Date __/__/__

Hashing.

Use to impliment dictionary where we have key value pairs.

we can do all operations search, insert, delete. all in Big O(1) average.

Hashing Not Use For :-

① Finding closest value.
② sorted data
③ Prefix searching.

Application of hashing:

- After Array Hashing is second most used data structure.
- To impliment cache.
- Database indexing.

* - Hashing

Use keys as indexes in Array, & do insert, delete & Search in O(1) because Array can Access Randomly index.

For hashing we have to create Hash function & hash function shoud Achive.

① should always map large key to same small keys.
② shoud generate value from 1 to m-1.
③ shoud be fast, O(1) for integer & O(len) for string.
④ should Uniformly distribute large keys into Hash table slots.

→ At the time of insertion you have to check this element is present in Hash or not duplicates not allowed in Hash table.

Two methods to Avoid colision.

#1 Chaining :- (when Two value comes to same place then it create seporate node for that value & Join to that index).

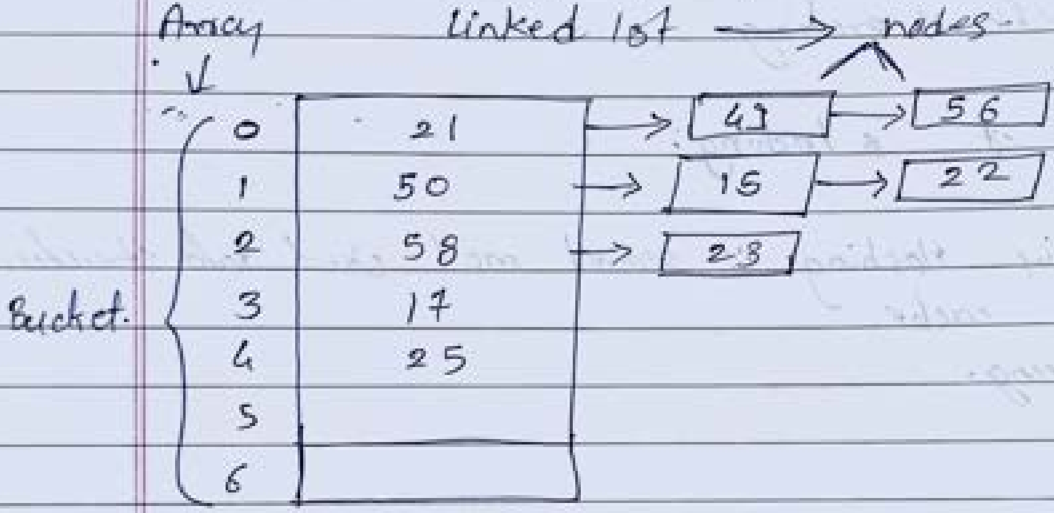
hash (key) = key % 7 (Remainder of 7)

hash function

input Key = 50, 21, 58, 17, 15, 49, 56, 22, 23, 25

key % 7 = 1 0 2 3 1 0 0 1 2 4

Here Divisen is 7 then remainder cannot go above 6.



Hash Table (Array of Linked list Headers).

search (15) → True

search (48) → False.

Implementation of chaining:

```
struct MyHash
{ int Bucket;
  list <int> *table;          // creating Pointer
                                 to array
  MyHash (int b)              // Initializing value
  {                              by constructor of structure.
       Bucket = b;
       table = new list <int>[b];
  }

  void  insert (int key) { - - - - }
  bool  search (int key) { - - - - }
  void  remove (int key) { - - - - }.
};

void insert (int key)
{  int i = key % Bucket;
   table [i]. Pushback (Key);
}

void remove (int key)
{  int i = key % Bucket;
   table [i]. remove (key);
}

bool search (int key)
{

  int i = key % Bucket;
  for ( auto x : table [i] )
        if (x = =key)
              return True;
  return False;
}.
```

2] Open Addressing:-

no. of ~~slot~~ slots in Hash table ≥ No. of keys to be inserted.

$\underline{hash(key) = key \% 7}$

key = 50, 51, 49, 16, 56, 15, 19.

key % 7 = 1, 2, 0, 2, 0, 1, 5

* #① Here 7 keys then we have to create Array of minimum size 7.

* #② It uses linear probing to enter value when colision accures.

Linear probing: Linear search for Next empty slot in Array when there is collision

0	49
1	50
2	51
3	16
4	56
5	15
6	19.

// Here 16 % 7 is 2 but there collision occures then value is stored in next empty slot.

// same for 56. 56 % 7 = 0.

// when yo enter value & last slot is full then again search from first in circular manner.

Date __/__/__

Implementation of open Addresing :-

```
struct Myhash()
{
    int *arr;
    int cap, size;
    Myhash (int c)
    {
        cap = c;
        size =0 ;
        for (int i=0 ; i<cap ; i++)
            arr[i]=-1
    }.
    int hash (int key)
    {
        return key % cap;
    }
    bool search (int key) { - - - }
    bool insert ( int key) { - - - }
    bool erase ( int key). { - - - }.
}.
```

// initialize all value in Array as -1 for empty notation

```
bool search (int key)
{
    int h = hash (key);
    int i =h;
    while ( arr[i] != -1)
    {
        if ( arr[i] = = key)
            return True;
        i = (i+1) % cap;
        if ( i = = h)
            return False;
    }
```

We linearly search in Hash table.
Stop searching.
① when element found in table.
② when empty slot occures having value -1
③ when you traverse through all location

Page No.

Date ___ /___ /_____

```
bool insert (int key)
{
     if (size == cap)
           return False;
     int i = hash (key);
     while (arr[i] != -1 && arr[i] != -2 && arr[i] != key)
           i = (i+1) % cap;
     if (arr[i] == key)
           return False;
     else
     {
           arr[i] = key;
           size ++;
           return True;
     }
};

bool erase (int key)
{    int h = hash (key);
     int i = h;
     while (arr[i] != -1)
     {    if (arr[i] == key)
          {
               arr[i] = -2;
               return True;
          }

          i = (i+1) % cap;
          if (i == h)
               return False;
     }
     return False;
}
```

when it False
for insertion
① when hash table
is full
② & key value is
already present.

// if we find key
then assign -2
value as deleted

// when we not found key
then return False