UNIT – II

DICTIONARY

- 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.

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.

DICTIONARY OPERATIONS

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, which is a collection of key and values. There are different ways of Linear List Representation of the Dictionaries.

1. Ordered List
2. Sparse Matrix Representation.

ORDERED LIST

- An ordered list is a list in which the elements must always be ordered in a particular way.
- The most common example of ordered lists, namely *sorted* lists, in which the elements are ordered according to their values.
- The ordering is typically either ascending or descending and we assume that list items have a meaningful comparison operation that is already defined.

Ordered List Operations

- **orderedList() :** Creates a new ordered list that is empty. It needs no parameters and returns an empty list.
- **add(item) :** Adds a new item to the list making sure that the order is preserved. It needs the item and returns nothing. Assume the item is not already in the list.
- **remove(item) :** Removes the item from the list. It needs the item and modifies the list. Assume the item is present in the list.
- **search(item) :** Searches for the item in the list. It needs the item and returns a boolean value.
- **isEmpty() :** Tests to see whether the list is empty. It needs no parameters and returns a boolean value.

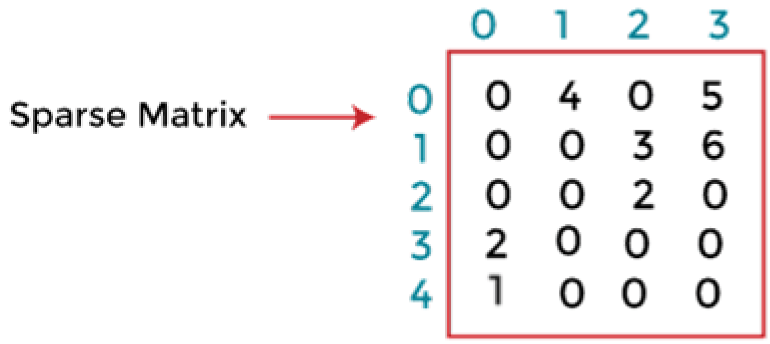
- **size() :** Returns the number of items in the list. It needs no parameters and returns an integer.
- **index(item) :** Returns the position of item in the list. It needs the item and returns the index. Assume the item is in the list.
- **pop() :** Removes and returns the last item in the list. It needs nothing and returns an item. Assume the list has at least one item.
- **pop(pos) :** Removes and returns the item at position pos. It needs the position and returns the item. Assume the item is in the list.

SPARSE MATRIX REPRESENTATION

What is a sparse matrix?

- Sparse matrix is a matrix which contains very few non-zero elements.
- In other words, the sparse matrix can be defined as the matrix that has a greater number of zero elements than the non-zero elements.

Example



$$\begin{array}{c|cccc} & 0 & 1 & 2 & 3 \\ \hline 0 & 0 & 4 & 0 & 5 \\ 1 & 0 & 0 & 3 & 6 \\ 2 & 0 & 0 & 2 & 0 \\ 3 & 2 & 0 & 0 & 0 \\ 4 & 1 & 0 & 0 & 0 \end{array}$$

Why is a sparse matrix required if we can use the simple matrix to store elements?

There are the following benefits of using the sparse matrix -

Storage

- We know that a sparse matrix contains lesser non-zero elements than zero, so less memory can be used to store elements.
- It evaluates only the non-zero elements.

Computing time

- In the case of searching in sparse matrix, we need to traverse only the non-zero elements rather than traversing all the sparse matrix elements.
- It saves computing time by logically designing a data structure traversing non-zero elements.

REPRESENTATION OF SPARSE MATRIX

The non-zero elements in the sparse matrix can be stored using triplets that are rows, columns, and values. There are two ways to represent the sparse matrix that are listed as follows -

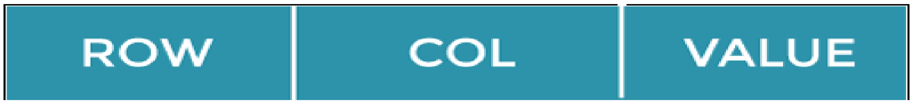
1. Array representation

2. Linked list representation

ARRAY REPRESENTATION OF THE SPARSE MATRIX

- Representing a sparse matrix by a 2D array leads to the wastage of lots of memory.
- This is because zeroes in the matrix are of no use, so storing zeroes with non-zero elements is wastage of memory.
- To avoid such wastage, we can store only non-zero elements. If we store only non-zero elements, it reduces the traversal time and the storage space.

In 2D array representation of sparse matrix, there are three fields used that are named as –



ROW	COL	VALUE

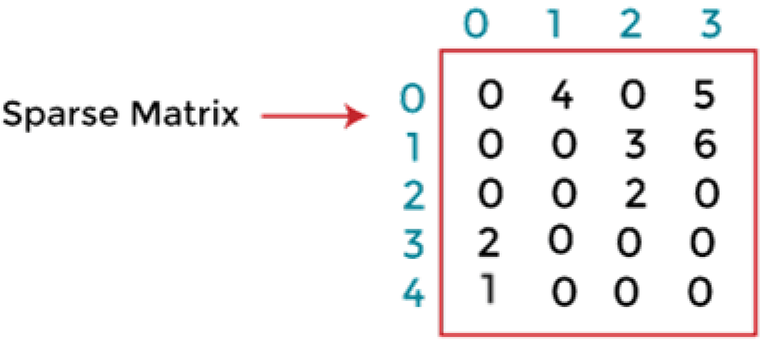
Row - It is the index of a row where a non-zero element is located in the matrix.

Column - It is the index of the column where a non-zero element is located in the matrix.

Value - It is the value of the non-zero element that is located at the index (row, column).

Example

Let's understand the array representation of sparse matrix with the help of the example given below Consider the sparse matrix –



$$\begin{array}{c|cccc} & 0 & 1 & 2 & 3 \\ \hline 0 & 0 & 4 & 0 & 5 \\ 1 & 0 & 0 & 3 & 6 \\ 2 & 0 & 0 & 2 & 0 \\ 3 & 2 & 0 & 0 & 0 \\ 4 & 1 & 0 & 0 & 0 \end{array}$$

- In the above figure, we can observe a 5x4 sparse matrix containing 7 non-zero elements and 13 zero elements.
- The above matrix occupies 5x4 = 20 memory space. Increasing the size of matrix will increase the wastage space.

The tabular representation of the above matrix is given below –

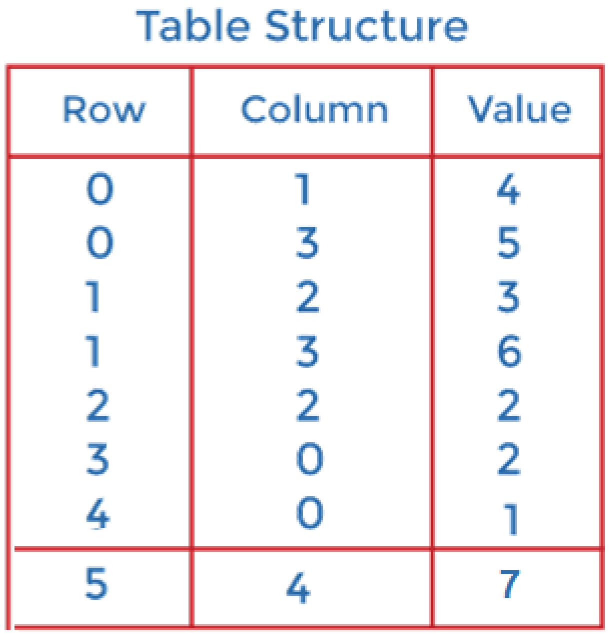


Table Structure

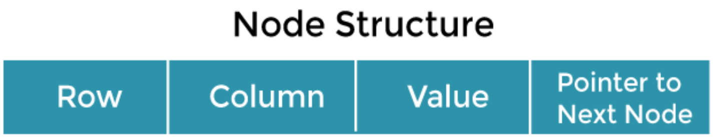
Row	Column	Value
0	1	4
0	3	5
1	2	3
1	3	6
2	2	2
3	0	2
4	0	1
5	4	7

- In the above structure, first column represents the rows, the second column represents the columns, and the third column represents the non-zero value.
- The first row of the table represents the triplets.
- The first triplet represents that the value 4 is stored at 0th row and 1st column.
- Similarly, the second triplet represents that the value 5 is stored at the 0th row and 3rd column.
- In a similar manner, all triplets represent the stored location of the non-zero elements in the matrix.
- The size of the table depends upon the total number of non-zero elements in the given sparse matrix.
- Above table occupies 8x3 = 24 memory space which is more than the space occupied by the sparse matrix.
- So, what's the benefit of using the sparse matrix?
- Consider the case if the matrix is 8*8 and there are only 8 non-zero elements in the matrix, then the space occupied by the sparse matrix would be 8*8 = 64, whereas the space occupied by the table represented using triplets would be 8*3 = 24.

LINKED LIST REPRESENTATION OF THE SPARSE MATRIX

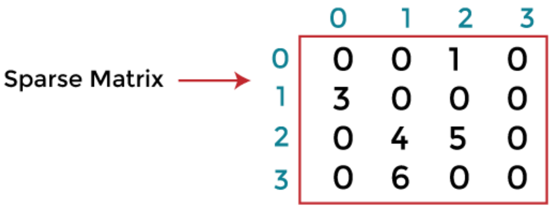
- In a linked list representation, the linked list data structure is used to represent the sparse matrix.
- The advantage of using a linked list to represent the sparse matrix is that the complexity of inserting or deleting a node in a linked list is lesser than the array.
- A node in the linked list representation consists of four fields. The four fields of the linked list are given as follows –
  - **Row -** It represents the index of the row where the non-zero element is located.
  - **Column -** It represents the index of the column where the non-zero element is located.
  - **Value -** It is the value of the non-zero element that is located at the index (row, column).
  - **Next node -** It stores the address of the next node.

The node structure of the linked list representation of the sparse matrix is –



Example

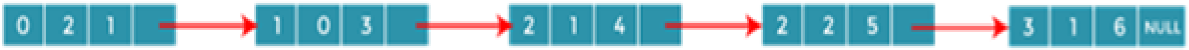
Consider the sparse matrix



$$\begin{array}{c|cccc|} & 0 & 1 & 2 & 3 \\ \hline 0 & 0 & 0 & 1 & 0 \\ 1 & 3 & 0 & 0 & 0 \\ 2 & 0 & 4 & 5 & 0 \\ 3 & 0 & 6 & 0 & 0 \\ \hline \end{array}$$

In the above figure, we can observe a 4x4 sparse matrix containing 5 non-zero elements and 11 zero elements. Above matrix occupies 4x4 = 16 memory space. Increasing the size of matrix will increase the wastage space.

The linked list representation of the above matrix is given below -



- The sparse matrix is represented in the linked list form.
- In the node, the first field represents the index of the row, the second field represents the index of the column, the third field represents the value, and the fourth field contains the address of the next node.
- In the above figure, the first field of the first node of the linked list contains 0, which means 0th row, the second field contains 2, which means 2nd column, and the third field contains 1 that is the non-zero element.
- So, the first node represents that element 1 is stored at the 0th row-2nd column in the given sparse matrix. In a similar manner, all of the nodes represent the non-zero elements of the sparse matrix.

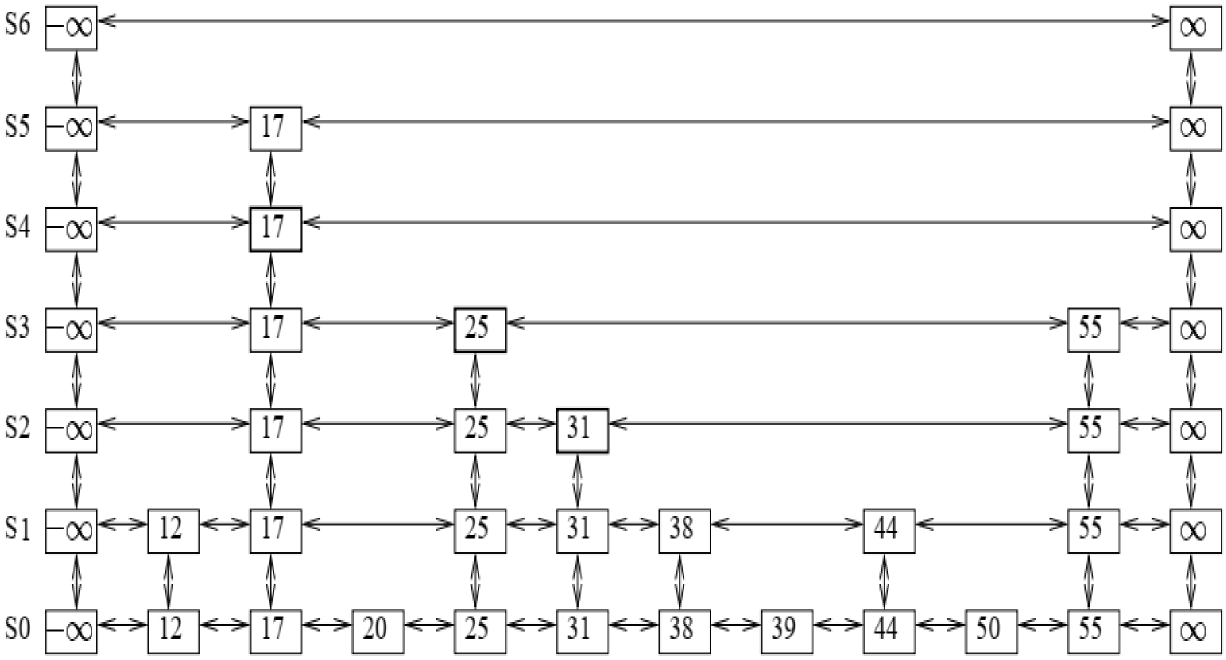
SKIP LIST REPRESENTATION

- A skip list is a data structure that is used for storing a sorted list of items with a help of hierarchy of linked lists that connect increasingly sparse subsequences of the items.
- A skip list allows the process of item look up in efficient manner. The skip list data structure skips over many of the items of the full list in one step, that's why it is known as skip list.
- A ***skip list*** *S* for a map M consists of a series of lists *{S_o,S_1,. . . , S_h,}.*
- Each list S_i stores a subset of the entries of M sorted by increasing keys plus entries with two special keys, denoted -∞ and +∞,
- where -∞ is smaller than every possible key that can be inserted in M and +∞ is larger than every possible key that can be inserted in M.

In addition, the lists in *S* satisfy the following:

1. List S_0 contains every entry of the map M (plus the special entries with keys -∞ and +∞).
2. For i = 1,. . . , h - 1, list S_i contains (in addition to -∞ and +∞) a randomly generated subset of the entries in list S_{i} -**1.**
3. List S_h, contains only -∞ and +∞.

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 Operation : To Search any element in a list

2. Insertion Operation : To Insert any element in a list

3. Deletion Operation : To Delete any element from a list

1.Searching in a Skip List

Searching for a key K in a skip list is quite simple. we always start from the top-most, left position storing the special entry with key -∞.

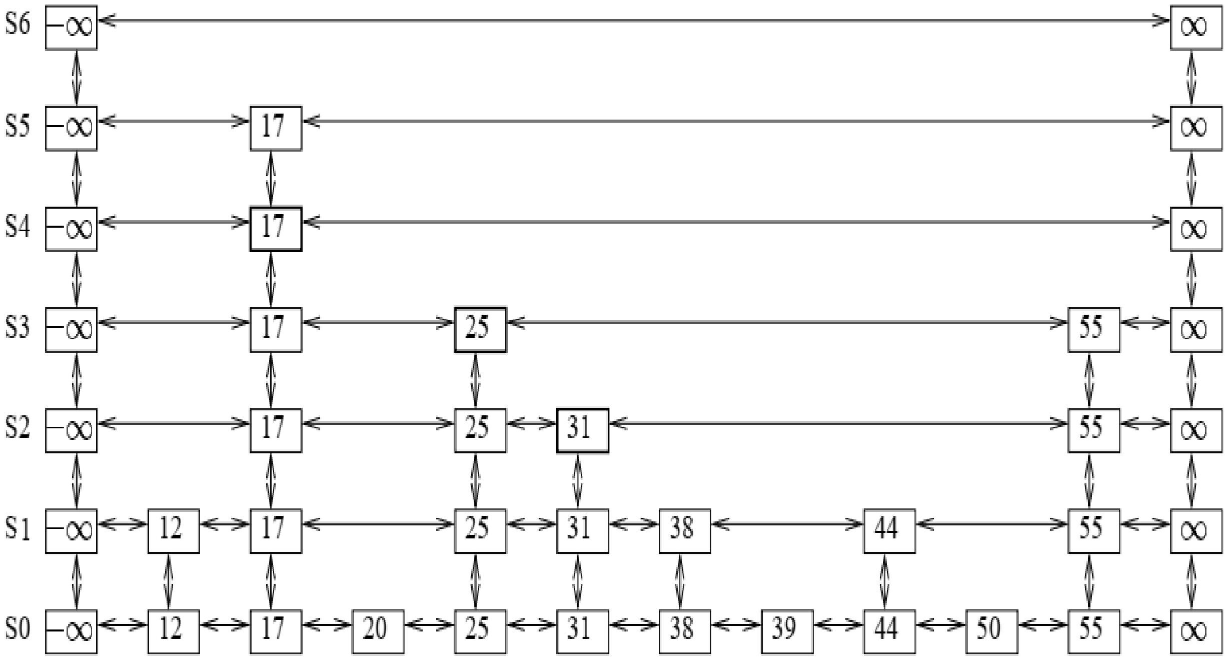
The following Procedure for Searching in a Skip List

1.If key = == current element, Search is successful!

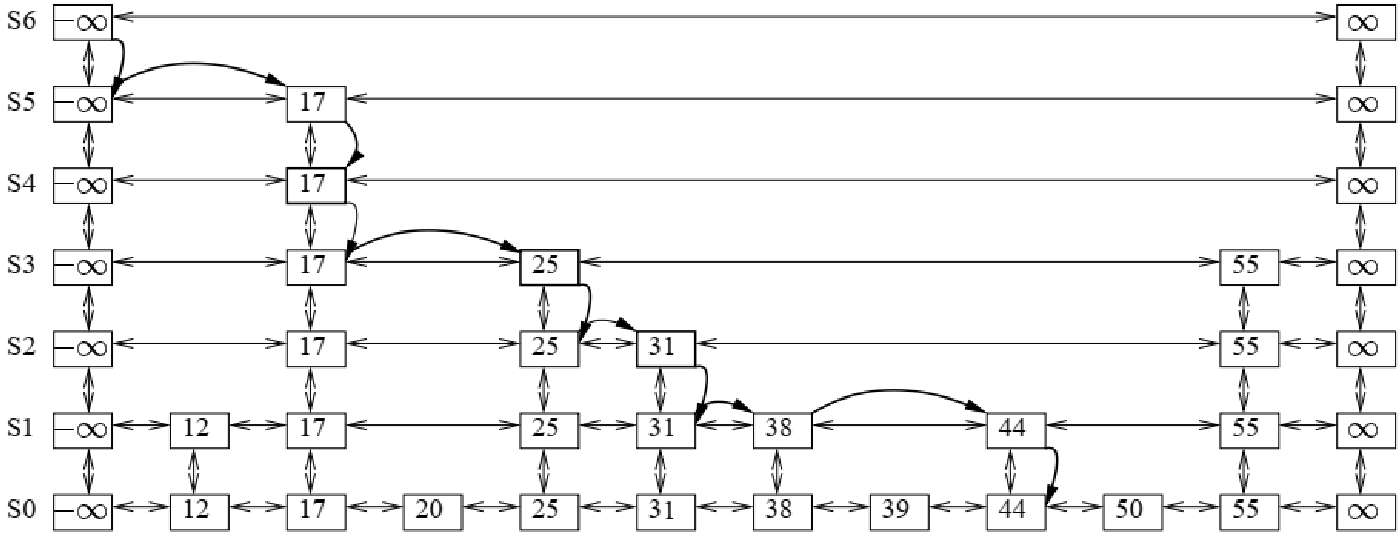
2.Otherwise If key <= current element, go down a level

3. Otherwise If key >= current element, go right

Example :



if we want to searching for the key 44



1. Start at the first position of top list.i.e,S6

2. 44 < ∞, “go down a level”

3. 44 > 17, " go right", and 44 < ∞, “go down a level”

4. 44 < ∞, “go down a level”

5. 44 > 25, " go right", 44 < 55, “go down a level”

6. 44 > 31, " go right", 44 < 55, “go down a level”

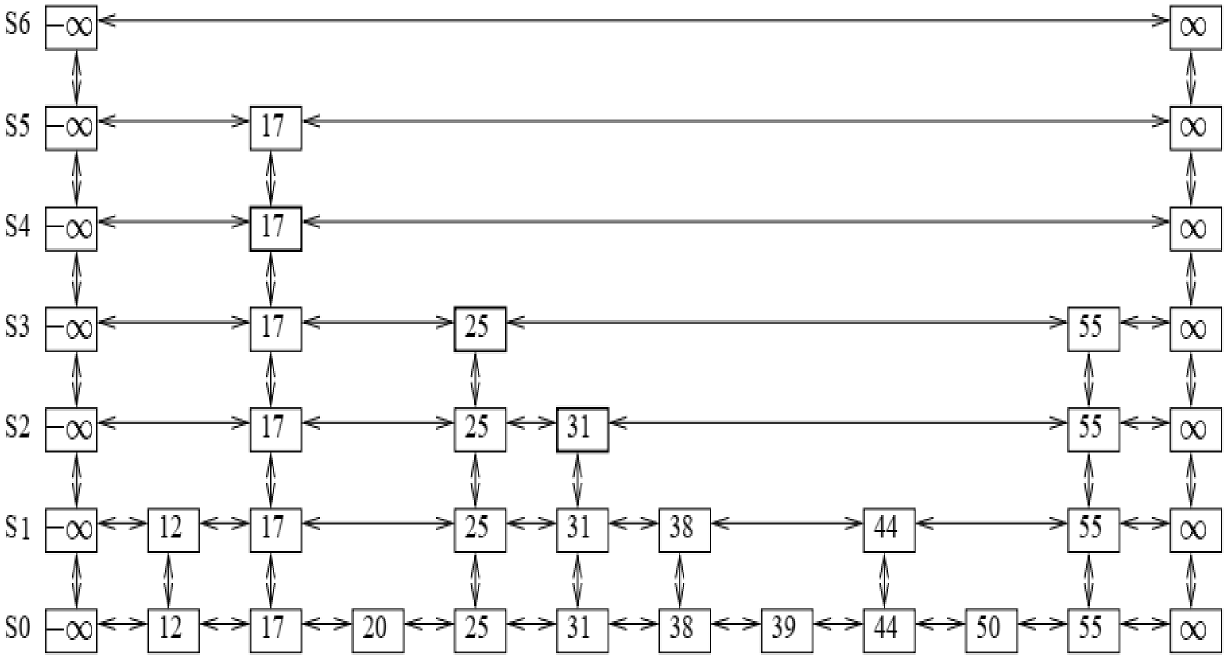
7. 44 > 38, " go right", 44 <= 44, Search is Successful

Time Complexity of search time is O(log n)

2. Inserting in a Skip List :

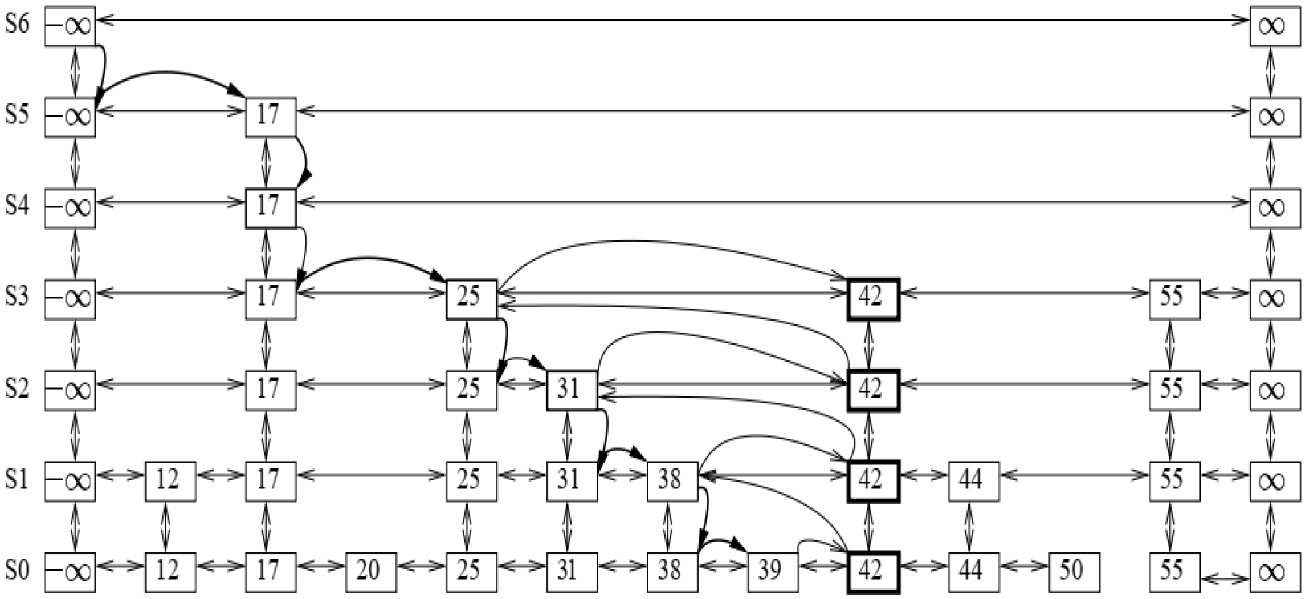
- The Insertion of an entry with key K is a skip list is essentially the same as searching.
- we first call the search operation to find the largest entry in S_0 with key less than or equal to K, then insert the new entry immediately after.
- We use coin flips to decide in how many levels K will appear, and then backtrack to insert K into higher levels.Š

Example :



S6 −∞ ∞
S5 −∞ 17 ∞
S4 −∞ 17 ∞
S3 −∞ 17 25 55 ∞
S2 −∞ 17 25 31 55 ∞
S1 −∞ 12 17 25 31 38 44 55 ∞
S0 −∞ 12 17 20 25 31 38 39 44 50 55 ∞

If we want to inserting an entry with key 42.



S6 −∞ ∞
S5 −∞ 17 ∞
S4 −∞ 17 ∞
S3 −∞ 17 25 42 55 ∞
S2 −∞ 17 25 31 42 55 ∞
S1 −∞ 12 17 25 31 38 42 44 55 ∞
S0 −∞ 12 17 20 25 31 38 39 42 44 50 55 ∞

Inserting an element in a skip list has the following steps

1. Start at the first position of top list.i.e,S6

2. 42 < ∞, “go down a level”

3. 42 > 17, " go right", and 42 < ∞, “go down a level”

4. 42 < ∞, “go down a level”

5. 42 > 25, " go right", 42 < 55, “go down a level”

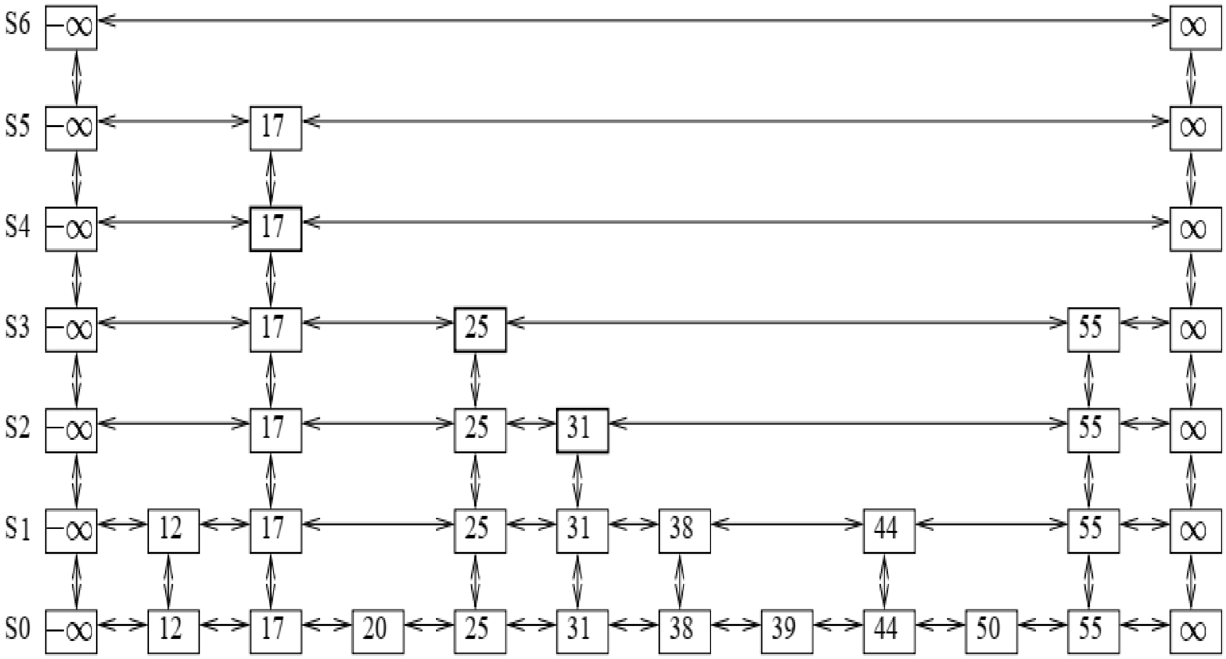
6. 42 > 31, " go right", 42 < 55, “go down a level”

7. 42 > 38, " go right", 42 <= 44, Insert element 42 and insertion is successful.

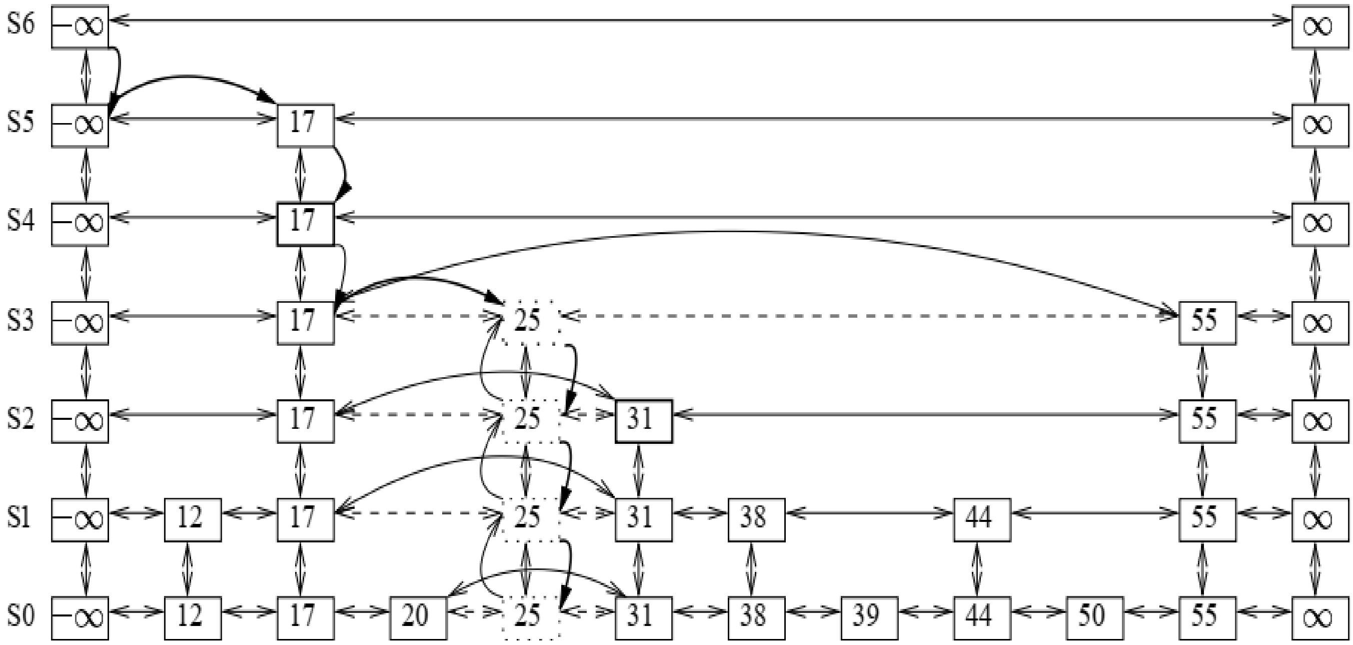
3.Deletion in a Skip list :

- The deletion of an entry with key K is a skip list is similar. we first call the search operation to find the position of the entry with key K, then remove the entry and all the positions it from the skip list.
- The double linked list becomes handy in this operation, but they are not necessary.
- In fact, we can replace all the double linked lists in the skip list with linked lists without affecting the asymptotic performance.Š

Example :



If we want to deleting an entry with key 25



Deletion an element from a skip list has the following steps

1. Start at the first position of top list.i.e,S6

2. 25 < ∞, “go down a level”

3. 25 > 17, " go right", and 25 < ∞, “go down a level”

4. 25 < ∞, “go down a level”

5. 25 <= 25, remove the entry and all the positions it from the skip list.

Applications of Skip List

1. Skip list are used in distributed applications. In distributed systems, the nodes of skip list represents the computer systems and pointers represent network connection.

2. Skip list are used for implementing highly scalable concurrent priority queues with less lock contention (struggle for having a lock on a data item).

HASHING

In data structures,

- There are several searching techniques like linear search, binary search, search trees etc.
- In these techniques, time taken to search any particular element depends on the total number of elements.

Example-

- Linear Search takes O(n) time to perform the search in unsorted arrays consisting of n elements.
- Binary Search takes O(log n) time to perform the search in sorted arrays consisting of n elements.
- It takes O(log n) time to perform the search in Binary Search Tree consisting of n elements.

Drawback-

The main drawback of these techniques is-

- As the number of elements increases, time taken to perform the search also increases.
- This becomes problematic when total number of elements become too large.

Hashing in Data Structure-

In data structures,

- Hashing is a well-known technique to search any particular element among several elements.
- It minimizes the number of comparisons while performing the search.

Advantage-

Unlike other searching techniques,

- Hashing is extremely efficient.
- The time taken by it to perform the search does not depend upon the total number of elements.
- It completes the search with constant time complexity O(1).

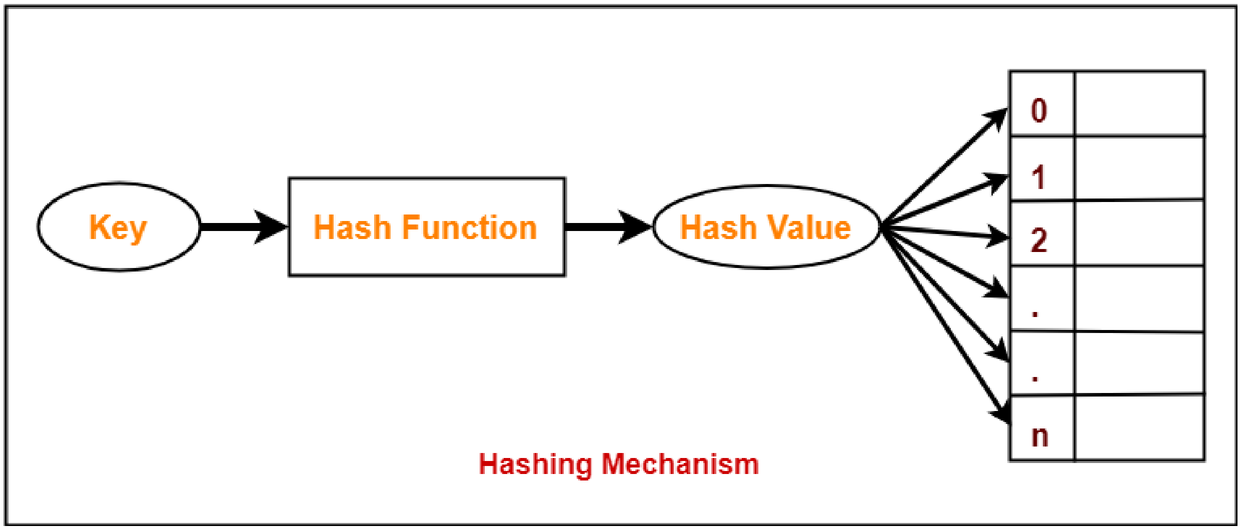
Hashing Mechanism :

In hashing,

- An array data structure called as **Hash table** is used to store the data items.
- Based on the hash key value, data items are inserted into the hash table.

Hash Key Value-

- Hash key value is a special value that serves as an index for a data item.
- It indicates where the data item should be be stored in the hash table.
- Hash key value is generated using a hash function.



Hashing Mechanism

Hash Function :

- Hash function is a function that maps any big number or string to a small integer value.
- Hash function takes the data item as an input and returns a small integer value as an output.
- The small integer value is called as a hash value.
- Hash value of the data item is then used as an index for storing it into the hash table.

Types of Hash Functions :

There are various types of hash functions available such as-

1.Division Method

2.Mid Square Method

3.Folding Method

It depends on the user which hashes function he wants to use.

1.Division Method :

- In this method the key (k) is divided by the number of slots (n) in the table and Take the remainder.
- **The hash function is**

h(k) = k mod N

Where,

h(k) = Hash key

k = k is keys

N = Size of the Hash Table.

Example : Consider the keys to be placed in their home buckets (or) slots are

54, 72, 89, 37, 88, 121

Hash Table size = 10.

Solution :

h(k) = k mod N

h (54) = 54 mod 10 = 4

h (72) = 72 mod 10 = 2

h (89) = 89 mod 10 = 9

h (37) = 37 mod 10 = 7

h (88) = 88 mod 10 = 8

h (121) = 121 mod 10 = 1

Buckets	Key
0	
1	121
2	72
3	
4	54
5	
6	
7	37
8	88
9	89

2.Mid Square Method :

- The Key 'K' is multiplied by itself and the address is obtained by selecting an appropriate number of digits from the middle of the square.
- The number of digits selected depends on the size of the hash table.

Example1 : if Key =123456, hash table size= 1000

$(123456)^2$ = 1524138 3936

H(K) = 138

Example2 : if Key =3205, hash table size= 100

$(3205)^2$ = 10272025

H(K) = 72

Example3 : if Key =87431, hash table size= 1000

$(87431)^2$ = 7644179761

The possible 3 digit mids of **7644179761** are 417 or 179, we can pick any of those mids. If we pick 417 then the element x=87431 will be stored at the index=417 in the hash table with the size of 1000 slots.

H(K) = 417

Note: In the case of two mids, we have to remember that if pick the left side mid then during searching for the same element or inserting more elements, we must pick the left mid.

3.Folding Method : There are two types of Folding Methods

1 : Fold Shift

2 : Fold Boundary

1 : Fold Shift :

- The key K is partitioned into a number of parts, each of which has the same length as the required address with the possible exception of the last part.
- The parts are then added together, ignoring the final carry, to form an address.

Example1 : if Key =356942781, hash table size= 1000

Solution :

- The key is partitioned into 3 parts from left to right. Therefore the parts are
  356, 942, 781
- These parts are then added together = 356 + 942 + 781 = 2079, ignore first 2.
- Finally the address is 079

Example2 : if Key =6217569, hash table size= 100

Solution :

- The key is partitioned into 2 parts from left to right. Therefore the parts are

  62, 17, 56, 9
- These parts are then added together = 62 + 17 + 56 + 9 = 144, ignore first 1.
- Finally the address is 44

2 : Fold Boundary :

- The left and right numbers are folded on a fixed boundary between them and the center number. The two outside values are thus reversed.
- The parts are then added together, ignoring the final carry, to form an address.

Example1 : if Key =356942781, hash table size= 1000

Solution :

- The key is partitioned into 3 parts from left to right. Therefore the parts are

  356, 942, 781
- The two outside values are thus reversed : 653, 942, 187
- These parts are then added together = 653 + 942 + 187 = 1782, ignore first 1.
- Finally the address is 782

Example2 : if Key =6217569, hash table size= 100

Solution :

- The key is partitioned into 2 parts from left to right. Therefore the parts are

  62, 17, 56, 9
- The two outside values are reversed : 26, 17, 56, 9
- These parts are then added together = 26 + 17 + 56 + 9 = 108, ignore first 1.
- Finally the address is 08

Properties of Hash Function-

The properties of a good hash function are-

1. It is efficiently computable.
2. It minimizes the number of collisions.
3.It distribute the keys uniformly over the table.

Collision: When two keys have the same location (or) same hash value in the hash table is called "Collision"

Example : Consider the following keys are to be inserted in the hash table.

131 44, 43, 78, 19, 36, 57 and 77.

Hash table Size = 10

Solution:

h(k) = k mod N

h(131)=131 mod 10 = 1

h(44)=44 mod 10 = 4

h(43)=43 mod 10 = 3

h(78)=78 mod 10 = 8

h(19)=19 mod 10 = 9

h(36)=36 mod 10 = 6

h(57)=57 mod 10 = 7

h(77)=77 mod 10 = 7

Hash table will be

Buckets	Key
0	
1	131
2	
3	43
4	44
5	
6	36
7	57
8	78
9	19

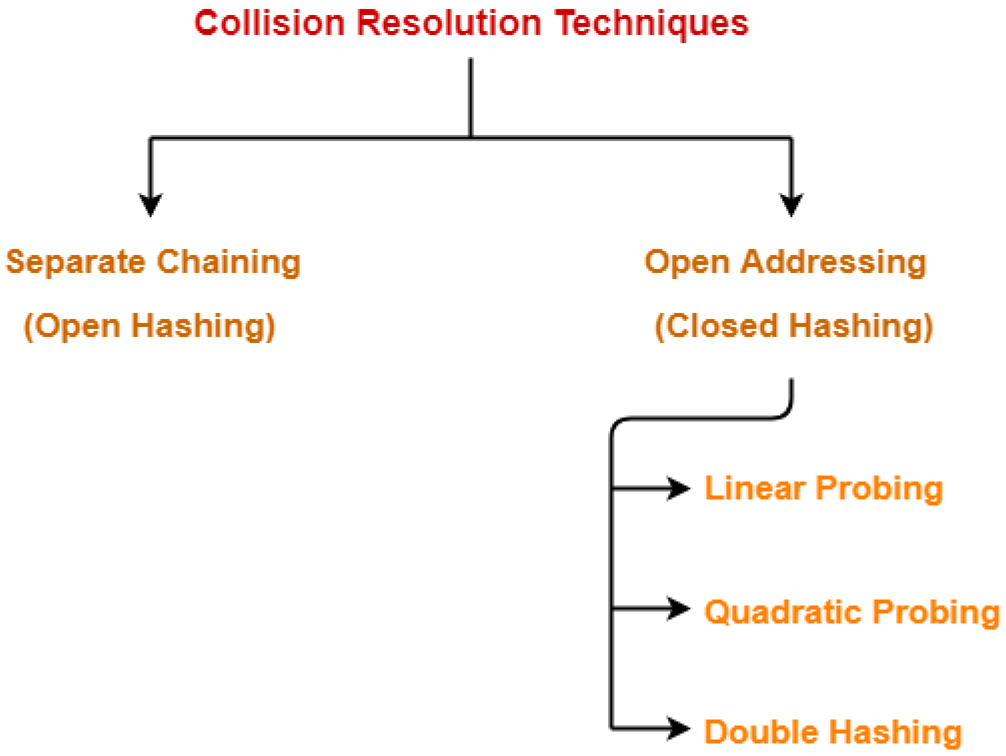
Collision occur →

Now, key 77 will be inserted in bucket-7 of the hash table. Since bucket-7 is already occupied, This situation is called collision.

Collision Resolution Techniques-

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

Collision resolution techniques are classified as-



1.Separate Chaining

2.Open Addressing

1.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**.

Example : Consider the keys to be placed in their home buckets are

131, 3, 4, 21, 61, 24, 7, 97, 8, 9 **Hash Table size(N) = 10**

Solution :

Step 1 : The first key to be inserted in the hash table = 131

h (k) = k mod N

h(131) = 131 mod 10 = 1.

So, key 131 will be inserted in bucket-1 of the hash table.

Buckets	Key
0	
1	131
2	
3	
4	
5	
6	
7	
8	
9	

Step 2 : The key to be inserted in the hash table = 3

h (k) = k mod N

h(3) = 3 mod 10 = 3.

So, key 3 will be inserted in bucket-3 of the hash table.

Buckets	Key
0	
1	131
2	
3	3
4	
5	
6	
7	
8	
9	

Step 3 : The key to be inserted in the hash table = 4

h (k) = k mod N

h(4) = 4 mod 10 = 4.

So, key 4 will be inserted in bucket-4 of the hash table.

Buckets	Key
0	
1	131
2	
3	3
4	4
5	
6	
7	
8	
9	

Step 4 : The key to be inserted in the hash table = 21

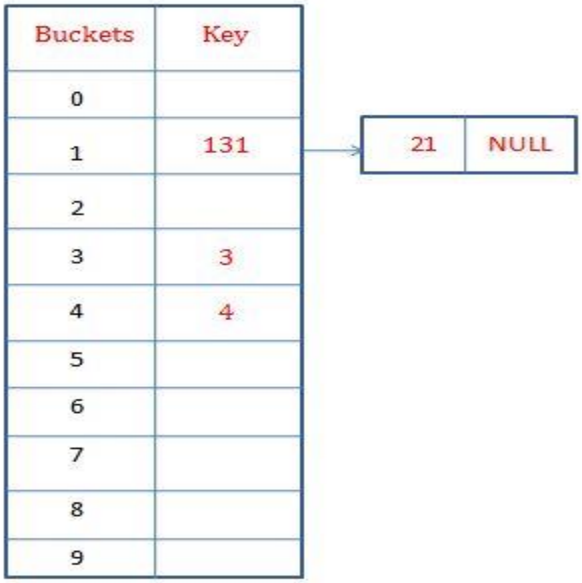
h (k) = k mod N

h(21) = 21 mod 10 = 1.

So, key 21 will be inserted in bucket-1 of the hash table.

Since bucket-1 is already occupied, so collision occurs.

Separate chaining handles the collision by creating a linked list to bucket-1.



Step 5 : The key to be inserted in the hash table = 61

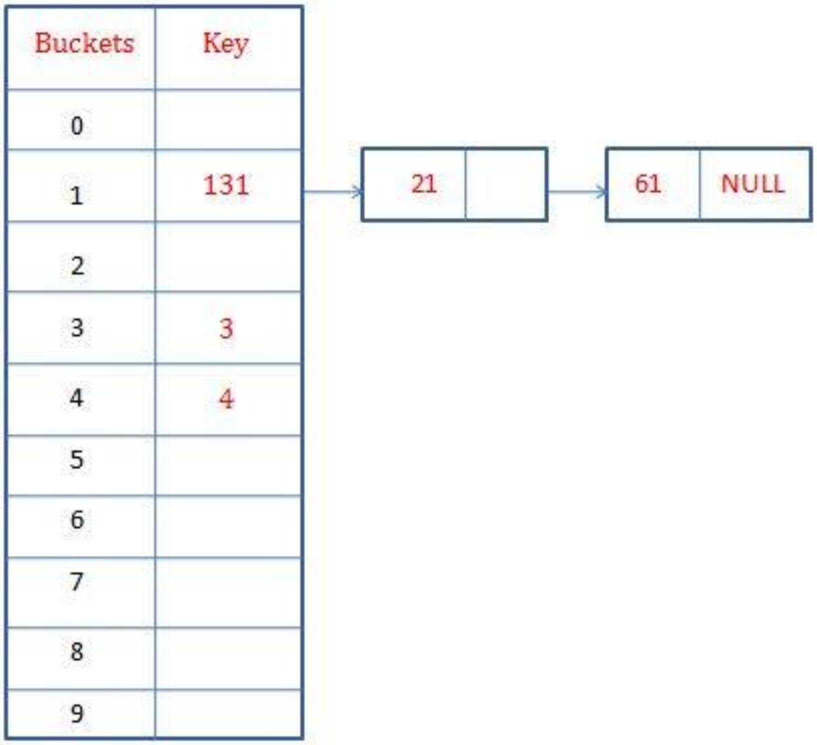
h (k) = k mod N

h(61) = 61 mod 10 = 1.

So, key 61 will be inserted in bucket-1 of the hash table.

Since bucket-1 is already occupied, so collision occurs.

Separate chaining handles the collision by creating a linked list to bucket-1.



Step 6 : The key to be inserted in the hash table = 24

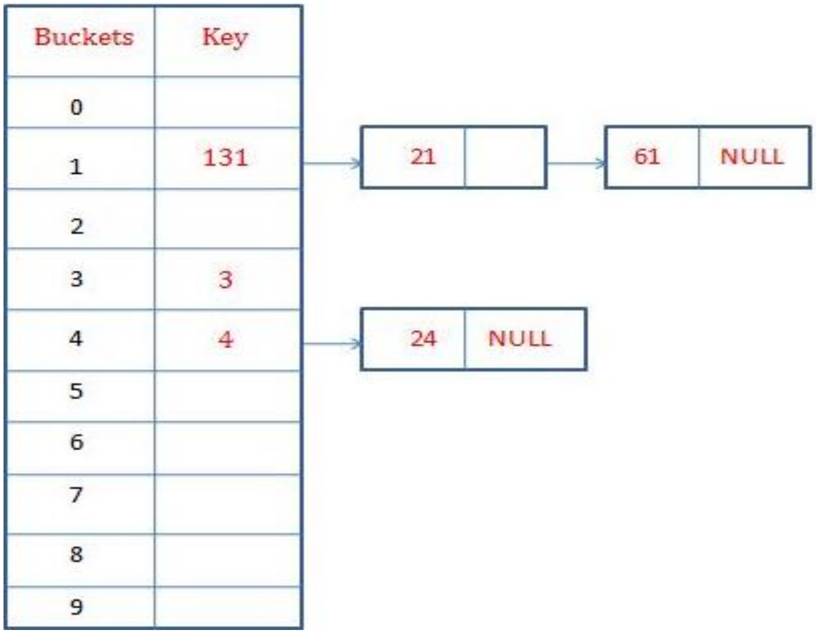
h (k) = k mod N

h(24) = 24 mod 10 = 4.

So, key 24 will be inserted in bucket-4 of the hash table.

Since bucket-4 is already occupied, so collision occurs.

Separate chaining handles the collision by creating a linked list to bucket-4.



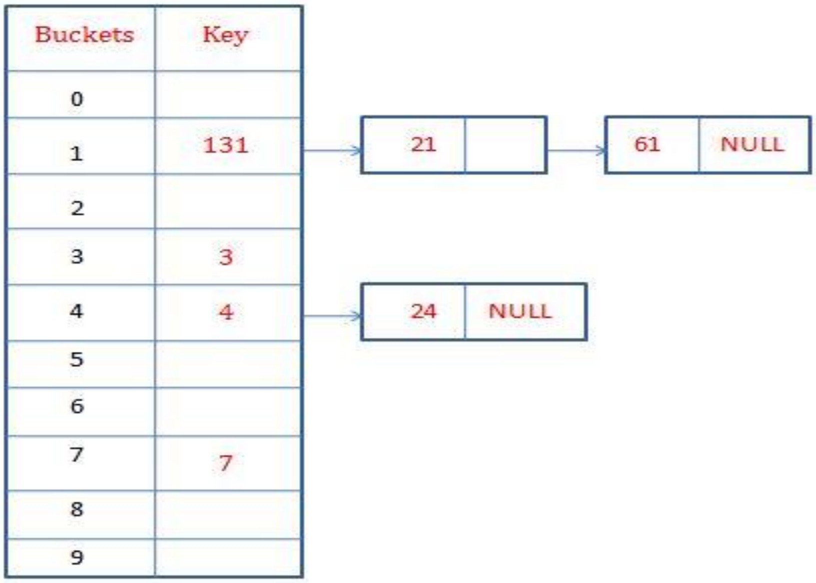
Buckets	Key
0	
1	131
2	
3	3
4	4
5	
6	
7	
8	
9	

Step 7 : The key to be inserted in the hash table = 7

h (k) = k mod N

h(7) = 7 mod 10 = 7.

So, key 7 will be inserted in bucket-7 of the hash table.



Buckets	Key
0	
1	131
2	
3	3
4	4
5	
6	
7	7
8	
9	

Step 8 : The key to be inserted in the hash table = 97

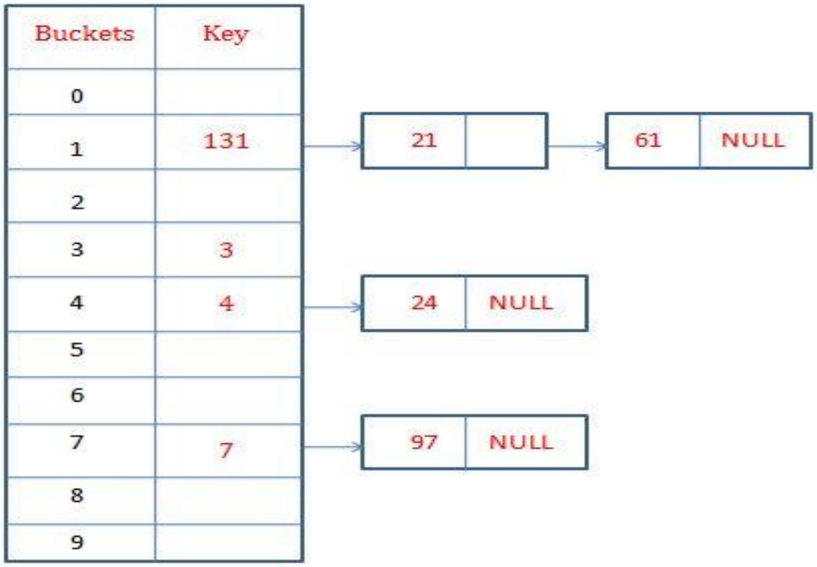
h (k) = k mod N

h(97) = 97 mod 10 = 7.

So, key 97 will be inserted in bucket-7 of the hash table.

Since bucket-7 is already occupied, so collision occurs.

Separate chaining handles the collision by creating a linked list to bucket-7.

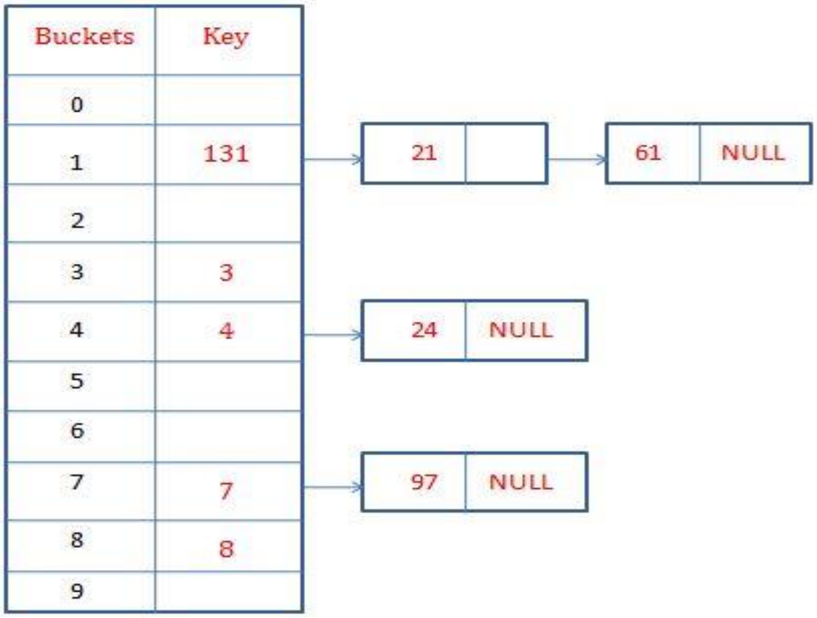


Step 9 : The key to be inserted in the hash table = 8

h (k) = k mod N

h(8) = 8 mod 10 = 8.

So, key 8 will be inserted in bucket-8 of the hash table.

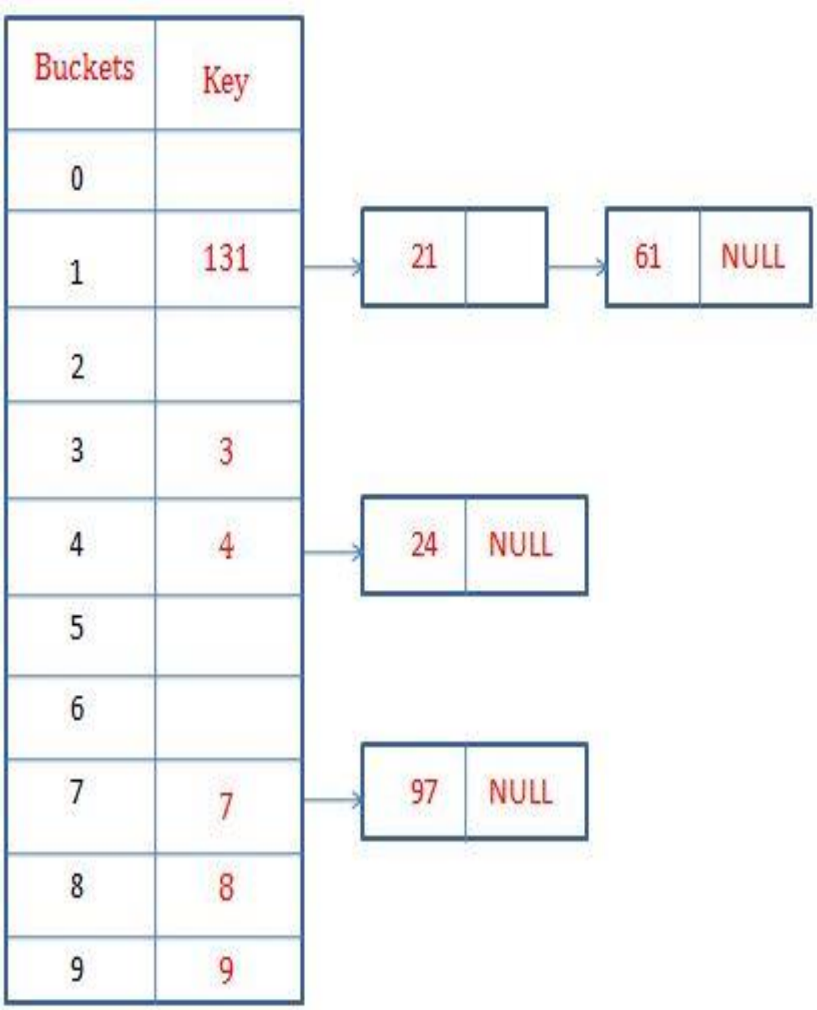


Step 10 : The key to be inserted in the hash table = 9

h (k) = k mod N

h(9) = 9 mod 10 = 9.

So, key 9 will be inserted in bucket-9 of the hash table.



Open Addressing : In Open Addressing, all elements are stored in the hash table itself. So at any point, size of the table must be greater than or equal to the total number of keys .

- In Open Addressing, all the keys are stored inside the hash table.
- No key is stored outside the hash table.

Open addressing Techniques are-

- Linear Probing
- Quadratic Probing
- Double Hashing

- **Probing :** The process of examining memory locations in the hash table is called *probing*.

1. Linear Probing :

- In linear probing, When collision occurs, we linearly probe for the next bucket.
- We keep probing until an empty bucket is found.
- In Linear probing uses a hash function of the form :

h(k , i) = [h(k) + i] mod N

Where ,

h(k) = k mod N

i = i is the probe number ranging from 0 to N-1

N = Size of the Hash Table

Example : Consider the keys to be placed in their home buckets are 126, 75, 37, 56, 29, 154, 10, 99 **Hash Table size = 10**

Step1 : The first key to be inserted in the hash table = 126

h(k , i) = [h (k) + i] mod N

h(126,0) =[(126 mod 10) + 0] mod 10

= [6 + 0] mod 10

= 6 mod 10 = 6

So, key 126 will be inserted in bucket-6 of the hash table.

Buckets	0	1	2	3	4	5	6	7	8	9
Keys							126			

Step2 : The key to be inserted in the hash table = 75

h(k , i) = [h(k) + i] mod N

h(75,0) =[(75 mod 10) + 0] mod 10

= [5 + 0] mod 10

= 5 mod 10 = 5

So, key 75 will be inserted in bucket-5 of the hash table.

Buckets	0	1	2	3	4	5	6	7	8	9
Keys						75	126			

Step 3 : The key to be inserted in the hash table = 37

h(k , i) = [h(k) + i] mod N

h(37,0) =[(37 mod 10) + 0] mod 10

= [7 + 0] mod 10

= 7 mod 10 = 7

So, key 37 will be inserted in bucket-7 of the hash table.

Buckets	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Keys						**75**	**126**	**37**		

Step 4 : The key to be inserted in the hash table = 56

h(k , i) = [h (k) + i] mod N

h(56,0) =[(56 mod 10) + 0] mod 10

= [6 + 0] mod 10

= 6 mod 10 = 6

Since bucket-6 is already occupied, so collision occurs. So the next probe sequence is

h(56,1) =[(56 mod 10) + 1] mod 10

= [6 + 1] mod 10

= 7 mod 10 = 7

Since bucket-7 is already occupied, so collision occurs. So the next probe sequence is

h(56,2) =[(56 mod 10) + 2] mod 10

= [6 + 2] mod 10

= 8 mod 10 = 8

So, key 56 will be inserted in bucket-8 of the hash table.

Buckets	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Keys						**75**	**126**	**37**	**56**	

Step 5 : The key to be inserted in the hash table = 29

h(k , i) = [h (k) + i] mod N

h(29,0) =[(29 mod 10) + 0] mod 10

= [9 + 0] mod 10

= 9 mod 10 = 9

So, key 29 will be inserted in bucket-9 of the hash table.

Buckets	0	1	2	3	4	5	6	7	8	9
Keys						75	126	37	56	29

Step 6 : The key to be inserted in the hash table = 154

h(k , i) = [h (k) + i] mod N

h(154,0) =[(154 mod 10) + 0] mod 10

= [4 + 0] mod 10

= 4 mod 10 = 4

So, key 154 will be inserted in bucket-4 of the hash table.

Buckets	0	1	2	3	4	5	6	7	8	9
Keys					154	75	126	37	56	29

Step 7 : The key to be inserted in the hash table = 10

h(k , i) = [h (k) + i] mod N

h(10,0) =[(10 mod 10) + 0] mod 10

= [0 + 0] mod 10

= 0 mod 10 = 0

So, key 10 will be inserted in bucket-0 of the hash table.

Buckets	0	1	2	3	4	5	6	7	8	9
Keys	10				154	75	126	37	56	29

Step 8 : The key to be inserted in the hash table = 99

h(k , i) = [h (k) + i] mod N

h(99,0) =[(99 mod 10) + 0] mod 10

= [9 + 0] mod 10

= 9 mod 10 = 9

Since bucket-9 is already occupied, so collision occurs. So the next probe sequence is

h(99,1) =[(99 mod 10) + 1] mod 10

= [9 + 1] mod 10

= 10 mod 10 = 0

Since bucket-0 is already occupied, so collision occurs. So the next probe sequence is

h(99,2) =[(99 mod 10) + 2] mod 10

= [9 + 2] mod 10

= 11 mod 10 = 1

So, key 99 will be inserted in bucket-1 of the hash table.

Buckets	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Keys	**10**	**99**			**154**	**75**	**126**	**37**	**56**	**29**

Therefore After Insertion of all keys the final Hash Table is

Buckets	**Key**
0	**10**
1	**99**
2	
3	
4	**154**
5	**75**
6	**126**
7	**37**
8	**56**
9	**29**

2. 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.
- In Quadratic probing uses a hash function of the form :

$$h(k, i) = [\, h(k) + i^2 \,] \bmod N$$

Where ,

h(k) = k mod N

i = i is the probe number ranging from 0 to N-1

N = N is the size of Hash Table

Example : Consider the keys to be placed in their home buckets are

126, 75, 37, 56, 29, 154, 10, 99 **Hash Table size = 11**

Step1 : The first key to be inserted in the hash table = 126

$$h(k, i) = [\, h(k) + i^2 \,] \bmod N$$

$$\begin{aligned} h(126,0) &= [\,(126 \bmod 11) + 0^2\,] \bmod 11 \\ &= [\,5 + 0\,] \bmod 11 \\ &= 5 \bmod 11 = 5 \end{aligned}$$

So, key 126 will be inserted in bucket-5 of the hash table.

Buckets	0	1	2	3	4	5	6	7	8	9	10
Key						126					

Step2 : The key to be inserted in the hash table = 75

$$h(k, i) = [\, h(k) + i^2 \,] \bmod N$$

$$\begin{aligned} h(75,0) &= [\,(75 \bmod 11) + 0^2\,] \bmod 11 \\ &= [\,9 + 0\,] \bmod 11 \\ &= 9 \bmod 11 = 9 \end{aligned}$$

So, key 75 will be inserted in bucket-9 of the hash table.

Buckets	0	1	2	3	4	5	6	7	8	9	10
Key						126				75	

Step3 : The key to be inserted in the hash table = 37

$h(k , i) = [h (k) + i^2] \bmod N$

$h(37,0) =[(37 \bmod 11) + 0^2] \bmod 11$

$= [4 + 0] \bmod 11$

$= 4 \bmod 11 = 4$

So, key 37 will be inserted in bucket-4 of the hash table.

Buckets	0	1	2	3	4	5	6	7	8	9	10
Key					37	126				75	

Step4 : The key to be inserted in the hash table = 56

$h(k , i) = [h (k) + i^2] \bmod N$

$h(56,0) =[(56 \bmod 11) + 0^2] \bmod 11$

$= [1 + 0] \bmod 11$

$= 1 \bmod 11 = 1$

So, key 56 will be inserted in bucket-1 of the hash table.

Buckets	0	1	2	3	4	5	6	7	8	9	10
Key		56			37	126				75	

Step5 : The key to be inserted in the hash table = 29

$h(k , i) = [h (k) + i^2] \bmod N$

$h(29,0) =[(29 \bmod 11) + 0^2] \bmod 11$

$= [7 + 0] \bmod 11$

$= 7 \bmod 11 = 7$

So, key 29 will be inserted in bucket-7 of the hash table.

Buckets	0	1	2	3	4	5	6	7	8	9	10
Key		56			37	126		29		75	

Step 6 : The key to be inserted in the hash table = 154

$h(k , i) = [h (k) + i^2] \bmod N$

$h(154,0) =[(154 \bmod 11) + 0^2] \bmod 11$

$= [0+ 0] \bmod 11$

$= 0 \bmod 11 = 0$

So, key 154 will be inserted in bucket-0 of the hash table.

Buckets	0	1	2	3	4	5	6	7	8	9	10
Key	154	56			37	126		29		75	

Step 7 : The key to be inserted in the hash table = 10

$h(k , i) = [h (k) + i^2] \bmod N$

$h(10,0) =[(10 \bmod 11) + 0^2] \bmod 11$

$= [10+ 0] \bmod 11$

$= 10 \bmod 11 = 10$

So, key 154 will be inserted in bucket-0 of the hash table.

Buckets	0	1	2	3	4	5	6	7	8	9	10
Key	154	56			37	126		29		75	10

Step 8 : The key to be inserted in the hash table = 99

$h(k , i) = [h (k) + i^2] \bmod N$

$h(99,0) =[(99 \bmod 11) + 0^2] \bmod 11$

$= [0 + 0] \bmod 11$

$= 0 \bmod 11 = 0$

Since bucket-0 is already occupied, so collision occurs. So the next probe sequence is

$h(99,1) =[(99 \bmod 11) + 1^2] \bmod 11$
$= [0 + 1] \bmod 11$
$= 1 \bmod 11 = 1$

Since bucket-1 is already occupied, so collision occurs. So the next probe sequence is

$h(99,2) =[(99 \bmod 11) + 2^2] \bmod 11$

$= [0 + 4] \bmod 11$

$= 4 \bmod 11 = 4$

Since bucket-4 is already occupied, so collision occurs. So the next probe sequence is

$h(99,3) = [(99 \bmod 11) + 3^2] \bmod 11$

$= [0 + 9] \bmod 11$

$= 9 \bmod 11 = 9$

Since bucket-9 is already occupied, so collision occurs. So the next probe sequence is

$h(99,4) = [(99 \bmod 11) + 4^2] \bmod 11$

$= [0 + 16] \bmod 11$

$= 16 \bmod 11 = 5$

Since bucket-5 is already occupied, so collision occurs. So the next probe sequence is

$h(99,5) = [(99 \bmod 11) + 5^2] \bmod 11$

$= [0 + 25] \bmod 11$

$= 25 \bmod 11 = 3$

So, key 99 will be inserted in bucket-3 of the hash table.

Buckets	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Key	**154**	**56**		**99**	**37**	**126**		**29**		**75**	**10**

Therefore After Insertion of all keys the final Hash Table is

Buckets	**Key**
0	**154**
1	**56**
2	
3	**99**
4	**37**
5	**126**
6	
7	**29**
8	
9	**75**
10	**10**

3.Double Hashing :

- In Double hashing uses one hash value and then repeatedly steps forward an interval until an empty location is reached.
- The interval is decided using a second, independent hash function, hence the name *double hashing*.
- In double hashing, we use two hash functions rather than a single function.

In Double Hashing uses a hash function of the form :

h(k , i) = [h(k) + i * h2(k)] mod N

Where ,

h(k) = k mod N

i = i is the probe number ranging from 0 to N-1

h2(k) = R-(k mod R)

Where ,

R = is a prime smaller than the TABLE_SIZE.

Example: Consider inserting the keys 76, 26, 37, 59, 21, 65 into a hash table of size n = 11 using Double Hashing

Solution :

Step 1 : The first key to be inserted in the hash table = 76

h(k , i) = [h(k) + i * h2(k)] mod N

h(76,0) =[(76 mod 11) + 0 * (7-(76 mod 7))] mod 11

= [10 + 0 * (7 - 6)] mod 11

= [10 + 0 * 1] mod 11

= 10+0 mod 11 = 10

So, key 76 will be inserted in bucket-10 of the hash table.

Buckets	0	1	2	3	4	5	6	7	8	9	10
Key											76

Step 2 : The key to be inserted in the hash table = 26

h(k , i) = [h(k) + i * h2(k)] mod N

h(26,0) =[(26 mod 11) + 0 * (7-(26 mod 7))] mod 11

= [4 + 0 * (7 - 5)] mod 11

= [4 + 0 * 2] mod 11

= 4+0 mod 11 = 4

So, key 26 will be inserted in bucket-4 of the hash table.

Buckets	0	1	2	3	4	5	6	7	8	9	10
Key					26						76

Step 3 : The key to be inserted in the hash table = 37

h(k , i) = [h(k) + i * h2(k)] mod N

h(37,0) =[(37 mod 11) + 0 * (7-(37 mod 7))] mod 11

= [4 + 0 * (7 - 2)] mod 11

= [4 + 0 * 5] mod 11

= 4+0 mod 11 = 4

Since bucket-4 is already occupied, so collision occurs. So the next probe sequence is

h(37,1) =[(37 mod 11) + 1 * (7-(37 mod 7))] mod 11

= [4 + 1 * (7 - 2)] mod 11

= [4 + 1 * 5] mod 11

= 4+5 mod 11 = 9

So, key 37 will be inserted in bucket-9 of the hash table.

Buckets	0	1	2	3	4	5	6	7	8	9	10
Key					26					37	76

Step 4 : The key to be inserted in the hash table = 59

h(k , i) = [h(k) + i * h2(k)] mod N

h(59,0) =[(59 mod 11) + 0 * (7-(59 mod 7))] mod 11

= [4 + 0 * (7 - 3)] mod 11

= [4 + 0 * 4] mod 11

= 4+0 mod 11 = 4

Since bucket-4 is already occupied, so collision occurs. So the next probe sequence is

h(59,1) =[(59 mod 11) + 1 * (7-(59 mod 7))] mod 11

= [4 + 1 * (7 - 3)] mod 11

= [4 + 1 * 4] mod 11

= 4+4 mod 11 = 8

So, key 59 will be inserted in bucket-8 of the hash table.

Buckets	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Key					**26**				**59**	**37**	**76**

Step 5 : The key to be inserted in the hash table = 21

h(k , i) = [h(k) + i * h2(k)] mod N

h(21,0) =[(21 mod 11) + 0 * (7-(21 mod 7))] mod 11

= [10 + 0 * (7 - 0)] mod 11

= [10 + 0 * 7] mod 11

= 10+0 mod 11 = 10

Since bucket-10 is already occupied, so collision occurs. So the next probe sequence is

h(21,1) =[(21 mod 11) + 1 * (7-(21 mod 7))] mod 11

= [10 + 1 * (7 - 0)] mod 11

= [10 + 1 * 7] mod 11

= 10+7 mod 11

= 17 mod 11 = 6

So, key 21 will be inserted in bucket-6 of the hash table.

Buckets	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Key					**26**		**21**		**59**	**37**	**76**

Step 6 : The key to be inserted in the hash table = 65

h(k , i) = [h(k) + i * h2(k)] mod N

h(65,0) =[(65 mod 11) + 0 * (7-(65 mod 7))] mod 11

= [10 + 0 * (7 - 2)] mod 11

= [10 + 0 * 5] mod 11

= 10+0 mod 11 = 10

Since bucket-10 is already occupied, so collision occurs. So the next probe sequence is

h(65,1) =[(65 mod 11) + 1 * (7-(65 mod 7))] mod 11
= [10 + 1 * (7 - 2)] mod 11
= [10 + 1 * 5] mod 11
= 10+5 mod 11 = 10
= 15 mod 11 = 4

Since bucket-4 is already occupied, so collision occurs. So the next probe sequence is

h(65,2) =[(65 mod 11) + 2 * (7-(65 mod 7))] mod 11
= [10 + 2 * (7 - 2)] mod 11
= [10 + 2 * 5] mod 11
= 10+10 mod 11 = 10
= 20 mod 11 = 9

Since bucket-9 is already occupied, so collision occurs. So the next probe sequence is

h(65,3) =[(65 mod 11) + 3 * (7-(65 mod 7))] mod 11
= [10 + 3 * (7 - 2)] mod 11
= [10 + 3 * 5] mod 11
= 10+15 mod 11 = 10
= 25 mod 11 = 3

So, key 65 will be inserted in bucket-3 of the hash table.

Buckets	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Key				**65**	**26**		**21**		**59**	**37**	**76**

Therefore After Insertion of all keys the final Hash Table is

Buckets	**Key**
0	
1	
2	
3	**65**
4	**26**
5	
6	**21**
7	
8	**59**
9	**37**
10	**76**

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.

Note: In open addressing, the value of load factor always lies 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.
- Basically, when the load factor increases to more than its pre-defined value (default value of load factor is 0.75), the complexity increases.
- So to overcome this, the size of the hash table is increased (doubled) and all the values are hashed again and stored in the new double-sized Hash table to maintain a low load factor and low complexity. Usually to extend the size of the table, to the **next prime from 2 * current_table_size.**

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

Table size is 10

Solution :

h(k) = k mod N

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

Now the Hash Table is

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 next prime from 2 * current_table_size.
- Now new Hash table size is 23.

Now the New Hash Table is

h(k) = k mod N

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.

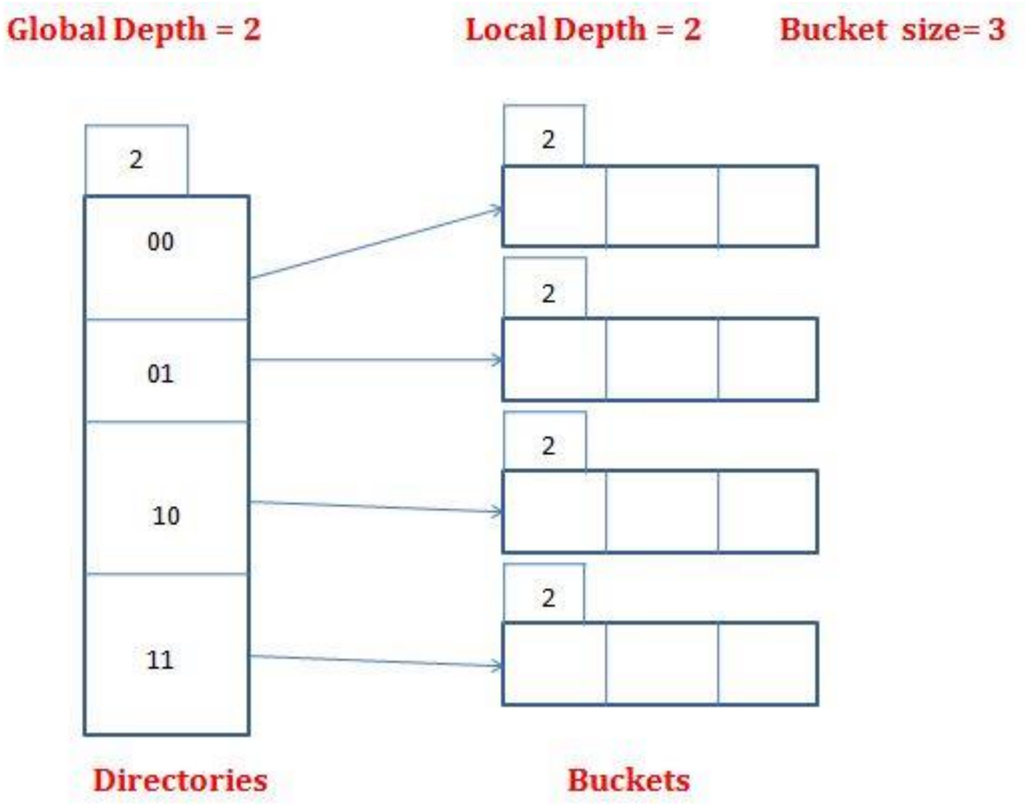
EXTENDABLE HASHING:_Extendible Hashing is a dynamic hashing method wherein directories, and buckets are used to hash data. It is an aggressively flexible method in which the hash function also experiences dynamic changes.

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 – Data elements may exist in various forms eg. Integer, String, Float, etc.. Currently, let us consider data elements of type integer.

Step 2 – 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.

Example: 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.

Example: The binary obtained is: 110001 and the global-depth is 3. So, the hash function will return 3 LSBs of 110**001**. Which is 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 –First, Check if the local depth is less than or equal to the global depth. Then choose one of the cases below.

Case1: 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.

Case2: If 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.

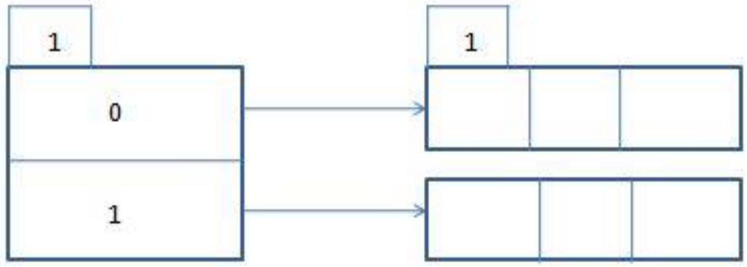
Step 9 – The element is successfully hashed.

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.

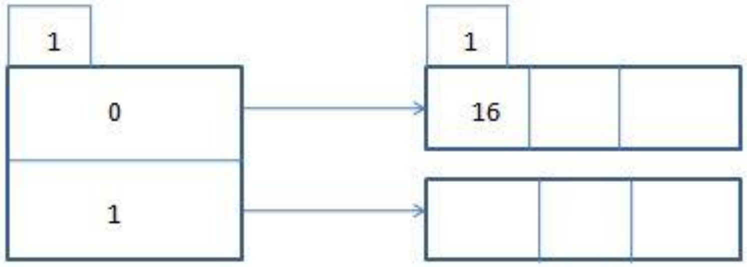
Keys	Binary Form
16	10000
4	00100
6	00110
22	10110
24	11000
10	01010
31	11111
7	00111
9	01001
20	10100
26	11010

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



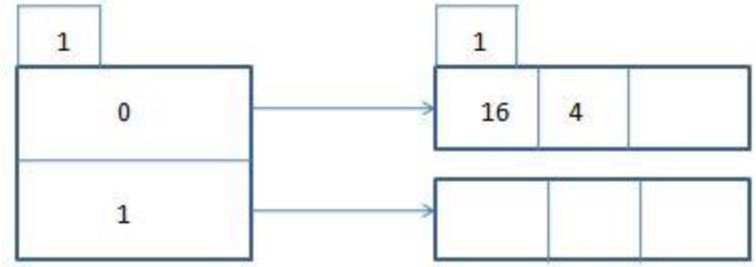
Step2 : Inserting 16 : The binary format of 16 is 10000 and Global depth is 1

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



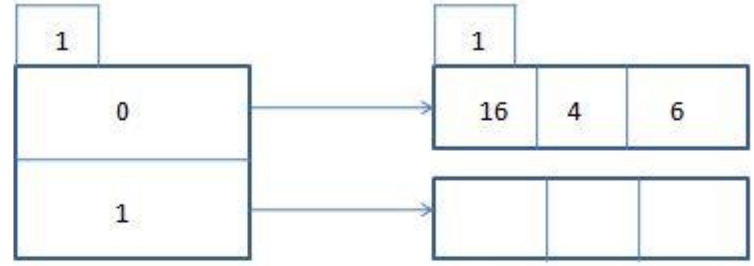
Step3: Inserting 4 : The binary format of 4 is 00100 and Global depth is 1

It is LSB of 0010**0** which is 0. Hence, 4 is mapped to the directory with id=0.



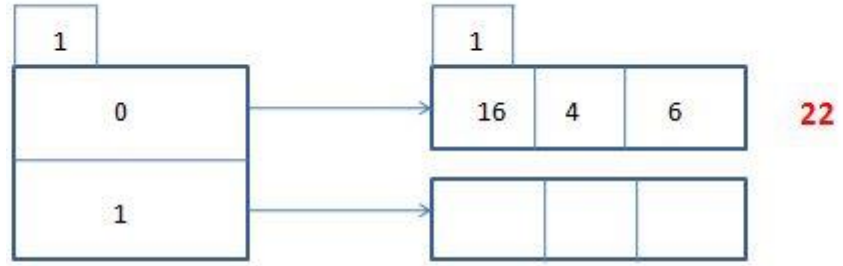
Step 4: Inserting 6 : The binary format of 6 is 00110 and Global depth is 1

It is LSB of 0011**0** 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

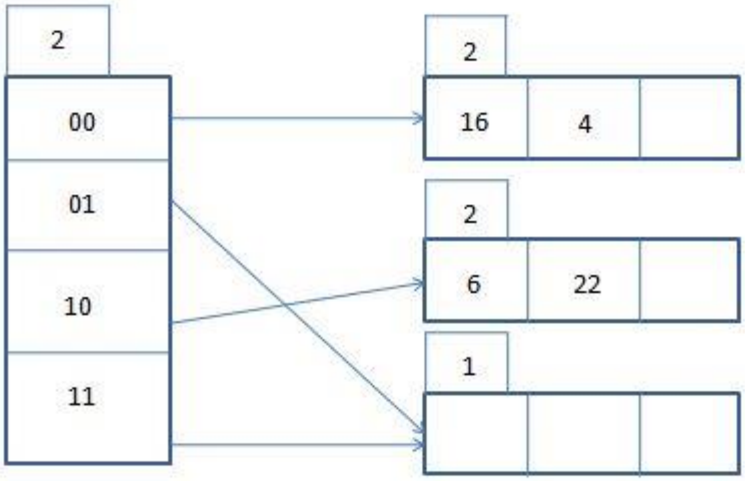
It is LSB of 1011**0** which is 0. Hence, 22 is mapped to the directory with id=0.



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

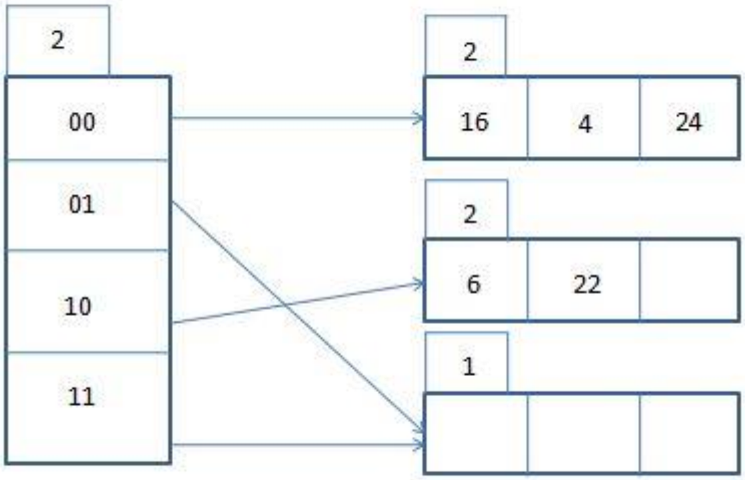
Since Local Depth = Global Depth, the bucket splits and directory expansion takes place. Also, rehashing of numbers present in the overflowing bucket takes place after the split. And, since the global depth is incremented by 1, now,the global depth is 2. Hence, 16,4,6,22 are now rehashed w.r.t 2 LSBs.[16(100**00**),4(1**00**),6(1**10**),22(101**10**)]

After Bucket split and Directory Expansion



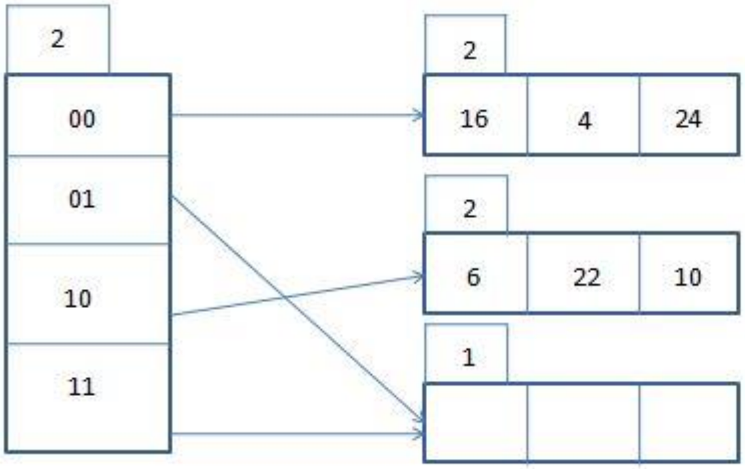
Step 6: Inserting 24 : The binary format of 24 is 11000 and Global depth is 2

It is LSB of 110**00** which is 00. Hence, 24 is mapped to the directory with id=00.

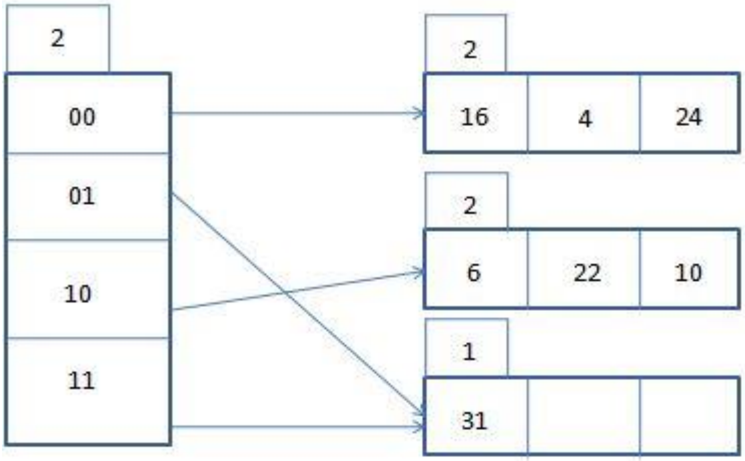


Step 7: Inserting 10 : The binary format of 10 is 01010 and Global depth is 2.

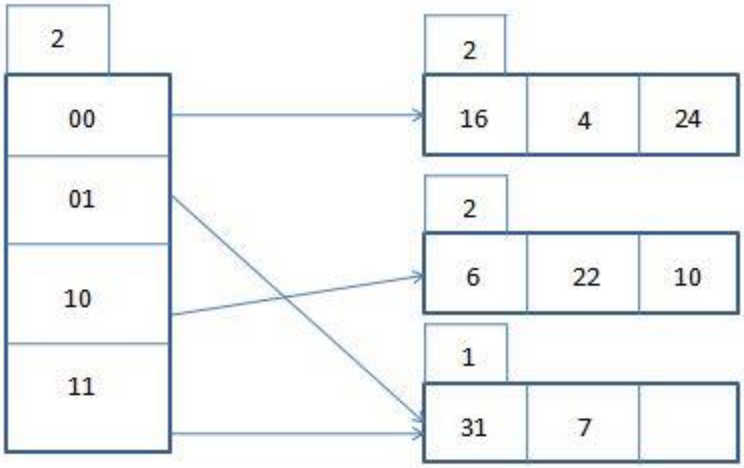
It is LSB of 010**10** which is 10. Hence, 10 is mapped to the directory with id=10.



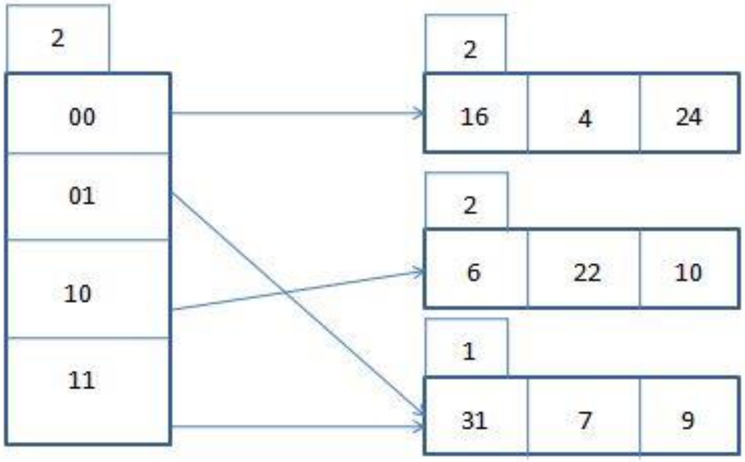
Step 8: Inserting 31 : The binary format of 31 is 11111 and Global depth is 2. It is LSB of 11111 which is 11. Hence, 31 is mapped to the directory with id=11.



Step 9: Inserting 7 : The binary format of 7 is 00111 and Global depth is 2. It is LSB of 00111 which is 11. Hence, 7 is mapped to the directory with id=11.

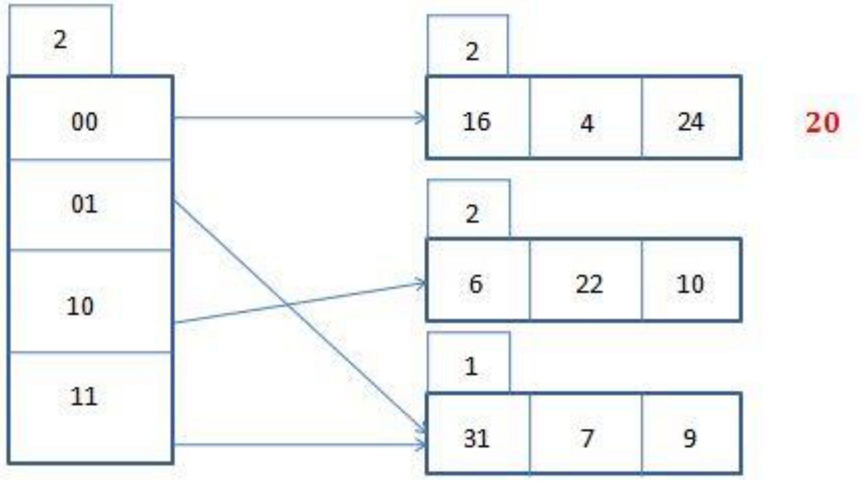


Step 10: Inserting 9 : The binary format of 9 is 01001 and Global depth is 2. It is LSB of 01001 which is 01. Hence, 9 is mapped to the directory with id=01.



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

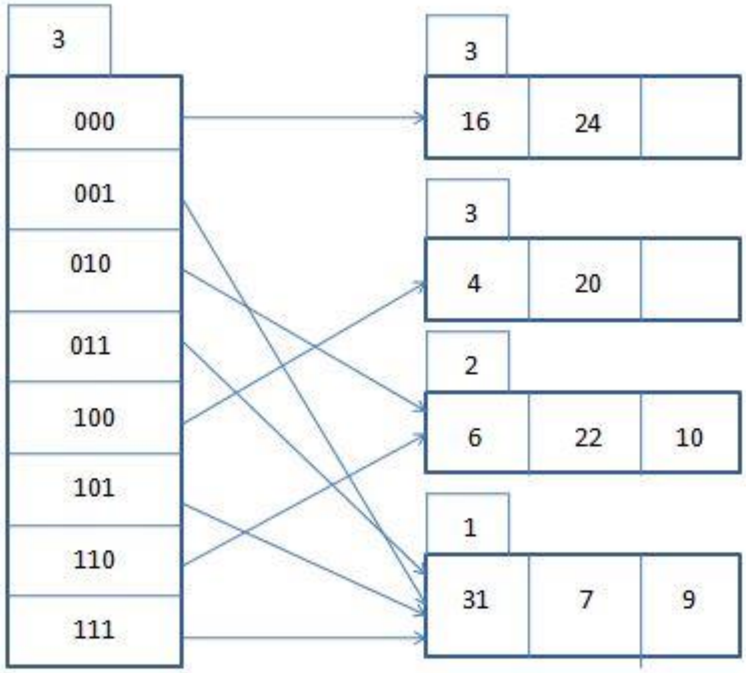
It is LSB of 10100 which is 00. Hence, 20 is mapped to the directory with id=00.



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

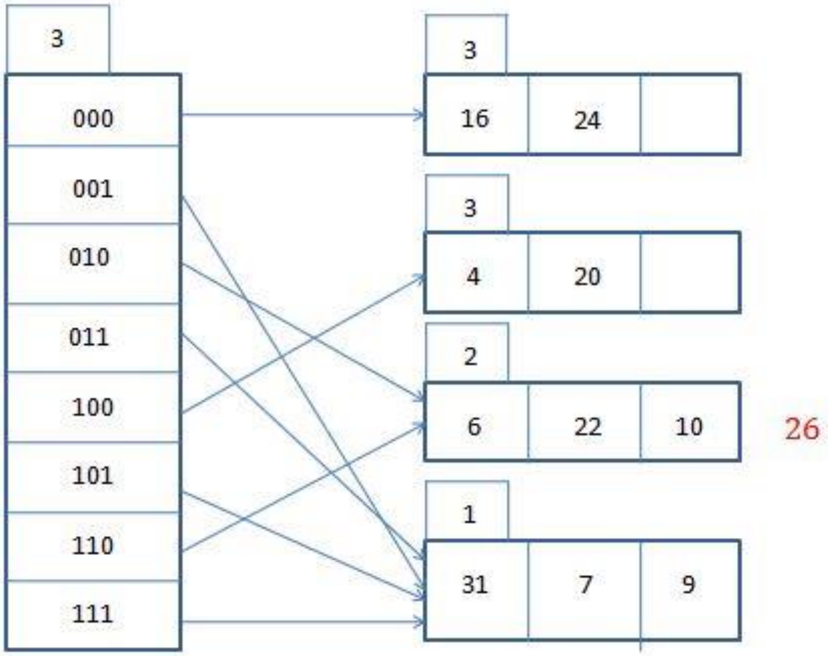
Since Local Depth = Global Depth, the bucket splits and directory expansion takes place. Also, rehashing of numbers present in the overflowing bucket takes place after the split. And, since the global depth is incremented by 1, now, the global depth is 3. Hence, 16,4,24,20 are now rehashed w.r.t 3 LSBs.[16(10**000**),4(00**100**),24(11**000**),20(10**100**)]

After Bucket split and Directory Expansion



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

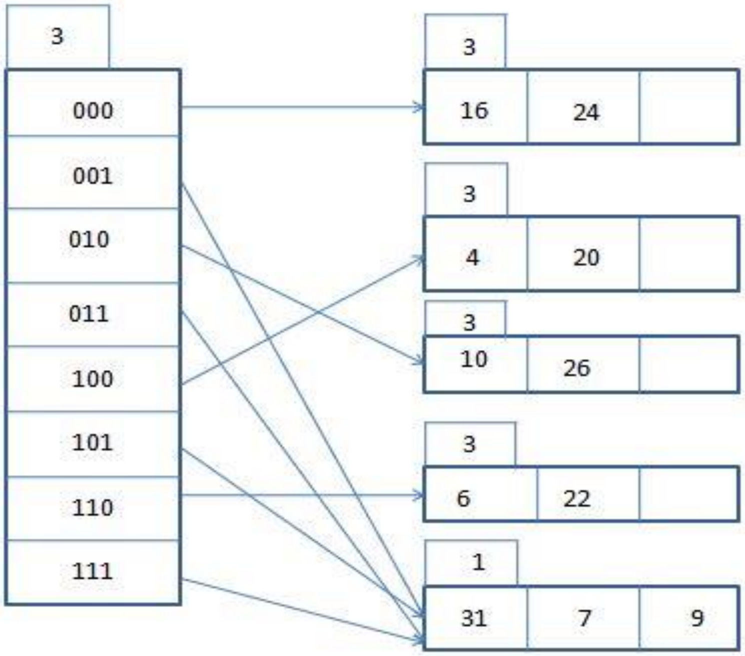
It is LSB of 11010 which is 010. Hence, 26 is mapped to the directory with id=010.



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


3
000
001
010
011
100
101
110
111
3
16
24
3
4
20
3
10
26
3
6
22
1
31
7
9