3 Sept'20

DSA - GFG - 20 hours - (24)

(9/24) HASHING (70 min)

* (20 min)

Hashing is a DS that supports search, insert & delete in O(1)

C++ →
unordered_set , unordered_map

Java →
HashSet , HashMap , LinkedHashSet , LinkedHM,.....

Set stores single item (key)
Maps stores key-value pair

Set & Map don't insert duplicate values

Q1) Given an array, count all the distinct elements in it.
A) Unordered_Set OR HashSet

Q2) Given an array, count frequencies of all elements in it.
A) HashMap

LinkedHashSet/Map maintains the insertion order of traversal of the array while normal HashSet / Map does not.

Self Balancing BST ;

C++	Set , Map
Java	TreeSet , TreeMap

Search, Delete & Insert in O(log n)
It maintains the sorted order and supports many extra functions that are not made by Hash Set / HashMap.
Hence it helps it lowerbound , upper, etc.

* Code : C++

unordered-set ; insert(), find(), erase(), size()

```
unordered_set <int>   us ;
us. insert (15) ;
if ( us. find (15)  !=  us.end())
        cout << "Present" ;
else
        cout << "Not Present" ;
us. size() ;
us. erase (15) ;
```

Code : Java

```
HashSet < Integer >  hs =  new HashSet<Integer>();
hs.add(10);
hs.add(20);
hs.size();
hs.remove(10);
if ( hs.contains(20) == true)
      syso(" Present");
```

* Interal Working of Hashing (15 mins)

It is assumed that the keys are spread in a uniform distribution.

If input size is small ;
bool set [26] = { false ..... false }
set [x - 'a'] = true ;

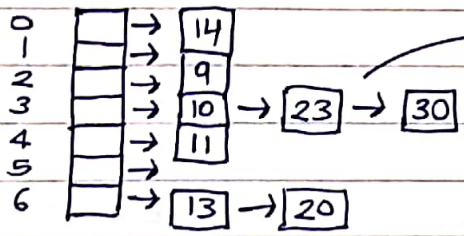
If input size is big ;
Hashing function converts the big keys into small values under modular arithmetic with a prime number.
Eg ; key % p

Birthday Paradox ;
23 people : 50%
70 people : 99.9 %

There are two ways to handle collisions :
(i) Linear Chaining
(ii) Open Addressing

(i) Linear Chaining

hashfunc (key) = key % 7
{ 10, 11, 9, 13, 14, 23, 30, 20 }



Java : Self Balance BST : O(log n)
C++ : Linear List : O(n)

In worst case, it may become a full one chain & hence we assume uniform distribution of keys.

(ii) Open Addressing

We do the same hash func but now the size ~~the~~ elem. of array is = to no. of elements in array & not the prime no.

We insert elements & if space is occupied, we linearly search for next empty space & add it there.

Open Addressing is cache friendly while Linear ~~Addressing~~ is not. But due to linear probing, it becomes sensetive to hashing while ~~LA~~ is not.

* Q/A (22 mins)

Q1) Find the most frequent element in array
arr[] = { 5, 6, 8, 3, 6, 6, 6, 2 }

A) Iterative : $O(n^2)$
Hashing : $O(n)$

Insert all items in HM with frequency
Traverse the HM
Find the maximum frequency

Q2) Given array, find if there is a subarray with 0 sum.
arr[] = { 5, 6, -4, -2, 10 } True
arr[] = { 10, -1, 3, 2 } False

A) Naive Solution W/O Hashing : $O(n^2)$

```
for ( i= 0; i<n ; i++ ) {
    int  sub_sum = 0
    for ( j= i ; j<n ; j++) {
            sub_sum += arr[j]
    }
}
```

10 Jan'21

Hashing Method : O(n)
Traverse the array, find the prefix sum.
If the prefix sum already exists in set, then it means sub array with 0 sum exists.
Otherwise, keep inserting the elements.

(Q3) Same as Q2 but now check for given sum

(A) Check for prefix-sum - x in the hashset and extend the previous answer.

(Q4) Given a binary array, find largest subarray with equal no. of 0 and 1.

(A) This is an extension of Q2. Just take all 0 as -1 and check if already exists.