Hash Functions and Hash Tables

What is Hashing?

- Hashing is the **process of mapping large amount of data item to smaller table with the help of hashing function.**
- Hashing is also known as **Hashing Algorithm** or **Message Digest Function**.
- It is a technique **to convert a range of key values into a range of indexes of an array.**
- It is **used to facilitate the next level searching method** when compared with the linear or binary search.
- Hashing allows to **update and retrieve any data entry in a constant time** O(1).
- Constant time O(1) means the operation does not depend on the size of the data.
- Hashing is used with a database to enable items to be retrieved more quickly.
- It is used in the **encryption and decryption** of digital signatures.

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.

Hashing is implemented in two steps:

1. **An element is converted into an integer by using a hash function. This element can be used as an index to store the original element, which falls into the hash table.**
2. The **element is stored in the hash table where it can be quickly retrieved using hashed key**.
    - hash = hashfunc(key)
    - index = hash % array_size

In this method, the hash is independent of the array size and it is then reduced to an index (a number between 0 and array_size − 1) by using the modulo operator (%).

What is Hash Function?

- A **fixed process converts a key to a hash key is known as a Hash Function.**
- This **function takes a key and maps it to a value of a certain length which is called a Hash value or Hash.**
- **Hash value represents the original string of characters, but it is normally smaller than the original.**
- It transfers the digital signature and then both hash value and signature are sent to the receiver. Receiver uses the same hash function to generate the hash value and then compares it to that received with the message.
- If the hash values are same, the message is transmitted without errors.

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:

- **Easy to compute:** It should be easy to compute and must not become an algorithm in itself.
- **Uniform distribution**: It should provide a uniform distribution across the hash table and should not result in clustering.
- **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.**

What is Hash Table?

- **Hash table or hash map is a data structure used to store key-value pairs.**
- It is a collection of items stored to make **it easy to find them later**.
- It **uses a hash function to compute an index into an array of buckets or slots from which the desired value can be found.**
- It is an **array of list where each list is known as bucket**.
- It **contains value based on the key.**
- Hash table is used to implement the map interface and extends Dictionary class.
- **Hash table is synchronized and contains only unique elements**.

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)**.

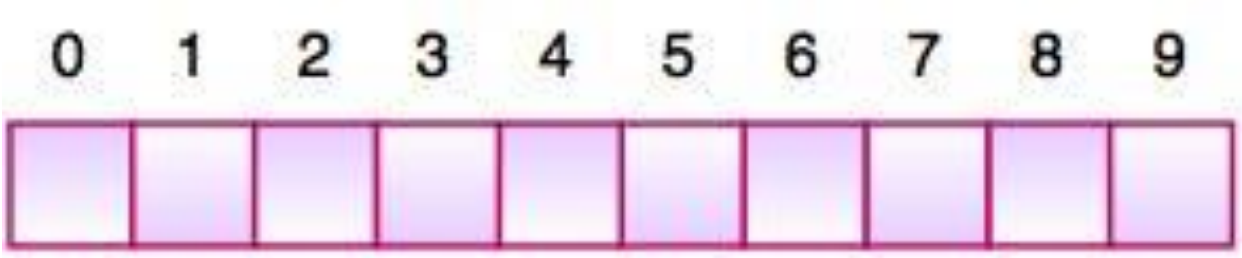


Fig. Hash Table

- The above figure shows the hash table with the size of n = 10. **Each position of the hash table is called as Slot**. In the above hash table, there are **n slots in the table**, names = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}. Slot 0, slot 1, slot 2 and so on. **Hash table contains no items, so every slot is empty.**
- As we know **the mapping between an item and the slot where item belongs in the hash table is called the hash function.** The hash function takes any item in the collection and returns an integer in the range of slot names between 0 to n-1.

Suppose we have integer items {26, 70, 18, 31, 54, 93}. One common method of determining a hash key is the division method of hashing and the formula is :

Hash Key = Key Value % Number of Slots in the Table

or

Hash Key = element % Number of Slots in the Table

or

Hash Key = value % Number of Slots in the Table

- Division method or reminder method takes an item and divides it by the table size and returns the remainder as its hash value.

Data Item	Value % No. of Slots	Hash Value
26	26 % 10 = 6	6
70	70 % 10 = 0	0
18	18 % 10 = 8	8
31	31 % 10 = 1	1
54	54 % 10 = 4	4
93	93 % 10 = 3	3

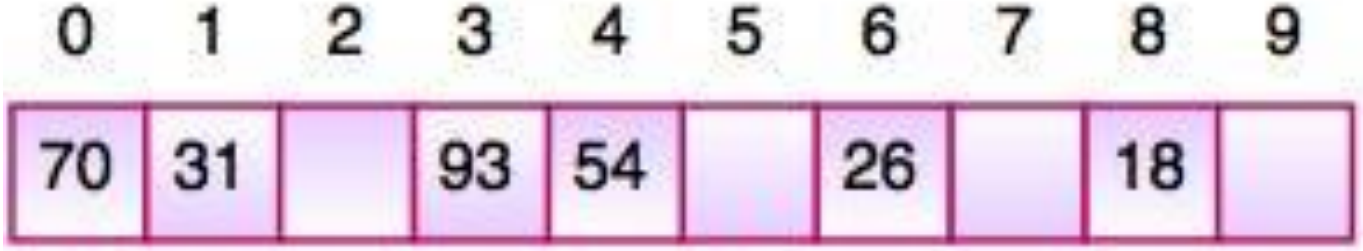


Fig. Hash Table

- **After computing the hash values, we can insert each item into the hash table at the designated position a**s shown in the above figure. In the hash table, 6 of the 10 slots are occupied, it is referred to as the load factor and denoted by, λ = No. of items / table size. For example , λ = 6/10.
- **It is easy to search for an item using hash function where it computes the slot name for the item and then checks the hash table to see if it is present.**
- Constant amount of time O(1) is required to compute the hash value and index of the hash table at that location.

Hash Table

Here all strings are sorted at same index

Index				
0				
1				
2	abcdef	bcdefa	cdefab	defabc
3				
4				
-				
-				
-				
-				

Here, it will take **O(n)** time (where n is the number of strings) to access a specific string. This shows that the hash function is not a good hash function.

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 + 101*3 + 102*4 + 97*5 + 98*6)%2069	14
defabc	(100*1 + 101*2 + 102*3 + 97*4 + 98*5 + 99*6)%2069	11

Hash Table

Here all strings are stored at different indices

Index	
0	
1	
-	
-	
-	
11	defabc
12	
13	
14	cdefab
-	
-	
-	
-	
23	bcdefa
-	
-	
-	
38	abcdef
-	
-	

Hash table is a type of data structure which is used for storing and accessing data very quickly. Insertion of data in a table is based on a key value. **Hence every entry in the hash table is defined with some key.** By using this key data can be searched in the hash table by **few key comparisons** and then searching time is dependent upon the size of the hash table.

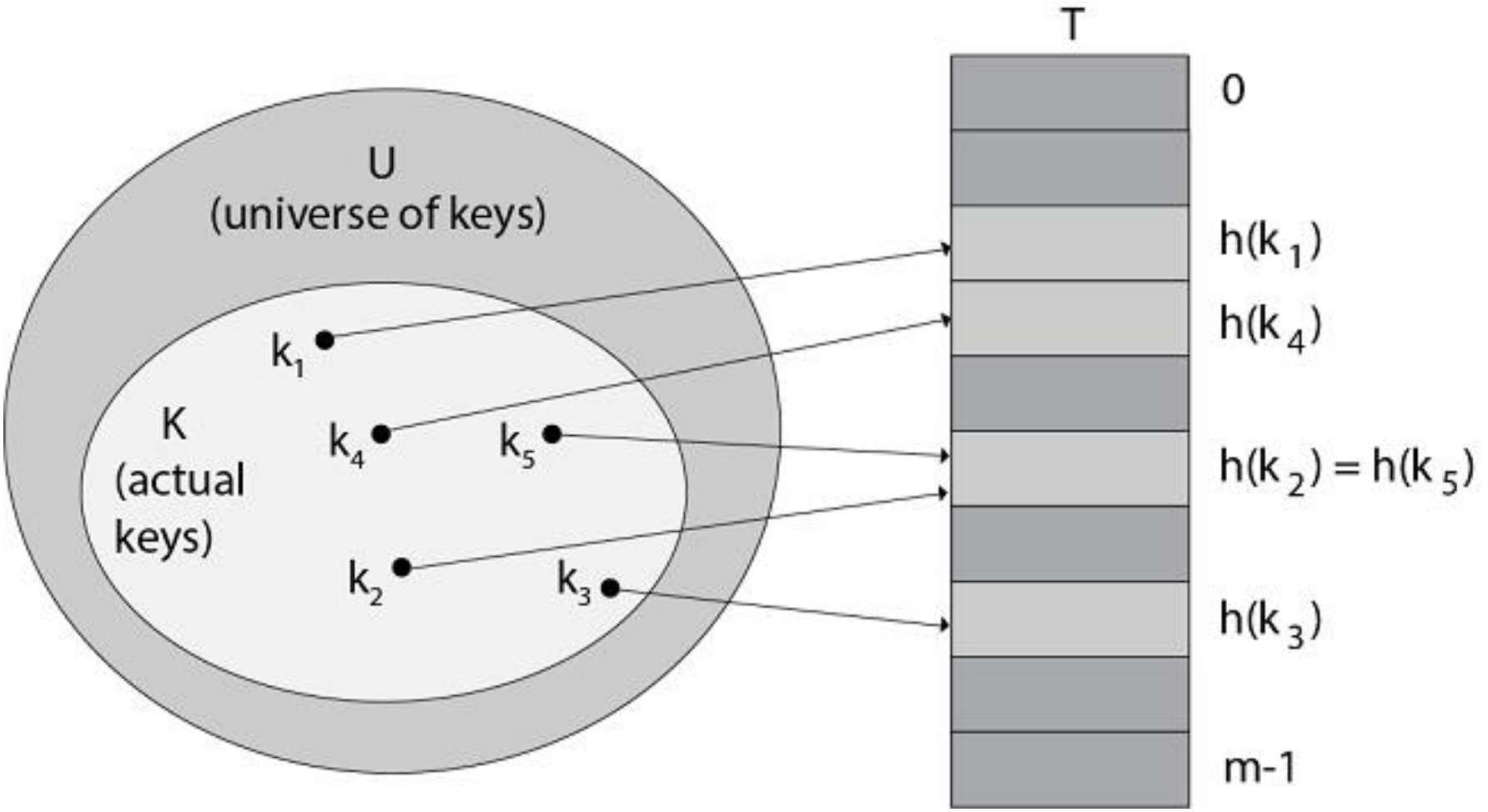


Figure: using a hash function h to map keys to hash-table slots. Because keys K2 and k5 map to the same slot, they collide.

Why use HashTable?

If **U** (Universe of keys) is large, storing a table T of size [U] may be impossible.

Set **k of keys may be small relative to U so space allocated for T will waste.**

So Hash Table requires less storage. Indirect addressing element with key k is stored in slot k with hashing it is stored in h (k) where h is a hash f^n and hash (k) is the value of key k. Hash f^n required array range.

Application of Hash Tables:

Some application of Hash Tables are:

- **Database System:** Specifically, those that are required efficient random access. Usually, database systems try to develop between two types of access methods: sequential and random. Hash Table is an integral part of efficient random access because they provide a way to locate data in a constant amount of time.
- **Symbol Tables:** The tables utilized by compilers to maintain data about symbols from a program. Compilers access information about symbols frequently. Therefore, it is essential that symbol tables be implemented very efficiently.
- **Data Dictionaries:** Data Structure that supports adding, deleting, and searching for data. Although the operation of hash tables and a data dictionary are similar, other Data Structures may be used to implement data dictionaries.
- **Associative Arrays:** Associative Arrays consist of data arranged so that n^{th} elements of one array correspond to the n^{th} element of another. Associative Arrays are helpful for indexing a logical grouping of data by several key fields.

Hash Function

Hash function is a function which is applied on a key by which it produces an integer, which can be used as an address of hash table. Hence one can use the same hash function for accessing the data from the hash table. In this the integer returned by the hash function is called hash key.

Hash Function is used to index the original value or key and then used later each time the data associated with the value or key is to be retrieved. Thus, hashing is always a one-way operation. There is no need to "reverse engineer" the hash function by analyzing the hashed values.

Types of hash function

There are various types of hash function which are used to place the data in a hash table,

1. Division method

Choose a number m smaller than the number of n of keys in k (The number m is usually chosen to be a prime number or a number without small divisors, since this frequently a minimum number of collisions).

$$h(k) = k \bmod m$$

$$h(k) = k \bmod m + 1$$

For Example: if the hash table has size m = 12 and the key is k = 100, then h (k) = 4. Since it requires only a single division operation, hashing by division is quite fast.

The hash function is:

In this the hash function is dependent upon the remainder of a division. For example:-if the record 52,68,99,84 is to be placed in a hash table and let us take the table size is 10.

Then:

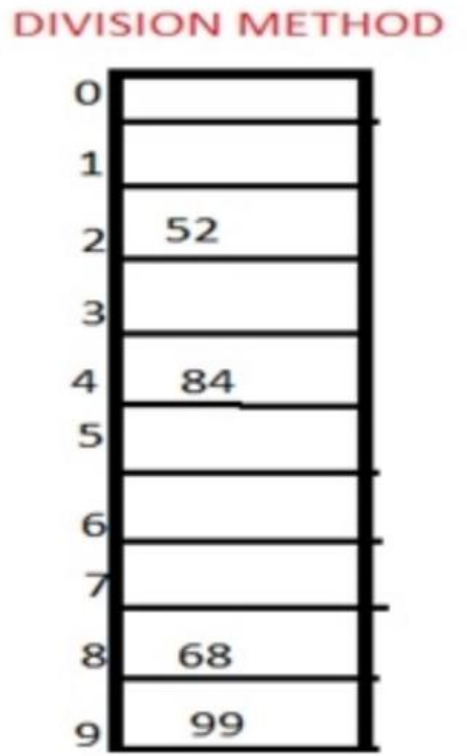
h(key)=record% table size.

2=52%10

8=68%10

9=99%10

4=84%10



2. Mid square method

The key k is squared. Then function H is defined by

H (k) = L

Where L is obtained by deleting digits from both ends of k^2. We emphasize that the same position of k^2 must be used for all of the keys.

In this method firstly key is squared and then mid part of the result is taken as the index. For example: consider that if we want to place a record of 3101 and the size of table is 1000. So 3101*3101=9616201 i.e. **h (3101) = 162 (middle 3 digit)**

3. Digit folding method

The key k is partitioned into a number of parts k_1, k_2.... k_n where each part except possibly the last, has the same number of digits as the required address.

Then the parts are added together, ignoring the last carry.

$H(k) = k^1 + k^2 + ... + k^n$

In this method the key is divided into separate parts and by using some simple operations these parts are combined to produce a hash key. For example: consider a record of 12465512 then it will be divided into parts i.e. 124, 655, 12. After dividing the parts combine these parts by adding it.

H(key)=124+655+12

=791

4. **Multiplication Method:**

The multiplication method for creating hash functions operates in two steps. First, we multiply the key k by a constant A in the range $0 < A < 1$ and extract the fractional part of kA. Then, we increase this value by m and take the floor of the result.

The hash function is:

$$h(k) = \lfloor m(k\,A \bmod 1) \rfloor$$

Where "k A mod 1" means the fractional part of k A, that is, k A -⌊k A⌋.

Example: Company has 68 employees, and each is assigned a unique four- digit employee number. Suppose L consist of 2- digit addresses: 00, 01, and 02....99. We apply the above hash functions to each of the following employee numbers:

3205, 7148, 2345

Division Method: Choose a Prime number m close to 99, such as m =97, Then

H (3205) = 4, H (7148) = 67, H (2345) = 17.

That is dividing 3205 by 97 gives a remainder of 4, dividing 7148 by 97 gives a remainder of 67, dividing 2345 by 97 gives a remainder of 17.

Mid-Square Method:

k =	3205	7148	2345
k^2=	10272025	51093904	5499025
h (k) =	72	93	99

Observe that fourth & fifth digits, counting from right are chosen for hash address.

Folding Method: Divide the key k into 2 parts and adding yields the following hash address:

H (3205) = 32 + 50 = 82

H (7148) = 71 + 84 = 55

H (2345) = 23 + 45 = 68

Characteristics of good hashing function

1. The hash function should generate different hash values for the similar string.
2. The hash function is easy to understand and simple to compute.
3. The hash function should produce the keys which will get distributed, uniformly over an array.
4. A number of collisions should be less while placing the data in the hash table.
5. The hash function is a perfect hash function when it uses all the input data.

Example

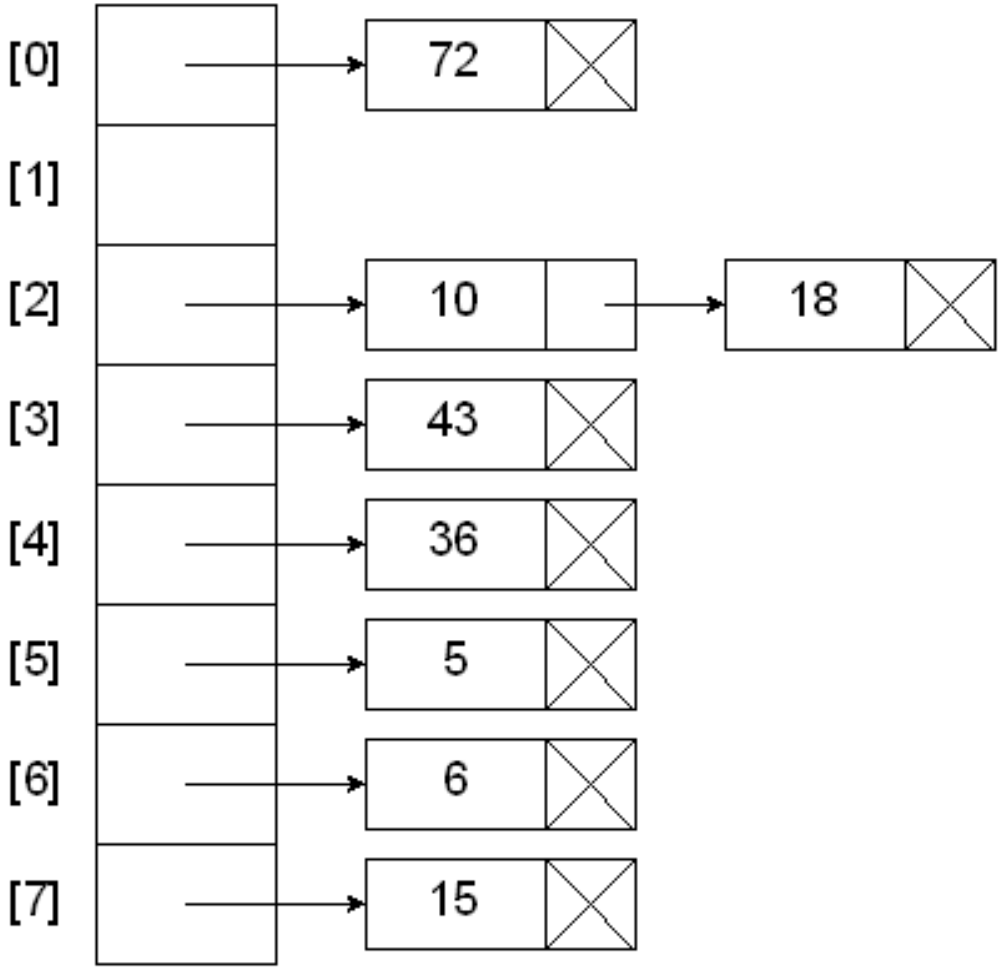
- Hash function
  - A mapping function that maps a key to a number in the range *0* to *TableSize -1*

```
int hashfunc(int integer_key)
{
    return integer_key % HASHTABLESIZE;
}
```

Hashing with Chains

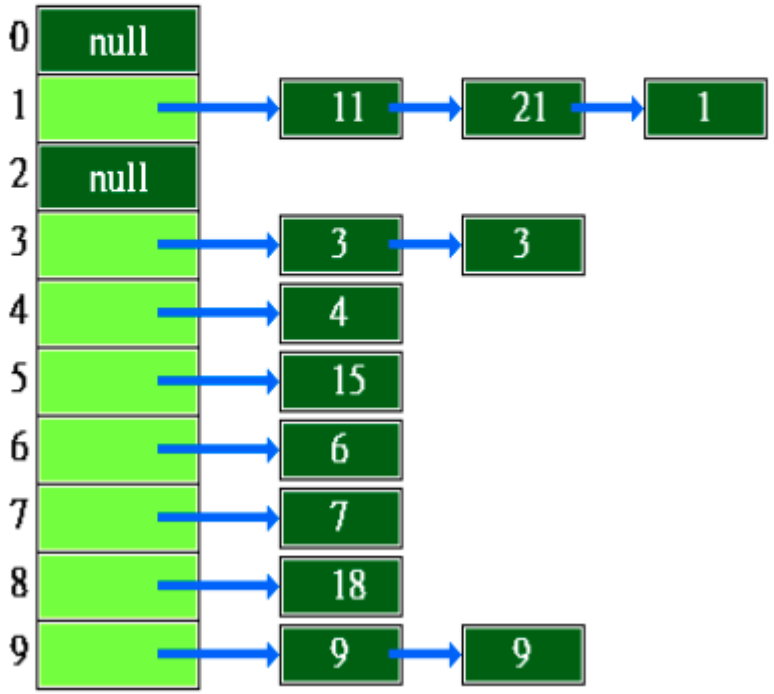
Hash key = key % table size

```
4  =  36 % 8
2  =  18 % 8
0  =  72 % 8
3  =  43 % 8
6  =   6 % 8
2  =  10 % 8
5  =   5 % 8
7  =  15 % 8
```



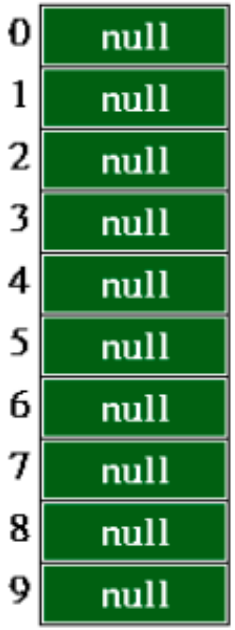
Hashing - Separate Chaining

- If two keys map to same value, the elements are chained together.



Example

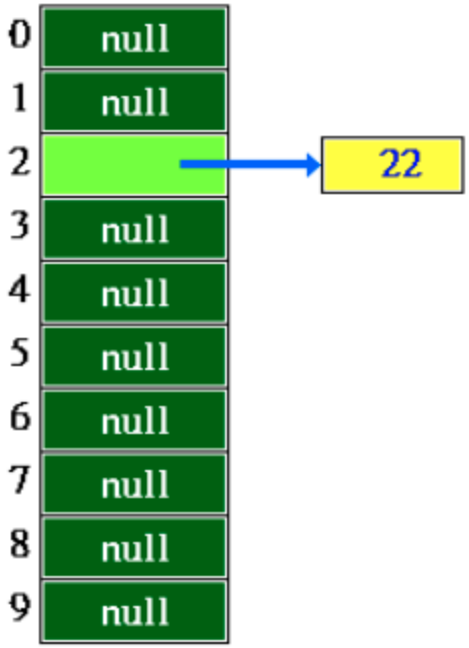
- Insert the following four keys 22 84 35 62 into hash table of size 10 using separate chaining.
- The hash function is key % 10



Initial hash table

- Insert the following four keys 22 84 35 62 into hash table of size 10 using separate chaining.
- The hash function is key % 10

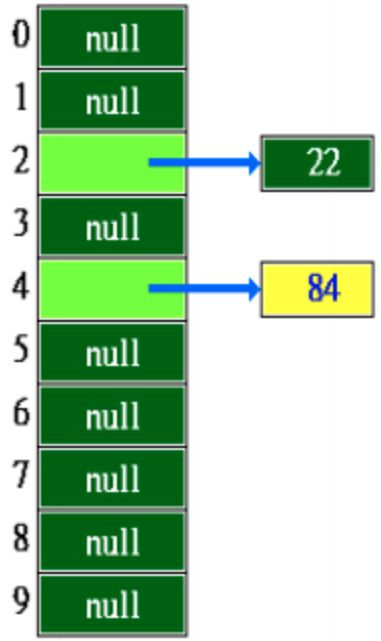
22 % 10 = 2



After insert 22

- Insert the following four keys 22 84 35 62 into hash table of size 10 using separate chaining.
- The hash function is key % 10

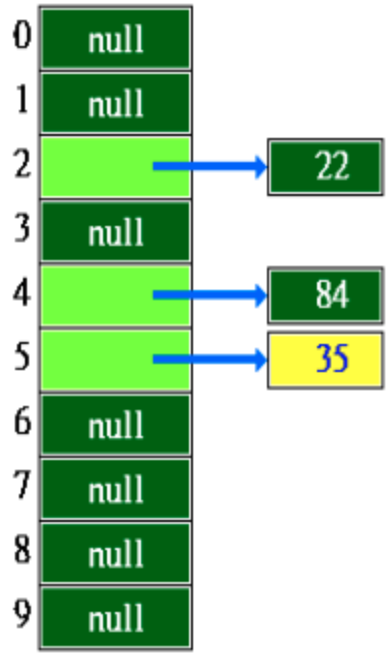
84 % 10 = 4



After insert 84

- Insert the following four keys 22 84 35 62 into hash table of size 10 using separate chaining.
- The hash function is key % 10

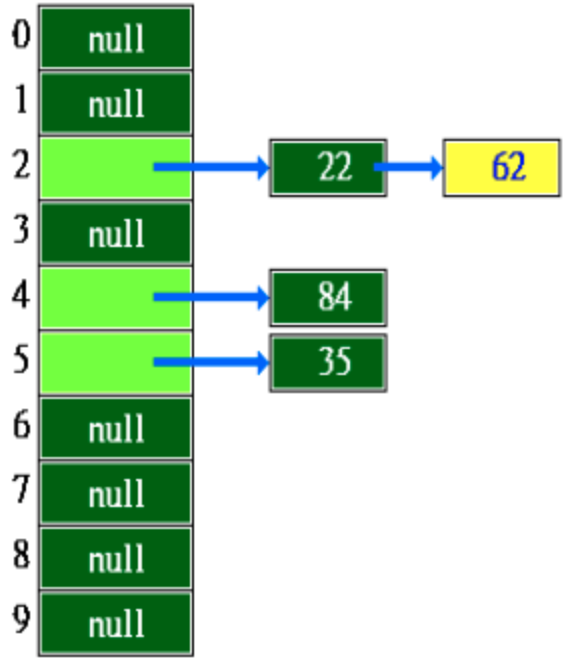
35 % 10 = 5



After insert 35

- Insert the following four keys 22 84 35 62 into hash table of size 10 using separate chaining.
- The hash function is key % 10

62 % 10 = 2



After insert 62

What is Collision?

Since a hash function gets us a small number for a key which is a big integer or string, there is a possibility that two keys result in the same value. **The situation where a newly inserted key maps to an already occupied slot in the hash table is called collision and must be handled using some collision handling technique.**

It is a situation in which the hash function returns the same hash key for more than one record, it is called as collision. Sometimes when we are going to resolve the collision it may lead to a overflow condition and this overflow and collision condition makes the poor hash function.

What are the chances of collisions with large table?

Collisions are very likely even if we have big table to store keys. An important observation is Birthday Paradox. With only 23 persons, the probability that two people have the same birthday is 50%.

How to handle Collisions?

There are mainly two methods to handle collision:

1) **Separate Chaining**

2) **Open Addressing**

Collision resolution technique

If there is a problem of collision occurs then it can be handled by apply some technique. These techniques are called as **collision resolution techniques.** There are generally four techniques which are described below.

Following are the ways to handle collisions:

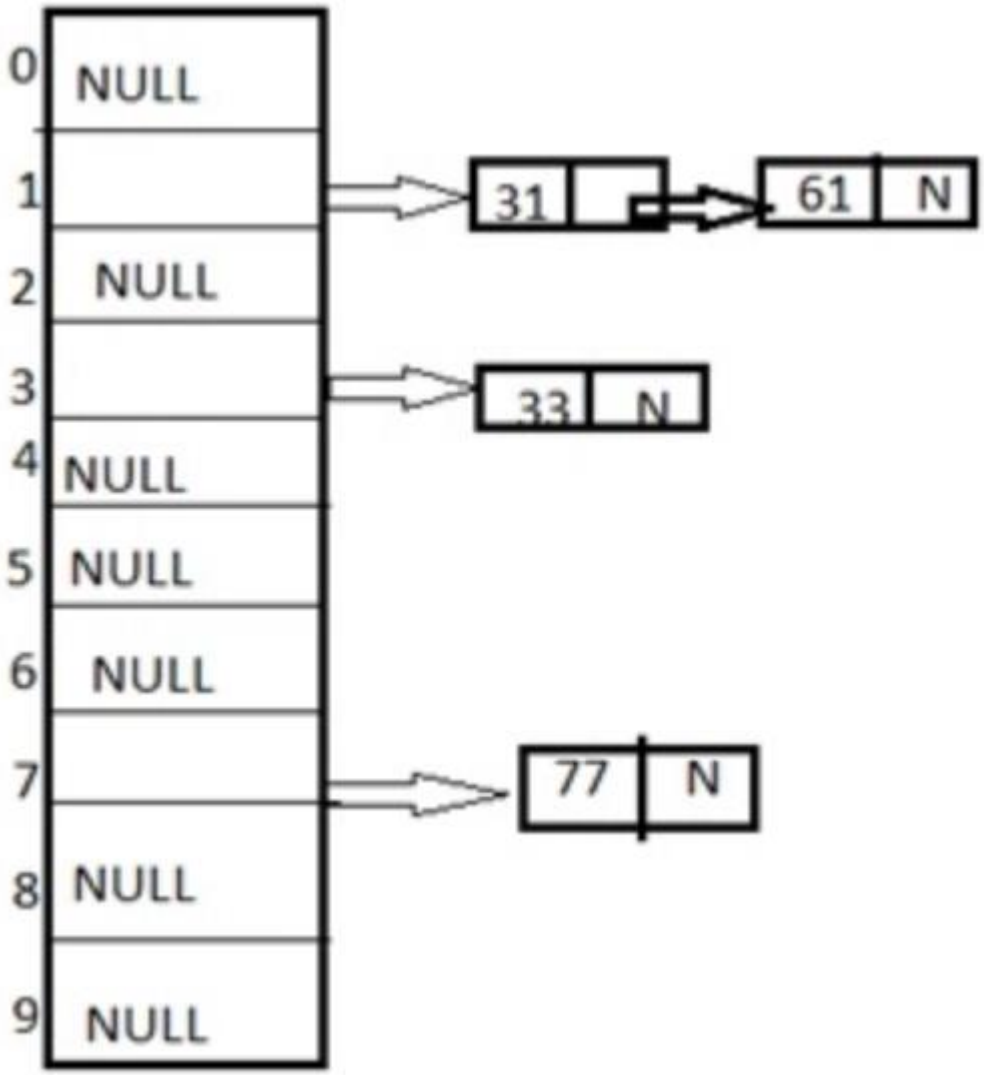
Chaining : The idea is to make each cell of **hash table point to a linked list of records** that have same hash function value. Chaining is simple, but requires additional memory outside the table.

Open Addressing : 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 one by one examine table slots until the desired element is found or it is clear that the element is not in the table.**

1) Chaining

It is a method in which additional field with data i.e. chain is introduced. A chain is maintained at the home bucket. In this when a collision occurs then a linked list is maintained for colliding data.

Example: Let us consider a hash table of size 10 and we apply a hash function of H(key)=key % size of table. Let us take the keys to be inserted are 31,33,77,61. In the above diagram we can see at same bucket 1 there are two records which are maintained by linked list or we can say by chaining method.

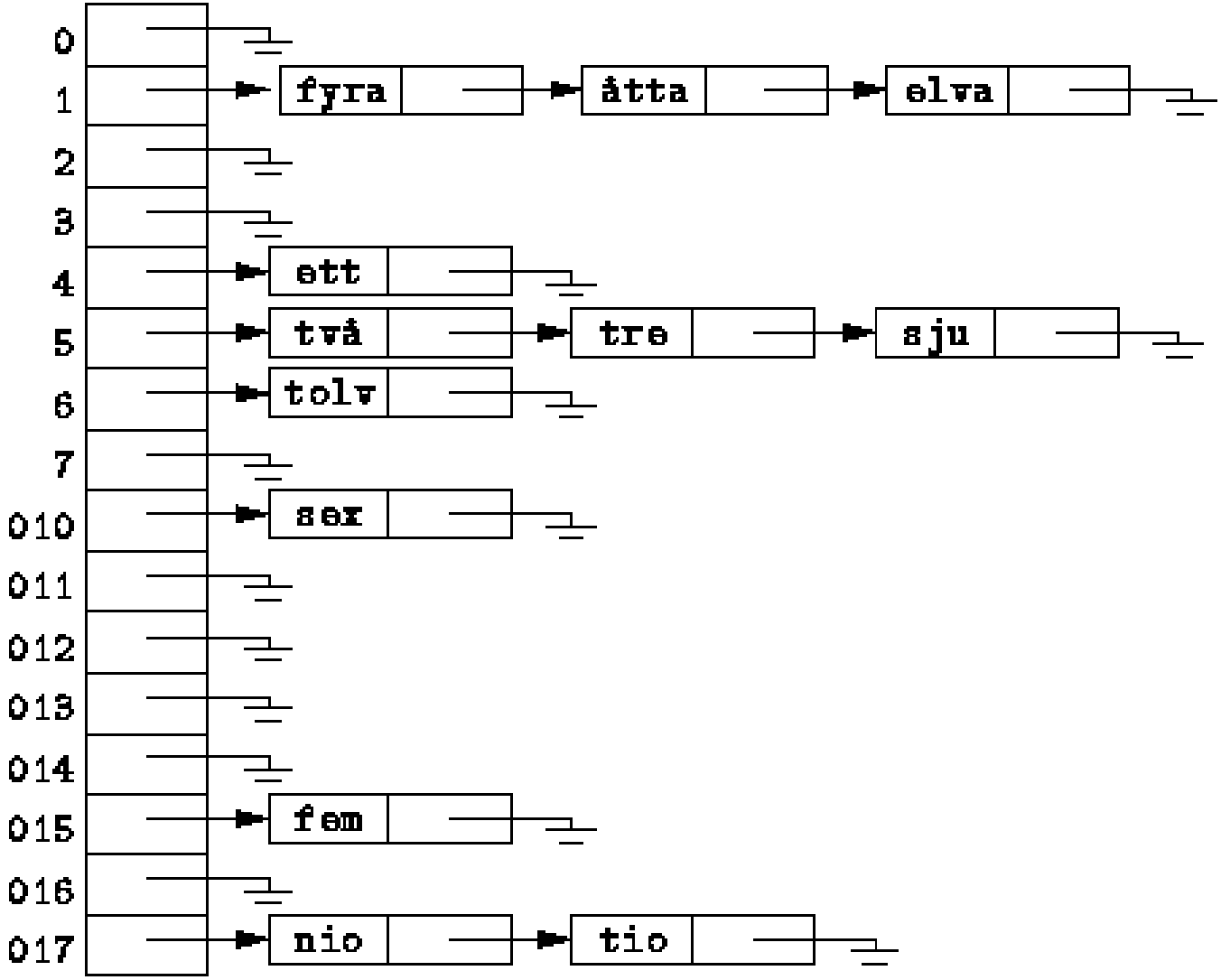

0
NULL
1
31
61
N
2
NULL
3
33
N
4
NULL
5
NULL
6
NULL
7
77
N
8
NULL
9
NULL

Scatter Tables

The separately chained hash table described in the preceding section is essentially a linked-list implementation. We have seen both linked-list and array-based implementations for all of the data structures considered so far and hash tables are no exception**. Array-based hash tables are called *scatter tables* .**

The essential idea behind a scatter table is that all of the information is stored within a fixed size array. Hashing is used to identify the position where an item should be stored. When a collision occurs, the colliding item is stored somewhere else in the array.

One of the motivations for using scatter tables can be seen by considering again the linked-list hash table shown in Figure . Since most of the linked lists are empty, much of the array is unused. At the same time, for each item that is added to the table, dynamic memory is consumed. Why not simply store the data in the unused array positions?



Hash table using separate chaining.

Separate Chaining

Above Hash table that uses *separate chaining* to resolve collisions. The hash table is implemented as an array of linked lists. To insert an item into the table, it is appended to one of the linked lists. The linked list to it is appended is determined by hashing that item.

Above figure illustrates an example in which there are *M*=16 linked lists. The twelve character strings "ett"-"tolv" have been inserted into the table using the hashed values and in the order given in Table . Notice that in this example since *M*=16, the linked list is selected by the least significant four bits of the hashed value given in Table . In effect, it is only the last letter of a string which determines the linked list in which that string appears.

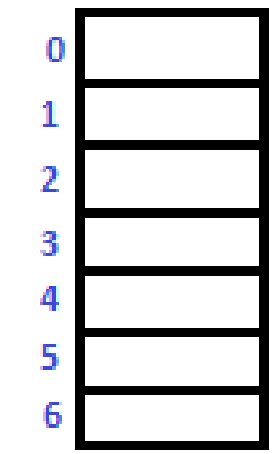
Storing all entries in a single big contiguous array is great for keeping the memory representation simple and fast. But it makes all of the operations on the hash table more complex. When inserting an entry, its bucket may be full, sending us to look at another bucket. That bucket itself may be occupied and so on. **This process of finding an available bucket is called probing and the order that you examine buckets is a probe sequence.**

There are a number of algorithms for determining which buckets to probe and how to decide which entry goes in which bucket. There's been a ton of research here because even slight tweaks can have a large performance impact. **And, on a data structure as heavily used as hash tables, that performance impact touches a very large number of real-world programs across a range of hardware capabilities.**

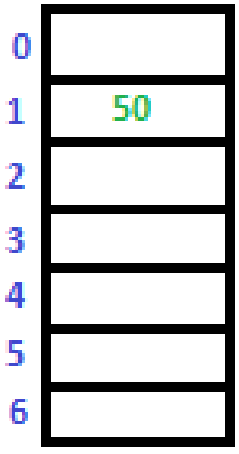
Separate Chaining:

The idea is to make each cell of hash table point to a linked list of records that have same hash function value.

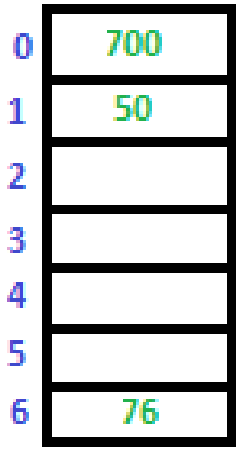
Let us consider a simple hash function as "**key mod 7**" and sequence of keys as 50, 700, 76, 85, 92, 73, 101.



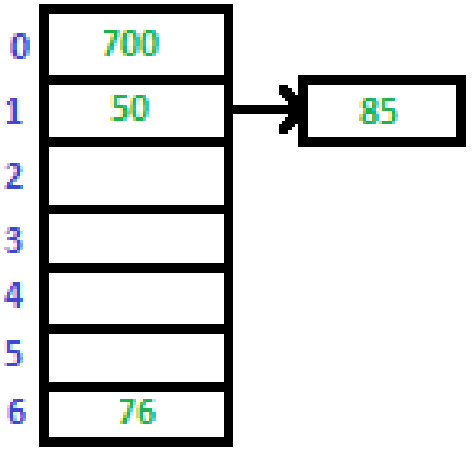
Initial Empty Table



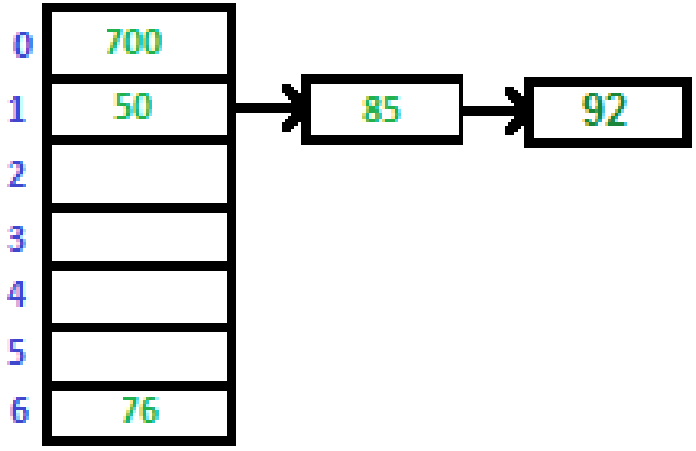
Insert 50



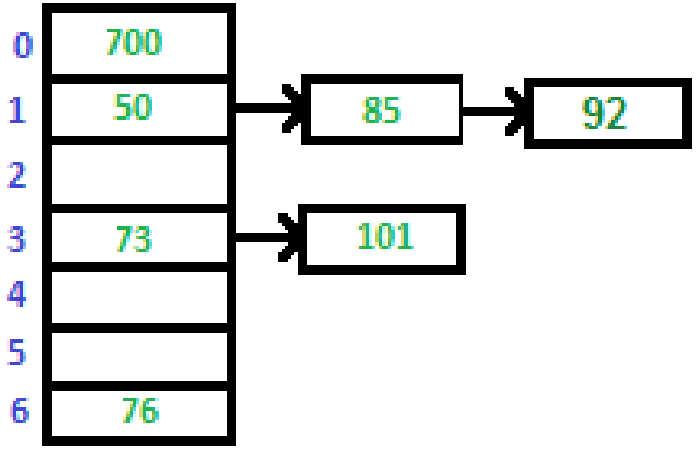
Insert 700 and 76



Insert 85: Collision
Occurs, add to chain



Inser 92 Collision
Occurs, add to chain



Insert 73 and 101

Advantages:

1) Simple to implement.

2) Hash table never fills up, we can always add more elements to the chain.

3) Less sensitive to the hash function or load factors.

4) It is mostly used when it is unknown how many and how frequently keys may be inserted or deleted.

Disadvantages:

1) Cache performance of chaining is not good as keys are stored using a linked list. Open addressing provides better cache performance as everything is stored in the same table.

2) Wastage of Space (Some Parts of hash table are never used)

3) If the chain becomes long, then search time can become O(n) in the worst case.

4) Uses extra space for links.

Open addressing

The other technique is called **“open addressing”** or (confusingly) **“closed hashing”**. **With this technique, all entries live directly in the bucket array, with one entry per bucket. If two entries collide in the same bucket, we find a different empty bucket to use instead.**

- Open addressing
  - Open addressing hash tables store the records directly within the array.
  - A hash collision is resolved by *probing*, or searching through alternate locations in the array.
    - Linear probing
    - Quadratic probing
    - Random probing
    - Double hashing

- Open addressing
  - if collision occurs, alternative cells are tried.
  - $h_0(X)$, $h_1(X)$, $h_2(X)$, ...

$$h_i(X) = (\text{Hash}(X) + F(i)) \bmod \textit{TableSize}$$

  - Linear probing : $F(i) = i$
  - Quadratic probing : $F(i) = i^2$
  - Double hashing : $F(i) = i * \text{Hash}_2(X)$

Open Addressing

Like separate chaining, open addressing is a method for handling collisions. In Open Addressing, **all elements are stored in the hash table itself**. So at any point, the size of the table must be greater than or equal to the total number of keys (Note that we can increase table size by copying old data if needed).

Insert(k): Keep probing until an empty slot is found. Once an empty slot is found, insert k.

Search(k): Keep probing until slot's key doesn't become equal to k or an empty slot is reached.

Delete(k): ***Delete operation is interesting***. If we simply delete a key, then the search may fail. So slots of deleted keys are marked specially as "deleted". The insert can insert an item in a deleted slot, but the search doesn't stop at a deleted slot.

Open Addressing is done in the following ways:

a) **Linear Probing:** In linear probing, we linearly probe for next slot. For example, the typical gap between two probes is 1 as taken in below example also.

let hash(x) be the slot index computed using a hash function and S be the table size

If slot hash(x) % S is full, then we try (hash(x) + 1) % S

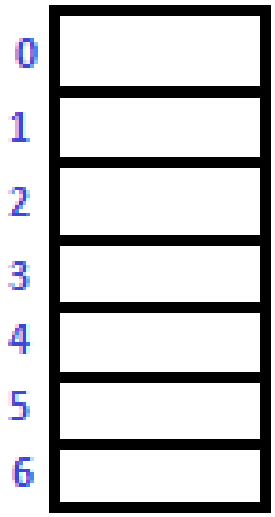
If (hash(x) + 1) % S is also full, then we try (hash(x) + 2) % S

If (hash(x) + 2) % S is also full, then we try (hash(x) + 3) % S

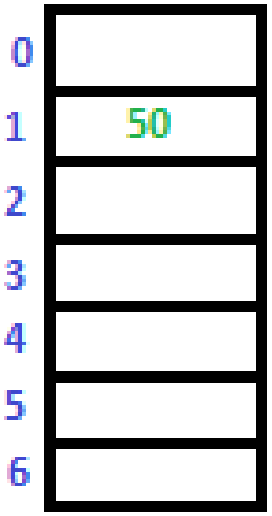
.....

.....

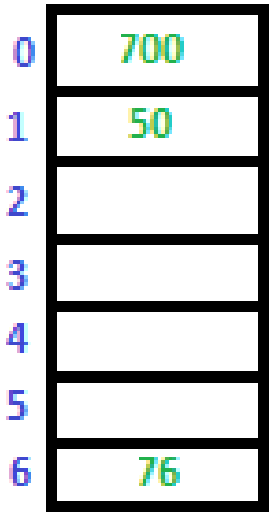
Let us consider a **simple hash function as "key mod 7"** and sequence of keys as 50, 700, 76, 85, 92, 73, 101.



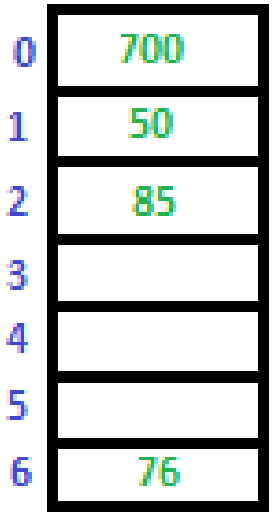
Initial Empty Table



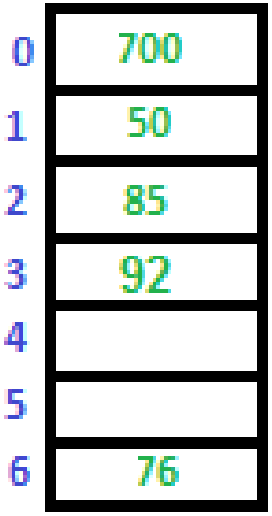
Insert 50



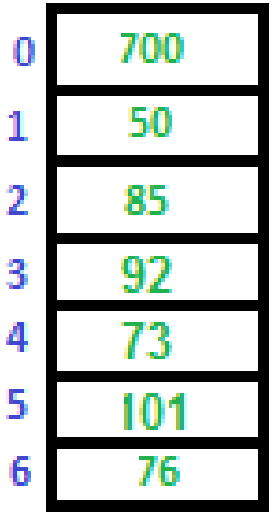
Insert 700 and 76



Insert 85: Collision Occurs, insert 85 at next free slot.



Insert 92, collision occurs as 50 is there at index 1. Insert at next free slot



Insert 73 and 101

- Linear probing
  - F(i) = i

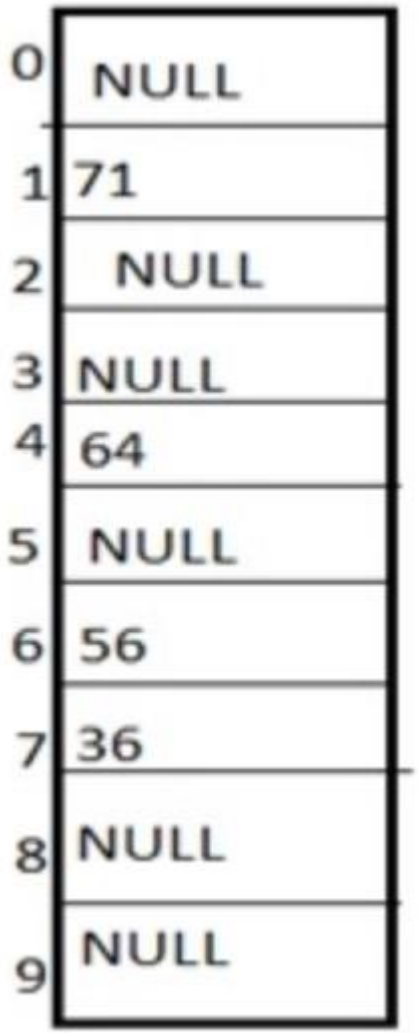
$h_i(X) = (\text{Hash}(X) + i) \bmod \textit{TableSize}$

$h_0(X) = (\text{Hash}(X) + 0) \bmod \textit{TableSize}$,
$h_1(X) = (\text{Hash}(X) + 1) \bmod \textit{TableSize}$,
$h_2(X) = (\text{Hash}(X) + 2) \bmod \textit{TableSize}$, ...

2) Linear probing

It is very easy and simple method to resolve or to handle the collision. In this collision can be solved by placing the second record linearly down, whenever the empty place is found. In this method there is a problem of clustering which means at some place block of a data is formed in a hash table.

Example: Let us consider a hash table of size 10 and hash function is defined as H(key)=key % table size. Consider that following keys are to be inserted that are 56,64,36,71.



In this diagram we can see that 56 and 36 need to be placed at same bucket but by linear probing technique the records linearly placed downward if place is empty i.e. it can be seen 36 is placed at index 7.

Hashing with Linear Probe

[0]	72
[1]	
[2]	18
[3]	43
[4]	36
[5]	
[6]	6
[7]	

Add the keys 10, 5, and 15 to the previous table .

Hash key = key % table size

2 = 10 % 8

5 = 5 % 8

7 = 15 % 8

[0]	72
[1]	15
[2]	18
[3]	43
[4]	36
[5]	10
[6]	6
[7]	5

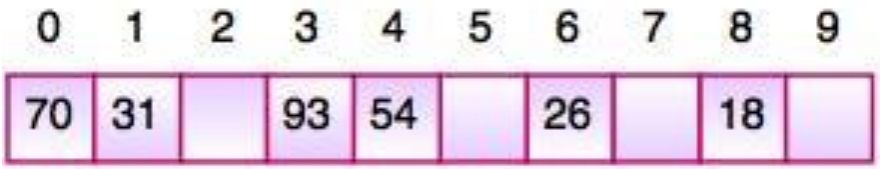


Fig. Hash Table

Linear Probing

- Take the above example, if we insert next item 40 in our collection, it would have a hash value of 0 (40 % 10 = 0). But 70 also had a hash value of 0, it becomes a problem. This problem is called as **Collision** or **Clash**. Collision creates a problem for hashing technique.
- **Linear probing is used for resolving the collisions in hash table**, data structures for maintaining a collection of key-value pairs.
- Linear probing was invented by Gene Amdahl, Elaine M. McGraw and Arthur Samuel in 1954 and analyzed by Donald Knuth in 1963.
- It is a component of open addressing scheme for using a hash table to solve the dictionary problem.

The simplest method is called Linear Probing.

Formula to compute linear probing is:

P = (1 + P) % (MOD) Table_size

Fig. Hash Table

If we insert next item 40 in our collection, it would have a hash value of 0 (40 % 10 = 0). But 70 also had a hash value of 0, it becomes a problem.

Linear probing solves this problem:

P = H(40)

44 % 10 = **0**

Position 0 is occupied by 70. so we look elsewhere for a position to store 40.

Using Linear Probing:

P= (P + 1) % table-size

0 + 1 % 10 = **1**

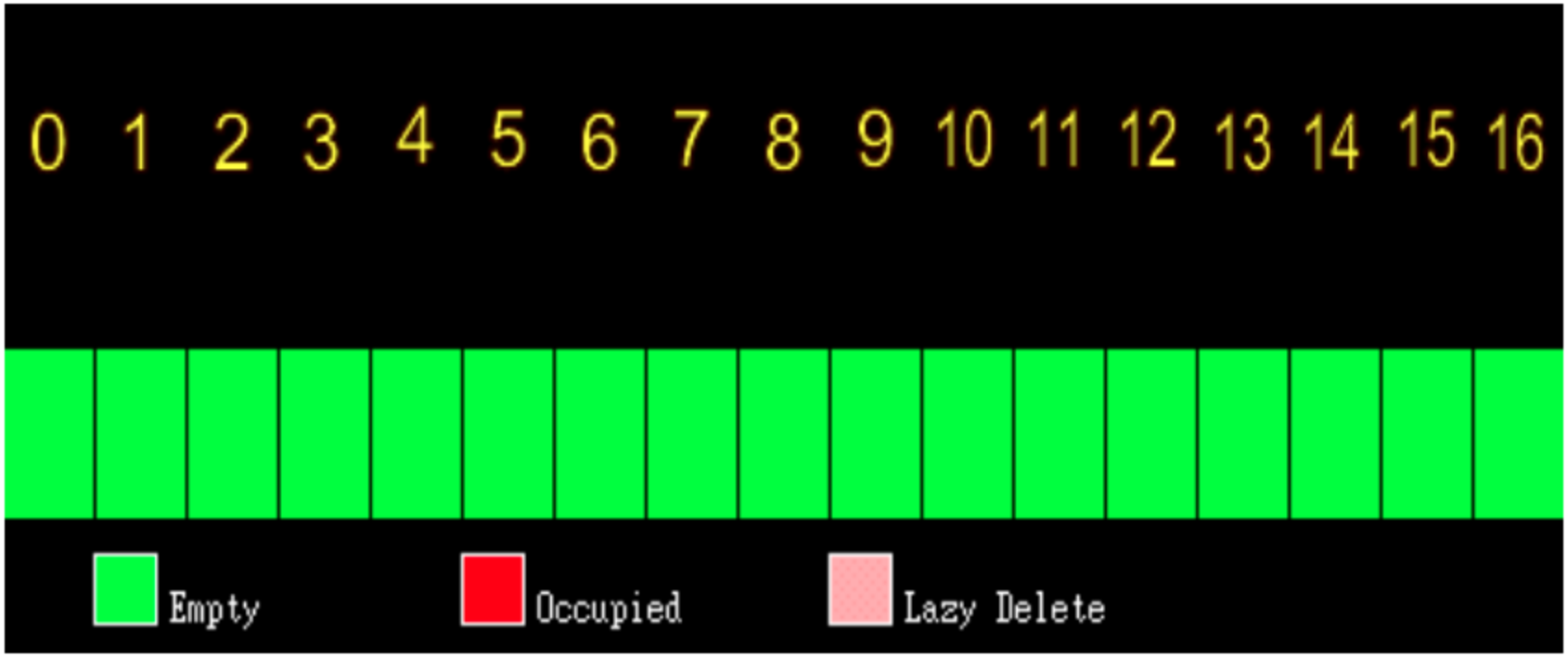
But, position 1 is occupied by 31, so we look elsewhere for a position to store 40.

Using linear probing, we try next position : 1 + 1 % 10 = 2

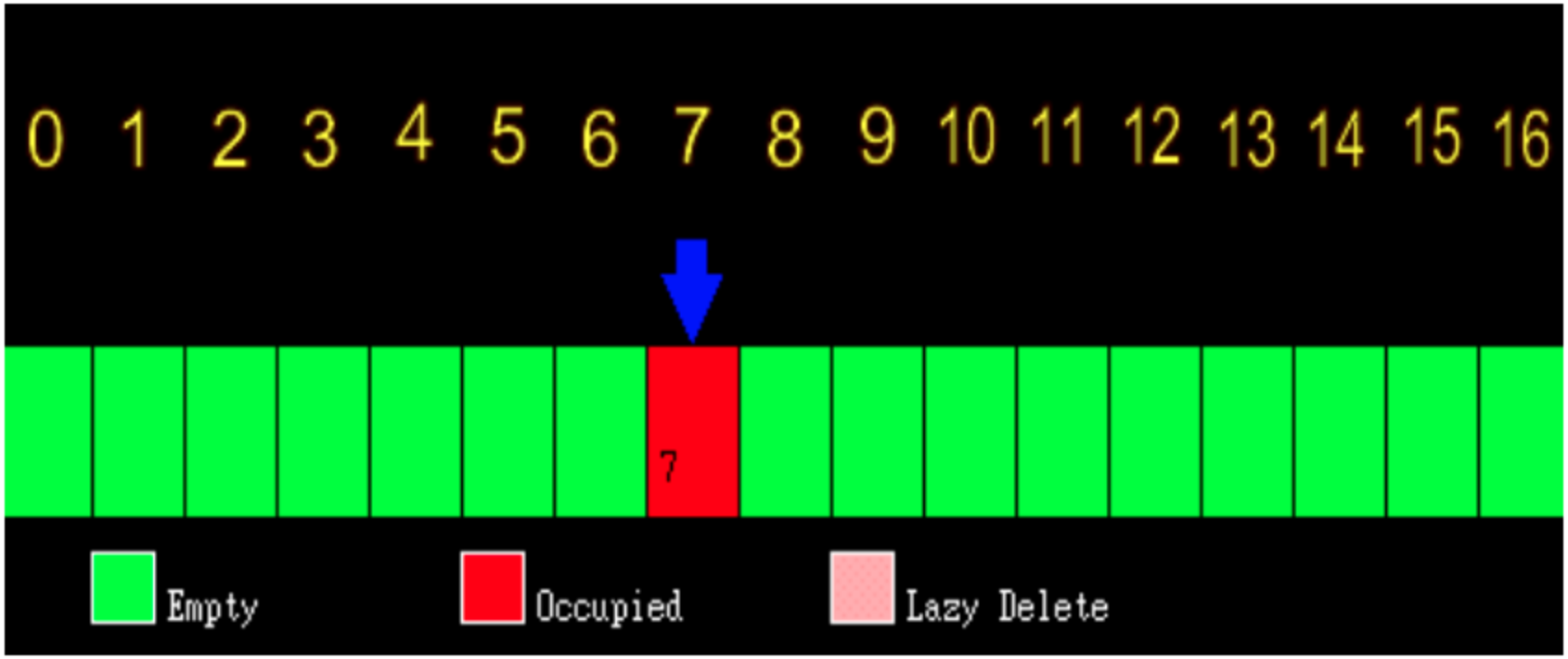
Position 2 is empty, so 40 is inserted there.

Fig. Hash Table

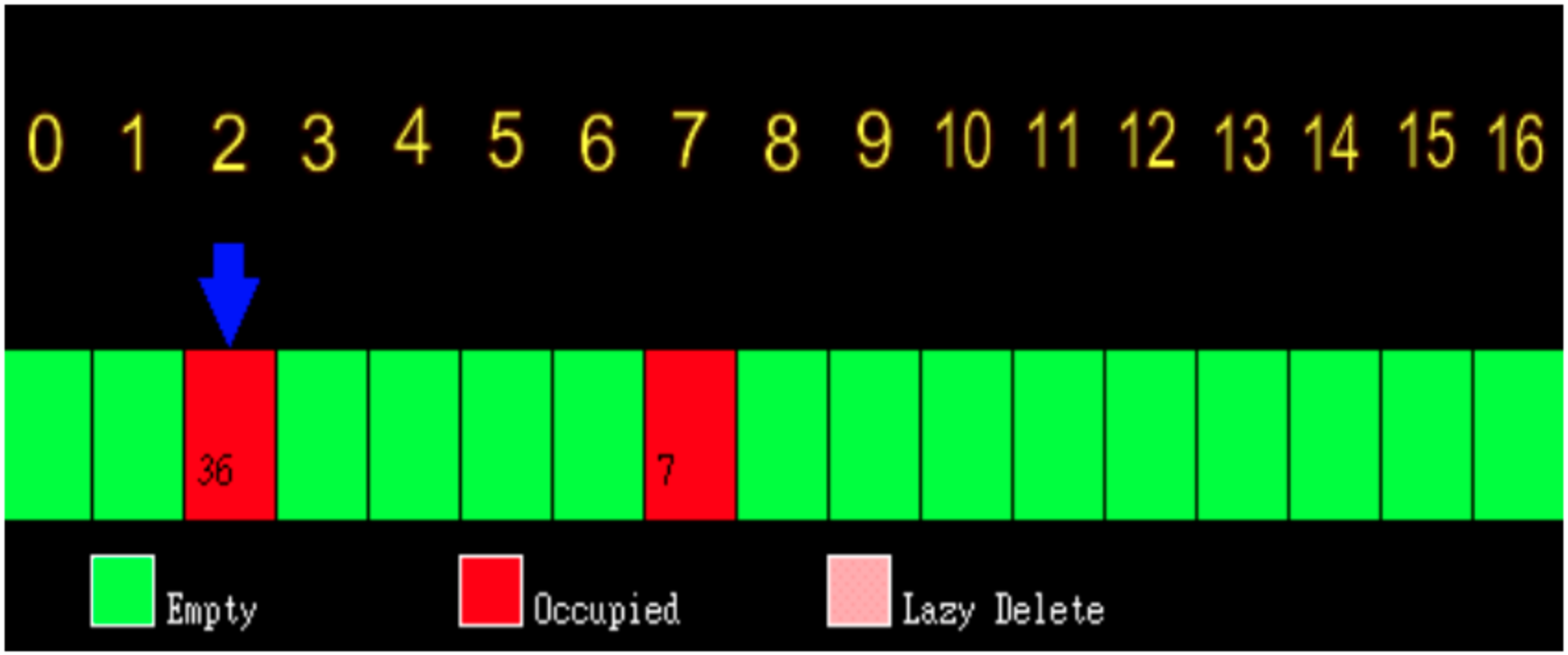
- Linear probing example
  - Initial hash table



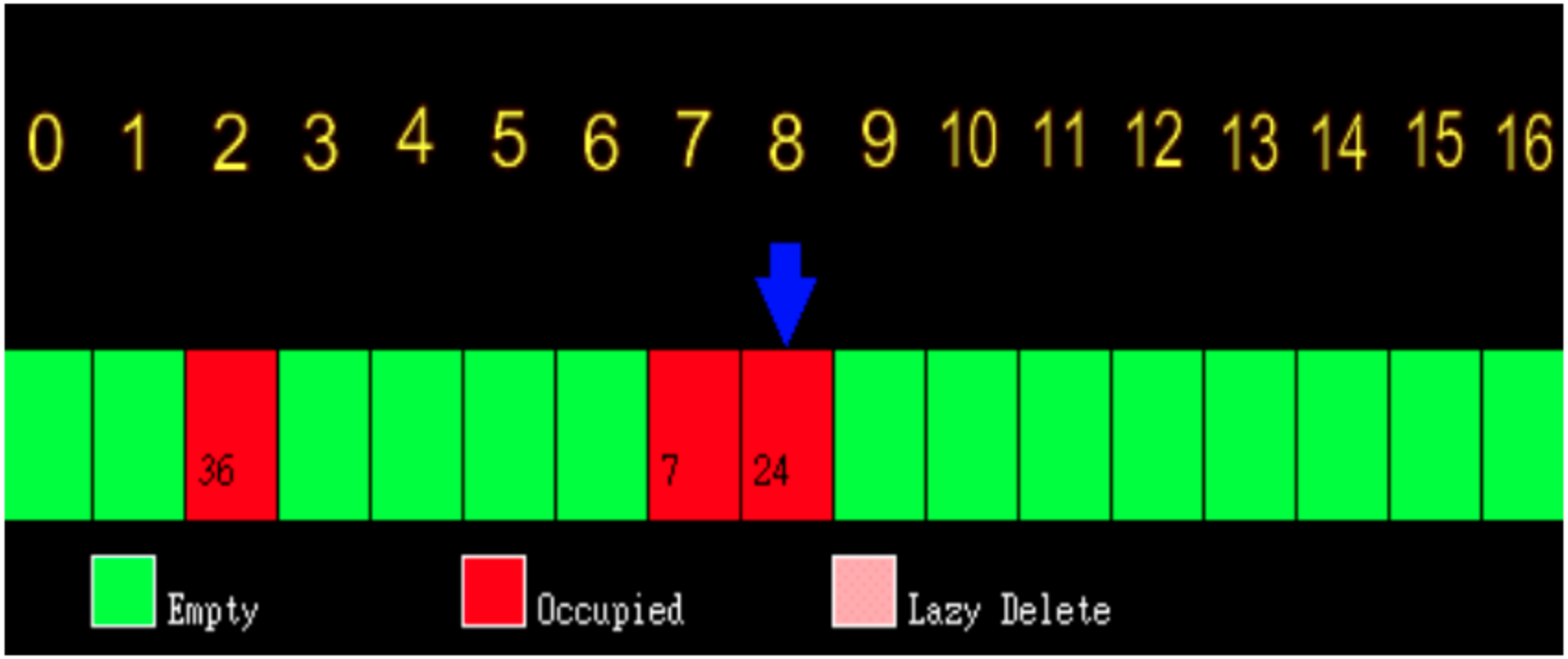
- Linear probing example
  - Insert 7 at $h_0(7)$ (7 mod 17) = 7



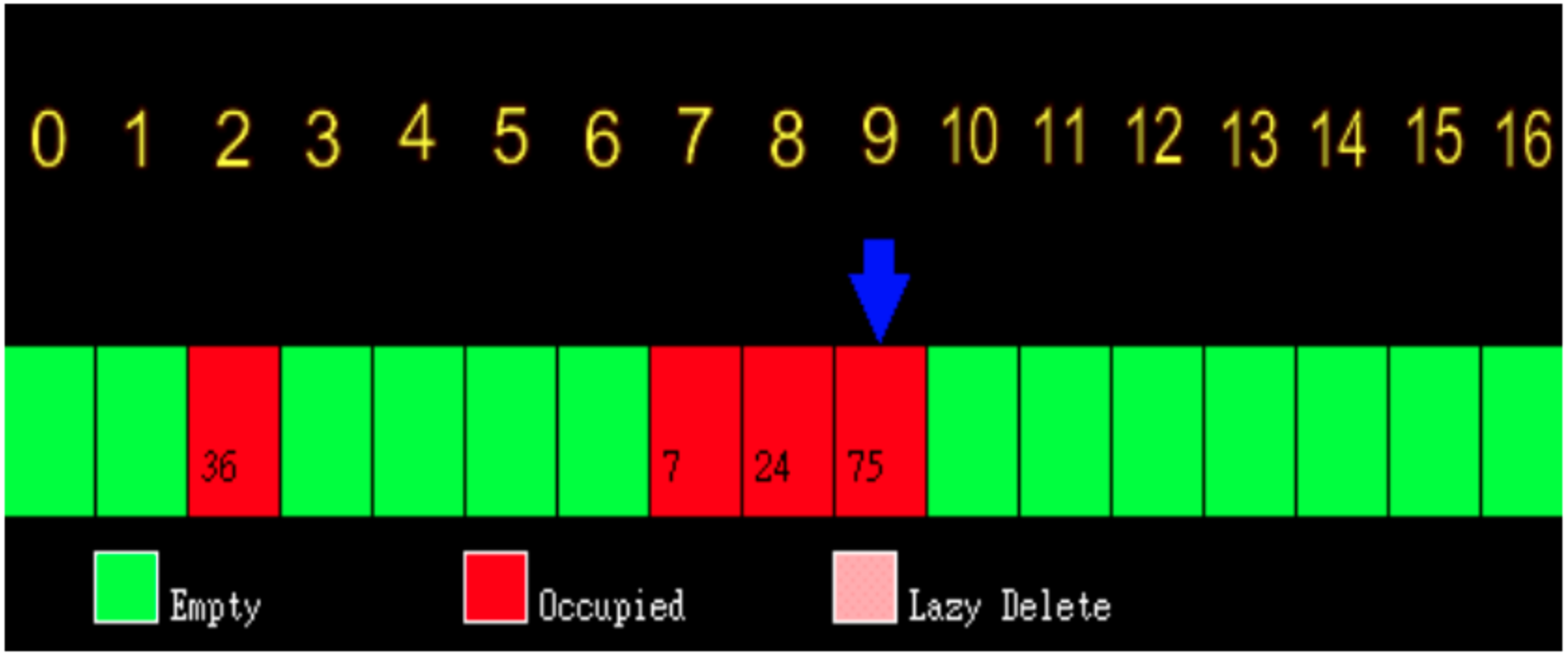
- Linear probing example
  - Insert 36 at $h_0(36)$ (36 mod 17) = 2



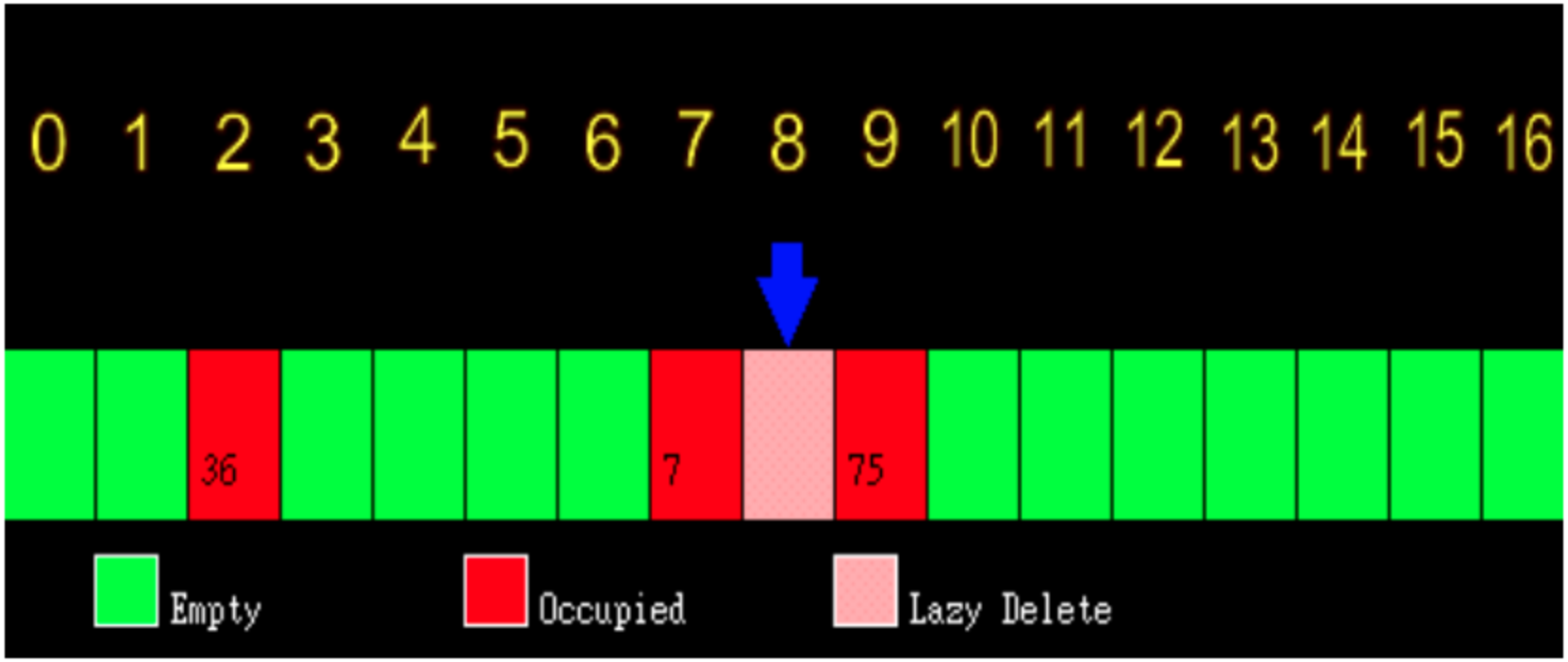
- Linear probing example
  - Insert 24 at $h_1(24)$ (24 mod 17) = 7



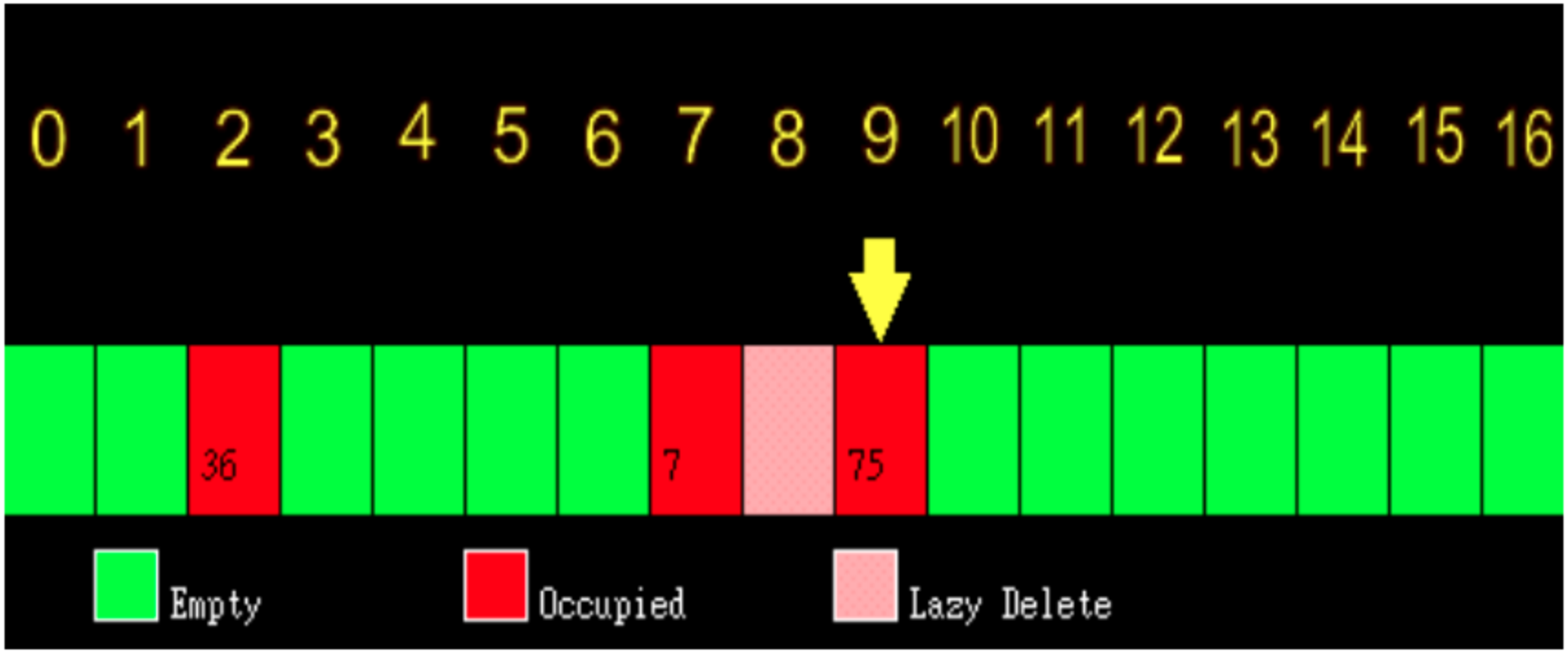
- Linear probing example
  - Insert 75 at $h_2(75)$ (75 mod 17) = 7



- Linear probing example
  - Delete 24



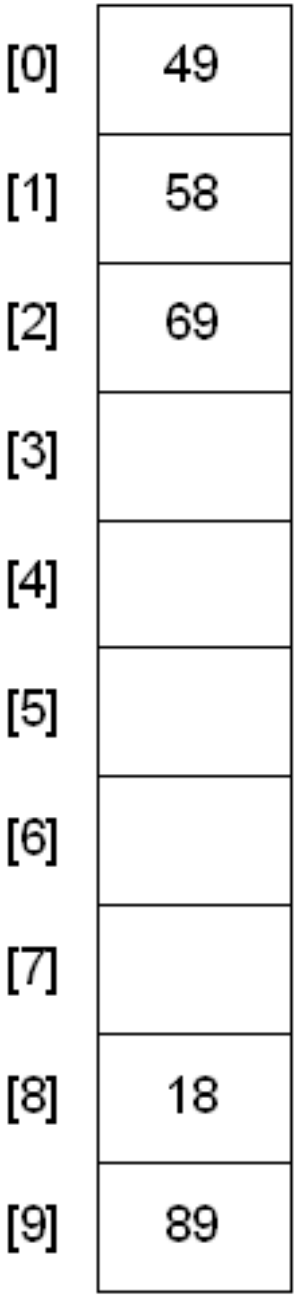
- Linear probing example
  - Find 75, found !



Challenges in Linear Probing :

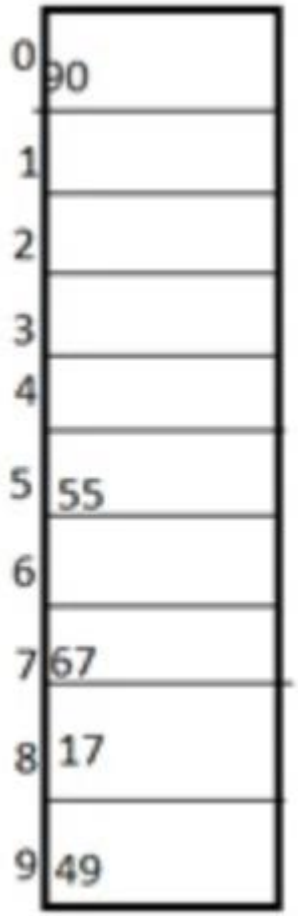
- **Primary Clustering:** One of the problems with linear probing is Primary clustering, **many consecutive elements form groups and it starts taking time to find a free slot or to search an element.**
- **Secondary Clustering*:*** Secondary clustering is less severe, **two records do only have the same collision chain(Probe Sequence) if their initial position is the same.**

A problem with the linear probe method is that it is possible for blocks of data to form when collisions are resolved. This is known as **primary clustering**. This means that any key that hashes into the cluster will require several attempts to resolve the collision. For example, insert the nodes 89, 18, 49, 58, and 69 into a hash table that holds 10 items using the division method:


[0]
49
[1]
58
[2]
69
[3]
[4]
[5]
[6]
[7]
[8]
18
[9]
89

3) Quadratic probing

This is a method in which solving of clustering problem is done. In this method the hash function is defined by the **H(key)=(H(key)+x*x)%table size**. Let us consider we have to insert following elements that are:-67, 90,55,17,49.



In this we can see if we insert 67, 90, and 55 it can be inserted easily but at case of 17 hash function is used in such a manner that :-(17+0*0)%10=7 (when x=0 it provide the index value 7 only) by making the increment in value of x. let x =1 so (17+1*1)%10=8.in this case bucket 8 is empty hence we will place 17 at index 8.

- Quadratic probing
  - $F(i) = i^2$

$h_i(X) = (\text{Hash}(X) + i^2) \bmod \textit{TableSize}$

$h_0(X) = (\text{Hash}(X) + 0^2) \bmod \textit{TableSize}$,
$h_1(X) = (\text{Hash}(X) + 1^2) \bmod \textit{TableSize}$,
$h_2(X) = (\text{Hash}(X) + 2^2) \bmod \textit{TableSize}$, ...

b) Quadratic Probing We look for i^2'th slot in i'th iteration.

let hash(x) be the slot index computed using hash function.

If slot hash(x) % S is full, then we try (hash(x) + 1*1) % S

If (hash(x) + 1*1) % S is also full, then we try (hash(x) + 2*2) % S

If (hash(x) + 2*2) % S is also full, then we try (hash(x) + 3*3) % S

.....

.....

Hashing with Quadratic Probe

89 % 10 = 9

18 % 10 = 8

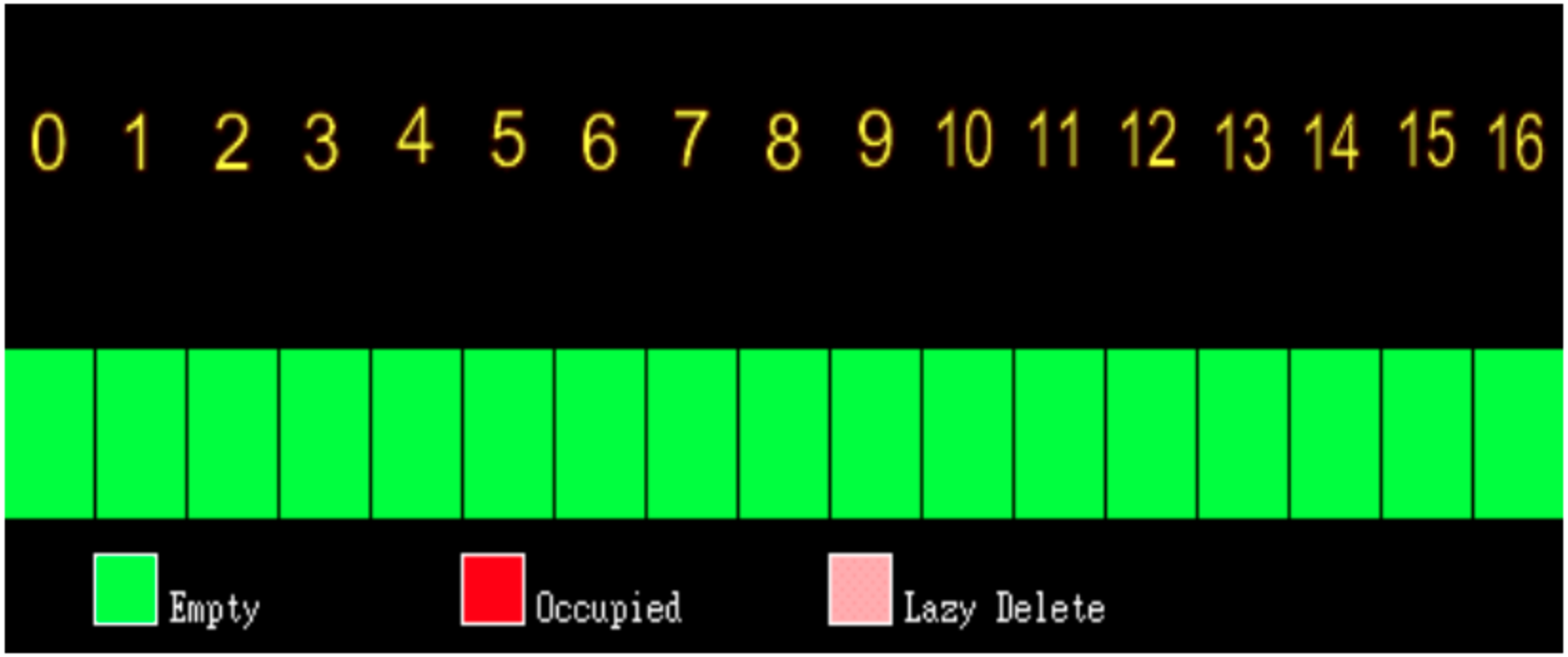
49 % 10 = 9 – 1 attempt needed – 1^2 = 1 spot

58 % 10 = 8 – 3 attempts – 3^2 = 9 spots

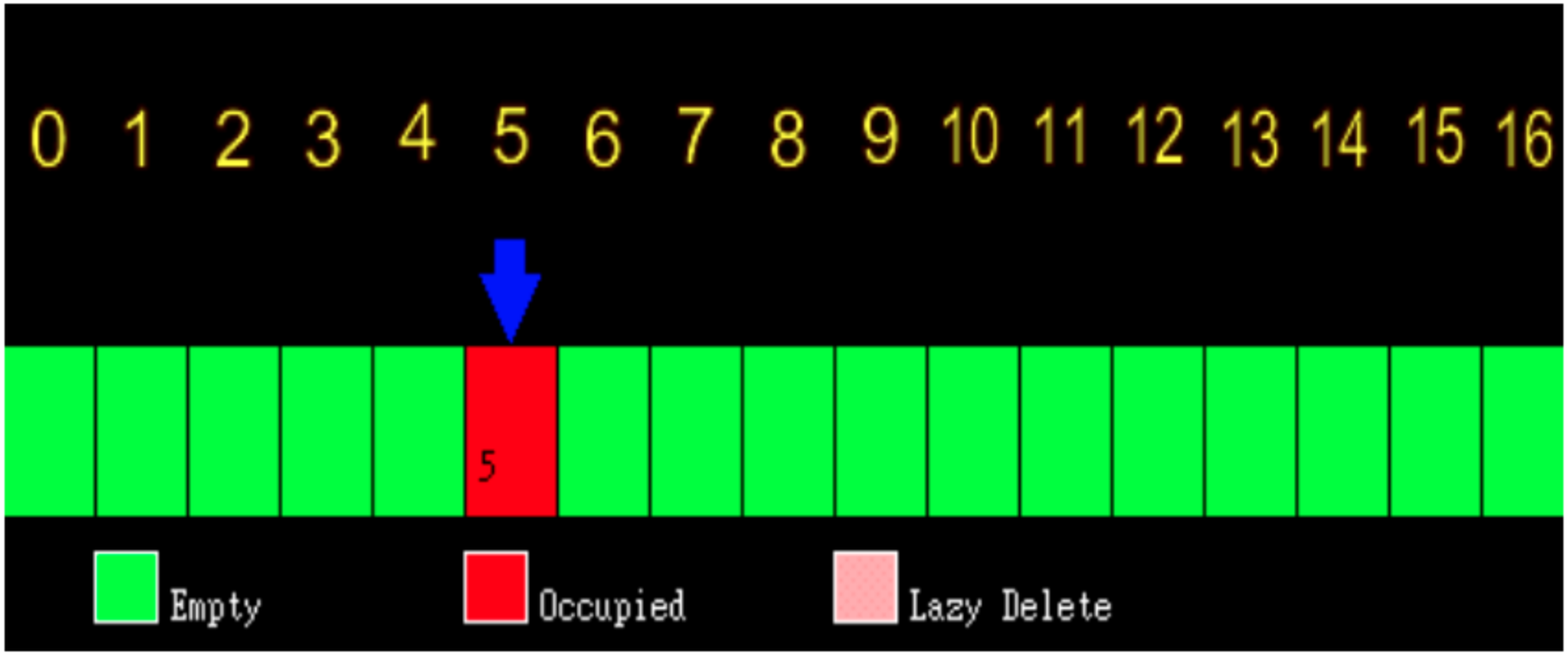
69 % 10 = 9 – 2 attempts – 2^2 = 4 spots

[0]	49
[1]	
[2]	
[3]	69
[4]	
[5]	
[6]	
[7]	58
[8]	18
[9]	89

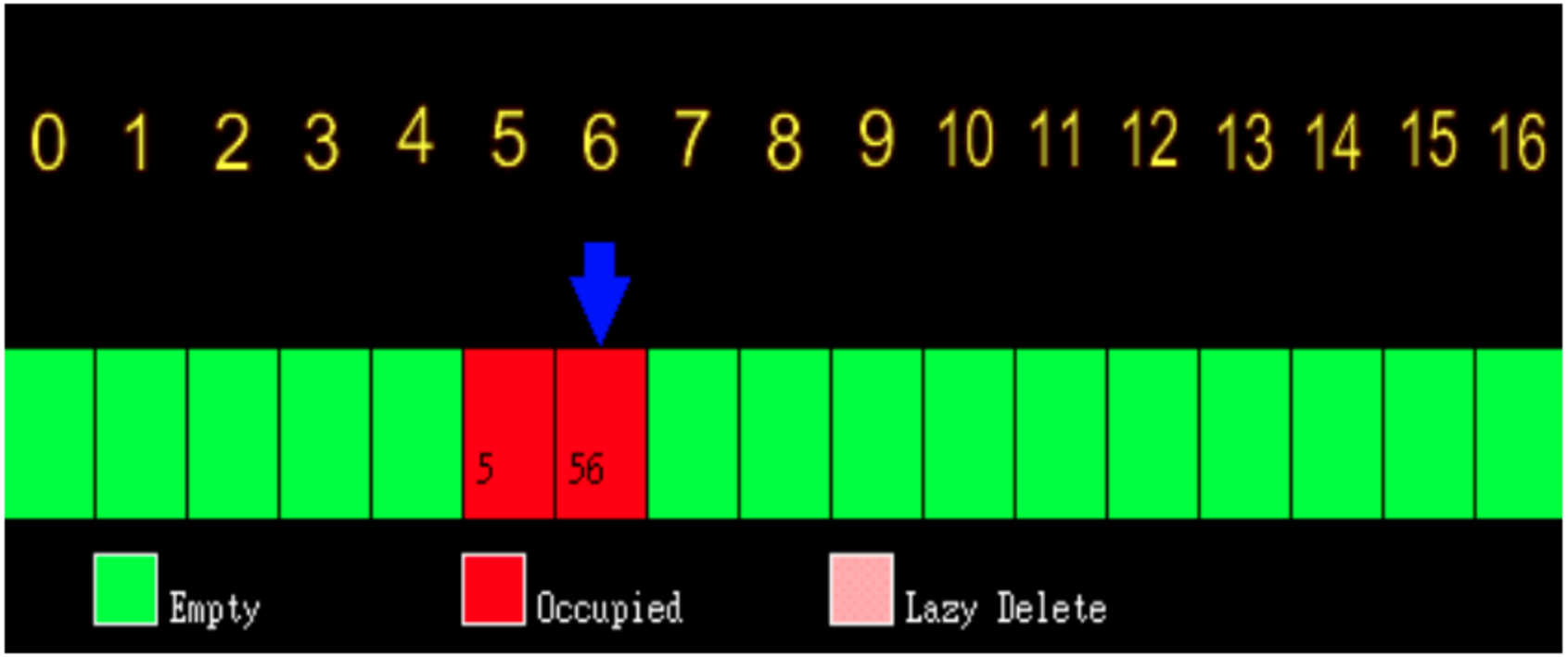
- Quadratic probing example
  - Initial hash table



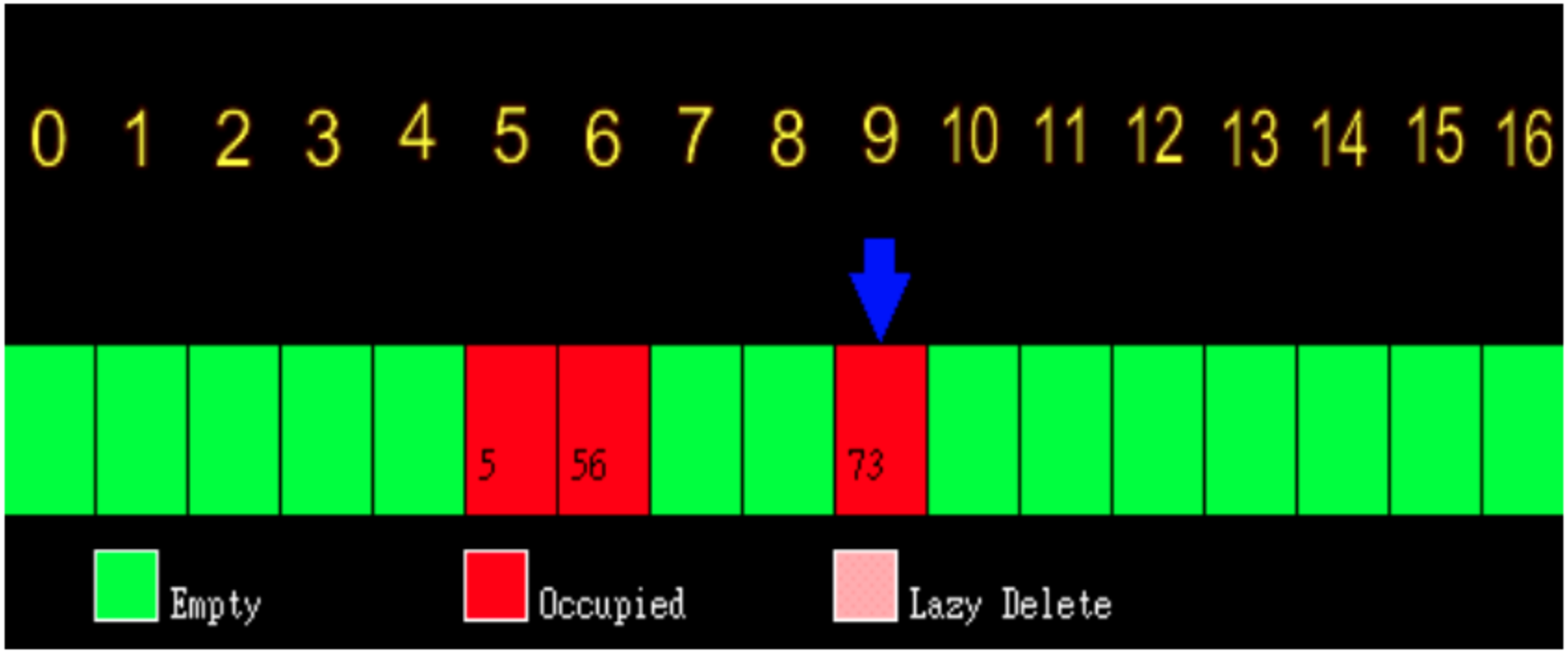
- Quadratic probing example
  - Insert 5 at $h_0(5)$ (5 mod 17) = 5



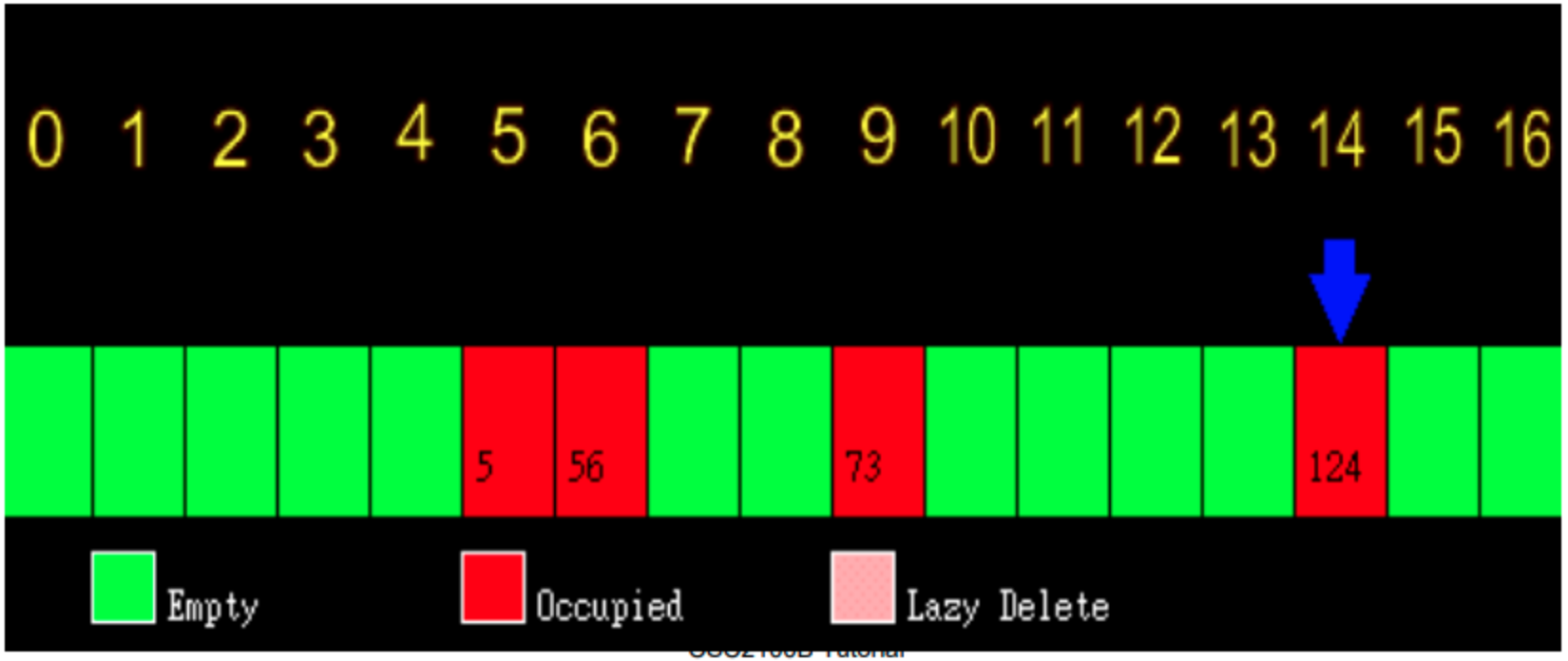
- Quadratic probing example
  - Insert 56 at $h_1(56)$ (56 mod 17) = 5
    ((56 + 1*1) mod 17) = 6



- Quadratic probing example
  - Insert 73 at $h_2(56)$ (73 mod 17) = 5
    ((73 + 2*2) mod 17) = 9



- Quadratic probing example
  - Insert 124 at $h_3(124)$ (124 mod 17) = 5
    ((124 + 3*3) mod 17) = 6



4) Double hashing

It is a technique in which **two hash function are used when there is an occurrence of collision.** In this method 1 hash function is simple as same as division method. But for the second hash function there are two important rules which are

- It must never evaluate to zero.
- Must sure about the buckets, that they are probed.

The hash functions for this technique are:

H1(key)=key % table size

H2(key)=P-(key mod P)

Where, **P** is a prime number which should be taken smaller than the size of a hash table.

c) Double Hashing We use another hash function hash2(x) and look for i*hash2(x) slot in i'th rotation.

let hash(x) be the slot index computed using hash function.

If slot hash(x) % S is full, then we try (hash(x) + 1*hash2(x)) % S

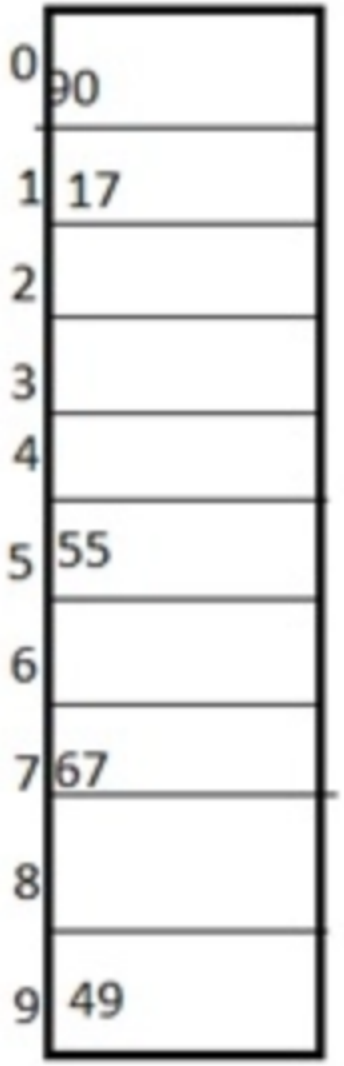
If (hash(x) + 1*hash2(x)) % S is also full, then we try (hash(x) + 2*hash2(x)) % S

If (hash(x) + 2*hash2(x)) % S is also full, then we try (hash(x) + 3*hash2(x)) % S

.....

.....

Example: Let us consider we have to insert 67, 90,55,17,49.



In this we can see 67, 90 and 55 can be inserted in a hash table by using first hash function but in case of 17 again the bucket is full and in this case we have to use the second hash function which is H2(key)=P-(key mode P) here p is a prime number which should be taken smaller than the hash table so value of p will be the 7.

i.e. H2(17)=7-(17%7)=7-3=4 that means we have to take 4 jumps for placing the 17. Therefore 17 will be placed at index 1.

Hashing with Double Hashing

- Double hashing uses the idea of applying a second hash function to the key when a collision occurs. The result of the second hash function will be the number of positions form the point of collision to insert.
- There are a couple of requirements for the second function:
- it must never evaluate to 0
- must make sure that all cells can be probed
- A popular second hash function is: $Hash_2(key) = R - (key \% R)$ where R is a prime number that is smaller than the size of the table.

Table Size = 10 elements
$Hash_1(key) = key \% 10$
$Hash_2(key) = 7 - (k \% 7)$

Insert keys : 89, 18, 49, 58, 69

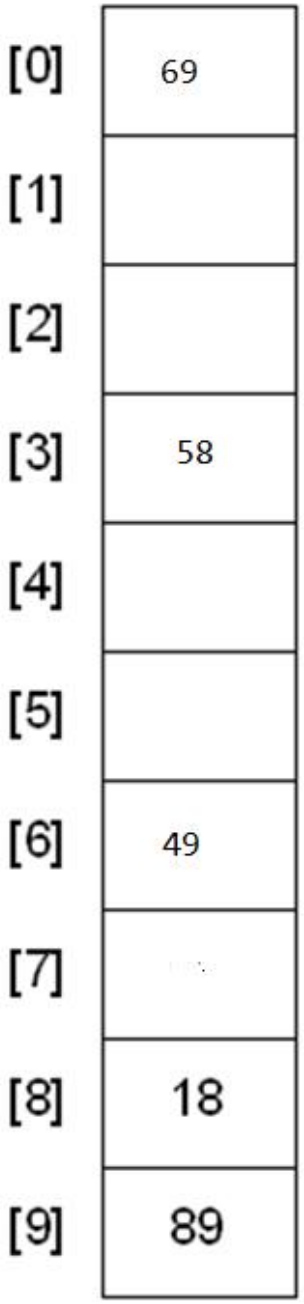
Hash(89) = 89 % 10 = 9

Hash(18) = 18 % 10 = 8

Hash(49) = 49 % 10 = 9 a collision !
= 7 – (49 % 7)
= 7 positions from [9]

Hash(58) = 58 % 10 = 8
= 7 – (58 % 7)
= 5 positions from [8]

Hash(69) = 69 % 10 = 9
= 7 – (69 % 7)
= 1 position from [9]



Comparison of above three:

- Linear probing has the best cache performance but suffers from clustering. One more advantage of Linear probing is easy to compute.
- Quadratic probing lies between the two in terms of cache performance and clustering.
- Double hashing has poor cache performance but no clustering. Double hashing requires more computation time as two hash functions need to be computed.

S.NO.	SEPARATE CHAINING	OPEN ADDRESSING
1.	Chaining is Simpler to implement.	Open Addressing requires more computation.
2.	In chaining, Hash table never fills up, we can always add more elements to chain.	In open addressing, table may become full.
3.	Chaining is Less sensitive to the hash function or load factors.	Open addressing requires extra care for to avoid clustering and load factor.
4.	Chaining is mostly used when it is unknown how many and how frequently keys may be inserted or deleted.	Open addressing is used when the frequency and number of keys is known.
5.	Cache performance of chaining is not good as keys are stored using linked list.	Open addressing provides better cache performance as everything is stored in the same table.
6.	Wastage of Space (Some Parts of hash table in chaining are never used).	In Open addressing, a slot can be used even if an input doesn’t map to it.
7.	Chaining uses extra space for links.	No links in Open addressing