Dictionary

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

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

results = {'Anik' : 75, 'Aftab' :80, 'James' : 85, 'Manisha': 77, 'Suhana' :87, 'Margaret': 82}

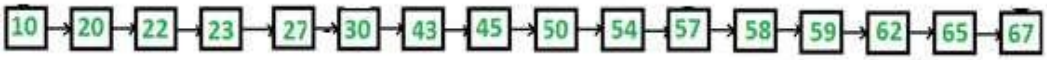
Main operations of dictionaries

- **retrieve a value:** based on language, attempting to retrieve a missing key may provide a default value or throw an exception
- **inserting or updating a value**: 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 or delete a key-value pair**
- **test or verify for existence of a key**

Types of Representation

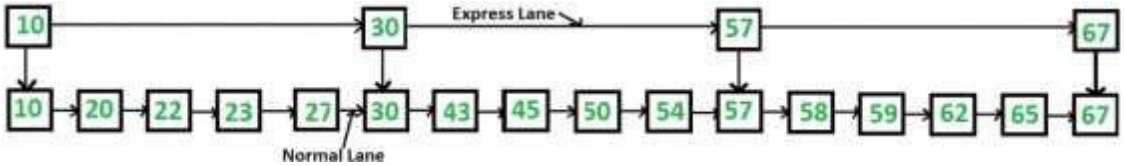
1. **Linear List**

   Sorted linked list as we can only linearly traverse the list and cannot skip nodes while searching.



2. **Skip List**

   The idea is simple, we create multiple layers so that we can skip some nodes. See the following example list with 16 nodes and two layers. The upper layer works as an "express lane" that connects only the main outer stations, and the lower layer works as a "normal lane" that connects every station. Suppose we want to search for 50, we start from the first node of the "express lane" and keep moving on the "express lane" till we find a node whose next is greater than 50. Once we find such a node (30 is the node in the following example) on "express lane", we move to "normal lane" using a pointer from this node, and linearly search for 50 on "normal lane". In the following example, we start from 30 on the "normal lane" and with linear search, we find 50.



Types of Skip List

- Perfect Skip List
- Randomized Skip List

(Perfect) Skip Lists

- A skip list is a collection of lists at different *levels*
- The lowest level (0) is a sorted, singly linked list of all nodes
- The first level (1) links alternate nodes
- The second level (2) links every fourth node
- In general, level i links every 2^i*th* node
- In total, $\lceil \log_2 n \rceil$ levels (i.e. $O(\log_2 n)$ levels).
- Each level has half the nodes of the one below it

In perfect skip list, after insertion and deletion, the nodes in every level has to be modified. Its difficult task and we can move to randomized skip list for the operation.

Randomized Skip List

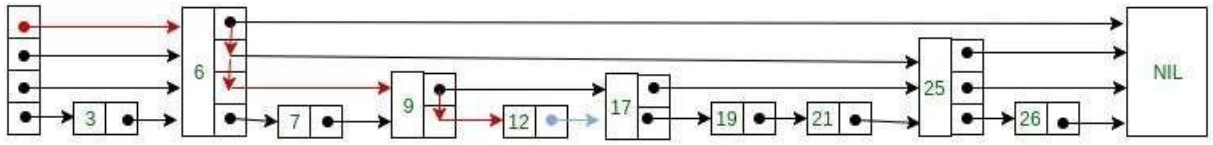
Searching

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

1. If Key of next node is less than search key then we keep on moving forward on the same level.
2. If Key of next node is greater than the key to be inserted then we store the pointer to current node **i** at **update[i]** and move one level down and continue our search.
3. 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.

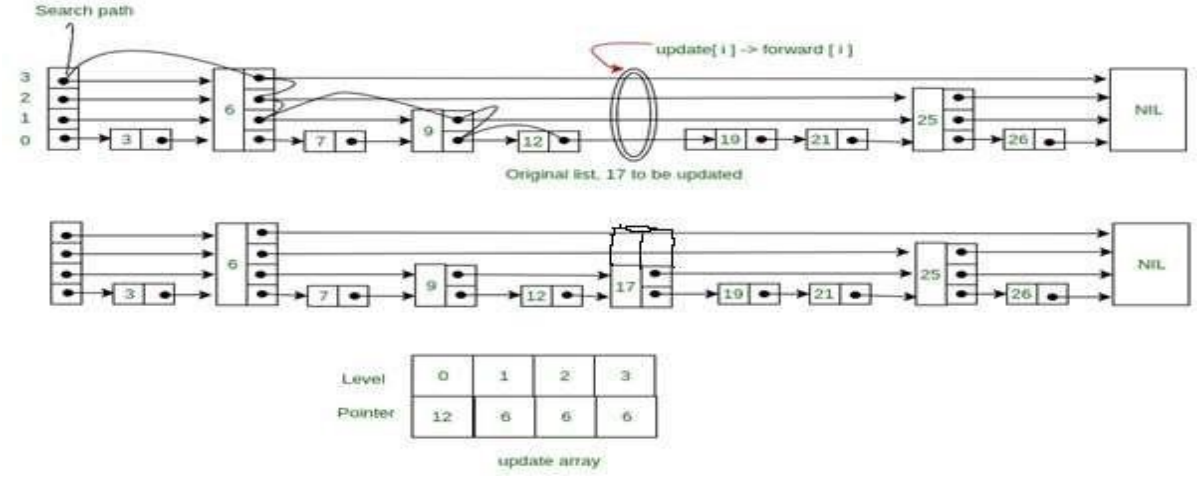


Insertion

We will start from the highest level as like search and find a position to insert the given key.

Example

Consider this example where we want to insert key 17.

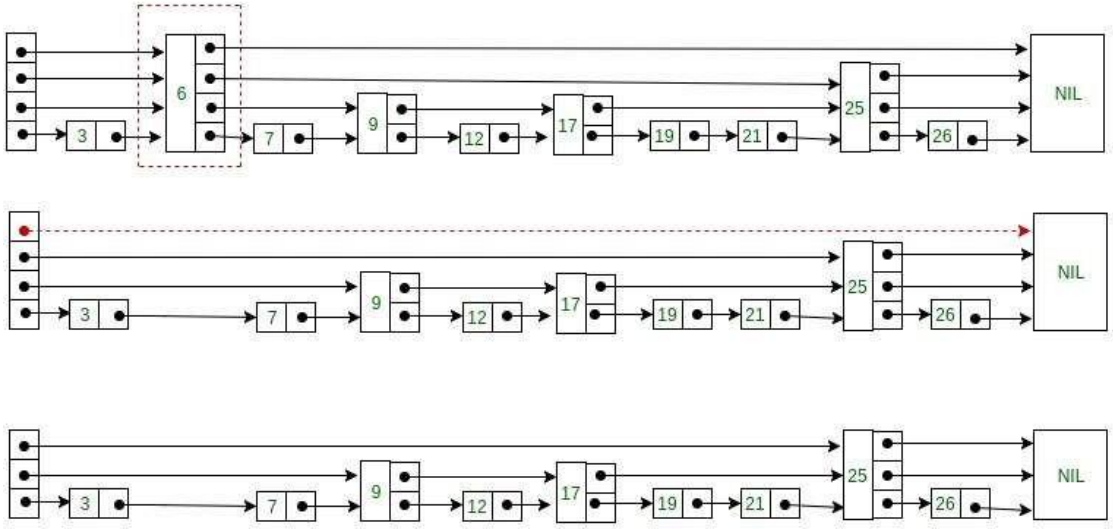


Deletion

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 to update[i] 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.

Example

Consider this example where we want to delete element 6 –



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

Advantages of Skip List

- The skip list is solid and trustworthy.
- To add a new node to it, it will be inserted extremely quickly.
- Easy to implement compared to the hash table and binary search tree
- The number of nodes in the skip list increases, and the possibility of the worst-case decreases
- Requires only Θ(logn) time in the average case for all operations.
- Finding a node in the list is relatively straightforward.

Disadvantages of Skip List

- It needs a greater amount of memory than the balanced tree.
- Reverse search is not permitted.
- Searching is slower than a linked list
- Skip lists are not cache-friendly because they don't optimize the locality of reference

HASHING

Hashing refers to the process of generating a fixed-size output from an input of variable size using the mathematical formulas known as hash functions. This technique determines an index or location for the storage of an item in a data structure.

Components of Hashing

There are majorly three components of hashing:

- **Key:**

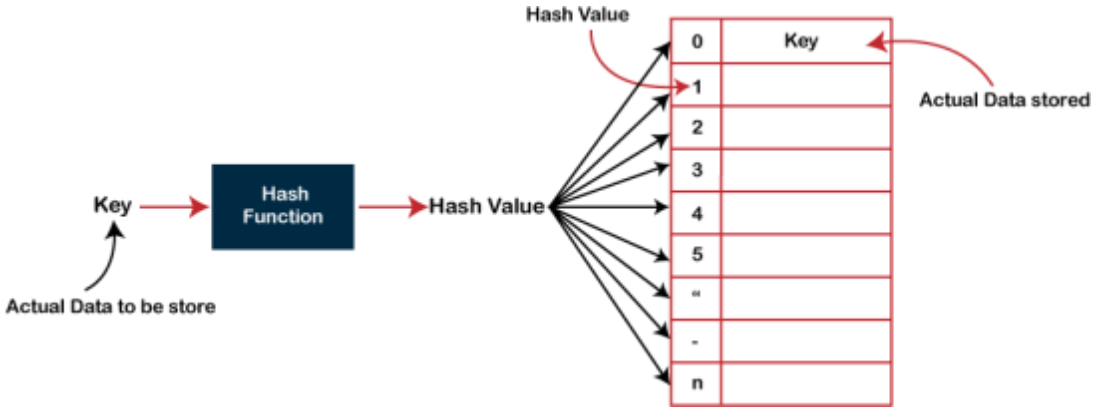
  A **Key** can be anything string or integer which is fed as input in the hash function the technique that determines an index or location for storage of an item in a data structure.

- **Hash Function:**

  The **hash function** receives the input key and returns the index of an element in an array called a hash table. The index is known as the **hash index**.

- **Hash Table:**

  Hash table is a data structure that maps keys to values using a special function called a hash function. Hash stores the data in an associative manner in an array where each data value has its own unique index.



Example for working of Hashing

Suppose we have a set of strings {“ab”, “cd”, “efg”} and we would like to store it in a table.

Our main objective here is to search or update the values stored in the table quickly in O(1) time and we are not concerned about the ordering of strings in the table. So the given set of strings can act as a key and the string itself will act as the value of the string.

Step 1: Let’s assign “a” = 1, “b”=2, .. etc, to all alphabetical characters.

Step 2: Therefore, the numerical value by summation of all characters of the string:

“ab” = 1 + 2 = 3,

“cd” = 3 + 4 = 7 ,

“efg” = 5 + 6 + 7 = 18

Step 4: Now, assume that we have a table of size 7 to store these strings. The hash function that is used here is the sum of the characters in **key mod Table size**. We can compute the location of the string in the array by taking the **sum(string) mod 7**.

Step 5: So we will then store

"ab" in 3 mod 7 = 3,

"cd" in 7 mod 7 = 0, and

"efg" in 18 mod 7 = 4.

0	1	2	3	4	5	6
cd			ab	efg		

The above technique enables us to calculate the location of a given string by using a simple hash function and rapidly find the value that is stored in that location. Therefore the idea of hashing seems like a great way to store (key, value) pairs of the data in a table.

Problem with Hashing

If we consider the above example, the hash function we used is the sum of the letters, but if we examined the hash function closely then the problem can be easily visualized that for different strings same hash value is begin generated by the hash function.

For example: {"ab", "ba"} both have the same hash value, and string {"cd","be"} also generate the same hash value, etc. This is known as **collision** and it creates problem in searching, insertion, deletion, and updating of value.

Hash function

The hash function creates a mapping between key and value, this is done through the use of mathematical formulas known as **hash functions**. The result of the hash function is referred to as a hash value or hash. The hash value is a representation of the original string of characters but usually smaller than the original.

Types of Hash functions

There are many hash functions that use numeric or alphanumeric keys.

- Division Method.
- Mid Square Method.
- Digit Folding Method.
- Multiplication Method.

Properties of a Good hash function

- Efficiently computable.
- Should uniformly distribute the keys (Each table position is equally likely for each.
- Should minimize collisions.
- Should have a low load factor(number of items in the table divided by the size of the table).

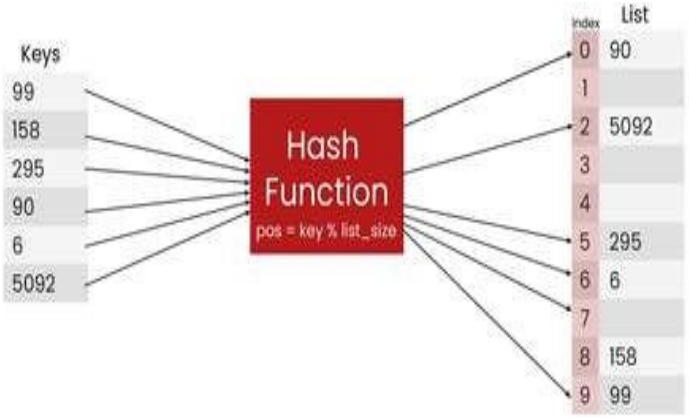
Division Method.

Division Modulo Method is the simplest method of hashing. In this method, we divide the element with the size of the hash table and use the remainder as the index of the element in the hash table. **H(key) = k mod m**, where m is size of hash table

Example

Size of Hash Table (m) = 10
Key = { 99, 158, 295, 90, 6, 5092 }
H(key) = k mod m

H(99) = 99 % 10 = 9
H(158) = 158 % 10 = 8
H(295) = 295 % 10 = 5
H(90) = 90 % 10 = 0
H(6) = 6 % 10 = 6
H(5092)= 5092 % 10 = 2



Mid Square Method

The mid square method is a very good hash function. It involves squaring the value of the key and then extracting the middle **r** digits as the hash value. The value of **r** can be decided according to the size of the hash table.

Example

Suppose the hash table has 100 memory locations. So **r=2** because two digits are required to map the key to memory location.

(i)	k = 50	
	k*k = 2500	
	h(50) = 50	The hash value obtained is 50
(ii)	k = 25	
	k*k = 625	
	h(25) = 62	The hash value obtained is 62

Digit Folding Method.

We can break a key into groups of digits and then do the addition of groups. This ensures that all the digits contribute the hash code. The number of digits in a group corresponds to the size of the array.

There are 2 types of folding methods used **Fold shift** and **Fold boundary**.

Fold Shift

We divide the key in parts whose size matches the size of required address. The parts are simply added to get the required address.

Key: 123456789 and size of required address is 3 digits.

123+456+789 = 1368.

To reduce the size to 3, either 1 or 8 is removed and accordingly the key would be 368 or 136 respectively.

Fold Boundary

We again divide the key in parts whose size matches the size of required address. But now you also applying folding, except for the middle part, if its there.

Key:123456789 and size of required address is 3 digits

321 (folding applied)+456+987 (folding applied) = 1764

(discard 1 or 4 and accordingly the key would be 764 or 176 respectively)

Multiplication Method

The hash function used for the multiplication method is **h(k) = floor(m (kA mod 1))**

Here, k is the key and A can be any constant value between 0 and 1. Both k and A are multiplied and their fractional part is separated. This is then multiplied with **m** to get the hash value.

Example

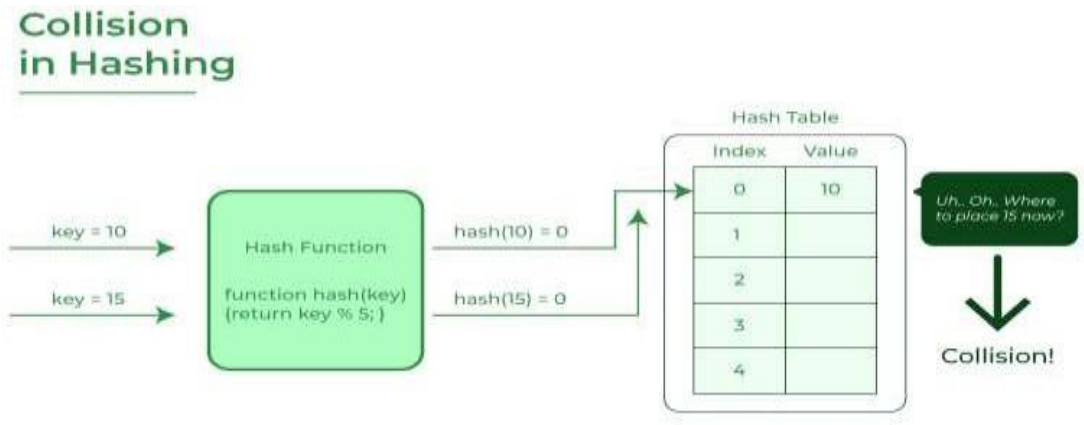
k = 123
m =100
A=0.618033
h(123) = 100 (123 * 0.618033 mod 1) = 100 (76.018059 mod 1) = 100 (0.018059) = 1

The hash value obtained is 1

An advantage of the multiplication method is that it can work with any value of A, although some values are believed to be better than others.

Collision

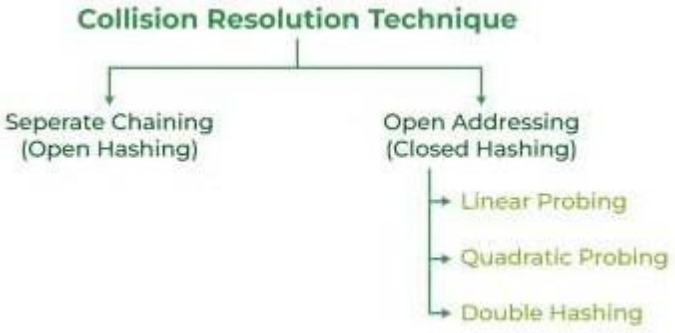
The hashing process generates a small number for a big key, so there is a possibility that two keys could produce the same value. The situation where the newly inserted key maps to an already occupied, and it must be handled using some collision handling technology.



Collision Resolution

There are mainly two methods to handle collision

1. Closed Addressing / Open Hashing / Separate Chaining
2. Open Addressing / Closed Hashing



1) Closed Addressing / Open Hashing / Separate Chaining

The idea is to make each cell of the hash table point to a linked list of records that have the same hash function value. Chaining is simple but requires additional memory outside the table.

Example

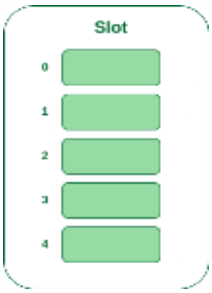
We have given a hash function and we have to insert some elements in the hash table using a separate chaining method for collision resolution technique.

Hash function = key % 5,

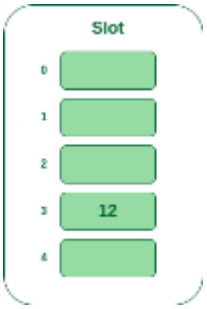
Key = {12, 22, 15, 25}

Step 1:

First draw the empty hash table which will have a possible range of hash values from 0 to 4 according to the hash function provided.



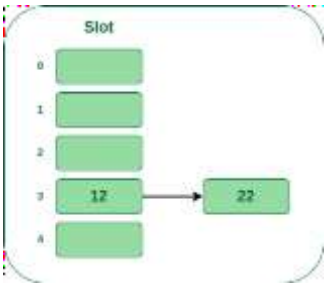
Step 2:



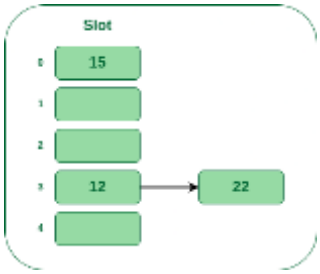
Now insert all the keys in the hash table one by one. The first key to be inserted is 12 which is mapped to bucket number 2 which is calculated by using the hash function 12%5=2.

Step 3:

Now the next key is 22. It will map to bucket number 2 because 22%5=2. But bucket 2 is already occupied by key 12.



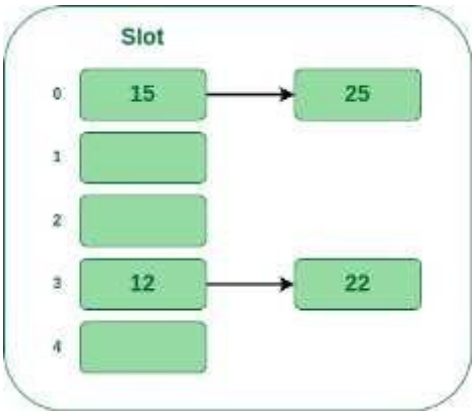
Step 4:



The next key is 15. It will map to slot number 0 because 15%5=0.

Step 5:

Now the next key is 25. Its bucket number will be 25%5=0. But bucket 0 is already occupied by key 25. So separate chaining method will again handle the collision by creating a linked list to bucket 0.



Hence, in this way, the separate chaining method is used as the collision resolution technique.

2) Open Addressing / Closed Hashing

In open addressing, all elements are stored in the hash table itself. Each table entry contains either a record or NIL. When searching for an element, we examine the table slots one by one until the desired element is found or it is clear that the element is not in the table.

2.a) Linear Probing

In linear probing, the hash table is searched sequentially that starts from the original location of the hash. If in case the location that we get is already occupied, then we check for the next location.

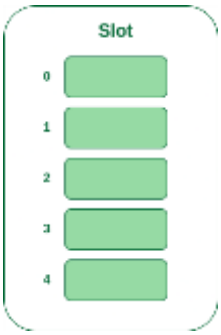
1. If the hash index already has some value then check for next index using
   key = (H(key)+i) % size where i = 0 to m-1
2. Check, if the next index is available hashTable[key] then store the value. Otherwise try for next index.
3. Repeat the above process till we find the space.

Example 1

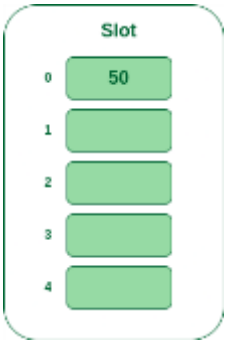
Let us consider a simple hash function as "key mod 5" and a sequence of keys that are to be inserted are 50, 70, 76, 93.

Step1:

First draw the empty hash table which will have a possible range of hash values from 0 to 4 according to the hash function provided.



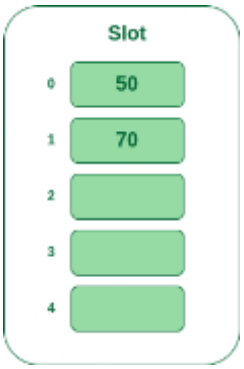
Step 2:



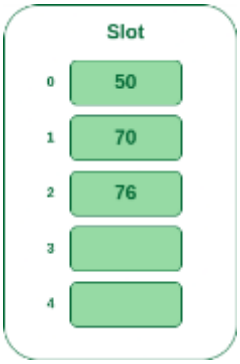
Now insert all the keys in the hash table one by one. The first key is 50. It will map to slot number 0 because 50%5=0. So insert it into slot number 0.

Step 3:

The next key is 70. It will map to slot number 0 because 70%5=0 but 50 is already at slot number 0 so, search for the next empty slot and insert it.



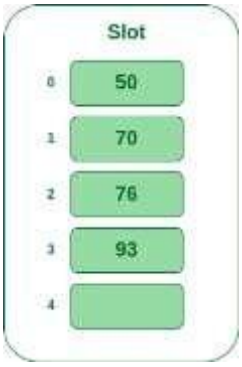
Step 4:



The next key is 76. It will map to slot number 1 because 76%5=1 but 70 is already at slot number 1 so, search for the next empty slot and insert it.

Step 5:

The next key is 93 It will map to slot number 3 because 93%5=3, So insert it into slot number 3.



Example 2

- Table Size is 11 (0..10)
- Hash Function: **h(x) = x mod 11**
- Insert keys: 20, 30, 2, 13, 25, 24, 10, 9
  - 20 mod 11 = 9
  - 30 mod 11 = 8
  - 2 mod 11 = 2
  - 13 mod 11 = 2 □ 2+1=3
  - 25 mod 11 = 3 □ 3+1=4
  - 24 mod 11 = 2 □ 2+1, 2+2, 2+3=5
  - 10 mod 11 = 10
  - 9 mod 11 = 9 □ 9+1, 9+2 mod 11 =0

0	9
1	
2	2
3	13
4	25
5	24
6	
7	
8	30
9	20
10	10

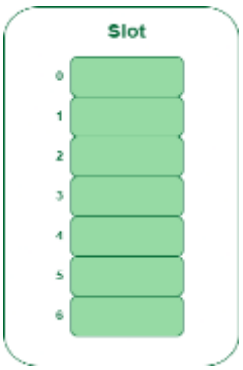
2.b) Quadratic Probing

Quadratic probing is an open addressing scheme in computer programming for resolving hash collisions in hash tables. Quadratic probing operates by taking the original hash index and adding successive values of an arbitrary quadratic polynomial until an open slot is found.

- Let **H(key)** be the slot index computed using the hash function and **n** be the size of the hash table.
- If the slot **H(key) % n** is full, then we try **(H(key) + i^2) % n** where i=1 to n-1.
- If it is free, then store the value. Otherwise repeat the above step for next value of i.
- This process will be repeated for all the values of **i** until an empty slot is found.

Example 1

Let us consider table Size = 7, hash function as Hash(x) = x % 7 and collision resolution strategy to be f(i) = i^2 . Insert = 22, 30, 50

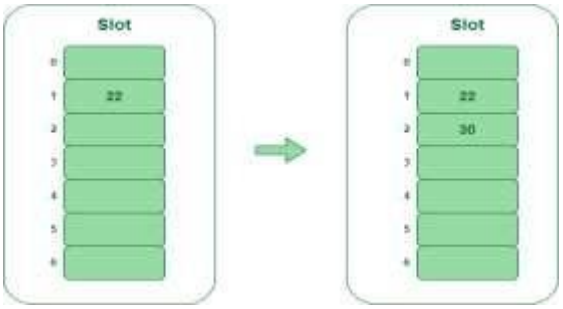


Step 1:

Create a table of size 7.

Step 2:

Inserting 22 and 30



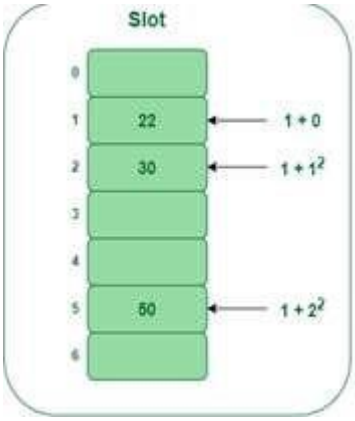
Hash(22) = 22 % 7 = 1, Since the cell at index 1 is empty, we can easily insert 22 at slot 1.

Hash(30) = 30 % 7 = 2, Since the cell at index 2 is empty, we can easily insert 30 at slot 2.

Step 3:

Inserting 50

- Hash(50) = 50 % 7 = 1
- In our hash table slot 1 is already occupied. So, we will search for slot 1+1^2, i.e. 1+1 = 2,
- Again slot 2 is found occupied, so we will search for cell 1+2^2, i.e.1+4 = 5,
- Now, cell 5 is not occupied so we will place 50 in slot 5.



Example 2

0	49
1	
2	58
3	79
4	
5	
6	
7	
8	18
9	89

TableSize = 10
insert(89)
insert(18)
insert(49)
insert(58)
insert(79)
79 % 10 = 9 collision!
(79 + 1) % 10 = 0 collision!
(79 + 4) % 10 = 3

2.c) Double Hashing

Double hashing is a collision resolving technique in Open Addressed Hash tables. Double hashing make use of two hash function,

- The first hash function is **$h_1(k)$** which takes the key and gives out a location on the hash table. But if the new location is not occupied or empty then we can easily place our key.
- But in case the location is occupied (collision) we will use secondary hash- function **$h_2(k)$** in combination with the first hash-function **$h_1(k)$** to find the new location on the hash table.

This combination of hash functions is of the form **$h(k, i) = (h_1(k) + i * h_2(k)) \% n$**
where **i** is a non-negative integer that indicates a collision number,
k = element/key which is being hashed
n = hash table size.

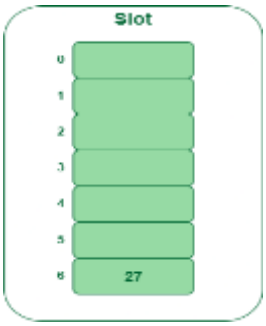
Example 1

Insert the keys 27, 43, 692, 72 into the Hash Table of size 7 where first hash-function is **$h_1(k) = k \bmod 7$** and second hash-function is **$h_2(k) = 1 + (k \bmod 5)$**

Step 1:

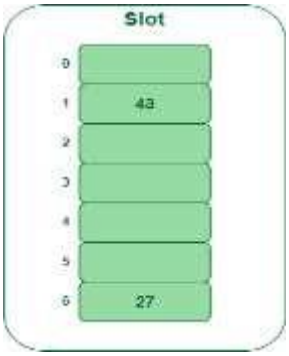
Insert 27

27 % 7 = 6, location 6 is empty so insert 27 into 6th slot.

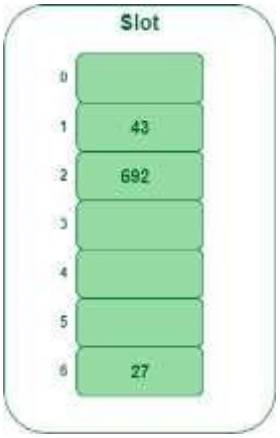


Step 2:

Insert 43



43 % 7 = 1, location 1 is empty so insert 43 into 1st slot.



Step 3:

Insert 692

692 % 7 = 6, but location 6 is already being occupied and this is a collision. So we need to resolve this collision using double hashing.

$h_{new} = [h_1(692) + i * (h_2(692)] \% 7$

$= [6 + 1 * (1 + (692 \% 5))] \% 7$

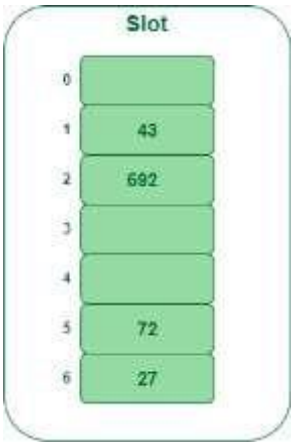
$= [6+1(3)] \% 7$

$= 2$

As 2 is an empty slot, so we can insert 692 into 2nd slot.

Step 4:

Insert 72



72 % 7 = 2, but location 2 is already being occupied and this is a collision. So we need to resolve this collision using double hashing.

$h_{new} = [h_1(72) + i * (h_2(72)] \% 7$

$= [2 + 1 * (1 + 72 \% 5)] \% 7$

$= 5 \% 7$

$= 5$

Now, as 5 is an empty slot, so we can insert 72 into 5th slot.

Example 2

$h(key, i) = (h_1(key) + i * h_2(key)) \% n$

Table Size is 11 (1...10)

Hash function: assume h1(key) = key mod 11 and

h2(key) = 7 – (key mod 7)

Insert keys:

58 mod 11 = 3

14 mod 11 = 3 -> 3+7 =10

91 mod 11 = 3 -> 3+7, 3+2*7 mod 11 =6

25 mod 11 = 3 -> 3+3, 3+2*3 = 9

0	
1	
2	
3	58
4	
5	
6	91
7	
8	
9	25
10	14

Load Factor in Hashing

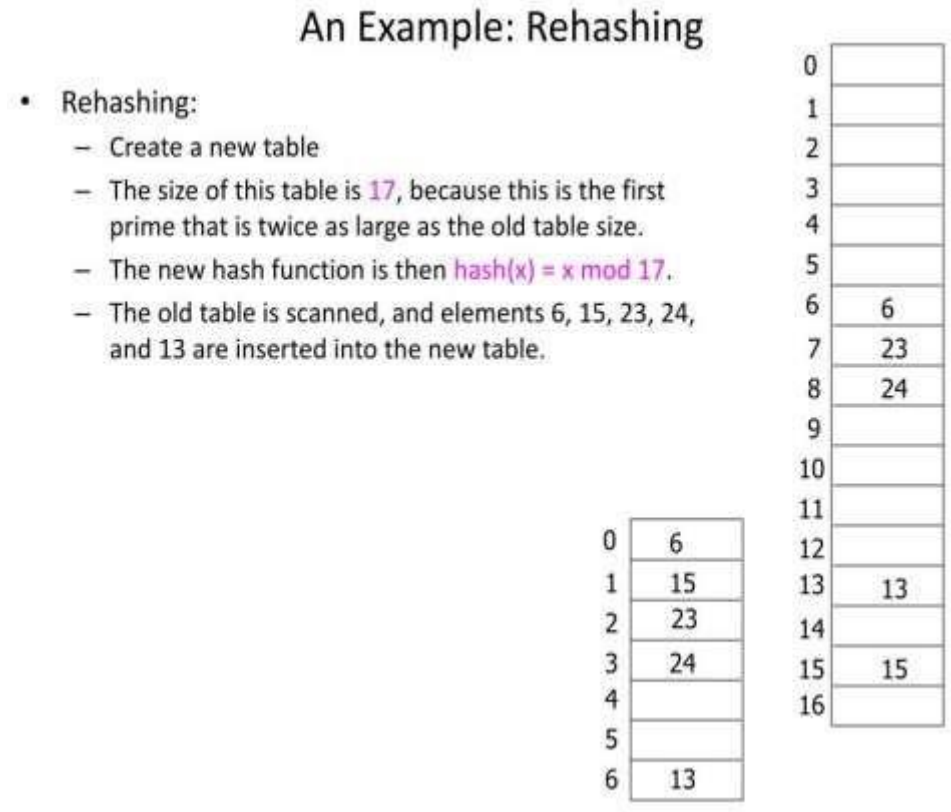
The load factor of the hash table can be defined as the number of items the hash table contains divided by the size of the hash table. Load factor is the decisive parameter that is used when we want to rehash the previous hash function or want to add more elements to the existing hash table.

It helps us in determining the efficiency of the hash function i.e. it tells whether the hash function which we are using is distributing the keys uniformly or not in the hash table.

$$Load\ \ Factor\ (LF) = \frac{\text{Total elements in hash table}}{\text{Size of hash table}}$$

Rehashing

As the name suggest, rehashing means hashing again. Basically, when the load factor increases to more than its predefined value (the default value of the load factor is 0.75), the complexity increases. So to overcome this, the size of the array is increased (doubled) and all the values are hashed again and stored in the new double-sized array to maintain a low load factor and low complexity.



Applications of Hash

- Hash is used in databases for indexing.
- Hash is used in disk-based data structures.
- In some programming languages like Python, JavaScript hash is used to implement objects.

Real-Time Applications of Hash

- Hash is used for cache mapping for fast access to the data.
- Hash can be used for password verification.
- Hash is used in cryptography as a message digest.

Advantages of Hash Data structure

- Hash provides better synchronization than other data structures.
- Hash tables are more efficient than search trees or other data structures
- Hash provides constant time for searching, insertion, and deletion operations on average.

Disadvantages of Hash Data structure

- Hash is inefficient when there are many collisions.
- Hash collisions are practically not avoided for a large set of possible keys.
- Hash does not allow null values.