UNIT-II

Dictionary is a general-purpose data structure for storing a group of objects.
A dictionary has a set of *keys* and each key has a single associated *value*. When presented with a key, the dictionary will return the associated value.

For example, the results of a classroom test could be represented as a dictionary with students names as keys and their scores as the values:

results = {'Detra' : 17**,'Nova' :** 84**,'Charlie' :** 22**, 'Henry' :** 75**, 'Roxanne' :** 92**, 'Elsa' :** 29**}**

Note: The keys in a dictionary must be simple types such as integers or strings while the values can be of any type.

Operations of Dictionary:

1. **Insertion**
2. **Deletion**
3. **Search**

Example:
Consider an empty unordered dictionary and the following set of operations:

Operation	Dictionary	Output
insert(5,A)	{(5,A)}	
insert(7,B)	{(5,A), (7,B)}	
insert(2,C)	{(5,A), (7,B), (2,C)}	
insert(8,D)	{(5,A), (7,B), (2,C), (8,D)}	
insert(2,E)	{(5,A), (7,B), (2,C), (8,D), (2,E)}	
search(7)	{(5,A), (7,B), (2,C), (8,D), (2,E)}	B
search(4)	{(5,A), (7,B), (2,C), (8,D), (2,E)}	NO_SUCH_KEY
search(2)	{(5,A), (7,B), (2,C), (8,D), (2,E)}	C
size()	{(5,A), (7,B), (2,C), (8,D), (2,E)}	5
delete(5)	{(7,B), (2,C), (8,D), (2,E)}	A

Implementation of Dictionary:

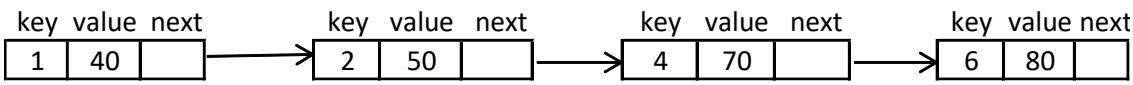
Dictionary is implemented in four ways:

1. **Linear List Representation**
2. **Skip List Representation**
3. **Hashing**
4. **Trees**

Linear List Representation

The dictionary can be represented as a linear list.The linear list is a collection of pairs(key & value).

EX:



Structure of Linked List for representing dictionary

```
struct node
{
        int key;
        int value;
        struct node *next;
}*head=NULL;
void insert();
void delete();
void search();
```

1.Insertion:

Step 1: Create a new node say '**ptr**' with key and value.

Step 2: Check whether dictionary is EMPTY. (head==NULL)

Step 3: If Dictionary is Empty,then set head=ptr and define node pointer **curr** aand initialize it with head. i.e curr=head.

Step 4: If it is not Empty then check the condition ptr->key>curr->key.

Step 5: If it is True then, set curr->next=ptr,curr=ptr.

Step 6: If it is not true then,define two node pointers temp and temp1 and initialize them with head.

Step 7: Keep moving the temp to its next node until the condition is true.

Step 8: Move temp1 to next node until the condition is true.Then set ptr->next=temp1->next ,temp1->next=ptr.

//Program for Insertion//

```
void insert()
{
        int num;
        struct node *ptr,*curr;
         ptr=(struct node*)malloc(sizeof(struct node));
         printf("Enter key:");
         scanf("%d",&key);
        printf("Enter data:");
```

```
        scanf("%d",&num);
        ptr->key=key;
       ptr->data=num;
       ptr->next=NULL;
       if(head==NULL)
       {
            head=ptr;
            curr=head;
       }
       else
       {
             if(ptr->key>curr->key)
             {

             curr->next=ptr;
             curr=ptr;
              }
             else
             {
                   struct node *temp=*temp1=head;
                   while(temp->key<ptr->key)
                   temp=temp->next;
                   for(;temp1->next!=temp;temp1=temp1->next);
                   ptr->next=temp1->next;
                   temp1->next=ptr;
             }
       }
}
```

2.Deletion

Step 1: Initialize **curr** with **head**.i.e,curr=head
Step 2: Enter the Key element to be deleted
Step 3:Compare key element with curr key element.
Step 4:If it matches stop comparison and delete that node
Step 5: Othewise,keep moving the curr to its next node until key found then delete.
Step 6: If Key doesn't match then diplay Node not found.

//Program for Deletion//

```
void delete()
{
```

```
    struct node *prev,*curr=head;
    int key;
    printf("Enter Key element to be deleted");
    scanf("%d",&key);
    while(curr!=NULL)
    {
        if(curr->key==key)
            break;
        prev=curr;
        curr=curr->next;
    }
    if(curr==NULL)
       printf("Dictionary is Empty");
    else
    {
      if(curr==head)
        head=curr->next;
      else
         prev->next=curr->next;
     }
     free(curr);
}
```

3.Search

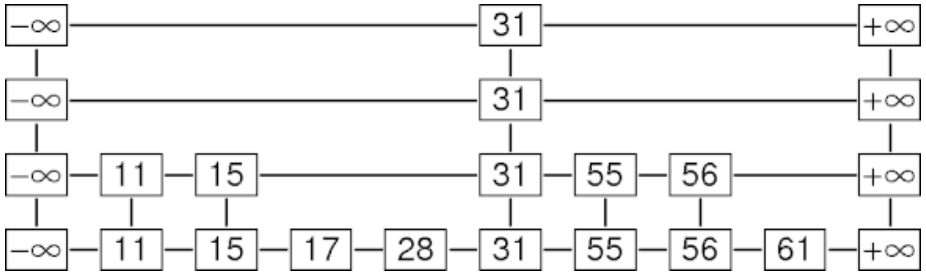
Step 1: Initialize curr with head.i.e,curr=head
Step 2: Enter the Key element to be deleted
Step 3:Compare key element with curr key element.
Step 4:If it matches stop comparison and display the Key is found message.
Step 5: Othewise,keep moving the curr to its next node until key found.
Step 6: If Key doesn't match then diplay key is not found.
Step 7: If curr=NULL then display Dictionary is Empty.

//Program for Search//

```
void search()
{
    struct node *curr=head;
    int key;
    printf("Enter Key element,that we want to search");
    scanf("%d",&key);
    if(curr==NULL)
```

```
        printf("Dictionary is Empty");
        while(curr!=NULL)
        {
                if(curr->key==key)
                {
                        print("Key is found);
                        break;
                 }
                 curr=curr->next;
        }
}
```

Skip List Representation



- Skip list is a randomized data structure.
- It uses coin flips to build the data structure.
- It consists of different layers. Bottom most layer contains all elements.
- As the layer increases, the number of nodes decreases.
- All layers are in the sorted order.
- Choosing elements for the upper layer depends on the probability.

Structure of Skip List

A skip list is built up of layers. The lowest layer (i.e. bottom layer) is an ordinary ordered linked list. The higher layers are like 'express lane' where the nodes are skipped (observe the figure).

Skip List Operations

1. **Search**
2. **Insertion**
3. **Deletion**

1.Searching Process

- When an element is tried to search, the search begins at the head element of the top list.

- It proceeds horizontally until the current element is greater than or equal to the target.
- If current element and target are matched, it means they are equal and search gets finished.
- If the current element is greater than target, the search goes on and reaches to the end of the linked list, the procedure is repeated after returning to the previous element and the search reaches to the next lower list (vertically).

We search for a key x in a skip list as follows:

Step 1: We start at the first position of the top list.

Step 2: At the current position p, we compare x with y<-key(after(p)).

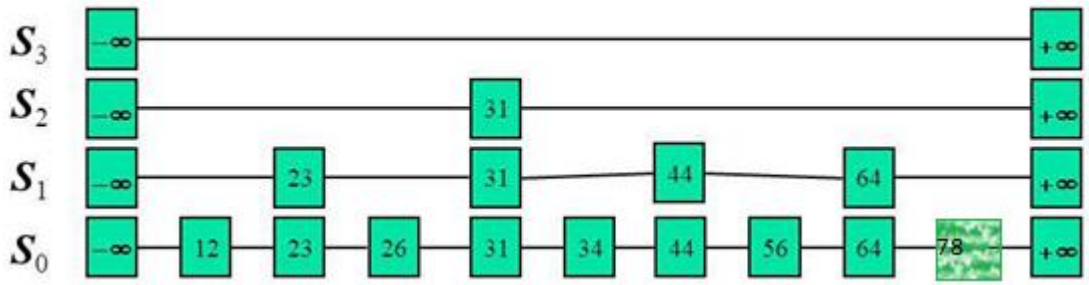
Step 3: x = y: we return element(after(p))

Step 4: x > y: we "scan forward"

Step 5: x < y: we "drop down"

Step 6: If we try to drop down past the bottom list, we return NO_SUCH_KEY

Example: Search for 78 in the below list.



1. Start at the first positon of top list.i.e,S3
2. At the current position p, we compare x with y<-key(after(p)).
3. i) 78<+∞ "drop down"
   ii) 78>31 "scan forward"
   iii) 78<+∞ "drop down"
   iv) 78>44 "scan forward"
   v) 78>64 "scan forward"
   vi) 78<+∞ "drop down"
   vii)78=78 return position(after(p))

2.Insertion

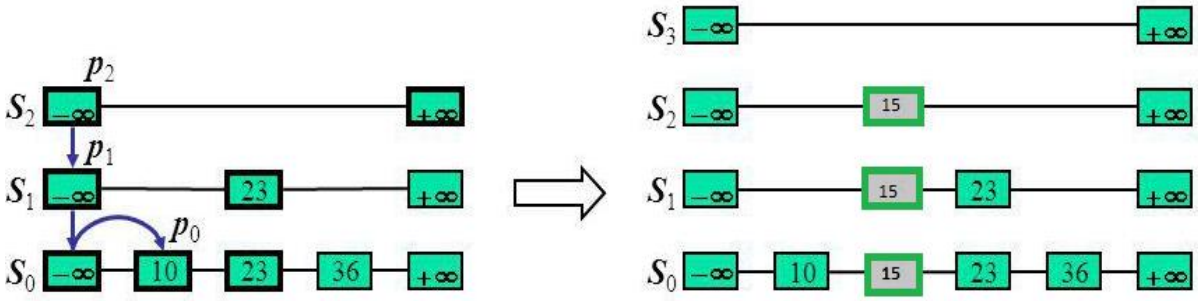
- The insertion algorithm for skip lists uses randomization to decide how many references to the new item (k,e) should be added to the skip list.
- We then insert (k,e) in this bottom-level list immediately after position p.
- After inserting the new item at this level we "flip a coin".

- If the flip comes up tails, then we stop right there.
- If the flip comes up heads, we move to next higher level and insert (k,e) in this level at the appropriate position.

Exapmle:

- Suppose we want to insert 15
- Do a search, and find the spot between 10 and 23

1. 15<+∞ “drop down”
2. 15<23 “drop down”
3. 15>**10** “scan forward”
4. 15<23 “drop down”
5. No level at down.So insert 15 after 10
6. After inserting 15 at this level “flip a coin”.
7. The flip comes up heads three times, we move to next higher level and insert 15 in this level at the appropriate position.
8. If the flip comes up tails, then we stop right there.



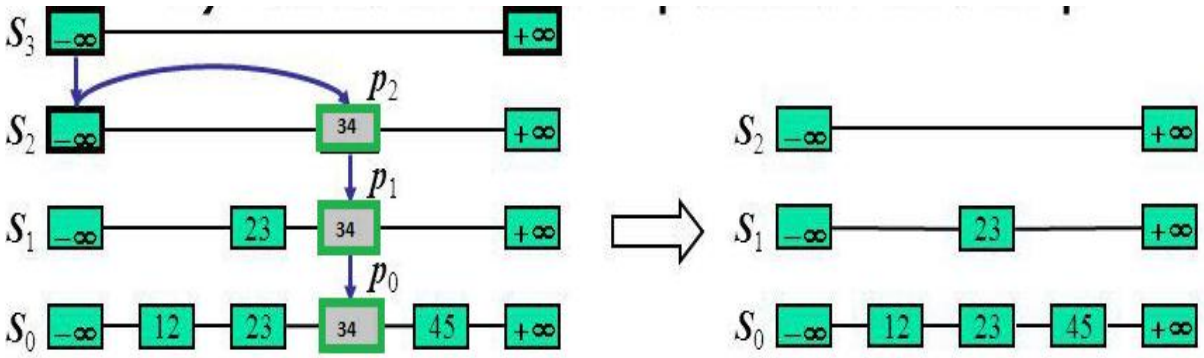
3.Deletion

- We begin by performing a search for the given key k.
- If a position p with key k is not found, then we return the NO SUCH KEY element.
- Otherwise, if a position p with key k is found (it would be found on the bottom level), then we remove all the position above p
- If more than one upper level is empty, remove it.

Example

1) Suppose we want to delete 34
2) Do a search, find the spot between 23 and 45
3) Remove all the position above p

1. 34<+∞ “drop down”
2. 34=34 “return element(after(p)”
3. Remove all the position above 34



Hash Table Representation

- **Hash table is a data structure that represents data in the form of** key-value **pairs.**
- It is a Data structure where the data elements are stored(inserted), searched, deleted based on the keys generated for each element, which is obtained from a hashing function.
- In a hashing system the keys are stored in an array which is called the Hash Table.
- In a hash table, data is stored in an array format, where each data value has its own unique index value.
- Access of data becomes very fast if we know the index of the desired data.

Hashing(Hash Technique)

- Hashing is a technique to convert a range of key values into a range of indexes of an array
- Use modulo operator to get a range of key values.

Hash Function

- The fixed process to convert a key to a hash key is known as a hash function.
- This function will be used whenever access to the table is needed.
- One common method of determining a hash key is the division method of hashing.
- The formula that will be used is:

  Hash(key) = key mod Table size

Example:

Assume a table size is 8.Put the values in the Hash table {36,18,72,43,6)

	Hash Table
0	72
1	
2	18
3	43
4	36
5	
6	6
7	

H(36)=36%8=4
H(18)=18%8=2
H(72)=72%8=0
H(43)=43%8=3
H(6)=6%8=6

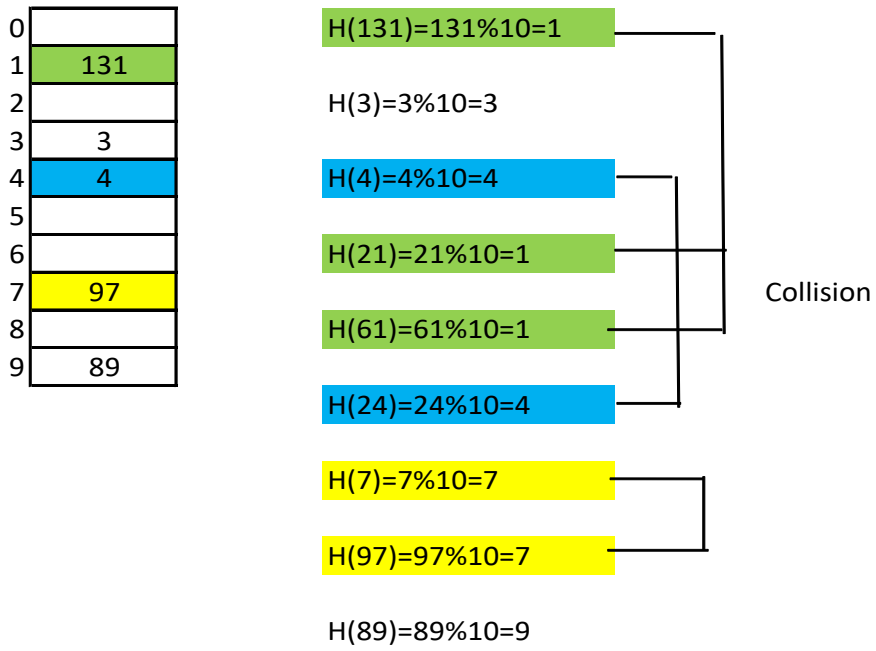
Collision

If the hash function returns same hash key for more than one element then this situation is called **Collision**.

(OR)

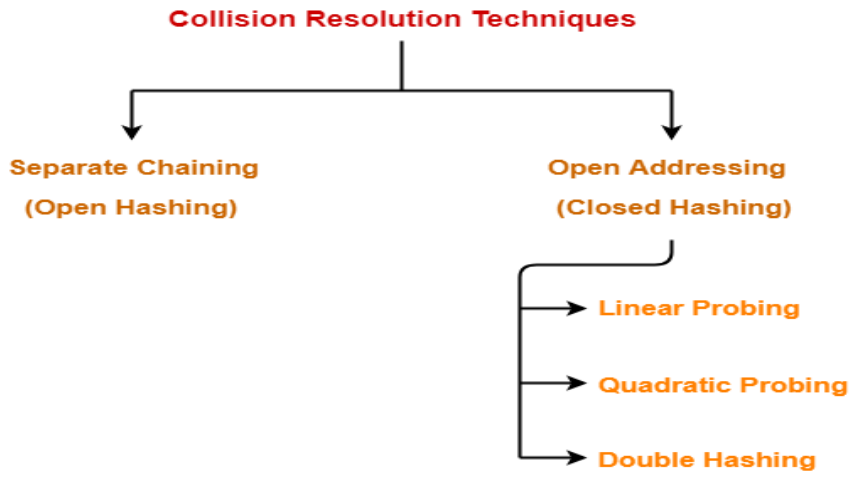
If x1 and x2 are two different keys,but the hash values of x1 and x2 are equal(i.e, h(x1)=h(x2)) then it is called as a **Collision**.

Example: {131,3,4,21,61,24,7,97,89},Table size=10



Collision Resolution Techniques

Collision Resolution Techniques are the techniques used for resolving or handling the collision.
Collision resolution techniques are classified as



Separate Chaining

To handle the collision,

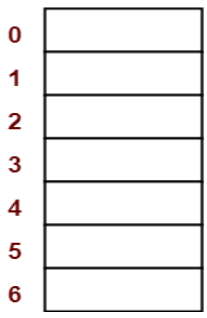
- This technique creates a linked list to the slot for which collision occurs.
- The new key is then inserted in the linked list.
- These linked lists to the slots appear like chains.
- That is why, this technique is called as **separate chaining**.

Problem

Insert the following sequence of keys in the hash table
50, 700, 76, 85, 92, 73 and 101.
Use separate chaining technique for collision resolution. Table size is 7.

Solution

- Draw an empty hash table consisting of 7 buckets.
- The possible range of hash values is [0, 6].
- Insert the given keys in the hash table one by one.
- **Hashing Formula is H(Key)=Key mod Table Size.**

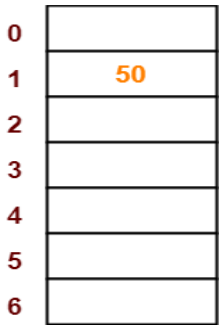


Insert 50:

H(50)=50 mod 7

H(50)=1

So,insert 50 in bucket-1 of the hash table.

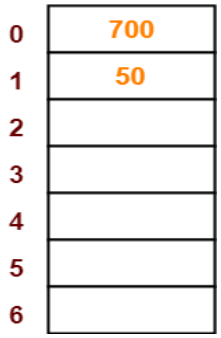


Insert 700:

H(700)=700 mod 7

H(700)=0

So,insert 700 in bucket-0 of the hash table.

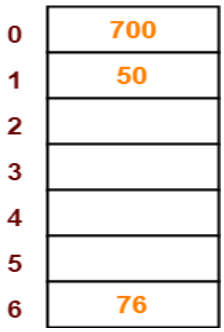


Insert 76:

H(76)=76 mod 7

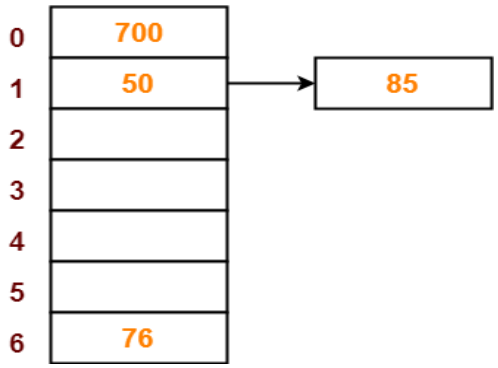
H(76)=6

So,insert 76 in bucket-6 of the hash table.



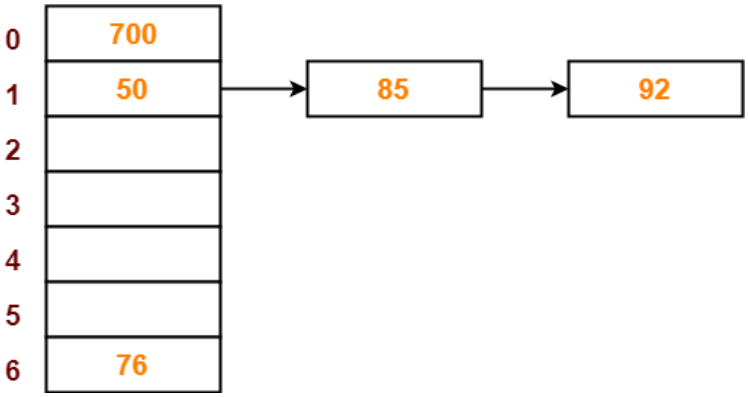
Insert 85:

H(85)=85 mod 7
H(85)=1
Since bucket-1 is already occupied, so collision occurs.
Separate chaining handles the collision by creating a linked list to bucket-1.
So,insert 85 in bucket-1 of the hash table.



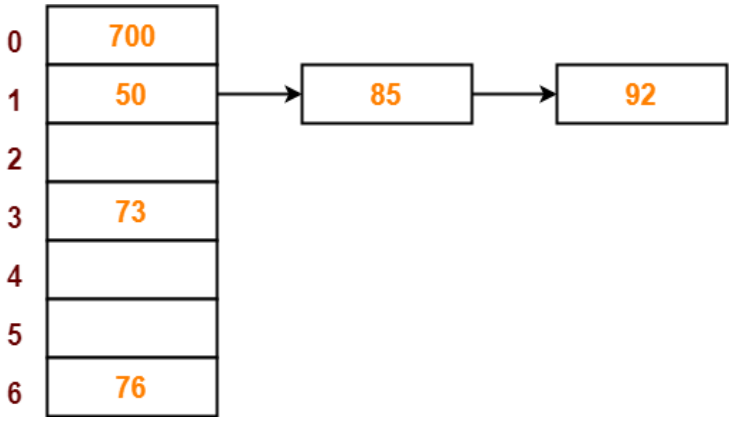
Insert 92:

H(92)=92 mod 7
H(92)=1
Since bucket-1 is already occupied, so collision occurs.
Separate chaining handles the collision by creating a linked list to bucket-1.
So,insert 92 in bucket-1 of the hash table.



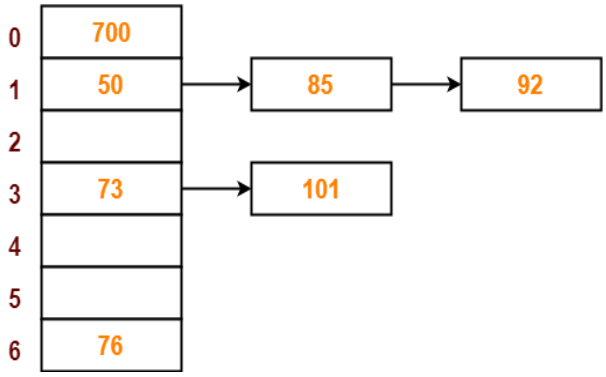
Insert 73:

H(73)=73 mod 7
H(73)=3
Since bucket-3 is already occupied, so collision occurs.
Separate chaining handles the collision by creating a linked list to bucket-3.
So,insert 73 in bucket-3 of the hash table.



Insert 101:

H(101)=101 mod 7
H(101)=3
Since bucket-3 is already occupied, so collision occurs.
Separate chaining handles the collision by creating a linked list to bucket-3.
So,insert 101 in bucket-3 of the hash table.



Open Addressing

In open addressing,

- Unlike separate chaining, all the keys are stored inside the hash table.
- No key is stored outside the hash table.

Techniques used for open addressing are:

1. **Linear Probing**
2. **Quadratic Probing**
3. **Double Hashing**

Linear Probing

In linear probing,

- When collision occurs, we linearly probe for the next bucket.
- We keep probing until an empty bucket is found.

Advantage

- It is easy to compute.

Disadvantage

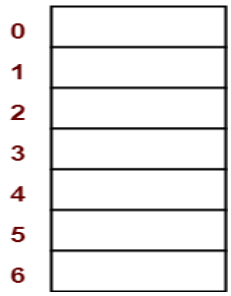
- The main problem with linear probing is clustering.
- Many consecutive elements form groups.
- Then, it takes time to search an element or to find an empty bucket.

Problem

Insert the following sequence of keys in the hash table
50, 700, 76, 85, 92, 73,101.
Use linear probing technique for collision resolution.Table size is 7

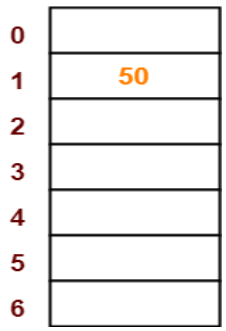
Solution

- Draw an empty hash table consisting of 7 buckets.
- The possible range of hash values is [0, 6].
- Insert the given keys in the hash table one by one.
- **Hashing Formula is H(Key)=Key mod Table Size.**



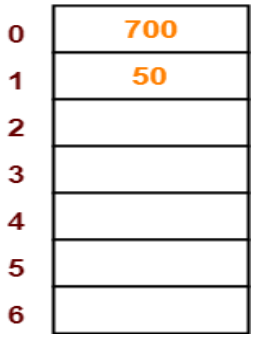
Insert 50:

- H(50)=50 mod 7
- H(50)=1
- So,insert 50 in bucket-1 of the hash table.



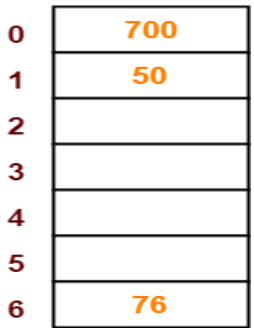
Insert 700:

- H(700)=700 mod 7
- H(700)=0
- So,insert 700 in bucket-0 of the hash table.



Insert 76:

- H(76)=76 mod 7
- H(76)=6
- So,insert 76 in bucket-6 of the hash table.



Insert 85:

- H(85)=85 mod 7
- H(85)=1
- Since bucket-1 is already occupied, so collision occurs.

- To handle the collision, linear probing technique keeps probing linearly until an empty bucket is found.
- The first empty bucket is bucket-2.
- So,insert 85 in bucket-2 of the hash table.

0	700
1	50
2	85
3	
4	
5	
6	76

Insert 92:

- H(92)=92 mod 7
- H(92)=1
- Since bucket-1 is already occupied, so collision occurs.
- To handle the collision, linear probing technique keeps probing linearly until an empty bucket is found.
- The first empty bucket is bucket-3.
- So,insert 92 in bucket-3 of the hash table.

0	700
1	50
2	85
3	92
4	
5	
6	76

Insert 73:

- H(73)=73 mod 7
- H(73)=3
- Since bucket-3 is already occupied, so collision occurs.
- To handle the collision, linear probing technique keeps probing linearly until an empty bucket is found.
- The first empty bucket is bucket-4.
- So,insert 73 in bucket-4 of the hash table.

0	700
1	50
2	85
3	92
4	73
5	
6	76

Insert 101:

- H(101)=101 mod 7
- H(101)=3
- Since bucket-3 is already occupied, so collision occurs.
- To handle the collision, linear probing technique keeps probing linearly until an empty bucket is found.
- The first empty bucket is bucket-5.
- So,insert 101 in bucket-5 of the hash table.

0	700
1	50
2	85
3	92
4	73
5	101
6	76

Quadratic Probing

In quadratic probing,

- When collision occurs, we probe for i^2'th bucket in i^{th} iteration.
- We keep probing until an empty bucket is found.

The formula that will be used is:

$$H_i(X)=((Hash(X)+f(i)) \bmod Tablesize$$

Where $f(i)=i^2$ (Initially i=0 when collision occurs i value is increased by one).
hash(X)=X mod Tablesize

Problem:

1. Insert the following list of keys in the hash table by using Quadratic Probing Technique for Collision Resolution.

23 13 12 22 18 28 32 53 6. Hash Table size is 11

Solution:

Insert 23

- $H_i(X)=((Hash(X)+f(i))$ mod Tablesize
- $H_0(23)=((Hash(23)+f(0))$ mod 11
- Hash(23)=23 mod 11
- Hash(23)=1
- $H_0(23)=((1+0))$ mod 11
- $H_0(23)=1$ mod 11
- $H_0(23)=1$

0	
1	**23**
2	
3	
4	
5	
6	
7	
8	
9	
10	

Insert 13

- $H_i(X)=((Hash(X)+f(i))$ mod Tablesize
- $H_0(13)=((Hash(13)+f(0))$ mod 11
- Hash(13)=13 mod 11
- Hash(13)=2
- $H_0(13)=((2+0))$ mod 11
- $H_0(13)=2$ mod 11
- $H_0(13)=2$

0	
1	**23**
2	**13**
3	
4	

5	
6	
7	
8	
9	
10	

Insert 12

- $H_i(X)=((Hash(X)+f(i))$ mod Tablesize
- $H_0(12)=((Hash(12)+f(0))$ mod 11
- Hash(12)=12 mod 11
- Hash(12)=1
- $H_0(12)=((1+0))$ mod 11
- $H_0(12)=1$ mod 11
- $H_0(12)=1$
- Collision is Occurred at Bucket-1.
- $H_1(12)=((Hash(12)+f(1))$ mod 11
- $H_1(12)=(1+1^2)$ mod 11
- $H_1(12)=2$ mod 11
- $H_1(12)=2$
- Collision is Occurred at Bucket-2
- $H_2(12)=((Hash(12)+f(2))$ mod 11
- $H_2(12)=(1+2^2)$ mod 11
- $H_2(12)=5$ mod 11
- $H_2(12)=5$

0	
1	**23**
2	**13**
3	
4	
5	**12**
6	
7	
8	
9	
10	

Insert 22

- $H_i(X)=((Hash(X)+f(i))$ mod Tablesize
- $H_0(22)=((Hash(22)+f(0))$ mod 11
- Hash(22)=22 mod 11

- Hash(22)=0
- H_0(22)=((0+0)) mod 11
- H_0(22)=0 mod 11
- H_0(22)=0

0	**22**
1	**23**
2	**13**
3	
4	
5	**12**
6	
7	
8	
9	
10	

Insert 18

- H_i(X)=((Hash(X)+f(i)) mod Tablesize
- H_0(18)=((Hash(18)+f(0)) mod 11
- Hash(18)= 18 mod 11
- Hash(18)=7
- H_0(18)=((7+0)) mod 11
- H_0(18)=7 mod 11
- H_0(18)=7

0	**22**
1	**23**
2	**13**
3	
4	
5	**12**
6	
7	**18**
8	
9	
10	

Insert 28

- $H_i(X)=((Hash(X)+f(i))$ mod Tablesize
- $H_0(28)=((Hash(28)+f(0))$ mod 11
- Hash(28)= 28 mod 11
- Hash(28)=6
- $H_0(28)=((6+0))$ mod 11
- $H_0(28)=6$ mod 11
- $H_0(28)=6$

0	**22**
1	**23**
2	**13**
3	
4	
5	**12**
6	**28**
7	**18**
8	
9	
10	

Double Hashing

Double hashing is a collision resolving technique in Open Addressed Hash tables. Double hashing uses the idea of applying a second hash function to key when a collision occurs.

The formula that will be used is:

$H_i(X)=((Hash(X)+f(i))$ mod Tablesize

Hash(X)=X mod Table size

$f(i)=i*Hash_2(X)$

$Hash_2(X)=R-(X \bmod R)$

Where,R is a last prime number smaller than Tablesize.

Problem

Insert the following list of keys in the hash table by using Double Hashing Technique for Collision Resolution.Table size is 10.

89,18,49,58,69,60

Solution:

Insert 89

- $H_i(X)=((Hash(X)+f(i))$ mod Tablesize

- $H_0(89)=((Hash(89)+f(0))$ mod 10
- Hash(89)= 89 mod 10
- Hash(89)=9
- $H_0(89)=((9+0))$ mod 10
- $H_0(89)=9$ mod 10
- $H_0(89)=9$

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	**89**

Insert 18

- $H_i(X)=((Hash(X)+f(i))$ mod Tablesize
- $H_0(18)=((Hash(18)+f(0))$ mod 10
- Hash(18)= 18 mod 10
- Hash(18)=8
- $H_0(18)=((8+0))$ mod 10
- $H_0(18)=8$ mod 10
- $H_0(18)=8$

0	
1	
2	
3	
4	
5	
6	
7	
8	**18**
9	**89**

Insert 49

- $H_i(X)=((Hash(X)+f(i))$ mod Tablesize

- $H_0(49)=((Hash(49)+f(0))$ mod 10
- Hash(49)= 49 mod 10
- Hash(49)=9
- $H_0(49)=((9+0))$ mod 10
- $H_0(49)=9$ mod 10
- $H_0(49)=9$
- Collision is Occurred at Bucket-9.
- $H_1(49)=((Hash(49)+f(1))$ mod 10
- $H_1(49)=((9+(1*Hash_2(49))$ mod 10
- $Hash_2(49)=7-(49$ mod 7)
- $Hash_2(49)=7-0$
- $Hash_2(49)=7$
- $H_1(49)=((9+7)$ mod 10
- $H_1(49)=16$ mod 10
- $H_1(49)=6$
-

0	
1	
2	
3	
4	
5	
6	**49**
7	
8	**18**
9	**89**

Insert 58

- $H_i(X)=((Hash(X)+f(i))$ mod Tablesize
- $H_0(58)=((Hash(58)+f(0))$ mod 10
- Hash(58)= 58 mod 10
- Hash(58)=8
- $H_0(58)=((8+0))$ mod 10
- $H_0(58)=8$ mod 10
- $H_0(58)=8$
- Collision is Occurred at Bucket-8.
- $H_1(58)=((Hash(58)+f(1))$ mod 10
- $H_1(58)=((8+(1*Hash_2(58))$ mod 10
- $Hash_2(58)=7-(58$ mod 7)

- $Hash_2(58)=7-2$
- $Hash_2(58)=5$
- $H_1(58)=((8+5) \bmod 10$
- $H_1(58)=13 \bmod 10$
- $H_1(58)=3$
-

0	
1	
2	
3	**58**
4	
5	
6	**49**
7	
8	**18**
9	**89**

Insert 69

- $H_i(X)=((Hash(X)+f(i)) \bmod$ Tablesize
- $H_0(69)=((Hash(69)+f(0)) \bmod 10$
- $Hash(69)= 69 \bmod 10$
- $Hash(69)=9$
- $H_0(69)=((9+0)) \bmod 10$
- $H_0(69)=9 \bmod 10$
- $H_0(69)=9$
- Collision is Occurred at Bucket-9.
- $H_1(69)=((Hash(69)+f(1)) \bmod 10$
- $H_1(69)=((9+(1*Hash_2(69)) \bmod 10$
- $Hash_2(69)=7-(69 \bmod 7)$
- $Hash_2(69)=7-6$
- $Hash_2(69)=1$
- $H_1(69)=((9+1) \bmod 10$
- $H_1(69)=10 \bmod 10$
- $H_1(69)=0$

0	**69**
1	
2	
3	**58**
4	

5	
6	**49**
7	
8	**18**
9	**89**

Insert 60

- $H_i(X)=((Hash(X)+f(i))$ mod Tablesize
- $H_0(60)=((Hash(60)+f(0))$ mod 10
- Hash(60)= 60 mod 10
- Hash(60)=0
- $H_0(60)=((0+0))$ mod 10
- $H_0(60)=0$ mod 10
- $H_0(60)=0$
- Collision is Occurred at Bucket-0.
- $H_1(60)=((Hash(60)+f(1))$ mod 10
- $H_1(60)=((0+(1*Hash_2(60))$ mod 10
- $Hash_2(60)=7-(60$ mod 7)
- $Hash_2(60)=7-4$
- $Hash_2(60)=3$
- $H_1(60)=((0+3)$ mod 10
- $H_1(60)=3$ mod 10
- $H_1(60)=3$
- Collision is Occurred at Bucket-3.
- $H_2(60)=((Hash(60)+f(2))$ mod 10
- $H_2(60)=((0+(2*Hash_2(60))$ mod 10
- $H_2(60)=((0+(2*3))$ mod 10
- $H_2(60)=((0+6)$ mod 10
- $H_2(60)=6$ mod 10
- $H_2(60)=6$
- Collision is Occurred at Bucket-6.
- $H_3(60)=((0+(3*Hash_2(60))$ mod 10
- $H_3(60)=((0+(3*3))$ mod 10
- $H_3(60)=9$ mod 10
- $H_3(60)=9$
- Collision is Occurred at Bucket-9.
- $H_4(60)=((0+(4*Hash_2(60))$ mod 10
- $H_4(60)=((0+(4*3))$ mod 10
- $H_4(60)=12$ mod 10
- $H_4(60)=2$

0	60
1	
2	60
3	58
4	
5	
6	49
7	
8	18
9	89

Load Factor (α)- Load factor (α) is defined as-

Load Factor (α) = M / N

Where,

M = Number of elements present in the hash table

N = Total size of the hash table.

The default load factor of Hash Map is **0.75f** (75% of the map size).

When the load factor ratio (m/n) reaches 0.75 at that time, hash map increases its capacity.

How Load Factor is calculated :

Now check that we need to increase the hashmap capacity or not Number of elements present in the hash table (M) = 9

Total size of the hash table (N) = 11

Load Factor (α) = M / N

= 9 / 11 = 0.81

Now compare this value with the default factor

0.81 > 0.75

Now we need to increase the hashmap size.

In open addressing, the value of load factor always lie between 0 and 1. This is because-

- In open addressing, all the keys are stored inside the hash table.
- So, size of the table is always greater or at least equal to the number of keys stored in the table.

REHASHING

Rehashing is a technique in which the table is resized, i.e., the size of table is doubled by creating a new table. It is preferable is the total size of table is a prime number. There are situations in which the rehashing is required.

- When table is completely full

- With quadratic probing when the table is filled half.
- When insertions fail due to overflow.

In such situations, we have to transfer entries from old table to the new table by re computing their positions using hash functions.

Example:
Consider we have to insert the elements 37, 90, 55, 22, 16, 49, 33 and 88.

Table size is 10

Solution:

h(X) = X mod Table size

h(37) = 37 mod 10 = 7
h(90) = 90 mod 10 = 0
h(55) = 55 mod 10 = 5
h(22) = 22 mod 10 = 2
h(16) = 16 mod 10 = 6
h(49) = 49 mod 10 = 9
h(33) = 33 mod 10 = 3
h(88) = 88 mod 10 = 8

Hash Table

Buckets	Key
0	90
1	
2	22
3	33
4	
5	55
6	16
7	37
8	88
9	49

Now this table is almost full and if we try to insert more elements collisions will occur and eventually further insertions will fail. Hence we will rehash by doubling the table size. The old table size is 10 then we should double this size for new table that becomes 20. But 20 is not a prime number, we will prefer to make the table size as 23.

New Hash Table

h(X) = X mod Table size

h(37) = 37 mod 23 = 14

h(90) = 90 mod 23 = 21

h(55) = 55 mod 23 = 9

h(22) = 22 mod 23 = 22

h(16) = 16 mod 23 = 16

h(49) = 49 mod 23 = 3

h(33) = 33 mod 23= 10

h(88) = 88 mod 23 = 19

Buckets	Key
0	
1	
2	
3	49
4	
5	
6	
7	
8	
9	55
10	33
11	
12	
13	
14	37
15	
16	16
17	
18	
19	88
20	
21	90
22	22

Now the hash table is sufficiently large to accommodate new insertions.

Advantages:

1. This technique provides the programmer a flexibility to enlarge the table size if required.
2. Only the space gets doubled with simple hash function which avoids occurrence of collisions.

EXTENDABLE HASHING

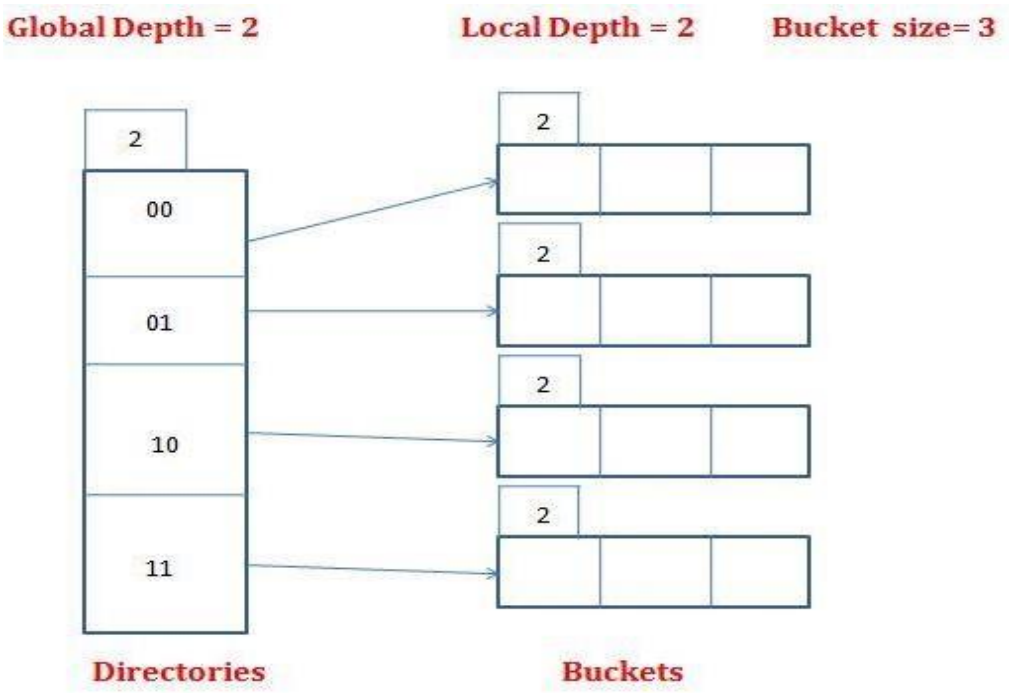
Extendible Hashing is a dynamic hashing method wherein directories, and buckets are used to hash data. It is an aggressively flexible method in which the hash function also experiences dynamic changes.

Main features of Extendible Hashing: The main features in this hashing technique are:

- **Directories:** The directories store addresses of the buckets in pointers. An id is assigned to each directory which may change each time when Directory Expansion takes place.
- **Buckets:** The buckets are used to hash the actual data.

Number of directory entries = 2^{GD}

Basic Structure of Extendable Hashing



Procedure of Extendable Hashing

Step 1 – Analyze Data Elements: Data elements may exist in various forms eg. Integer, String, Float, etc.. Currently, let us consider data elements of type integer. eg: 49.

Step 2 – Convert into binary format: Convert the data element in Binary form. For string elements, consider the ASCII equivalent integer of the starting character and then convert the integer into binary form. Since we have 49 as our data element, its binary form is 110001.

Step 3 – Check Global Depth of the directory. Suppose the global depth of the Hash-directory is 3.

Step 4 – Identify the Directory: Consider the 'Global-Depth' number of LSBs in the binary number and match it to the directory id.
Eg. The binary obtained is: 110001 and the global-depth is 3. So, the hash function will return 3 LSBs of 110**001** viz. 001.

Step 5 – Navigation: Now, navigate to the bucket pointed by the directory with directory-id 001.

Step 6 – Insertion and Overflow Check: Insert the element and check if the bucket overflows. If an overflow is encountered, go to **step 7** followed by **Step 8**, otherwise, go to **step 9**.

Step 7 – Tackling Over Flow Condition during Data Insertion: Many times, while inserting data in the buckets, it might happen that the Bucket overflows. In such cases, we need to follow an appropriate procedure to avoid mishandling of data.

First, Check if the local depth is less than or equal to the global depth. Then choose one of the cases below.

- **Case 1:** If the local depth of the overflowing Bucket is equal to the global depth, then Directory Expansion, as well as Bucket Split, needs to be performed. Then increment the global depth and the local depth value by 1. And, assign appropriate pointers.

  Directory expansion will double the number of directories present in the hash structure.

- **Case 2:** In case the local depth is less than the global depth, then only Bucket Split takes place. Then increment only the local depth value by 1. And, assign appropriate pointers.

Step 8 – Rehashing of Split Bucket Elements: The Elements present in the overflowing bucket that is split are rehashed w.r.t the new global depth of the directory.

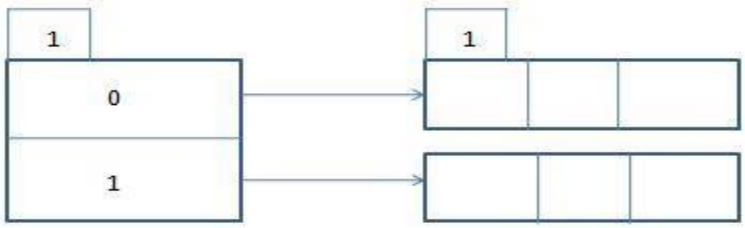
Step 9 – The element is successfully hashed.

Example : Hashing the following elements: 16,4,6,22,24,10,31,7,9,20,26.

Solution: First, calculate the binary forms of each of the given Keys.

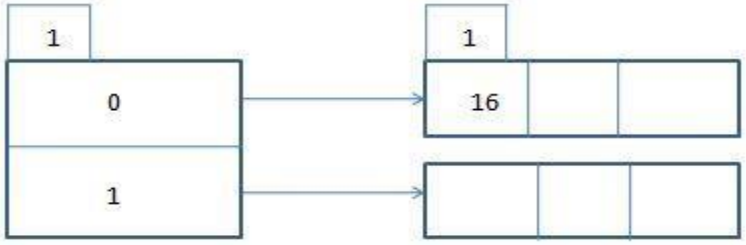
Keys	Binary Form
16	10000
4	00100
6	00110
22	10110
24	11000
10	01010
31	11111
7	00111
9	01001
20	10100
26	11010

Step1 : Initially, the global-depth and local-depth is always 1. Thus, the hashing frame is



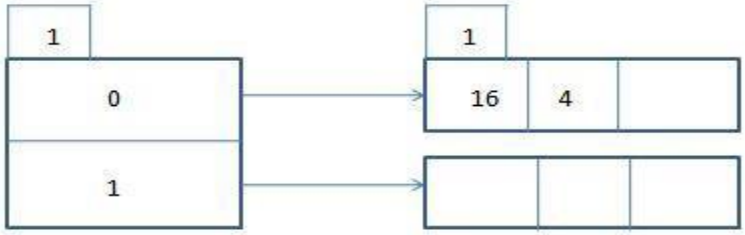
Step2 : Inserting 16 : The binary format of 16 is 10000 and Global depth is 1.

It is LSB of 1000**0** which is 0. Hence, 16 is mapped to the directory with id=0.



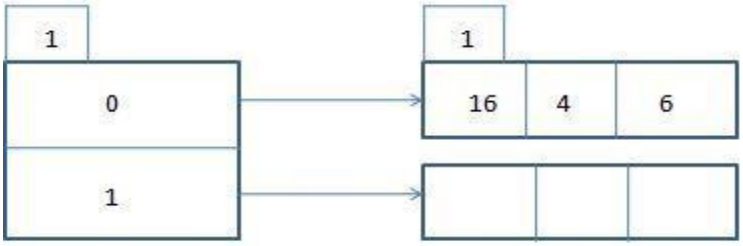
Step3: Inserting 4 : The binary format of 4 is 00100 and Global depth is 1.

It is LSB of 00100 which is 0. Hence, 4 is mapped to the directory with id=0.



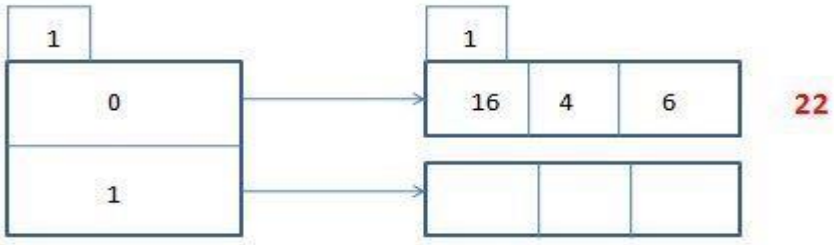
Step 4: Inserting 6 : The binary format of 6 is 00110 and Global depth is 1.

It is LSB of 00110 which is 0. Hence, 6 is mapped to the directory with id=0.



Step 5: Inserting 22 : The binary format of 22 is 10110 and Global depth is 1.

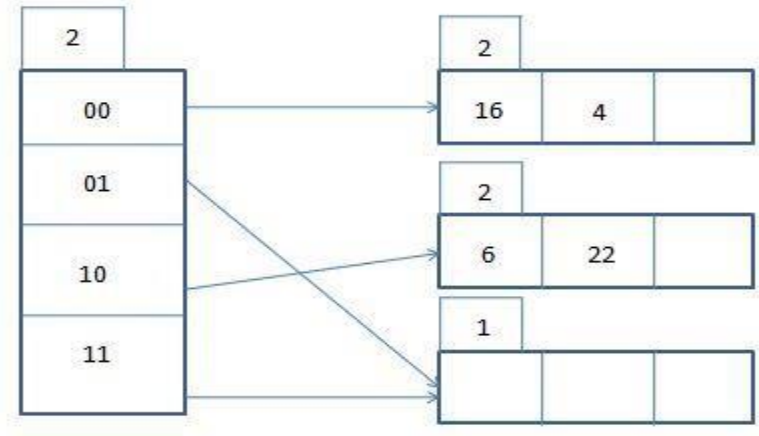
It is LSB of 10110 which is 0. Hence, 22 is mapped to the directory with id=0.



The bucket pointed by directory 0 is Already full. Hence , overflow occurs.

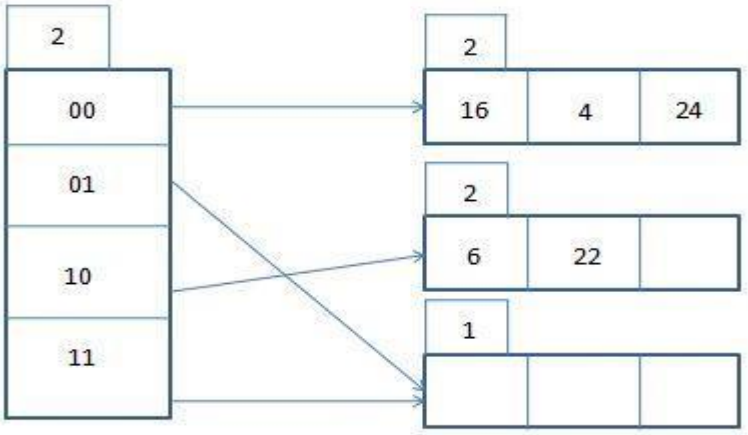
Since Local Depth = Global Depth, the bucket splits and directory expansion takes place. Also, rehashing of numbers present in the overflowing bucket takes place after the split. And, since the global depth is incremented by 1, now,the global depth is 2. Hence, 16,4,6,22 are now rehashed w.r.t 2 LSBs.[16(10000),4(100),6(110),22(10110)]

After Bucket split and Directory Expansion



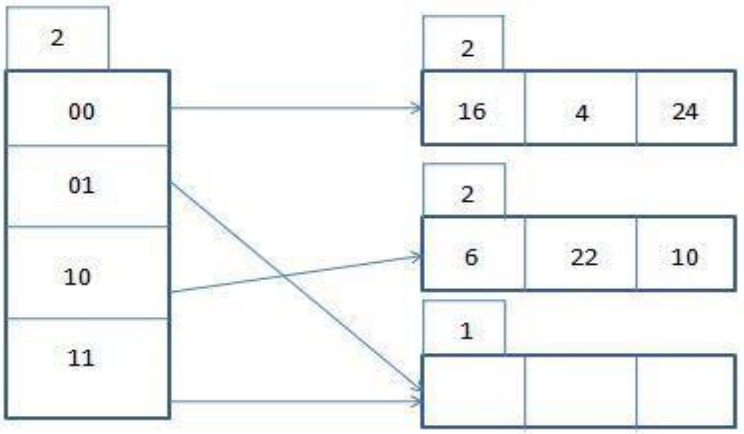
Step 6: Inserting 24 : The binary format of 24 is 11000 and Global depth is 2.

It is LSB of 11000 which is 00. Hence, 24 is mapped to the directory with id=00.



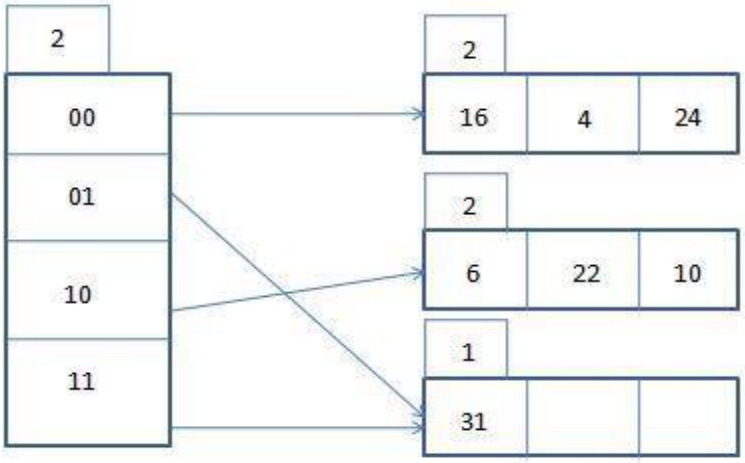
Step 7: Inserting 10 : The binary format of 10 is 01010 and Global depth is 2.

It is LSB of 01010 which is 10. Hence, 10 is mapped to the directory with id=10.



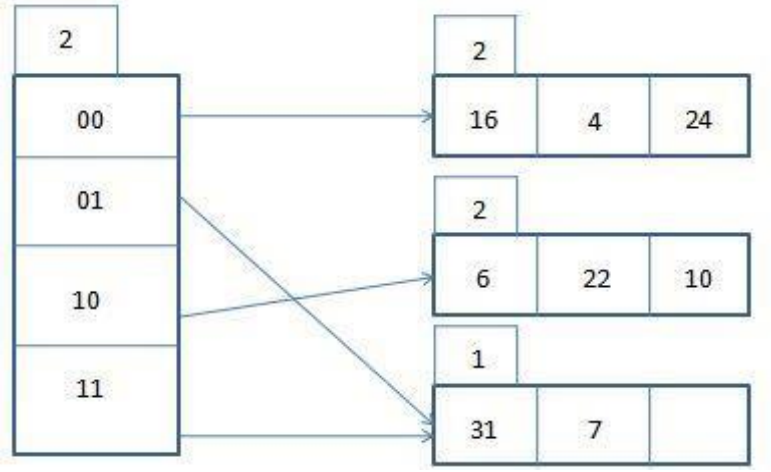
Step 8: Inserting 31 : The binary format of 31 is 11111 and Global depth is 2.

It is LSB of 111**11** which is 11. Hence, 31 is mapped to the directory with id=11.



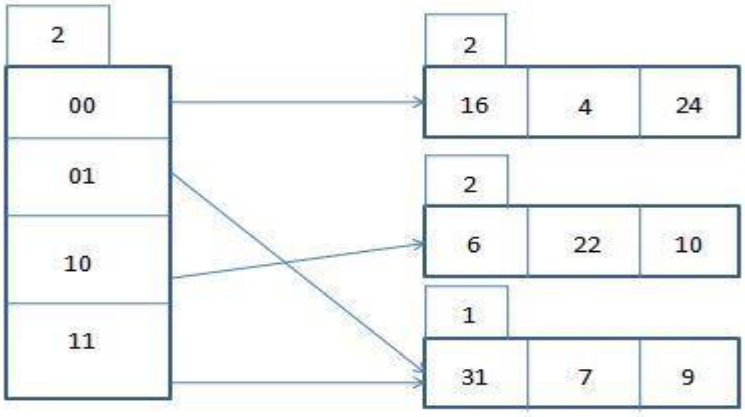
Step 9: Inserting 7 : The binary format of 7 is 00111 and Global depth is 2.

It is LSB of 001**11** which is 11. Hence, 7 is mapped to the directory with id=11.



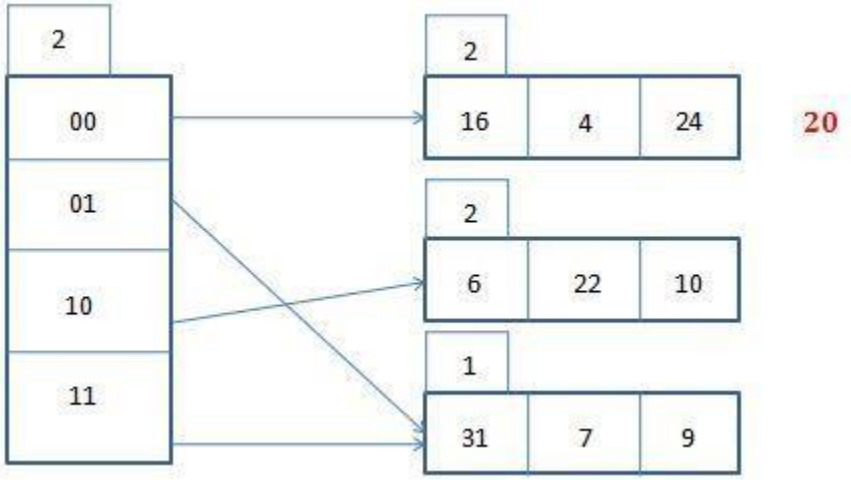
Step 10: Inserting 9 : The binary format of 9 is 01001 and Global depth is 2.

It is LSB of 010**01** which is 01. Hence, 9 is mapped to the directory with id=01.



Step 11: Inserting 20 : The binary format of 20 is 10100 and Global depth is 2.

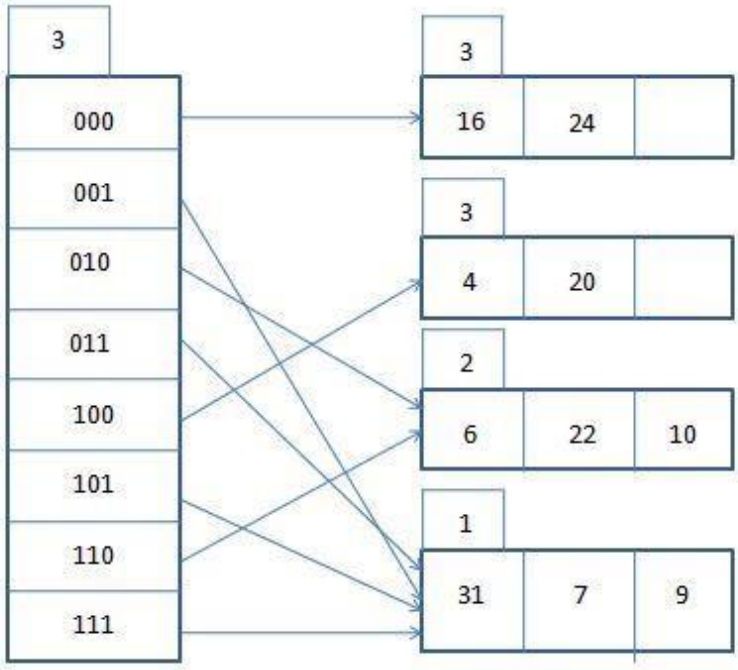
It is LSB of 101**00** which is 00. Hence, 20 is mapped to the directory with id=00.



The bucket pointed by directory 00 is Already full. Hence , overflow occurs.

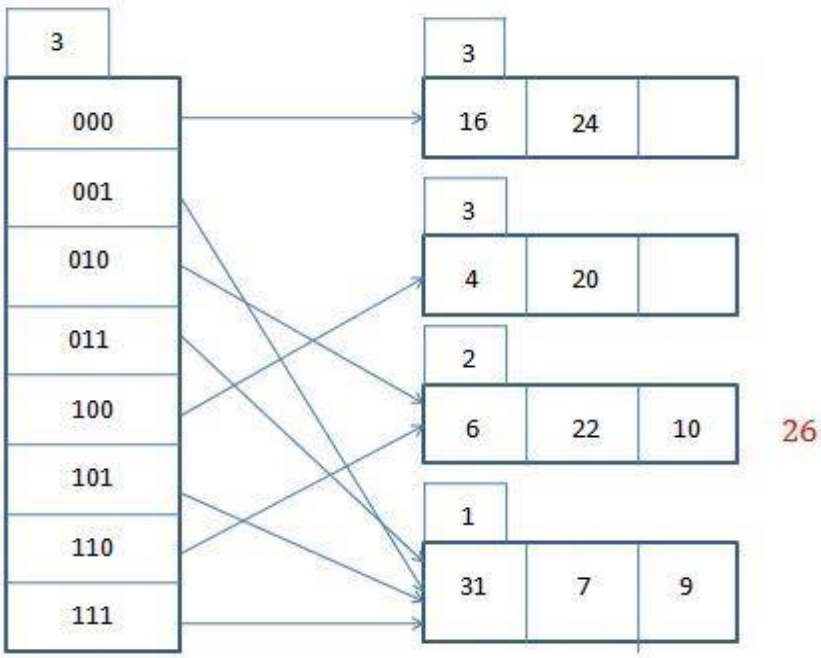
Since Local Depth = Global Depth, the bucket splits and directory expansion takes place. Also, rehashing of numbers present in the overflowing bucket takes place after the split. And, since the global depth is incremented by 1, now, the global depth is 3. Hence, 16,4,24,20 are now rehashed w.r.t 3 LSBs.[16(10**000**),4(00**100**),24(11**000**),20(10**100**)]

After Bucket split and Directory Expansion



Step 12: Inserting 26 : The binary format of 26 is 11010 and Global depth is 3.

It is LSB of 11010 which is 010. Hence, 26 is mapped to the directory with id=010.



The bucket pointed by directory 010 is Already full. Hence , overflow occurs.

since the **local depth of bucket < Global depth (2<3)**, directories are not doubled but, only the bucket is split and Hence, 6,22,10,26 are now rehashed w.r.t 3 LSBs.[6(00**110**), 22(10**110**), 10(01**010**), 26(11**010**)].

After Bucket split