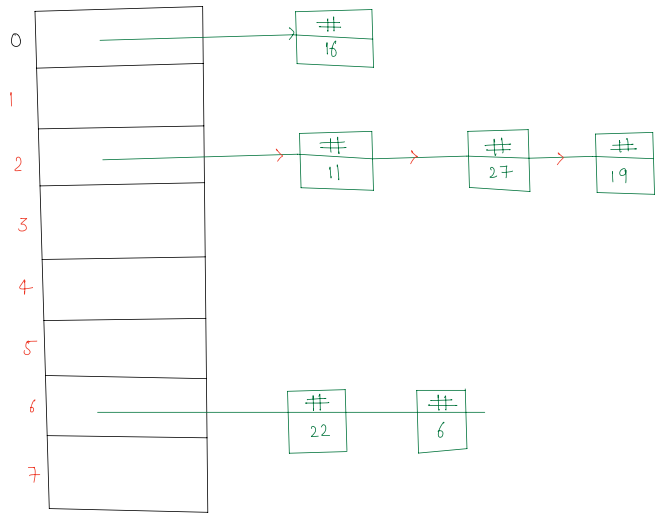


Table size ⇒ 8

Inserted elements ⇒ 6

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

Insert a new value

Prob of collision ⇒ $\frac{3}{8}$

Steps

- Hash table [2 * size of prev hash table]
- Redistribute all el again in new hash table.

Table size ⇒ 16

Inserted elements ⇒ 6

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

Insert a new value

Prob of collision ⇒ $\frac{1}{16}, \frac{2}{16}, \frac{3}{16} \dots \frac{6}{16}$

Break: 8:15 – 8:25

Code implementation

Structure

Put method

```
public void put(K key, V value) {
```

Hash Method

```
int hash(K key) {
```

Rehash method

```
void rehash() {
```

Get method

```
V get(K key) {
```

Contains key method

```
boolean containsKey(K key) {
```

Remove method

```
V remove( K Key) {
```

Size method

```
public int size() {

}
```

Key set method

ArrayList<K> keyset() {