HASHING

- Hashing in the data structure is a technique of mapping a large chunk of data into small tables using a hashing function. It is also known as the message digest function. It is a technique that uniquely identifies a specific item from a collection of similar items.
- It uses hash tables to store the data in an array format. Each value in the array has been assigned a unique index number. Hash tables use a technique to generate these unique index numbers for each value stored in an array format. This technique is called the hash technique.
- You only need to find the index of the desired item, rather than finding the data. With indexing, you can quickly scan the entire list and retrieve the item you wish. Indexing also helps in inserting operations when you need to insert data at a specific location. No matter how big or small the table is, you can update and retrieve data within seconds.
- The hash table is basically the array of elements, and the hash techniques of search are performed on a part of the item i.e. key. Each key has been mapped to a number, the range remains from 0 to table size 1
- Types of hashing in data structure is a two-step process.
    - The hash function converts the item into a small integer or hash value. This integer is used as an index to store the original data.
    - It stores the data in a hash table. You can use a hash key to locate data quickly.

Examples

- In schools, the teacher assigns a unique roll number to each student. Later, the teacher uses that roll number to retrieve information about that student.
- A library has an infinite number of books. The librarian assigns a unique number to each book. This unique number helps in identifying the position of the books on the bookshelf.

HASH FUNCTION

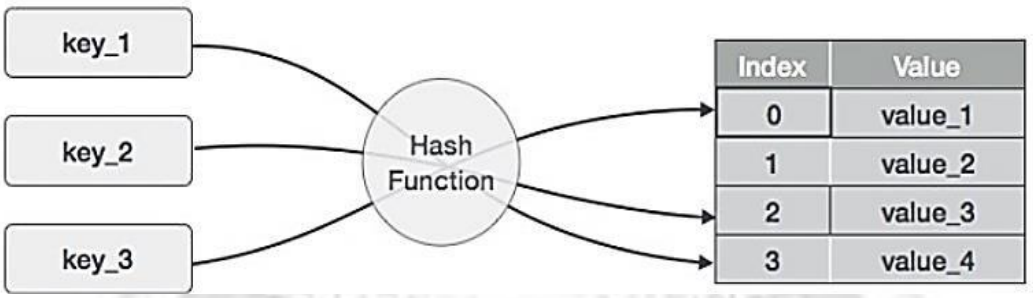
- The hash function in a data structure maps the arbitrary size of data to fixed-sized data. It returns the following values: a small integer value (also known as hash value), hash codes, and hash sums. The hashing techniques in the data structure are very interesting, such as:

- hash = hashfunc(key)
- index = hash % array_size

- The hash function must satisfy the following requirements:
  - A good hash function is easy to compute.
  - A good hash function never gets stuck in clustering and distributes keys evenly across the hash table.
  - A good hash function avoids collision when two elements or items get assigned to the same hash value.
- The three characteristics of the hash function in the data structure are:
  - Collision free
  - Property to be hidden
  - Puzzle friendly

Hash Table

- Hashing in data structure uses hash tables to store the key-value pairs. The hash table then uses the hash function to generate an index. Hashing uses this unique index to perform insert, update, and search operations.



- It can be defined as a bucket where the data are stored in an array format. These data have their own index value. If the index values are known then the process of accessing the data is quicker.

How does Hashing in Data Structure Works?

- In hashing, the hashing function maps strings or numbers to a small integer value. Hash tables retrieve the item from the list using a hashing function.
- The objective of hashing technique is to distribute the data evenly across an array. Hashing assigns all the elements a unique key. The hash table uses this key to

access the data in the list.

- Hash table stores the data in a key-value pair. The key acts as an input to the hashing function. Hashing function then generates a unique index number for each value stored.
- The index number keeps the value that corresponds to that key. The hash function returns a small integer value as an output. The output of the hashing function is called the hash value.
- Let us understand hashing in a data structure with an example. Imagine you need to store some items (arranged in a key-value pair) inside a hash table with 30 cells. The values are: (3,21) (1,72) (40,36) (5,30) (11,44) (15,33) (18,12) (16,80) (38,99)
- The hash table will look like the following:

Serial Number	Key	Hash	Array Index
1	3	3%30 = 3	3
2	1	1%30 = 1	1
3	40	40%30 = 10	10
4	5	5%30 = 5	5
5	11	11%30 = 11	11
6	15	15%30 = 15	15
7	18	18%30 = 18	18
8	16	16%30 = 16	16
9	38	38%30 = 8	8

- The process of taking any size of data and then converting that into smaller data value which can be named as hash value. This hash value can be used in an index accessible in hash table. This process define hashing in data structure.

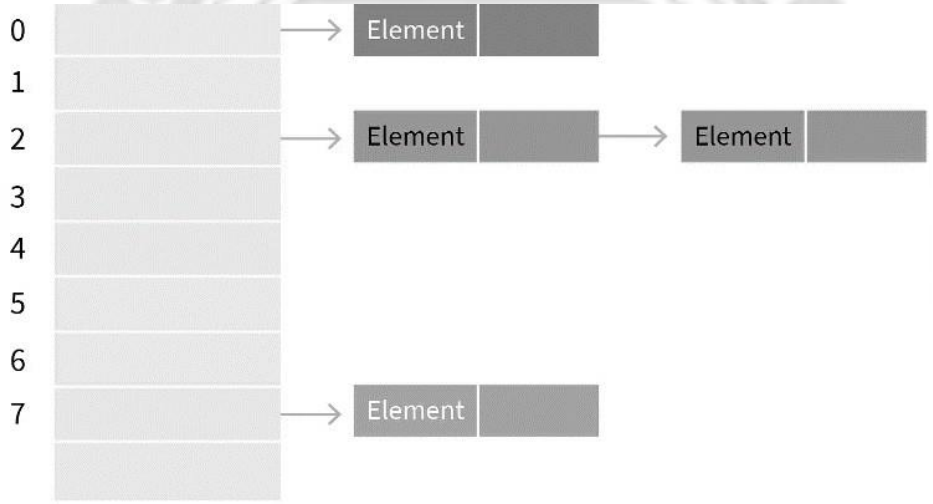
SEPARATE CHAINING

- Separate Chaining is the collision resolution technique that is implemented using linked list. When two or more elements are hash to the same location, these elements are represented into a singly linked list like a chain. Since this method uses extra memory to resolve the collision, therefore, it is also known as open

hashing.

Separate Chaining Hash Table

- In separate chaining, each slot of the hash table is a linked list. We will insert the element into a specific linked list to store it in the hash table. If there is any collision i.e. if more than one element after calculating the hashed value mapped to the same key then we will store those elements in the same linked list. Given below is the representation of the separate chaining hash table.

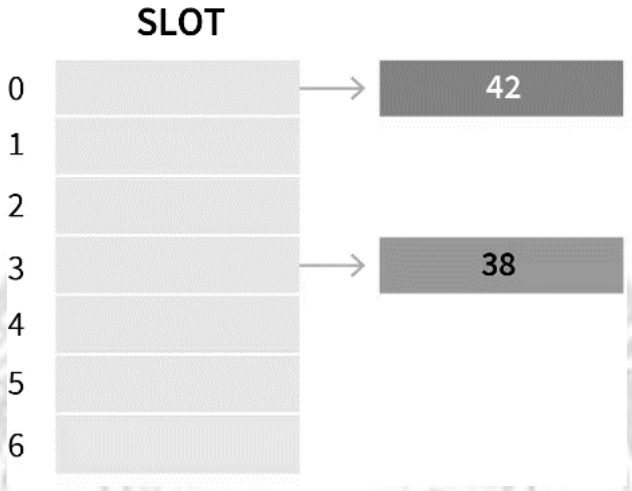


Example for Separate Chaining

- Let's understand with the help of examples. Given below is the hash

  function:h(key) = key % table size

- In a hash table with size 7, keys 42 and 38 would get 0 and 3 as hash indices respectively.



- If we insert a new element 52 , that would also go to the fourth index as 52%7 is 3.

- The lookup cost will be scanning all the entries of the selected linked list for the required key. If the keys are uniformly distributed, then the average lookup cost will be an average number of keys per linked list.

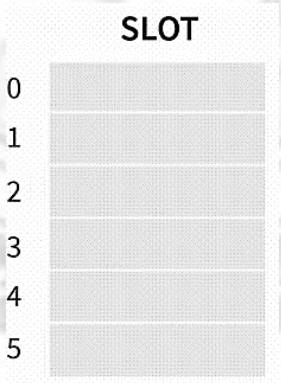
How to Avoid Collision in Separate Chaining Method

- Separate chaining method handles the collison by creating a linked list to the occupied buckets. So far we only looked at a simple hash function where collision is imminent.
- It is important to choose a good hash function in order to minimize the number of collisions so that all the key values are evenly distributed in the hash table.

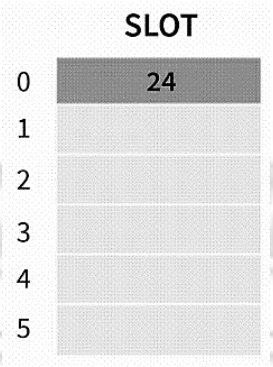
- Some characterstics of good hash function are:
  - ✓ Minimize collisions
  - ✓ Be easy and quick to compute
  - ✓ key values inserted evenly in the hash table
  - ✓ Have a high load factor for a given set of keys

Practice Problem Based on Separate Chaining

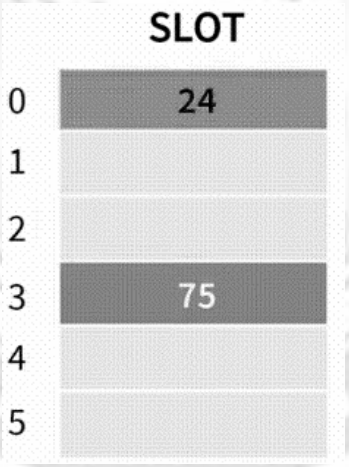
- Let's take an example to understand the concept more clearly. Suppose we have the following hash function, and we have to insert certain elements in the hash table by using separate chaining as the collision resolution technique.
- Hash function = key % 6 Elements = 24, 75, 65, 81, 42, and 63.
- **Step1:** First we will draw the empty hash table which will have possible range ofhash values from 0 to 5 according to the hash function provided.



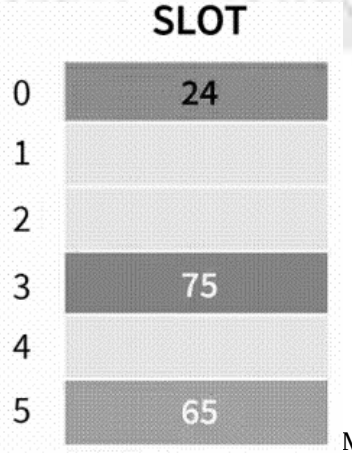
- **Step 2:** Now we will insert all the keys in the hash table one by one. First key to be inserted is 24. It will map to bucket number 0 which is calculated by using hash function 24%6=0.



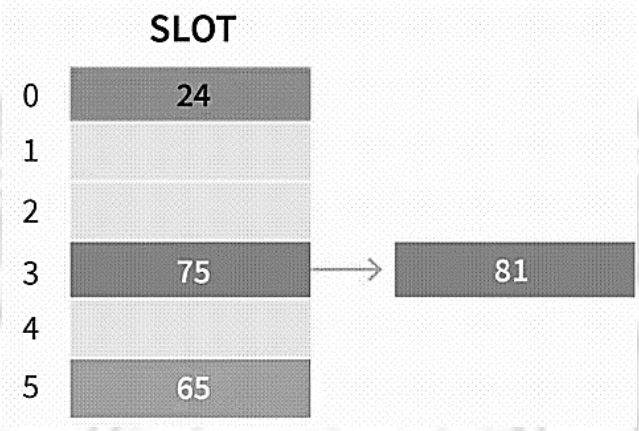
- Step 3: Now the next key that is need to be inserted is 75. It will map to the bucket number 3 because 75%6=3. So insert it to bucket number 3.



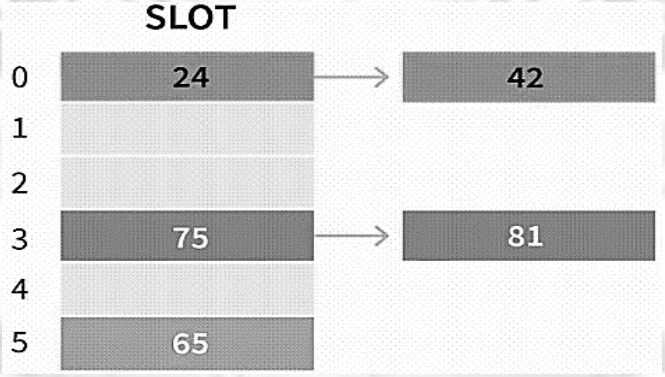
- Step 4: The next key is 65. It will map to bucket number 5 because 65%6=5. So, insert it to bucket number 5.



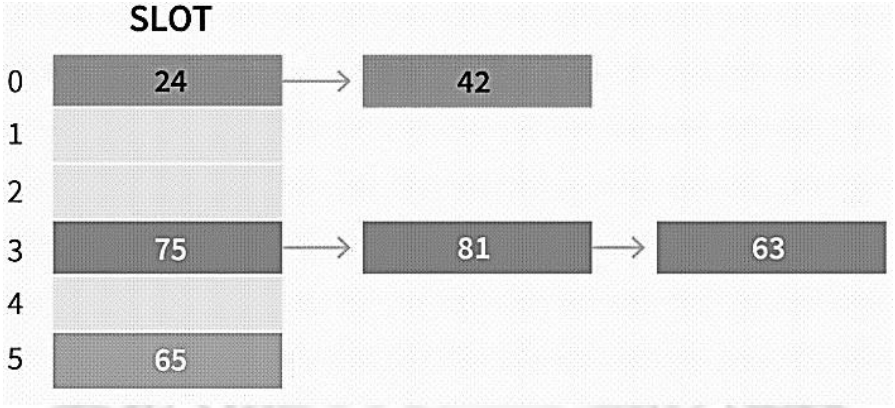
- Step 5: Now the next key is 81. Its bucket number will be 81%6=3. But bucket 3 is already occupied by key 75. So separate chaining method will handles the collision by creating a linked list to bucket 3.



- Step 6: Now the next key is 42. Its bucket number will be 42%6=0. But bucket 0 is already occupied by key 24. So separate chaining method will again handles the collision by creating a linked list to bucket 0.



- Step 7: Now the last key to be inserted is 63. It will map to the bucket number 63%6=3. Since bucket 3 is already occupied, so collision occurs but separate chaining method will handle the collision by creating a linked list to bucket 3.



- In this way the separate chaining method is used as the collision resolution technique.

Advantages and Disadvantages of Separate Chaining

Advantages

- Separate Chaining is one of the simplest methods to implement and understand.
- We can add any number of elements to the chain.
- It is frequently used when we don't know about the number of elements and the number of keys that can be inserted or deleted.

Disadvantages

- The keys in the hash table are not evenly distributed.
- Some amount of wastage of space occurs.
- The complexity of searching becomes O(n) in the worst case when the chain becomes long.

OPEN ADDRESSING

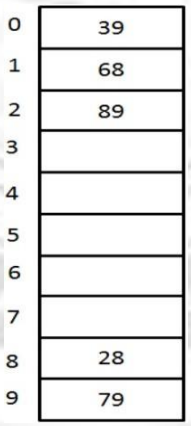
- The open addressing is another technique for collision resolution. Unlike chaining, it does not insert elements to some other data-structures. It inserts the data into the hash table itself. The size of the hash table should be larger than the number of keys.
- There are three different popular methods for open addressing techniques. These methods are −
  - ✓ Linear Probing
  - ✓ Quadratic Probing

- ✓ Double Hashing

LINEAR PROBING

- This is a simple method, sequentially tries the new location until an empty location is found in the table.
- For example: inserting the keys {79, 28, 39, 68, 89} into closed hash table by using same function and collision resolution technique as mentioned before and the table size is 10 (for easy undestanding we are not using prime number for table size).Here array or hash table is considered circular because when the last slot reached an empty location not found then the search proceeds to the first location of the array.
- The hash function is $h_i(X)=(Hash(X)+F(i))$ % TableSize for i = 0, 1, 2, 3,...etc.

4.10.1 Solution



A Closed Hash Table using Linear Probing

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	

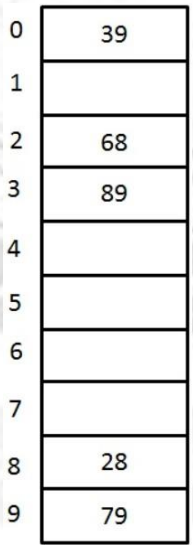
	$h_1(39)$ = (Hash (39) +F(0)) %10 = ((39%10) +1) %10 = 0	0		0
68	$h_0(68)$ = (Hash(68)+F(0))%10 = ((68%10)+0)%10 = 8	8	First collision occurs	
	$h_1(68)$ = (Hash(68)+F(1))%10 = ((68%10)+1)%10 = 9	9	Again collision occurs	
	$h_2(68)$ = (Hash(68)+F(2))%10 = ((68%10)+2)%10 = 0	0	Again collision occurs	
	$h_3(68)$ = (Hash(68)+F(3))%10 = ((68%10)+3)%10 = 1	1		1
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)%10 = 0	0	Again collision occurs	
	$h_2(89)$ = (Hash(89)+F(2))%10 = ((89%10)+2)%10 = 1	1	Again collision occurs	
	$h_3(89)$ = (Hash(89)+F(3))%10 = ((89%10)+3)%10 = 2	2		2

QUADTRATIC PROBING

- Quadratic probing is an open addressing method for resolving collision in the hash table. This method is used to eliminate the primary clustering problem of linear probing.
- This technique works by considering of original hash index and adding successive value of an arbitrary quadratic polynomial until the empty location is found. In linear probing, we would use H+0, H+1, H+2, H+3,.....H+K hash function sequence.
- Instead of using this sequence, the quadratic probing would use another sequence is that $H+1^2$, $H+2^2$, $H+3^2$,....$H+K^2$. Therefore, the hash function for quadratic probing is

$$h_i(X) = (Hash(X) + F(i)^2) \% TableSize \text{ for } i = 0, 1, 2, 3,...etc.$$

- Let us examine the linear probing with the same example



Key	Hash Function h(X)	Index	Collision	Alt Index
79	$h_0(79)$ $= (Hash (79) + F(0)^2) \% 10$ $= ((79 \% 10) + 0) \% 10$	9		

28	$h_0(28)$ $= (Hash(28) + F(0)^2) \% 10$ $= ((28 \% 10) + 0) \% 10$	8		
39	$h_0(39)$ $= (Hash(39) + F(0)^2) \% 10$ $= ((39 \% 10) + 0) \% 10$	9	The first collision occurs	
	$h_1(39)$ $= (Hash(39) + F(1)^2) \% 10$ $= ((39 \% 10) + 1) \% 10$	0		0
68	$h_0(68)$ $= (Hash(68) + F(0)^2) \% 10$ $= ((68 \% 10) + 0) \% 10$	8	The collision occurs	
	$h_1(68)$ $= (Hash(68) + F(1)^2) \% 10$ $= ((68 \% 10) + 1) \% 10$	9	Again collision occurs	
	$h_2(68)$ $= (Hash(68) + F(2)^2) \% 10$ $= ((68 \% 10) + 4) \% 10$	2		2
89	$h_0(89)$ $= (Hash(89) + F(0)^2) \% 10$ $= ((89 \% 10) + 0) \% 10$	9	The collision occurs	
	$h_1(89)$ $= (Hash(89) + F(1)^2) \% 10$ $= ((89 \% 10) + 1) \% 10$	0	Again collision occurs	
	$h_2(89)$ $= (Hash(89) + F(2)^2) \% 10$ $= ((89 \% 10) + 4) \% 10$	3		3

- Although, the quadratic probing eliminates the primary clustering, it still has the

problem.

- When two keys hash to the same location, they will probe to the same alternative location. This may cause secondary clustering. In order to avoid this secondary clustering, double hashing method is created where we use extra multiplications and divisions

DOUBLE HASHING

- Double Hashing uses 2 hash functions and hence called double hashing. The first hash function determines the initial location to locate the key and the second hash function is to determine the size of the jumps in the probe sequence.

Double Hashing - Hash Function 1 or First Hash Function – formula

- hi = (Hash(X) + F(i)) % Table

  Sizewere

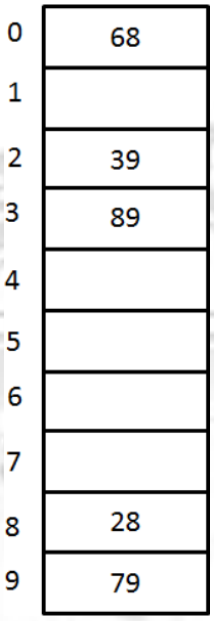
  - F(i) = i * hash2(X)
  - X is the Key or the Number for which the hashing is done
  - i is the ith time that hashing is done for the same value. Hashing is repeated only when collision occurs
  - Table size is the size of the table in which hashing is done
- This F(i) will generate the sequence such as $hash_2(X)$, 2 * $hash_2(X)$ and so on.

Double Hashing - Hash Function 2 or Second Hash Function – formula

- Second hash function is used to resolve collision in hashing We use second hash function as
  - hash2(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.



A Closed Hash Table using Double Hashing

Key	Hash Function h(X)	Index	Collision	At Index
79	h0(79)=(Hash(79)+F(0))% 10 = ((79 % 10) + 0) % 10 = 9	9		
28	h0(28)=(Hash(28)+F(0))% 10 = ((28 % 10) + 0) % 10 = 8	8		
39	h0(39)=(Hash(39)+F(0))% 10 = ((39 % 10) + 0) % 10 = 9	9	First collision occurs	
	h1(39)=(Hash(39)+F(1))%10 =((39% 10)+1(7-(39 % 7))) % 10 = (9 + 3) % 10 =12 % 10=2	2		2
68	h0(68) = (Hash (68) + F(0)) % 10 = ((68 % 10) + 0) % 10 = 8	8	collision occurs	
	h1(68) =(Hash(68) +F(1)) % 10 = ((68 % 10) +1(7-(68 % 7))) % 10 = (8 + 2) % 10 =10 % 10=0	0		0
89	h0(89) =(Hash(89) + F(0)) % 10	9	Collision	

	= ((89 % 10) + 0) % 10 = 9		occurs	
	h1(89) = (Hash(89) + F(1)) % 10 = ((89 % 10) + 1(7-(89 % 7))) % 10 = (9 + 2) % 10 =10 % 10 = 0	0	Again collision occurs	
	h2(89) = (Hash(89) + F(2)) % 10 = ((89 % 10) + 2(7-(89 % 7))) % 10 = (9 + 4) % 10=13 % 10 =3	3		3

RE-HASHING

- Rehashing is the process of re-calculating the hashcode of already stored entries (Key-Value pairs), to move them to another bigger size hashmap when the threshold is reached/crossed.

Why Rehashing is done?

- Rehashing is done because whenever a new key value pair is inserted into map, the load factor increases and due to which complexity also increases. And if complexity increases our HashMap will not have constant O(1) time complexity.
- Hence rehashing is done to distribute the items across the hashmap as to reduce both laod factor and complexity, So that get() and put() have constant time complexity of O(1).
- After rehashing is done existing items may fall in the same bucket or different bucket.

What is Load factor in HashMap?

- Load factor in HashMap is basically a measure that decides when exactly to increase the size of the HashMap to maintain the same time complexity of O(1).
- Load factor is defined as (m/n) where n is the total size of the hash table and m isthe preferred number of entries which can be inserted before an increment in the size of the underlying data structure is required.
- If you are going to store really large no of elements in the hashmap then it is always good to create HashMap with sufficient capacity upfront as rehashing will not be done frequently, this is more efficient than letting it to perform automatic rehashing.

How Rehashing is Done?

- Let's try to understand the above with an example: Say we had HashTable with Initial Capacity of 4. We need to insert 4 Keys: 100, 101, 102, 103
- Hash function used was division method: Key % ArraySize
- Element1: Hash(100) = 100%6 = 4, so Element1 will be rehashed and will be stored at 5th Index in this newly resized HashTable, instead of 1st Index as on previous HashTable.
- Element2: Hash(101) = 101%6 = 5, so Element2 will be rehashed and will be stored at 6th Index in this newly resized HashTable, instead of 2nd Index as on previous HashTable.
- Element3: Hash(102) = 102%6 = 6, so Element3 will be rehashed and will be stored at 4th Index in this newly resized HashTable, instead of 3rd Index as on previous HashTable.
- Since the Load Balance now is 3/6 = 0.5, we can still insert the 4th element now.
- Element4: Hash(103) = 103%6 = 1, so Element4 will be stored at 1st Index in this newly resized HashTable.

Rehashing Steps

- For each addition of a new entry to the map, check the current load factor.
- If it's greater than its pre-defined value, then Rehash.
- For Rehash, make a new array of double the previous size and make it the new bucket array.
- Then traverse to each element in the old bucketArray and insert them back so as to insert it into the new larger bucket array.

- If you are going to store a really large number of elements in the HashTable then it is always good to create a HashTable with sufficient capacity upfront as this is more efficient than letting it perform automatic rehashing