UNIT II - data structure notes r18 jntuh

Computer Science and Engineering (Jawaharlal Nehru Technological University, Hyderabad)

UNIT - II

Dictionaries: linear list representation, skip list representation, operations - insertion, deletion and searching.

Hash Table Representation: hash functions, collision resolution-separate chaining, open addressing linear probing, quadratic probing, double hashing, rehashing, extendible hashing.

DICTIONARIES:

Dictionary is a collection of pairs of key and value where every value is associated with the corresponding key.

Basic operations that can be performed on dictionary are:

1. Insertion of value in the dictionary

2. Deletion of particular value from dictionary

3. Searching of a specific value with the help of key

A 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.
- A dictionary is also called a hash, a map, a hashmap in different programming languages.
- The keys in a dictionary must be simple types (such as integers or strings) while the values can be of any type.
- Different languages enforce different type restrictions on keys and values in a dictionary.
- Dictionaries are often implemented as hash tables.
- Keys in a dictionary must be unique an attempt to create a duplicate key will typically overwrite the existing value for that key.
- Dictionary is an abstract data structure that supports the following operations: –

  search(K key) (returns the value associated with the given key)

  insert(K key, V value) – delete(K key)

• Each element stored in a dictionary is identified by a key of type K.

• Dictionary represents a mapping from keys to values.

Dictionaries have numerous applications.

– contact book • key: name of person; value: – telephone number

--table of program variable identiers • key: identier; value: address in memory

– property-value collection • key: property name; value: associated value –

--natural language dictionary • key: word in language X; value: word in language Y – etc

operations on dictionaries

Dictionaries typically support several operations:

– retrieve a value (depending on language, attempting to retrieve a missing key may give a default value or throw an exception)

– insert or update a value (typically, if the key does not exist in the dictionary, the key-value pair is inserted; if the key already exists, its corresponding value is overwritten with the new one)

– remove a key-value pair

– test for existence of a key

Note that items in a dictionary are unordered, so loops over dictionaries will return items in an arbitrary order.

Linear List Representation The dictionary can be represented as a linear list. The linear list is a collection of pair and value. There are two method of representing linear list.

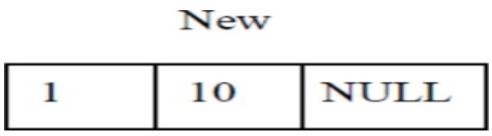
Structure of linear list for dictionary:

To Represent the dictionary with linear list , each node contain the 3 fields : those are key , value and pointer to the next node .

The below is the node structure :

Key	**value**	**Pointer to the next node**

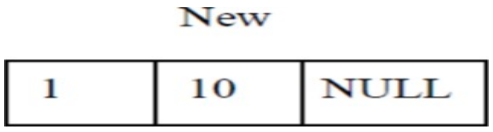
Example :




```
Struct node
{
   Int key;
   Int value;
  Struct node *next;
};
struct node *head;
```

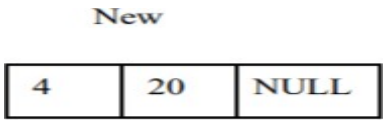
Insertion of new node in Dictionary

Consider that initially dictionary is empty then head = NULL
We will create a new node with some key and value contained in it.

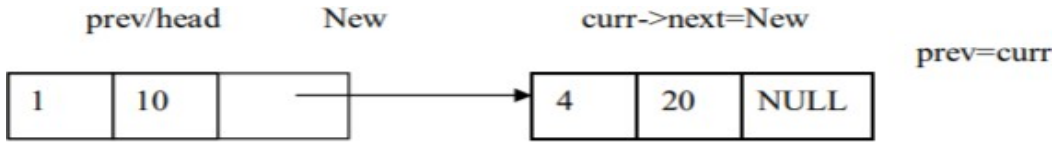


Now as head is NULL, this new node becomes head. Hence the dictionary contains only one record. this node will be 'curr' and 'prev' as well. The 'cuur' node will always point to current visiting node and 'prev' will always point to the node previous to 'curr' node. As now there is only one node in the list mark as 'curr' node as 'prev' node

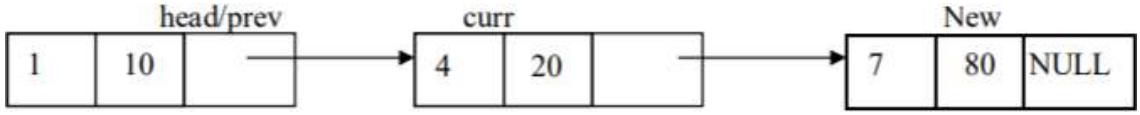
Insert a record, key=4 and value=20,



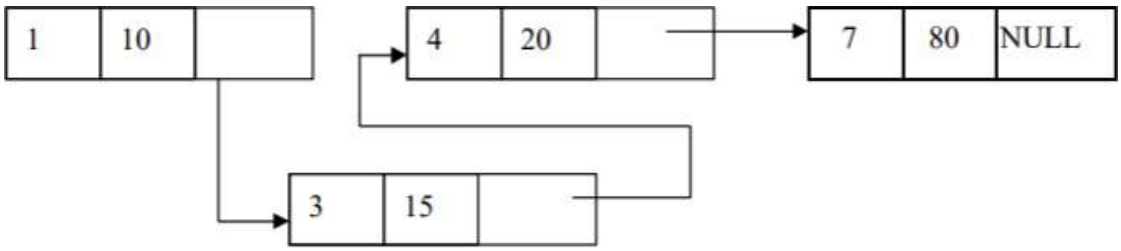
Compare the key value of 'curr' and 'New' node. If New->key > Curr->key then attach New node to 'curr' node.



Add a new node then

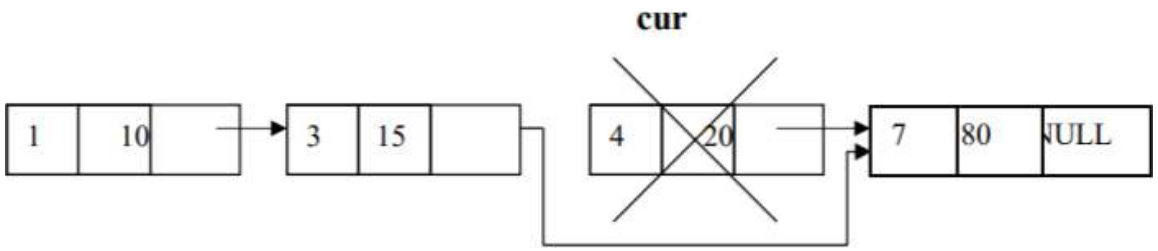


If we insert then we have to search for it proper position by comparing key value. (curr->key < New->key) is false. Hence else part will get executed.

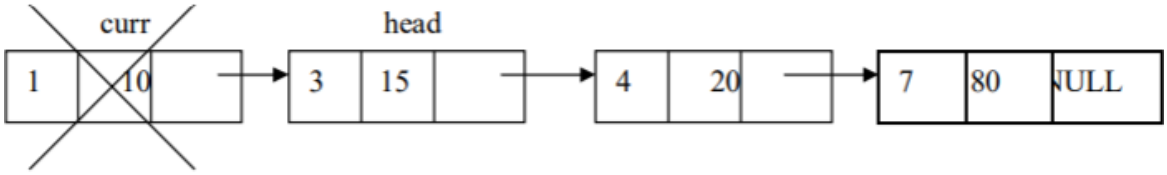


The delete operation:

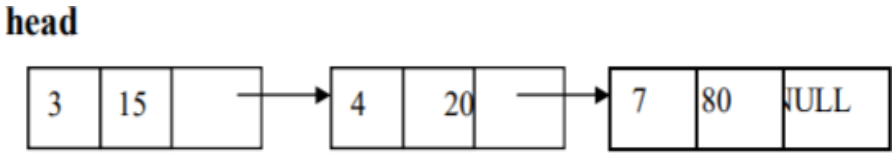
Case 1: Initially assign 'head' node as 'curr' node.Then ask for a key value of the node which is to be deleted. Then starting from head node key value of each jode is cked and compared with the desired node's key value. We will get node which is to be deleted in variable 'curr'. The node given by variable 'prev' keeps track of previous node of 'cuu' node. For eg, delete node with key value 4 then



Case 2: If the node to be deleted is head node i.e.. if(curr==head) Then, simply make 'head' node as next node and delete 'curr'



Hence the listbecomes

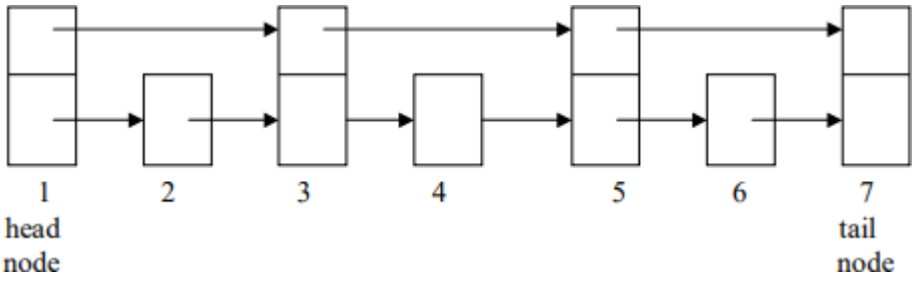


SKIP LIST REPRESENTATION

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.

Skip list is a variant list for the linked list. Skip lists are made up of a series of nodes connected one after the other. Each node contains a key and value pair as well as one or more references, or

pointers, to nodes further along in the list. The number of references each node contains is determined randomly. This gives skip lists their probabilistic nature, and the number of references a node contains is called its node level. There are two special nodes in the skip list one is head node which is the starting node of the list and tail node is the last node of the list.



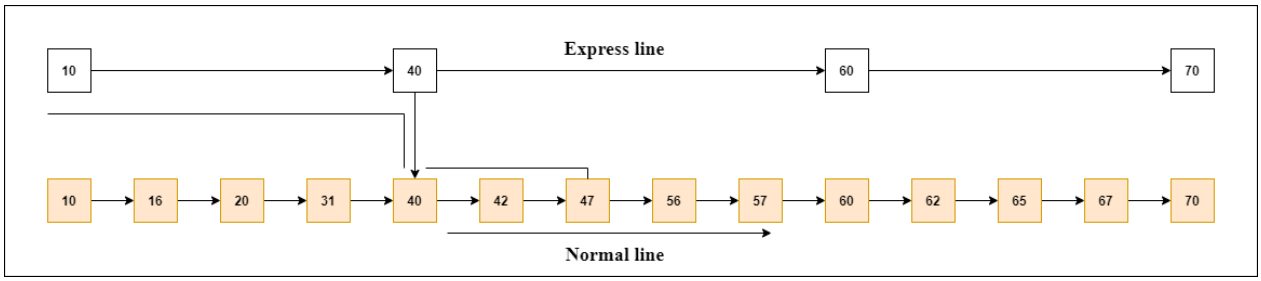
The skip list is an efficient implementation of dictionary using sorted chain. This is because in skip list each node consists of forward references of more than one node at a time.

Skip list structure

It is built in two layers: The lowest layer and Top layer.

The lowest layer of the skip list is a common sorted linked list, and the top layers of the skip list are like an "express line" where the elements are skipped.

You can see in the example that 47 does not exist in the express line, so you search for a node of less than 47, which is 40. Now, you go to the normal line with the help of 40, and search the 47, as shown in the diagram.

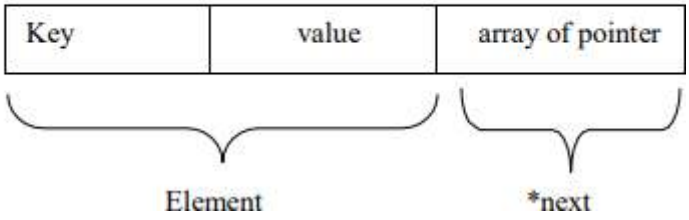


Skip List Basic Operations

There are the following types of operations in the skip list.

- **Insertion operation:** It is used to add a new node to a particular location in a specific situation.
- **Deletion operation:** It is used to delete a node in a specific situation.
- **Search Operation:** The search operation is used to search a particular node in a skip list

The individual node looks like this:

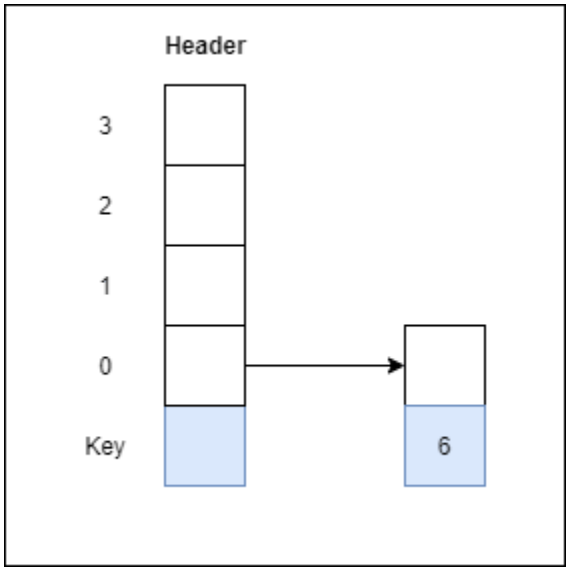


Example 1: Create a skip list, we want to insert these following keys in the empty skip list.

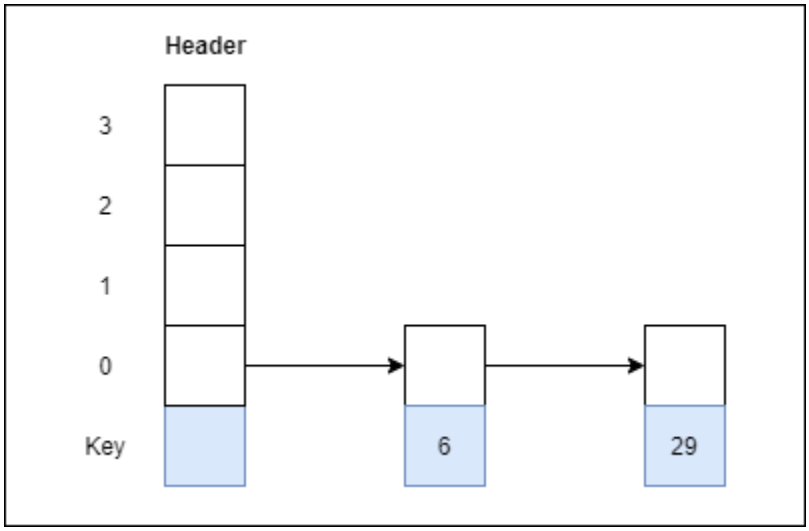
1. 6 with level 1.
2. 29 with level 1.
3. 22 with level 4.
4. 9 with level 3.
5. 17 with level 1.
6. 4 with level 2.

Ans:

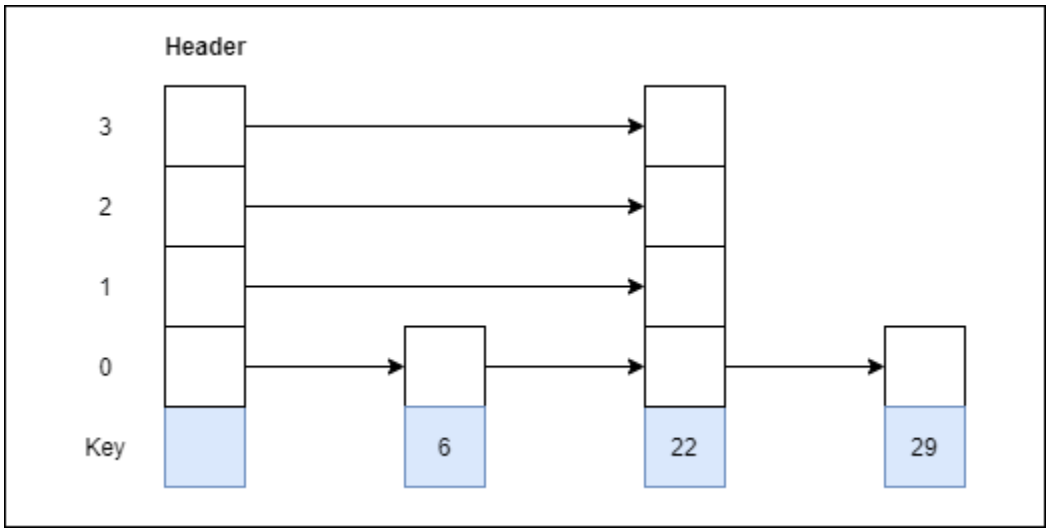
Step 1: Insert 6 with level 1



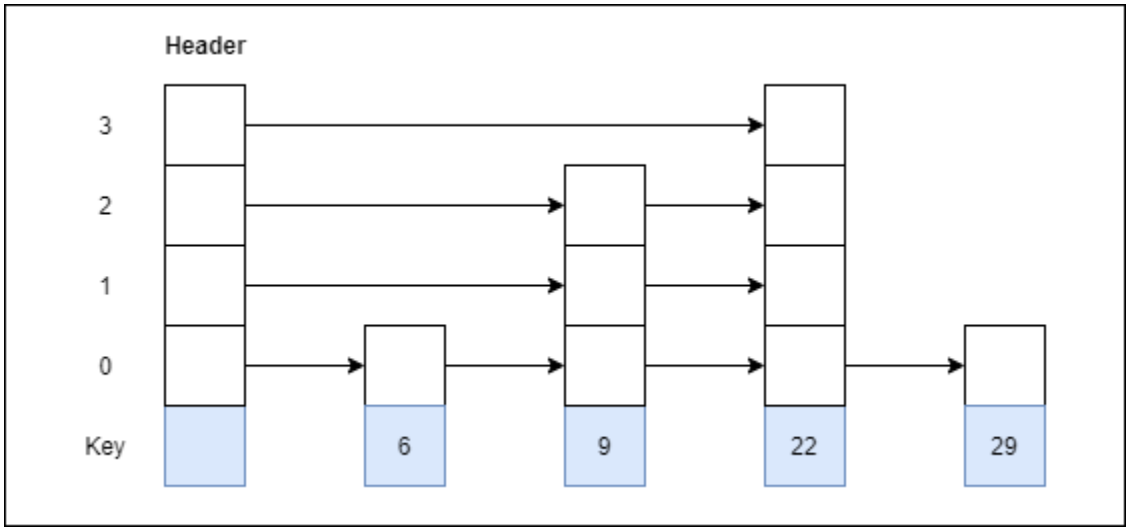
Step 2: Insert 29 with level 1



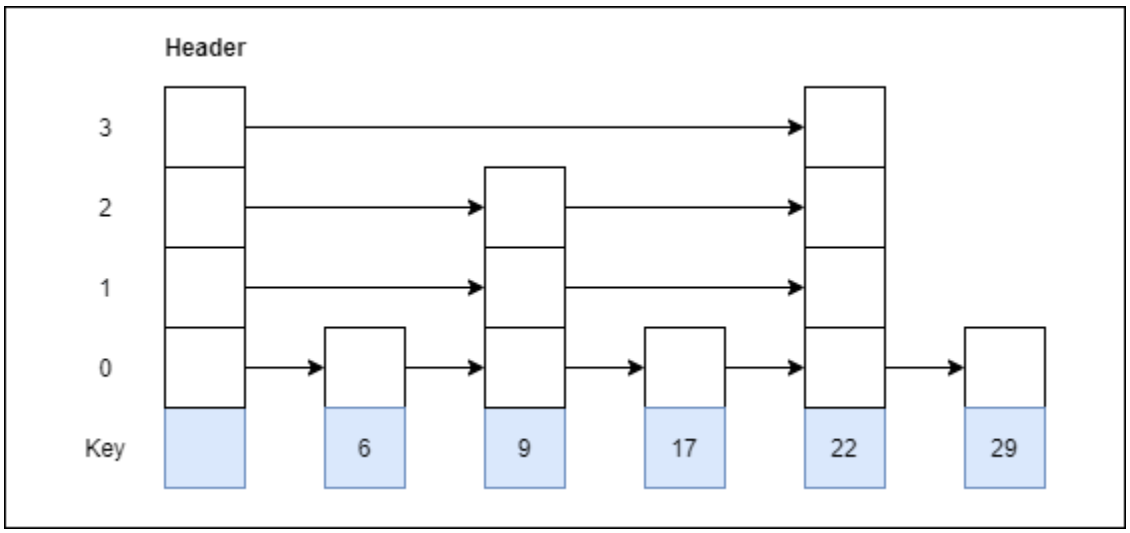
Step 3: Insert 22 with level 4



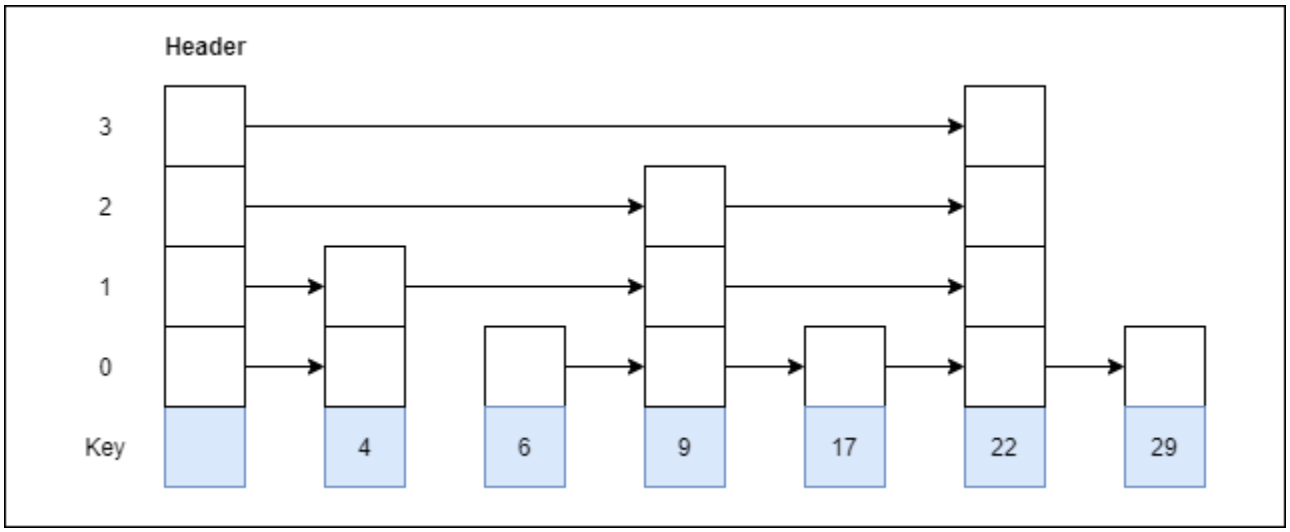
Step 4: Insert 9 with level 3



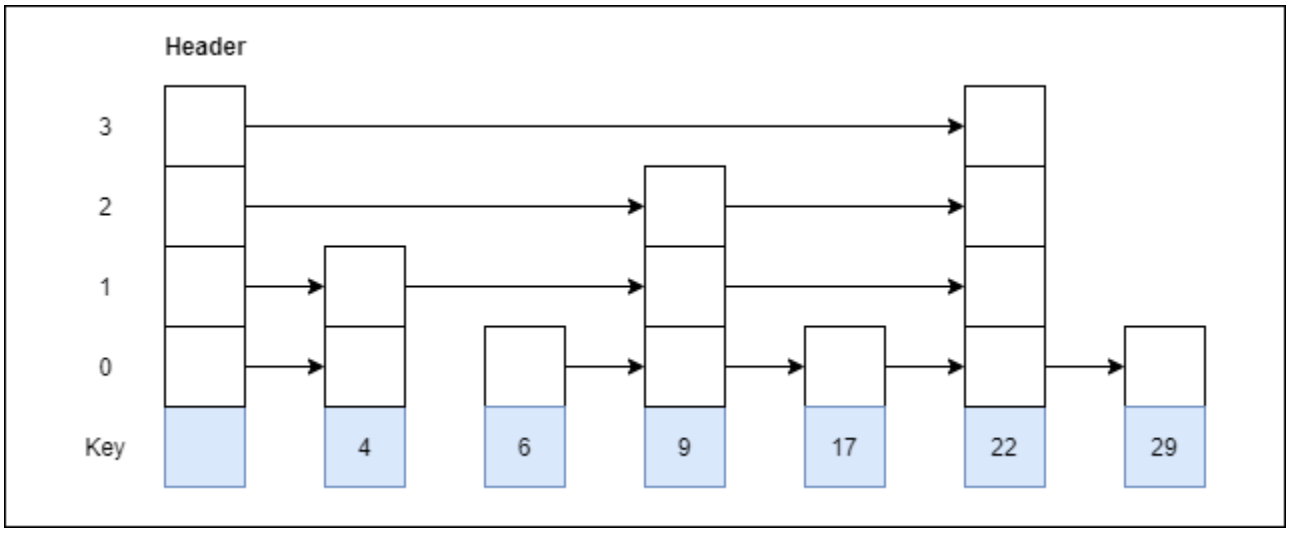
Step 5: Insert 17 with level 1



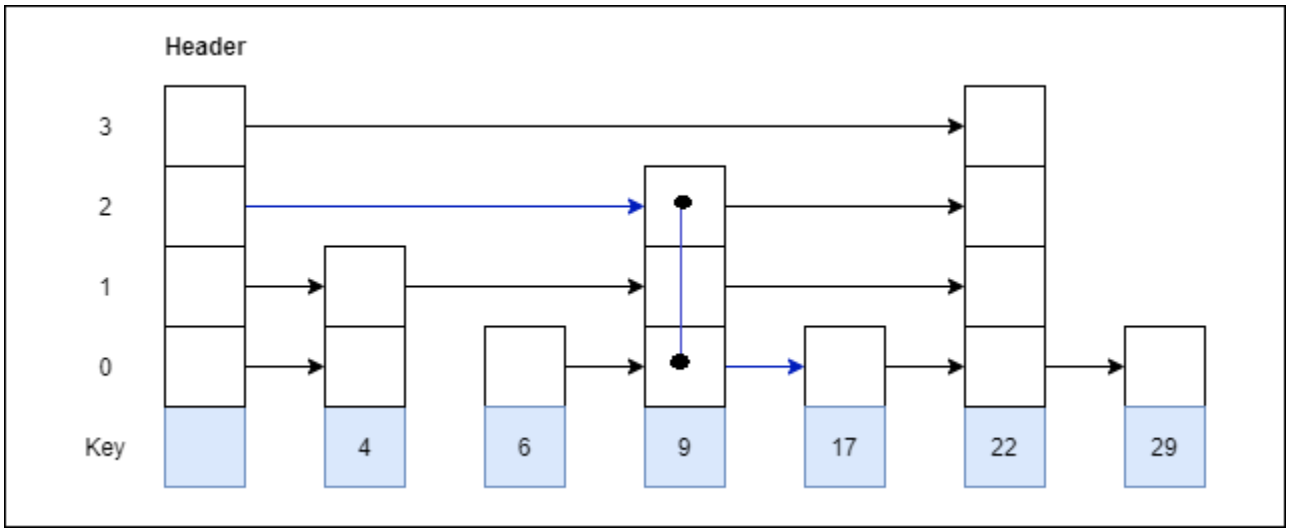
Step 6: Insert 4 with level 2



Example 2: Consider this example where we want to search for key 17.



Ans:

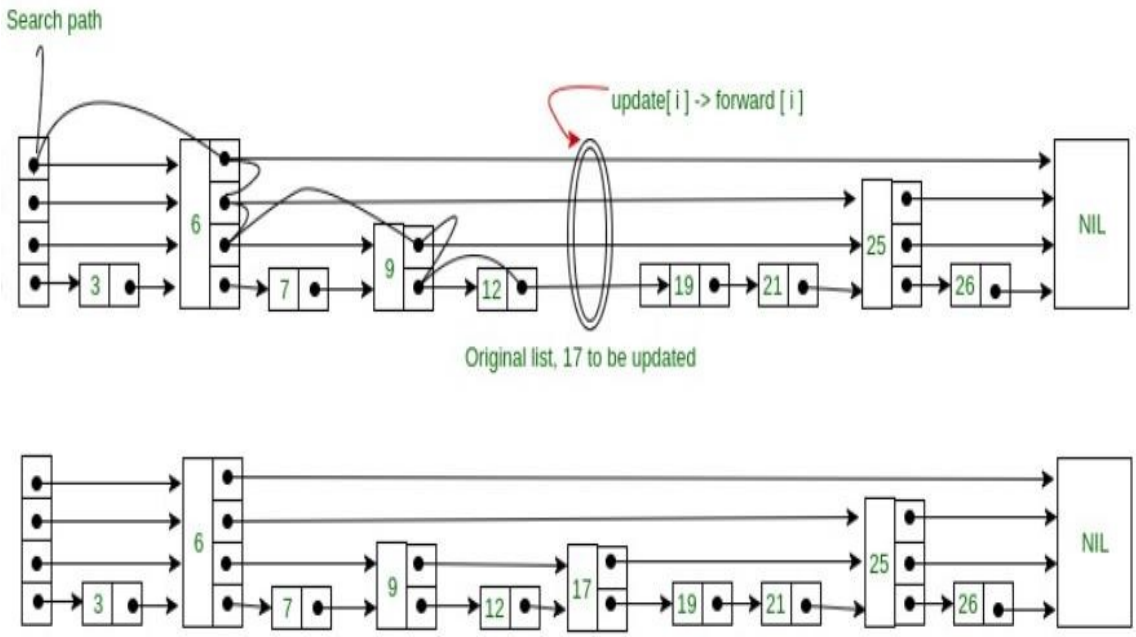


INSERTION IN SKIP LIST

We will start from highest level in the list and compare key of next node of the current node with the key to be inserted. Basic idea is If –

1. Key of next node is less than key to be inserted then we keep on moving forward on the same level
2. Key of next node is greater than the key to be inserted then we store the pointer to current node and move one level down and continue our search.

At the level 0, we will definitely find a position to insert given key.
Consider this example where we want to insert key 17 –



SEARCH IN SKIP LIST

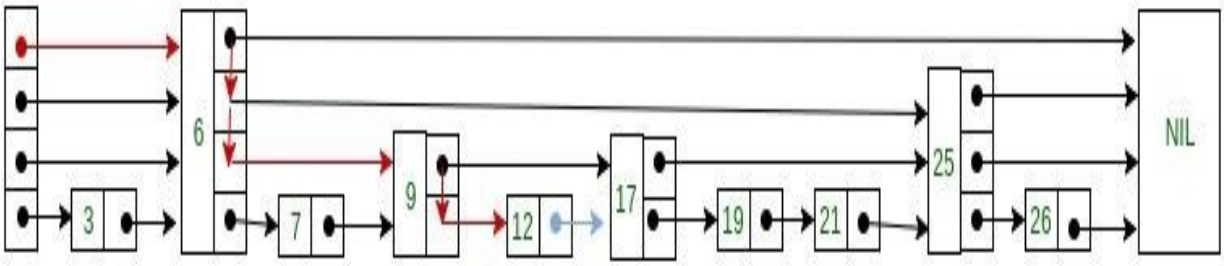
Searching an element is very similar to approach for searching a spot for inserting an element in Skip list. The basic idea is if –

1. Key of next node is less than search key then we keep on moving forward on the same level.
2. Key of next node is greater than the key to be inserted then we store the pointer to current node and move one level down and continue our search.

At the lowest level (0), if the element next to the rightmost element has key equal to the search key, then we have found key otherwise failure.

EXAMPLE:

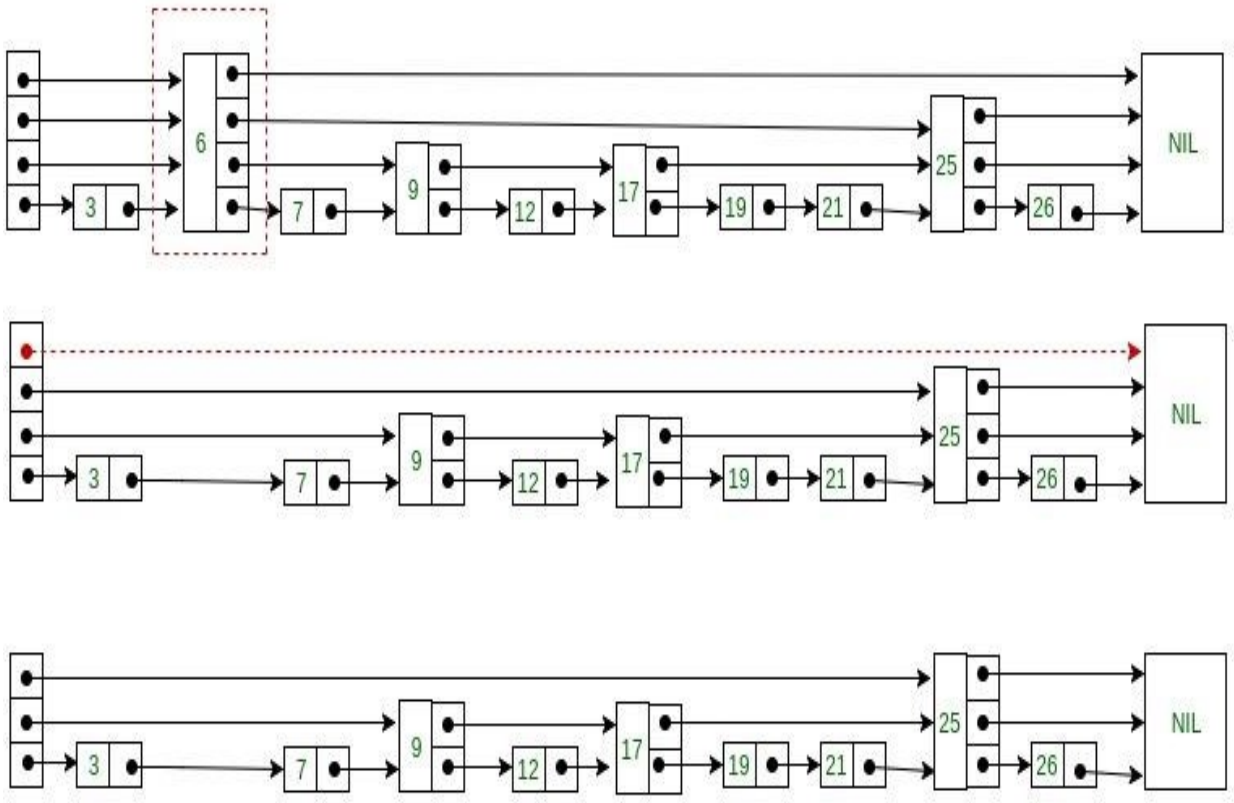
Consider this example where we want to search for key 17-



Deleting an element from the Skip list

Deletion of an element k is preceded by locating element in the Skip list using above mentioned search algorithm. Once the element is located, rearrangement of pointers is done to remove element form list just like we do in singly linked list. We start from lowest level and do rearrangement until element next is not k.
After deletion of element there could be levels with no elements, so we will remove these levels as well by decrementing the level of Skip list.

Consider this example where we want to delete element 6 –



Here at level 3, there is no element (arrow in red) after deleting element 6. So we will decrement level of skip list by 1.

Advantages of the Skip list

1. If you want to insert a new node in the skip list, then it will insert the node very fast because there are no rotations in the skip list.
2. The skip list is simple to implement as compared to the hash table and the binary search tree.
3. It is very simple to find a node in the list because it stores the nodes in sorted form.
4. The skip list algorithm can be modified very easily in a more specific structure, such as indexable skip lists, trees, or priority queues.
5. The skip list is a robust and reliable list.

Disadvantages of the Skip list

1. It requires more memory than the balanced tree.
2. Reverse searching is not allowed.
3. The skip list searches the node much slower than the linked list.

Applications of the Skip list

1. It is used in distributed applications, and it represents the pointers and system in the distributed applications.
2. It is used to implement a dynamic elastic concurrent queue with low lock contention.
3. The indexing of the skip list is used in running median problems.
4. The skip list is used for the delta-encoding posting in the Lucene search.

UNIT - II

Hash Table Representation: hash functions, collision resolution-separate chaining, open addressing linear probing, quadratic probing, double hashing, rehashing, extendible hashing.

HASHING

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

Drawbacks

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.

Advantages

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.

Hashing is a technique that is used to uniquely identify a specific object from a group of similar objects. Some examples of how hashing is used in our lives include:

- In universities, each student is assigned a unique roll number that can be used to retrieve information about them.
- In libraries, each book is assigned a unique number that can be used to determine information about the book, such as its exact position in the library or the users it has been issued to etc.

In both these examples the students and books were hashed to a unique number.

Assume that you have an object and you want to assign a key to it to make searching easy. To store the key/value pair, you can use a simple array like a data structure where keys (integers) can be used directly as an index to store values. However, in cases where the keys are large and cannot be used directly as an index, you should use *hashing*.

In hashing, large keys are converted into small keys by using **hash functions**. The values are then stored in a data structure called **hash table**. The idea of hashing is to distribute entries (key/value pairs) uniformly across an array. Each element is assigned a key (converted key). By using that key you can access the element in **O(1)** time. Using the key, the algorithm (hash function) computes an index that suggests where an entry can be found or inserted.

Hash function

A hash function is any function that can be used to map a data set of an arbitrary size to a data set of a fixed size, which falls into the hash table. The values returned by a hash function are called hash values, hash codes, hash sums, or simply hashes.

To achieve a good hashing mechanism, It is important to have a good hash function with the following basic requirements:

1. Easy to compute: It should be easy to compute and must not become an algorithm in itself.
2. Uniform distribution: It should provide a uniform distribution across the hash table and should not result in clustering.
3. Less collisions: Collisions occur when pairs of elements are mapped to the same hash value. These should be avoided.

Note: Irrespective of how good a hash function is, collisions are bound to occur. Therefore, to maintain the performance of a hash table, it is important to manage collisions through various collision resolution techniques.

Need for a good hash function

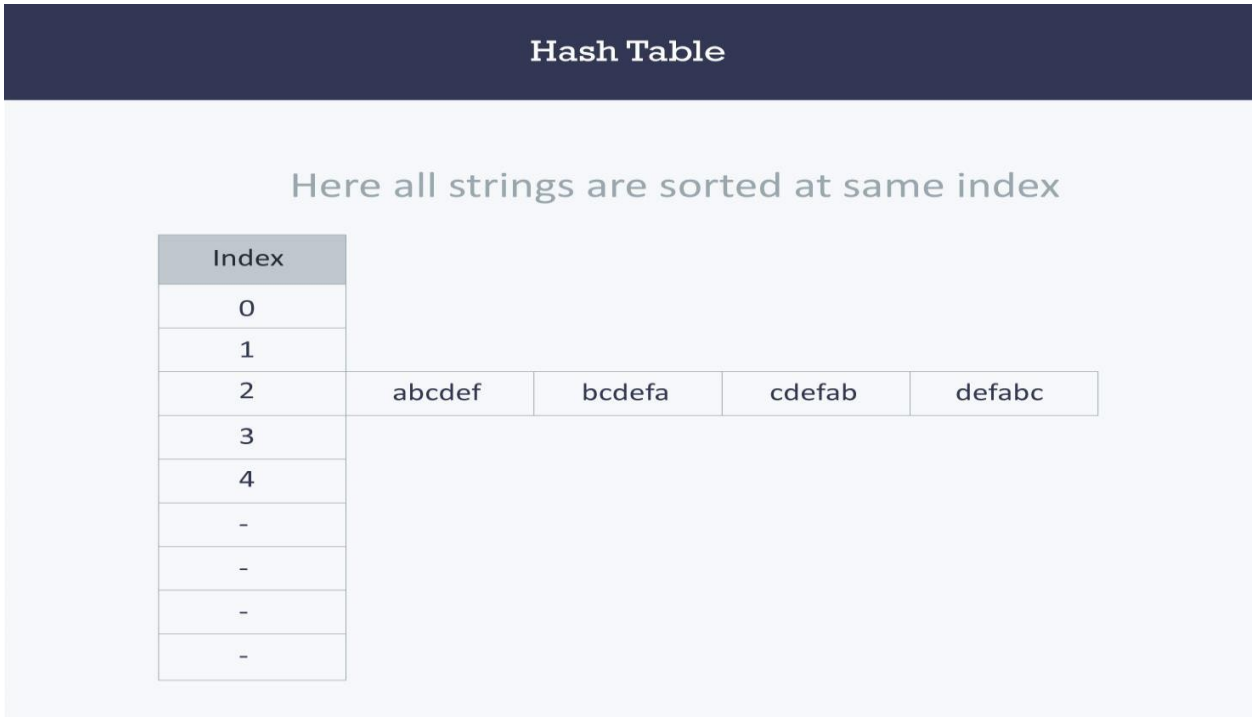
Let us understand the need for a good hash function. Assume that you have to store strings in the hash table by using the hashing technique {“abcdef”, “bcdefa”, “cdefab” , “defabc” }.

To compute the index for storing the strings, use a hash function that states the following:

The index for a specific string will be equal to the sum of the ASCII values of the characters modulo 599.

As 599 is a prime number, it will reduce the possibility of indexing different strings (collisions). It is recommended that you use prime numbers in case of modulo. The ASCII values of a, b, c, d, e, and f are 97, 98, 99, 100, 101, and 102 respectively. Since all the strings contain the same characters with different permutations, the sum will 599.

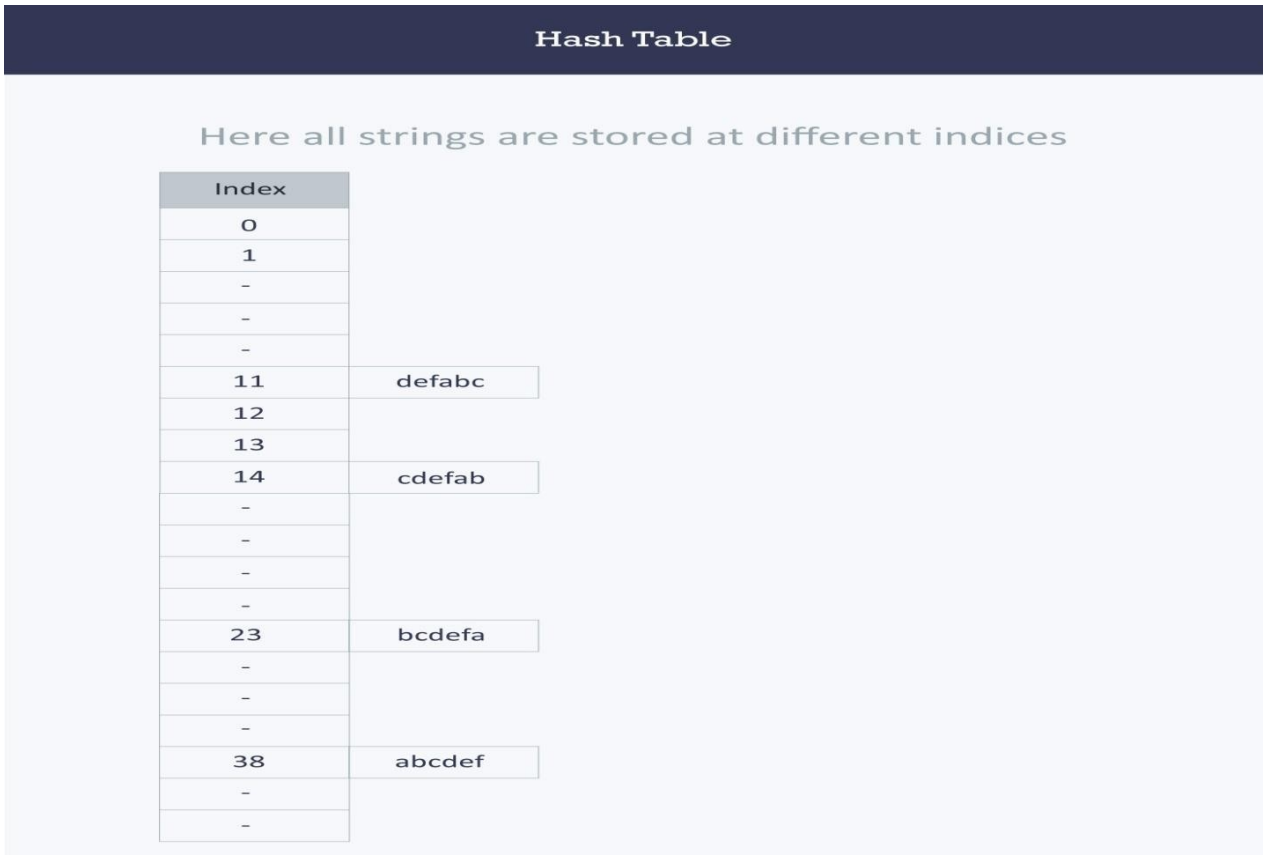
The hash function will compute the same index for all the strings and the strings will be stored in the hash table in the following format. As the index of all the strings is the same, you can create a list on that index and insert all the strings in that list.



Let’s try a different hash function. The index for a specific string will be equal to sum of ASCII values of characters multiplied by their respective order in the string after which it is modulo with 2069 (prime number).

String	Hash function	Index
abcdef	(97*1* + *98*2 + 99*3* + *100*4 + 101*5* + *102*6)%2069	38
bcdefa	(98*1* + *99*2 + 100*3* + *101*4 + 102*5* + *97*6)%2069	23

cdefab (99*1* + *100*2 + 1013 + *102*4 + 97*5* + *98*6)%2069 14
defabc (100*1* + *101*2 + 1023 + *97*4 + 98*5* + *99*6)%206911



Hash table

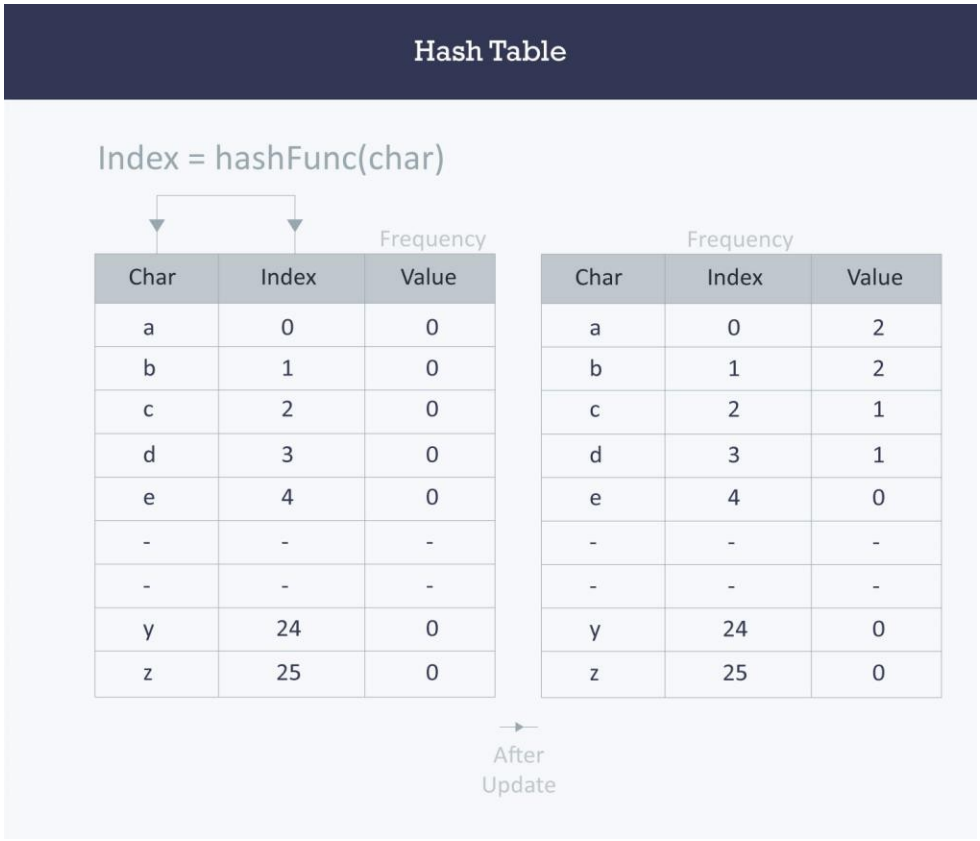
A hash table is a data structure that is used to store keys/value pairs. It uses a hash function to compute an index into an array in which an element will be inserted or searched. By using a good hash function, hashing can work well. Under reasonable assumptions, the average time required to search for an element in a hash table is **O(1)**.

Let us consider string S. You are required to count the frequency of all the characters in this string.

string S = “ababcd”

The simplest way to do this is to iterate over all the possible characters and count their frequency one by one. The time complexity of this approach is **O(26*N)** where **N** is the size of the string and there are 26 possible characters.

Let us apply hashing to this problem. Take an array frequency of size 26 and hash the 26 characters with indices of the array by using the hash function. Then, iterate over the string and increase the value in the frequency at the corresponding index for each character. The complexity of this approach is **O(N)** where **N** is the size of the string.



TYPES OF HASH FUNCTION

There are various types of hash functions that are used to place the record in the hash table-

1. Division Method:
2. Mid Square:
3. Digit Folding:

Division Method: The hash function depends upon the remainder of division.Typically the divisor is table length.
For eg; If the record 54, 72, 89, 37 is placed in the hash table and if the table size is 10 then

h(key) = record % table size

54%10=4
72%10=2
89%10=9
37%10=7

0	
1	
2	72
3	
4	54
5	
6	
7	37
8	
9	89

Mid Square:

In the mid square method, the key is squared and the middle or mid part of the result is used as the index. If the key is a string, it has to be preprocessed to produce a number.
Consider that if we want to place a record 3111 then

$3111^2 = 9678321$
for the hash table of size
1000 H(3111) = 783 (the
middle 3 digits)

Digit Folding:
The key is divided into separate parts and using some simple operation these parts arecombined to produce the hash key.
For eg; consider a record 12365412 then it is divided into separate parts as 123 654 12 and theseare added together
H(key) = 123+654+12= 789
The record will be placed at location 789

COLLISION

The hash function is a function that returns the key value using which the record can be placed in the hash table. Thus this function helps us in placing the record in the hash table at appropriate position and due to this we can retrieve the record directly from that location. This function need to be designed very carefully and it should not return the same hash key address for two different records. This is an undesirable situation in hashing.

> **Definition:** The situation in which the hash function returns the same hash key (home bucket) formore than one record is called collision and two same hash keys returned for different records is called synonym.

Similarly when there is no room for a new pair in the hash table then such a situation is called overflow. Sometimes when we handle collision it may lead to overflow conditions. Collision and overflow show the poor hash functions

EXAMPLE

For example,

Consider a hash function.

H(key) = recordkey%10 having the hash table size of 10

The record keys to be placed are

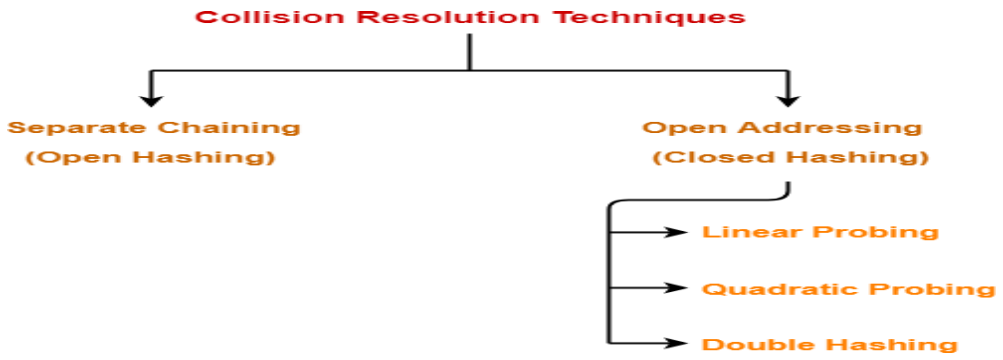
131, 44, 43, 78, 19, 36, 57 and 77
131%10=1
44%10=4
43%10=3
78%10=8
19%10=9
36%10=6
57%10=7
77%10=7

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

if we try to place 77 in the hash table then we get the hash key to be 7 and at index 7 already the record key 57 is placed. This situation is called **collision**. From the index 7 if we look for next vacant position at subsequent indices 8.9 then we find that there is no room to place 77 in the hash table. This situation is called **overflow**.

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

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**.

For Searching-

- In worst case, all the keys might map to the same bucket of the hash table.
- In such a case, all the keys will be present in a single linked list.
- Sequential search will have to be performed on the linked list to perform the search.
- So, time taken for searching in worst case is O(n).

For Deletion-

- In worst case, the key might have to be searched first and then deleted.
- In worst case, time taken for searching is O(n).
- So, time taken for deletion in worst case is O(n).

Load Factor (α)-

Load factor (α) is defined as-

$$\text{Load Factor } (\alpha) = \frac{\text{Number of elements present in the hash table}}{\text{Total size of the hash table}}$$

If Load factor (α) = constant, then time complexity of Insert, Search, Delete = Θ(1)

EXAMPLE:

Using the hash function 'key mod 7', insert the following sequence of keys in the hash table-

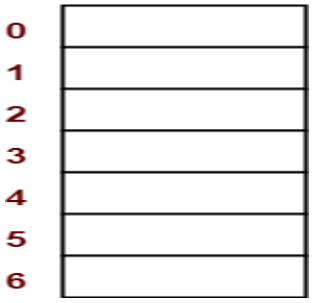
50, 700, 76, 85, 92, 73 and 101

Use separate chaining technique for collision resolution.

The given sequence of keys will be inserted in the hash table as-

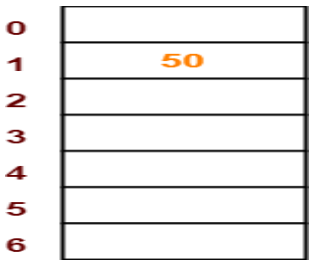
Step-01:

- Draw an empty hash table.
- For the given hash function, the possible range of hash values is [0, 6].
- So, draw an empty hash table consisting of 7 buckets as-



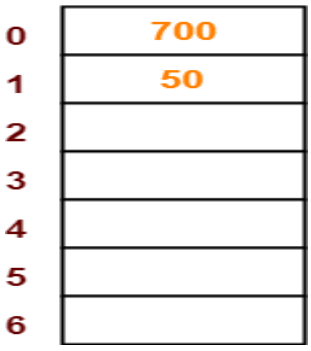
Step-02:

- Insert the given keys in the hash table one by one.
- The first key to be inserted in the hash table = 50.
- Bucket of the hash table to which key 50 maps = 50 mod 7 = 1.
- So, key 50 will be inserted in bucket-1 of the hash table as-



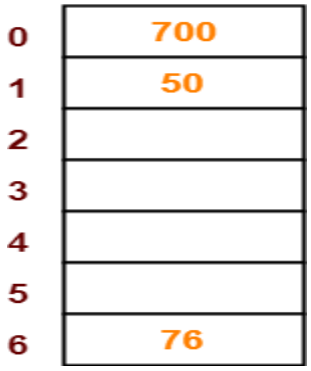
Step-03:

- The next key to be inserted in the hash table = 700.
- Bucket of the hash table to which key 700 maps = 700 mod 7 = 0.
- So, key 700 will be inserted in bucket-0 of the hash table as-



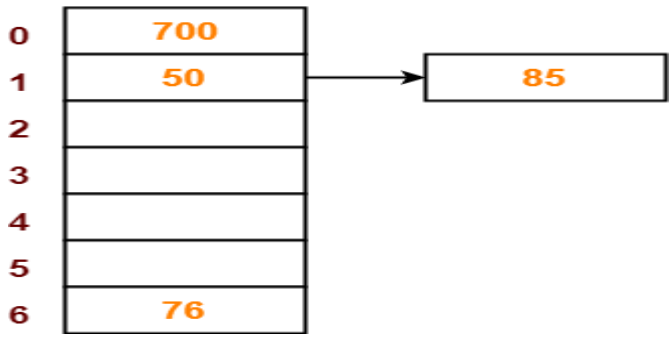
Step-04:

- The next key to be inserted in the hash table = 76.
- Bucket of the hash table to which key 76 maps = 76 mod 7 = 6.
- So, key 76 will be inserted in bucket-6 of the hash table as-



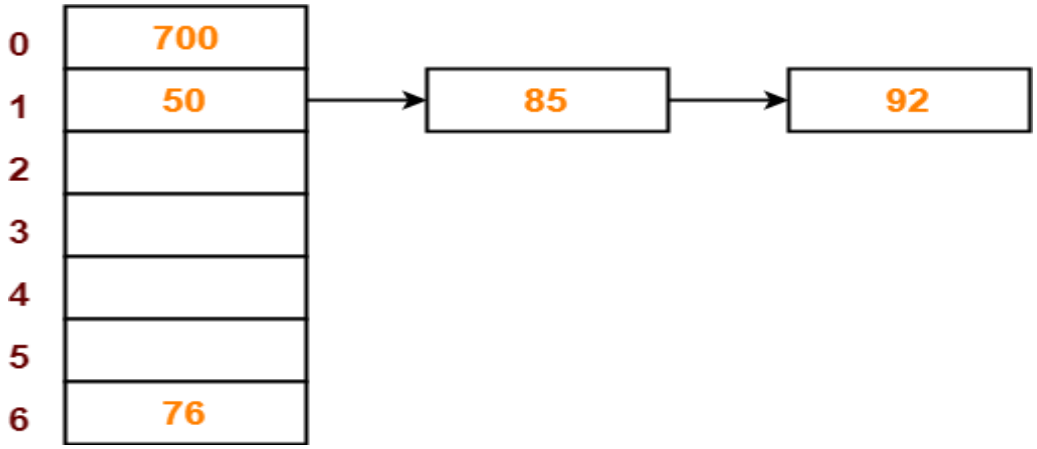
Step-05:

- The next key to be inserted in the hash table = 85.
- Bucket of the hash table to which key 85 maps = 85 mod 7 = 1.
- Since bucket-1 is already occupied, so collision occurs.
- Separate chaining handles the collision by creating a linked list to bucket-1.
- So, key 85 will be inserted in bucket-1 of the hash table as-



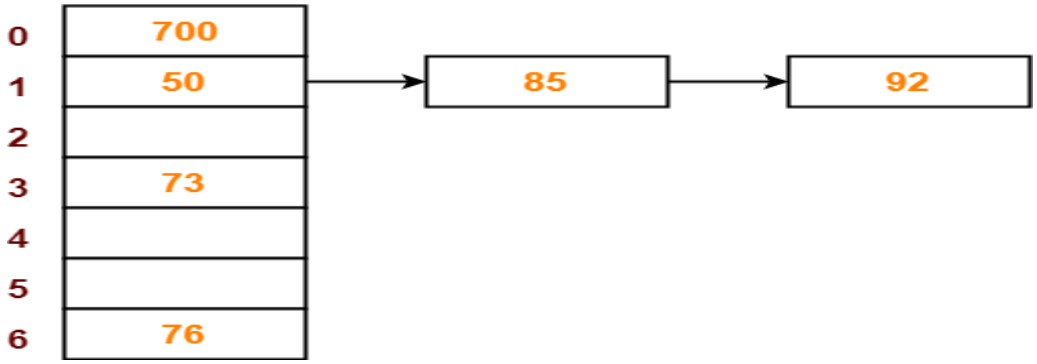
Step-06:

- The next key to be inserted in the hash table = 92.
- Bucket of the hash table to which key 92 maps = 92 mod 7 = 1.
- Since bucket-1 is already occupied, so collision occurs.
- Separate chaining handles the collision by creating a linked list to bucket-1.
- So, key 92 will be inserted in bucket-1 of the hash table as-



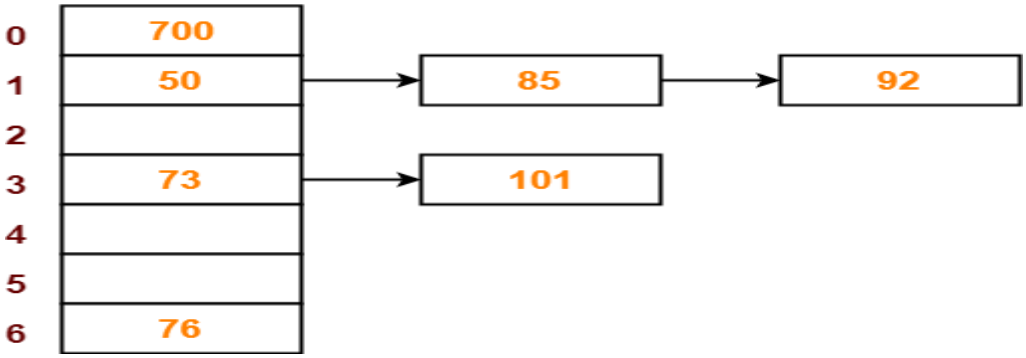
Step-07:

- The next key to be inserted in the hash table = 73.
- Bucket of the hash table to which key 73 maps = 73 mod 7 = 3.
- So, key 73 will be inserted in bucket-3 of the hash table as-



Step-08:

- The next key to be inserted in the hash table = 101.
- Bucket of the hash table to which key 101 maps = 101 mod 7 = 3.
- Since bucket-3 is already occupied, so collision occurs.
- Separate chaining handles the collision by creating a linked list to bucket-3.
- So, key 101 will be inserted in bucket-3 of the hash table as-



The **advantages** of separate chaining hashing are as follows −

- Separate chaining technique is not sensitive to the size of the table.
- The idea and the implementation are simple.

The **disadvantages** of separate chaining hashing are as follows −

- Keys are not evenly distributed in separate chaining.
- Separate chaining can lead to empty spaces in the table.
- The list in the positions can be very long.

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-

- Linear Probing
- Quadratic Probing
- Double Hashing

Operations in Open Addressing-

Let us discuss how operations are performed in open addressing-

Insert Operation-

- Hash function is used to compute the hash value for a key to be inserted.
- Hash value is then used as an index to store the key in the hash table.

In case of collision,

- Probing is performed until an empty bucket is found.
- Once an empty bucket is found, the key is inserted.
- Probing is performed in accordance with the technique used for open addressing.

Search Operation-

To search any particular key,

- Its hash value is obtained using the hash function used.
- Using the hash value, that bucket of the hash table is checked.

- If the required key is found, the key is searched.
- Otherwise, the subsequent buckets are checked until the required key or an empty bucket is found.
- The empty bucket indicates that the key is not present in the hash table.

Delete Operation-

- The key is first searched and then deleted.
- After deleting the key, that particular bucket is marked as "deleted".

NOTE-

- During insertion, the buckets marked as "deleted" are treated like any other empty bucket.
- During searching, the search is not terminated on encountering the bucket marked as "deleted".
- The search terminates only after the required key or an empty bucket is found.

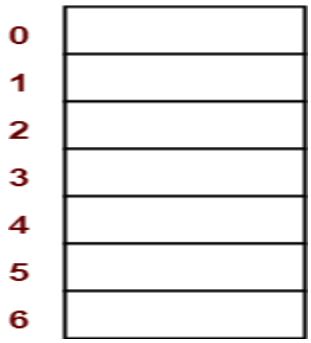
Linear Probing

Using the hash function 'key mod 7', insert the following sequence of keys in the hash table-
50, 700, 76, 85, 92, 73 and 101

Use linear probing technique for collision resolution.

Step-01:

- Draw an empty hash table.
- For the given hash function, the possible range of hash values is [0, 6].
- So, draw an empty hash table consisting of 7 buckets as-



Step-02:

- Insert the given keys in the hash table one by one.
- The first key to be inserted in the hash table = 50.
- Bucket of the hash table to which key 50 maps = 50 mod 7 = 1.
- So, key 50 will be inserted in bucket-1 of the hash table as-

0	
1	50
2	
3	
4	
5	
6	

Step-03:

- The next key to be inserted in the hash table = 700.
- Bucket of the hash table to which key 700 maps = 700 mod 7 = 0.
- So, key 700 will be inserted in bucket-0 of the hash table as-

0	700
1	50
2	
3	
4	
5	
6	

Step-04:

- The next key to be inserted in the hash table = 76.
- Bucket of the hash table to which key 76 maps = 76 mod 7 = 6.
- So, key 76 will be inserted in bucket-6 of the hash table as-

0	700
1	50
2	
3	
4	
5	
6	76

Step-05:

- The next key to be inserted in the hash table = 85.
- Bucket of the hash table to which key 85 maps = 85 mod 7 = 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, key 85 will be inserted in bucket-2 of the hash table as-

0	
1	50
2	
3	
4	
5	
6	

Step-06:

- The next key to be inserted in the hash table = 92.
- Bucket of the hash table to which key 92 maps = 92 mod 7 = 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, key 92 will be inserted in bucket-3 of the hash table as-

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

Step-07:

- The next key to be inserted in the hash table = 73.
- Bucket of the hash table to which key 73 maps = 73 mod 7 = 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, key 73 will be inserted in bucket-4 of the hash table as-

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

Step-08:

- The next key to be inserted in the hash table = 101.
- Bucket of the hash table to which key 101 maps = 101 mod 7 = 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, key 101 will be inserted in bucket-5 of the hash table as-

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

The **advantages** of linear probing are as follows −

- Linear probing requires very less memory.
- It is less complex and is simpler to implement.

The **disadvantages** of linear probing are as follows −

- Linear probing causes a scenario called "primary clustering" in which there are large blocks of occupied cells within the hash table.
- The values in linear probing tend to cluster which makes the probe sequence longer and lengthier.
-

QUADRATIC PROBING:

Quadratic probing operates by taking the original hash value and adding successive values of an arbitrary quadratic polynomial to the starting value. This method uses following formula.

$$H(key) = (Hash(key) + i^2) \% m)$$

where m can be table size or any prime number.

for eg; If we have to insert following elements in the hash table with table size 10:

37, 90, 55, 22, 17, 49, 87

37 % 10 = 7
90 % 10 = 0
55 % 10 = 5
22 % 10 = 2
11 % 10 = 1

0	90
1	11
2	22
3	
4	
5	55
6	
7	37
8	
9	

Now if we want to place 17 a collision will occur as 17%10 = 7 and bucket 7 has already an element 37. Hence we will apply quadratic probing to insert this record in the hash table.
Hi (key) = (Hash(key) + i^2) % m
Consider i = 0 then
(17 + 0^2) % 10 = 7
(17 + 1^2) % 10 = 8, when i =1

The bucket 8 is empty hence we will place the element at index 8.
Then comes 49 which will be placed at index 9.

49 % 10 = 9

0	90
1	11
2	22
3	
4	
5	55
6	
7	37
8	49
9	

Now to place 87 we will use quadratic probing.

$(87 + 0) \% 10 = 7$
$(87 + 1) \% 10 = 8$... but already occupied
$(87 + 2^2) \% 10 = 1$.. already occupied
$(87 + 3^2) \% 10 = 6$

It is observed that if we want place all the necessary elements in the hash table the size of divisor (m) should be twice as large as total number of elements.

0	90
1	11
2	22
3	
4	
5	55
6	87
7	37
8	49
9	

The **advantages** of quadratic probing is as follows −

- Quadratic probing is less likely to have the problem of primary clustering and is easier to implement than Double Hashing.

The **disadvantages** of quadratic probing are as follows −

- Quadratic probing has secondary clustering. This occurs when 2 keys hash to the same location, they have the same probe sequence. So, it may take many attempts before an insertion is being made.
- Also probe sequences do not probe all locations in the table.

Double hashing

Double Hashing is a hashing collision resolution technique where we use 2 hash functions.
Double Hashing - Hash Function 1
h_i = (Hash(X) + F(i)) % Table Size
where

- F(i) = i * $hash_2$(X)
- X is the Key or the Number for which the hashing is done
- i is the i^{th} time that hashing is done for the same value. Hashing is repeated only when collision occurs
- Table size is the size of the table in which hashing is done

This F(i) will generate the sequence such as $hash_2$(X), 2 * $hash_2$(X) and so on.
Double Hashing - Hash Function 2
We use second hash function as
$hash_2$(X) = R - (X mod R)
where

- R is the prime number which is slightly smaller than the Table Size.

- X is the Key or the Number for which the hashing is done

Double Hashing Example - Closed Hash Table

Let us consider the same example in which we choose R = 7.

0	68
1	
2	39
3	89
4	
5	
6	
7	
8	28
9	79

A Closed Hash Table using Double Hashing

Key	Hash Function h(X)	Index	Collision	Alt Index
79	$h_0(79)$ = (Hash(79) + F(0)) % 10 = ((79 % 10) + 0) % 10 = 9	9		
28	$h_0(28)$ = (Hash(28) + F(0)) % 10 = ((28 % 10) + 0) % 10 = 8	8		
39	$h_0(39)$ = (Hash(39) + F(0)) % 10 = ((39 % 10) + 0) % 10 = 9	9	first collision occurs	
	$h_1(39)$ = (Hash(39) + F(1)) % 10 = ((39 % 10) + 1(7-(39 % 7))) % 10 = (9 + 3) % 10 =12 % 10 = 2	2		2
68	$h_0(68)$ = (Hash(68) + F(0)) % 10 = ((68 % 10) + 0) % 10 = 8	8	collision occurs	
	$h_1(68)$ = (Hash(68) + F(1)) % 10 = ((68 % 10) + 1(7-(68 % 7))) % 10 = (8 + 2) % 10 =10 % 10 =0	0		0

89	$h_0(89)$ = (Hash(89) + F(0)) % 10 = ((89 % 10) + 0) % 10 = 9	9	collision occurs	
	$h_1(89)$ = (Hash(89) + F(1)) % 10 = ((89 % 10) + 1(7-(89 % 7))) % 10 = (9 + 2) % 10 =10 % 10 = 0	0	Again collision occurs	
	$h_2(89)$ = (Hash(89) + F(2)) % 10 = ((89 % 10) + 2(7-(89 % 7))) % 10 = (9 + 4) % 10=13 % 10 =3	3		3

he **advantage** of double hashing is as follows −

- Double hashing finally overcomes the problems of the clustering issue.

The **disadvantages** of double hashing are as follows:

- Double hashing is more difficult to implement than any other.

Comparison of Open Addressing Techniques-

	Linear Probing	Quadratic Probing	Double Hashing
Primary Clustering	Yes	No	No
Secondary Clustering	Yes	Yes	No
Number of Probe Sequence (m = size of table)	m	m	m^2
Cache performance	Best	Lies between the two	Poor

Separate Chaining Vs Open Addressing-

Separate Chaining	Open Addressing
Keys are stored inside the hash table as well as outside the hash table.	All the keys are stored only inside the hash table. No key is present outside the hash table.
The number of keys to be stored in the hash table can even exceed the size of the hash table.	The number of keys to be stored in the hash table can never exceed the size of the hash table.
Deletion is easier.	Deletion is difficult.
Extra space is required for the pointers to store the keys outside the hash table.	No extra space is required.
Cache performance is poor. This is because of linked lists which store the keys outside the hash table.	Cache performance is better. This is because here no linked lists are used.
Some buckets of the hash table are never used which leads to wastage of space.	Buckets may be used even if no key maps to those particular buckets.

Which is the Preferred Technique?

The performance of both the techniques depend on the kind of operations that are required to be performed on the keys stored in the hash table-

Separate Chaining-

Separate Chaining is advantageous when it is required to perform all the following operations on the keys stored in the hash table-

- Insertion Operation
- Deletion Operation
- Searching Operation

NOTE-

- Deletion is easier in separate chaining.
- This is because deleting a key from the hash table does not affect the other keys stored in the hash table.

Open Addressing-

Open addressing is advantageous when it is required to perform only the following operations on the keys stored in the hash table-

- Insertion Operation
- Searching Operation

NOTE-

- Deletion is difficult in open addressing.
- This is because deleting a key from the hash table requires some extra efforts.
- After deleting a key, certain keys have to be rearranged.

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 recomputing their positions using hash functions.

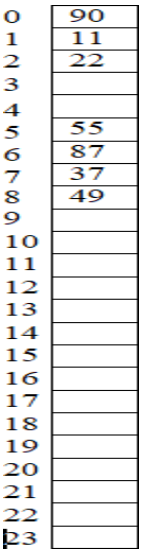
Consider we have to insert the elements 37, 90, 55, 22, 17, 49, and 87. the table size is 10 and will use hash function.,
H(key) = key mod tablesize
37 % 10 = 7
90 % 10= 0
55 % 10 = 5
22 % 10 = 2
17 % 10 = 7
Collision solved by linear probing
49 % 10 = 9

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. And new hash function will be

H(key) key mod 23

37 % 23 = 14
90 % 23 = 21
55 % 23 = 9
22 % 23 = 22
17 % 23 = 17
49 % 23 = 3
87 % 23 = 18



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

Advantages:

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

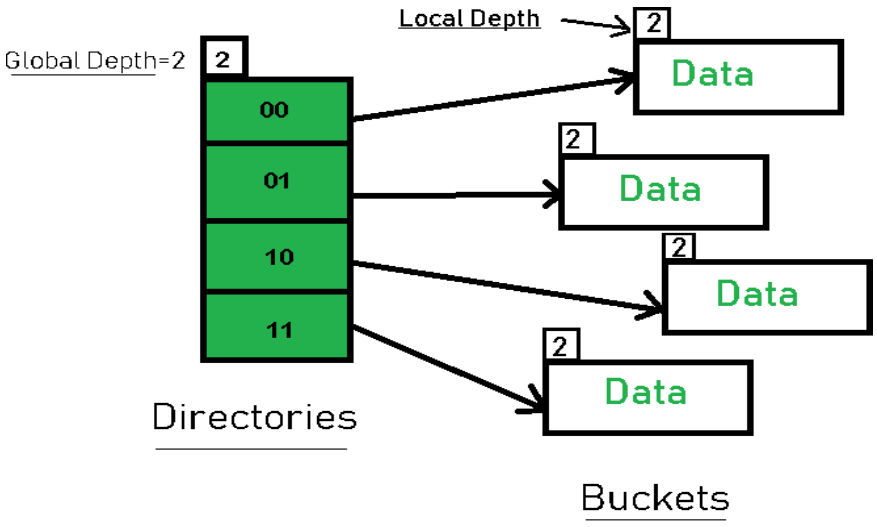
Extendible Hashing

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.

Basic Structure of Extendible Hashing:



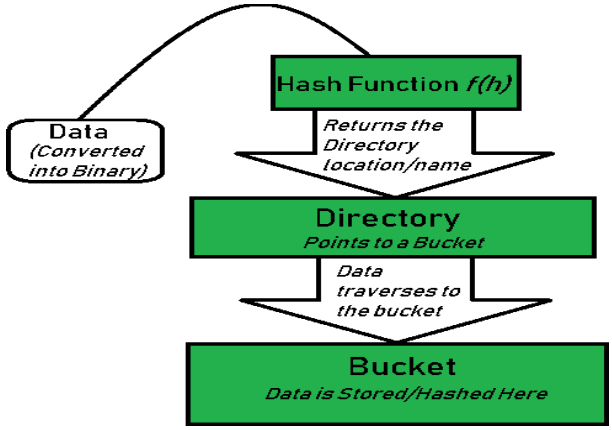
Extendible Hashing

Frequently used terms in Extendible Hashing:

- **Directories:** These containers store pointers to buckets. Each directory is given a unique id which may change each time when expansion takes place. The hash function returns this directory id which is used to navigate to the appropriate bucket. Number of Directories = 2^Global Depth.
- **Buckets:** They store the hashed keys. Directories point to buckets. A bucket may contain more than one pointers to it if its local depth is less than the global depth.
- **Global Depth:** It is associated with the Directories. They denote the number of bits which are used by the hash function to categorize the keys. Global Depth = Number of bits in directory id.

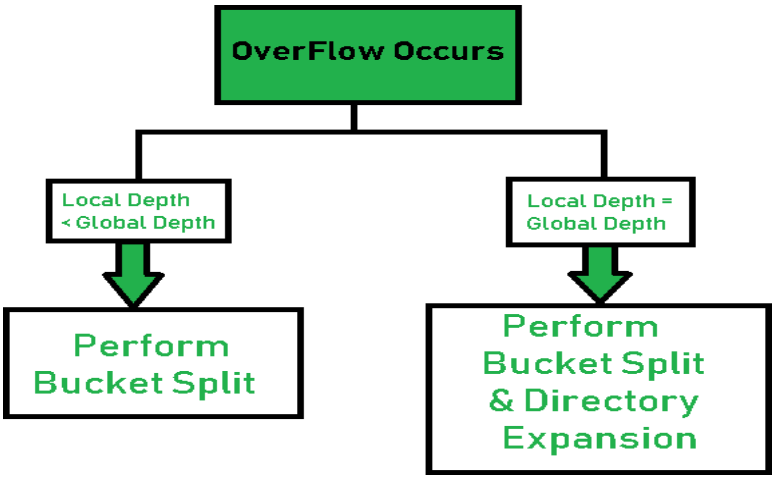
- **Local Depth:** It is the same as that of Global Depth except for the fact that Local Depth is associated with the buckets and not the directories. Local depth in accordance with the global depth is used to decide the action that to be performed in case an overflow occurs. Local Depth is always less than or equal to the Global Depth.
- **Bucket Splitting:** When the number of elements in a bucket exceeds a particular size, then the bucket is split into two parts.
- **Directory Expansion:** Directory Expansion Takes place when a bucket overflows. Directory Expansion is performed when the local depth of the overflowing bucket is equal to the global depth.

Basic Working of Extendible 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.
    - **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:** 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.

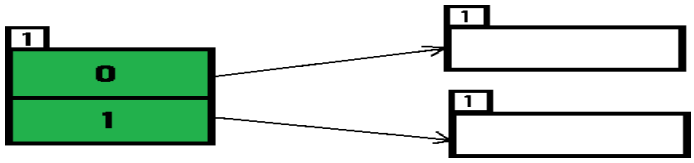


- **Step 8 – Rehashing of Split Bucket Elements:** The Elements present in the overflowing bucket that is split are rehashed w.r.t the new global depth of the directory.
- **Step 9 –** The element is successfully hashed.

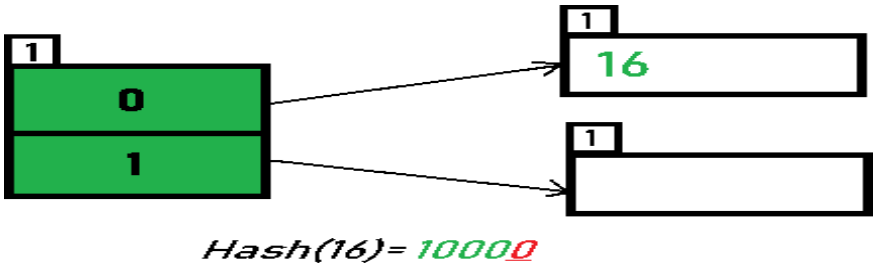
Example based on Extendible Hashing: Now, let us consider a prominent example of hashing the following elements: **16,4,6,22,24,10,31,7,9,20,26.**
Bucket Size: 3 (Assume)
Hash Function: Suppose the global depth is X. Then the Hash Function returns X LSBs.

- **Solution:** First, calculate the binary forms of each of the given numbers.
  16- 10000
  4- 00100
  6- 00110
  22- 10110
  24- 11000
  10- 01010
  31- 11111
  7- 00111
  9- 01001
  20- 10100
  26- 11010

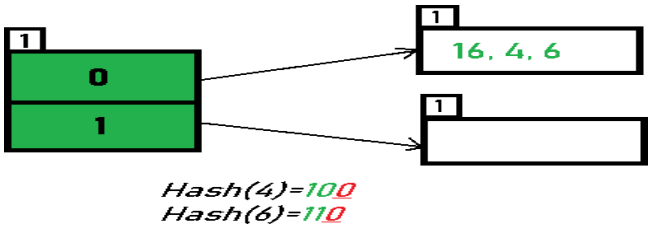
- Initially, the global-depth and local-depth is always 1. Thus, the hashing frame looks like this:



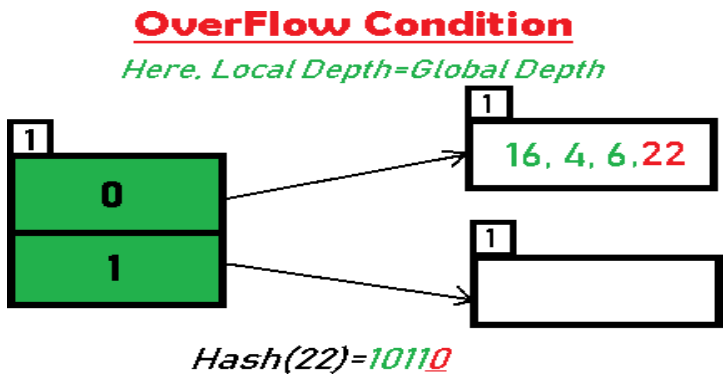
- **Inserting 16:**
  The binary format of 16 is 10000 and global-depth is 1. The hash function returns 1 LSB of 1000**0** which is 0. Hence, 16 is mapped to the directory with id=0.



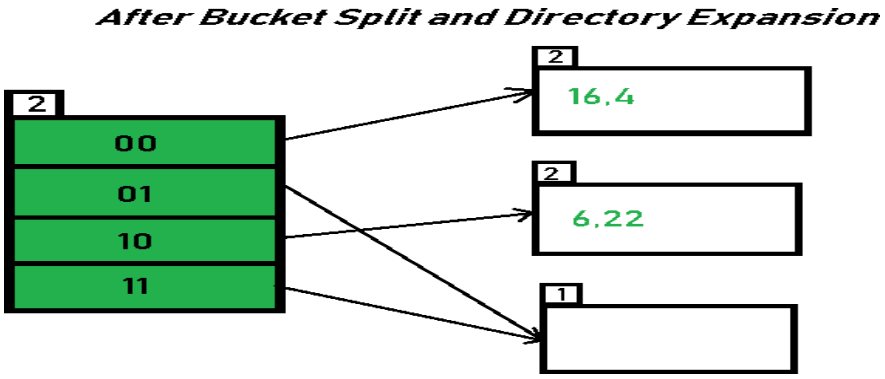
- **Inserting 4 and 6:**
  Both 4(10**0**) and 6(11**0**)have 0 in their LSB. Hence, they are hashed as follows:



- **Inserting 22:** The binary form of 22 is 1011**0**. Its LSB is 0. The bucket pointed by directory 0 is already full. Hence, Over Flow occurs.



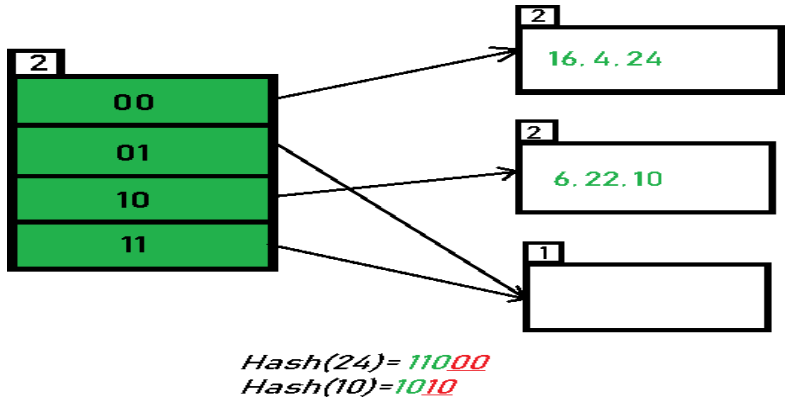
- As directed by **Step 7-Case 1**, Since Local Depth = Global Depth, the bucket splits and directory expansion takes place. Also, rehashing of numbers present in the overflowing bucket takes place after the split. And, since the global depth is incremented by 1, now,the global depth is 2. Hence, 16,4,6,22 are now rehashed w.r.t 2 LSBs.[16(100**00**),4(1**00**),6(1**10**),22(101**10**)]



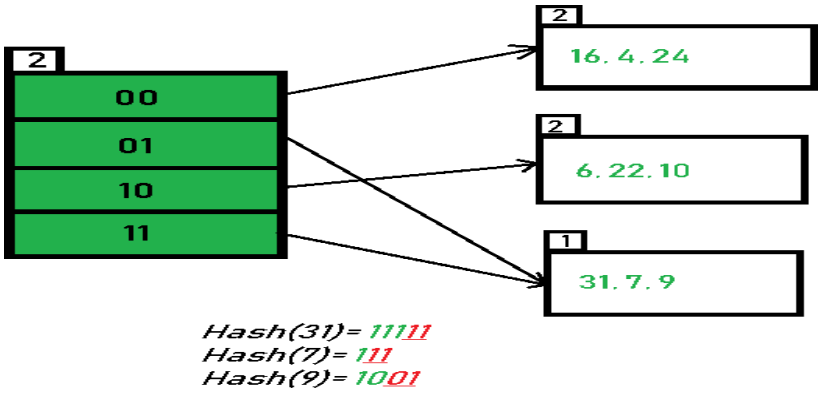
•

**Notice that the bucket which was underflow has remained untouched. But, since the number of directories has doubled, we now have 2 directories 01 and 11 pointing to the same bucket. This is because the local-depth of the bucket has remained 1. And, any bucket having a local depth less than the global depth is pointed-to by more than one directories.*

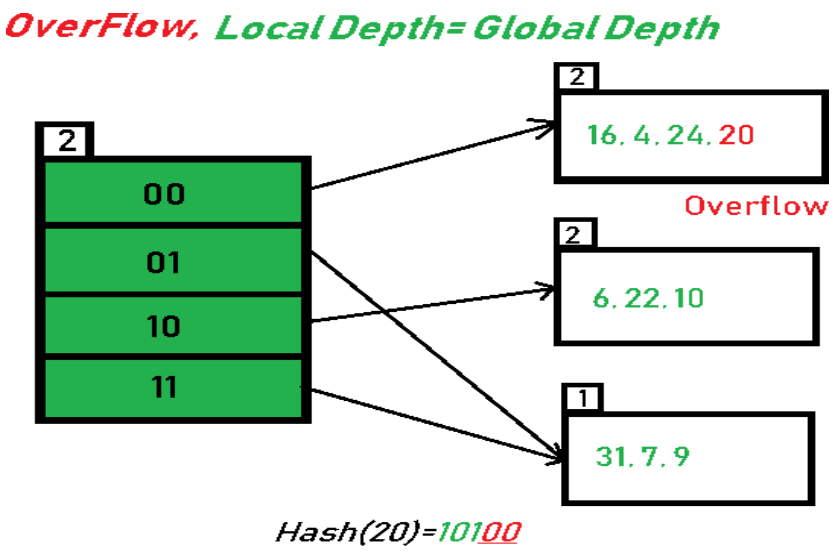
- **Inserting 24 and 10:** 24(110**00**) and 10 (10**10**) can be hashed based on directories with id 00 and 10. Here, we encounter no overflow condition.



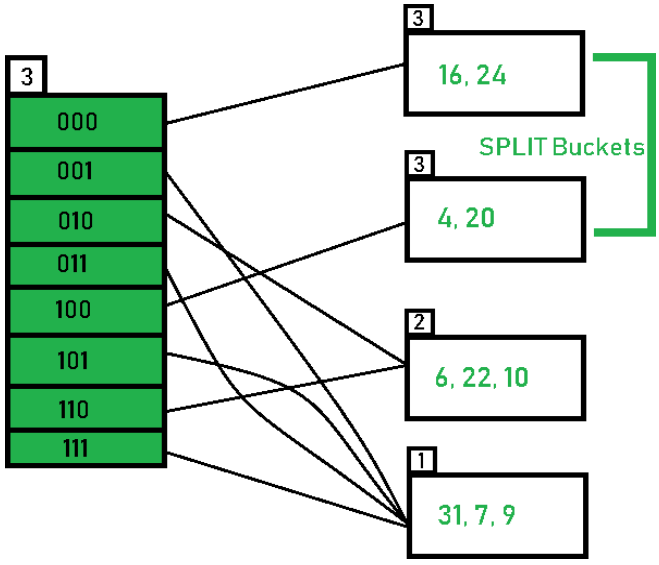
- **Inserting 31,7,9:** All of these elements[31(111**11**), 7(1**11**), 9(10**01**)] have either 01 or 11 in their LSBs. Hence, they are mapped on the bucket pointed out by 01 and 11. We do not encounter any overflow condition here.



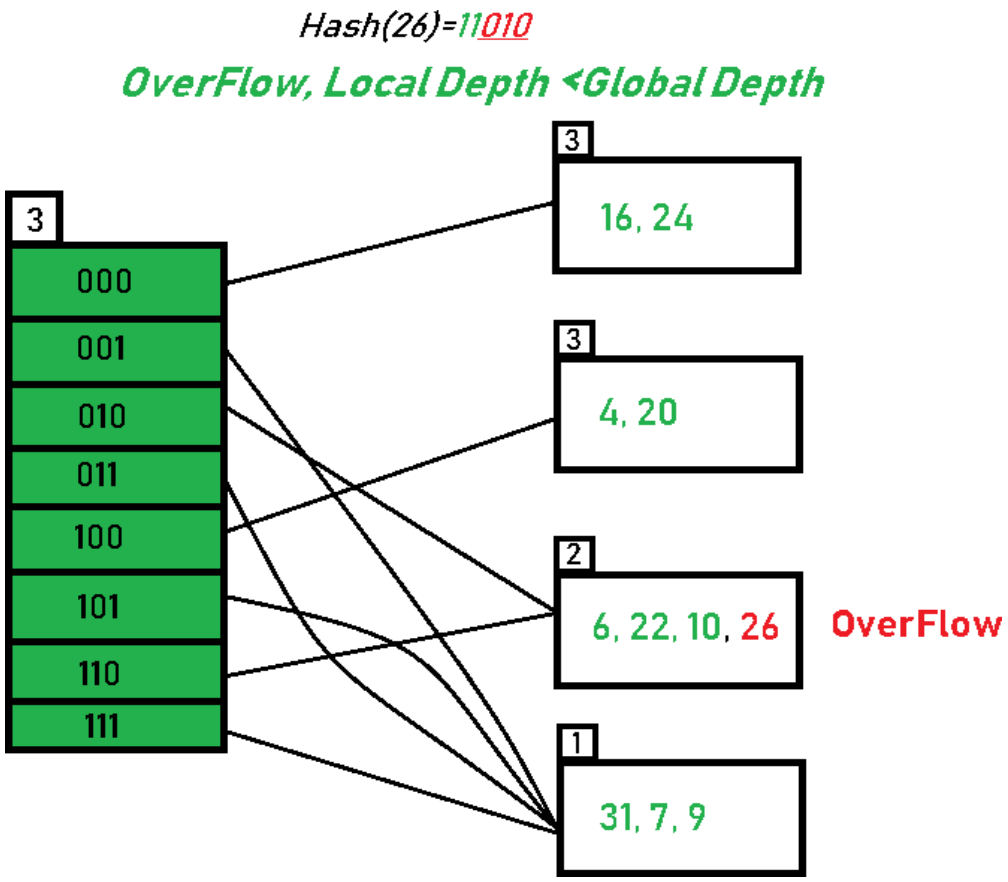
- **Inserting 20:** Insertion of data element 20 (101**00**) will again cause the overflow problem.



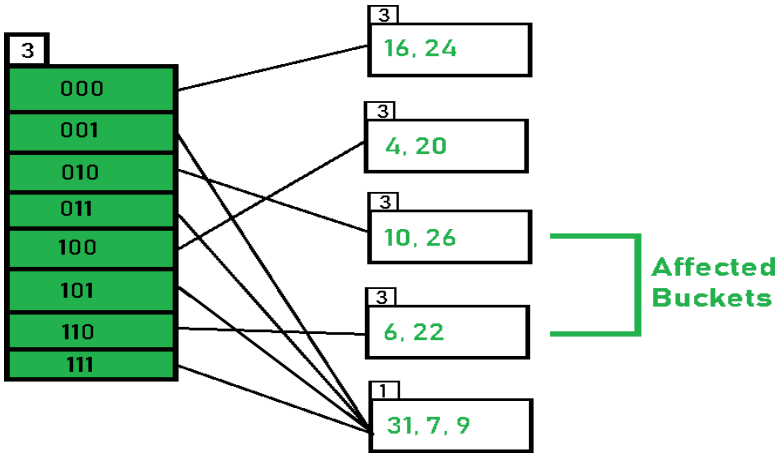
- 20 is inserted in bucket pointed out by 00. As directed by **Step 7-Case 1**, since the **local depth of the bucket = global-depth**, directory expansion (doubling) takes place along with bucket splitting. Elements present in overflowing bucket are rehashed with the new global depth. Now, the new Hash table looks like this:



- **Inserting 26:** Global depth is 3. Hence, 3 LSBs of 26(11**010**) are considered. Therefore 26 best fits in the bucket pointed out by directory 010.



- The bucket overflows, and, as directed by **Step 7-Case 2,** since the **local depth of bucket < Global depth (2<3)**, directories are not doubled but, only the bucket is split and elements are rehashed.



Finally, the output of hashing the given list of numbers is obtained.

Key Observations:

1. A Bucket will have more than one pointers pointing to it if its local depth is less than the global depth.
2. When overflow condition occurs in a bucket, all the entries in the bucket are rehashed with a new local depth.
3. If Local Depth of the overflowing bucket
4. The size of a bucket cannot be changed after the data insertion process begins.

Advantages:

1. Data retrieval is less expensive (in terms of computing).
2. No problem of Data-loss since the storage capacity increases dynamically.
3. With dynamic changes in hashing function, associated old values are rehashed w.r.t the new hash function.

Limitations Of Extendible Hashing:

1. The directory size may increase significantly if several records are hashed on the same directory while keeping the record distribution non-uniform.
2. Size of every bucket is fixed.
3. Memory is wasted in pointers when the global depth and local depth difference becomes drastic.
4. This method is complicated to code.