Searching
chapter 8

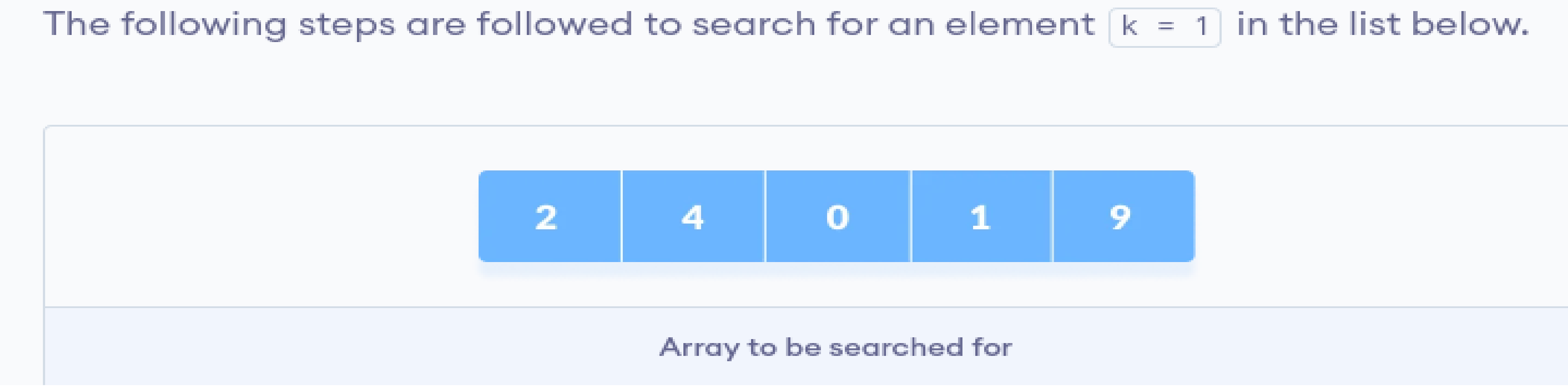
Searching

- Searching is an operation or technique that helps finds the place of a given element or value in the list.

- Any search is said to be successful depending upon whether the element that is being searched is found or not.

Linear Search or Sequential Search

- Linear search is a sequential searching algorithm where we start from one end and check every element of the list until the desired element is found.
- It is the simplest searching algorithm.

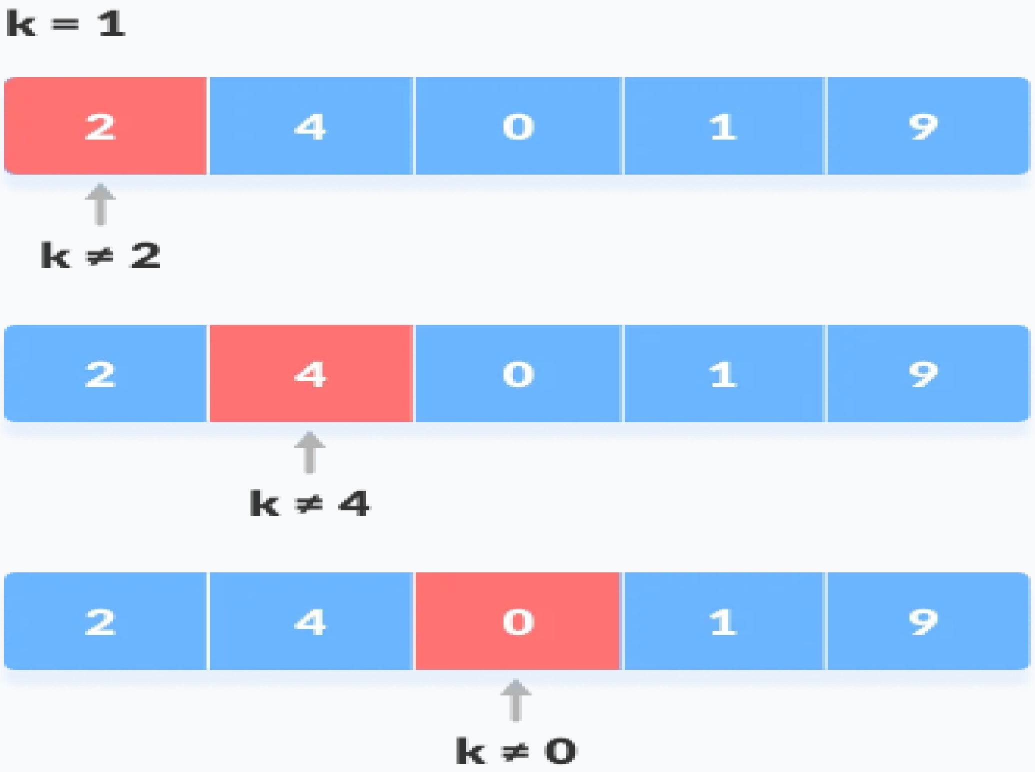


The following steps are followed to search for an element k = 1 in the list below.

2	4	0	1	9

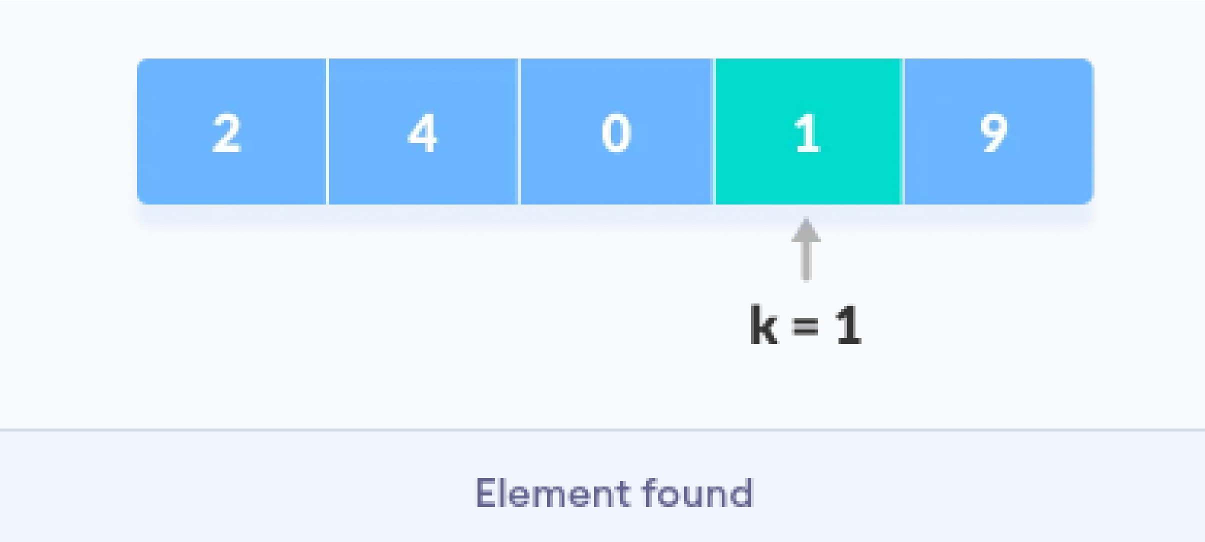
Array to be searched for

1. Start from the first element, compare `k` with each element `x`.



Compare with each element

If x==k, return the index



Else return not found

Linear Search Algorithm/ Sequential Search

Let A be an array of n elements, A [1], A[2],A[3], ... A[n]. “data” is the element to be searched. Then this algorithm will find the data and display the location if present otherwise display data not found

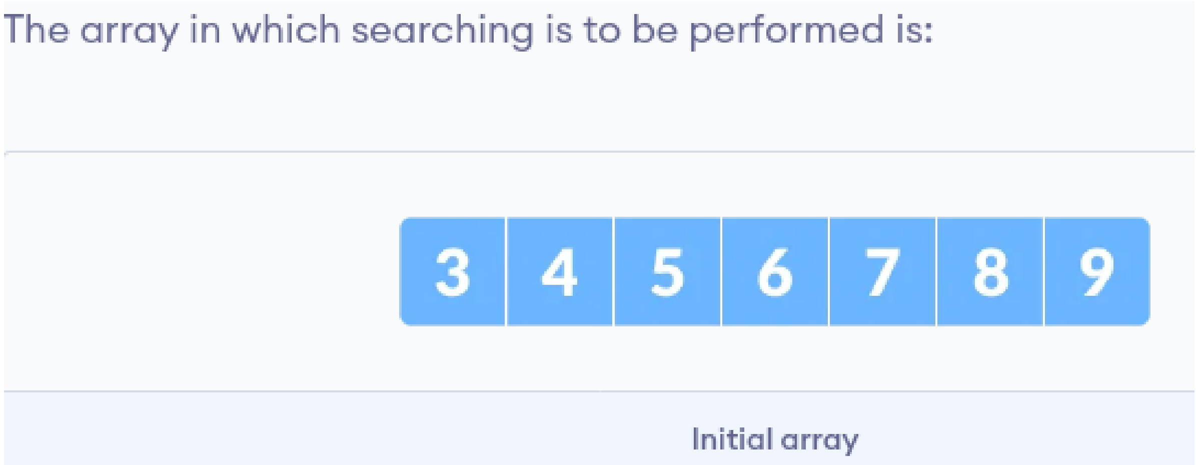
```
1.  Input an array A of n elements and “data” to be searched and initialize flag =0.

2.  Initialize $i = 0$; and repeat through step 3 if ($i < n$) by incrementing i by one
3.  If (data == A[i])
      i.   Display data is found at location i
      ii.  Flag = 1
      iii. return
4.  If (flag==0)
      i.   Display “data is not found and searching is unsuccessful”
5.  Exit
```

Binary Search

Binary Search is a searching algorithm for **finding an element's position in a sorted array.**

Binary search can be implemented only on a sorted list of items. If the elements are not sorted already, we need to sort them first.

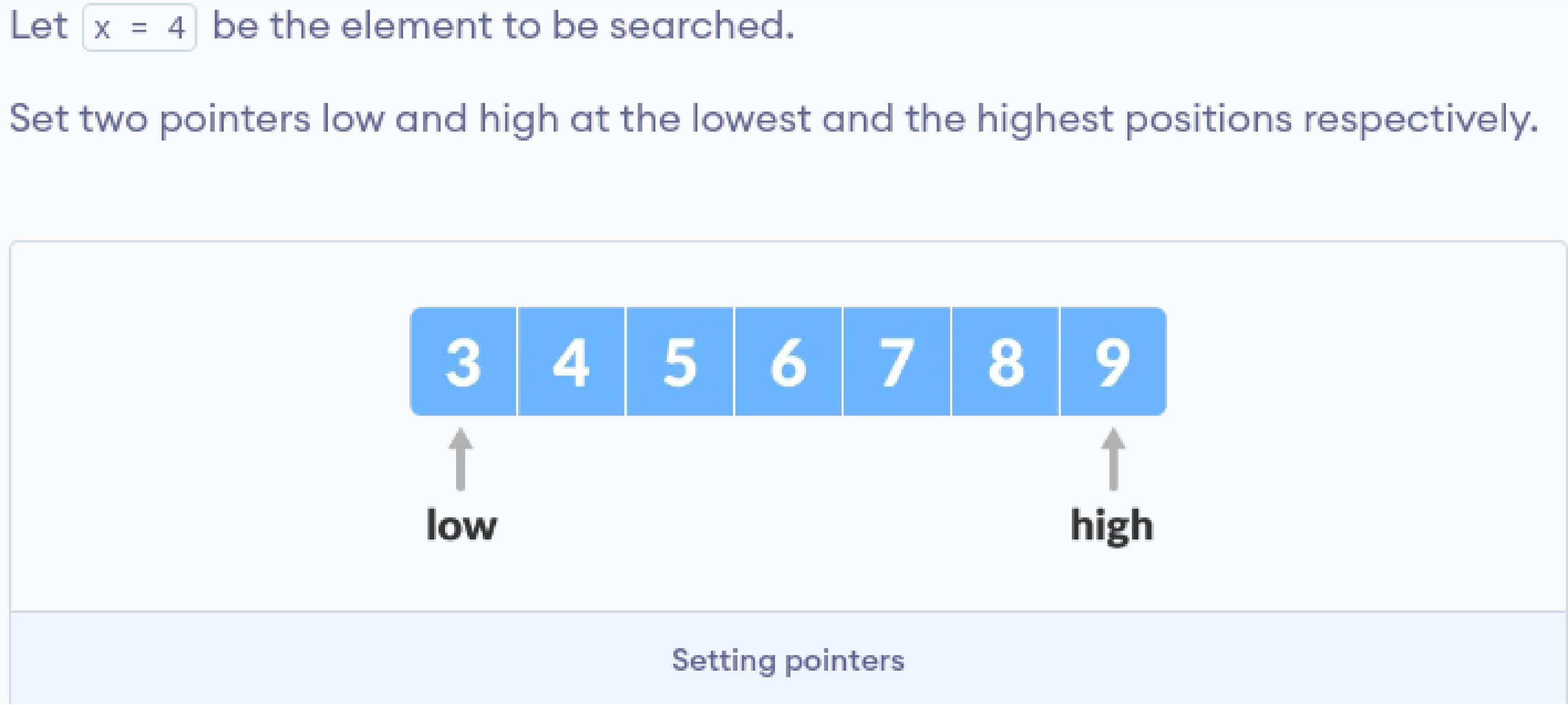
In this approach, the element is always searched in the middle of a portion of an array.



The array in which searching is to be performed is:

3	4	5	6	7	8	9

Initial array

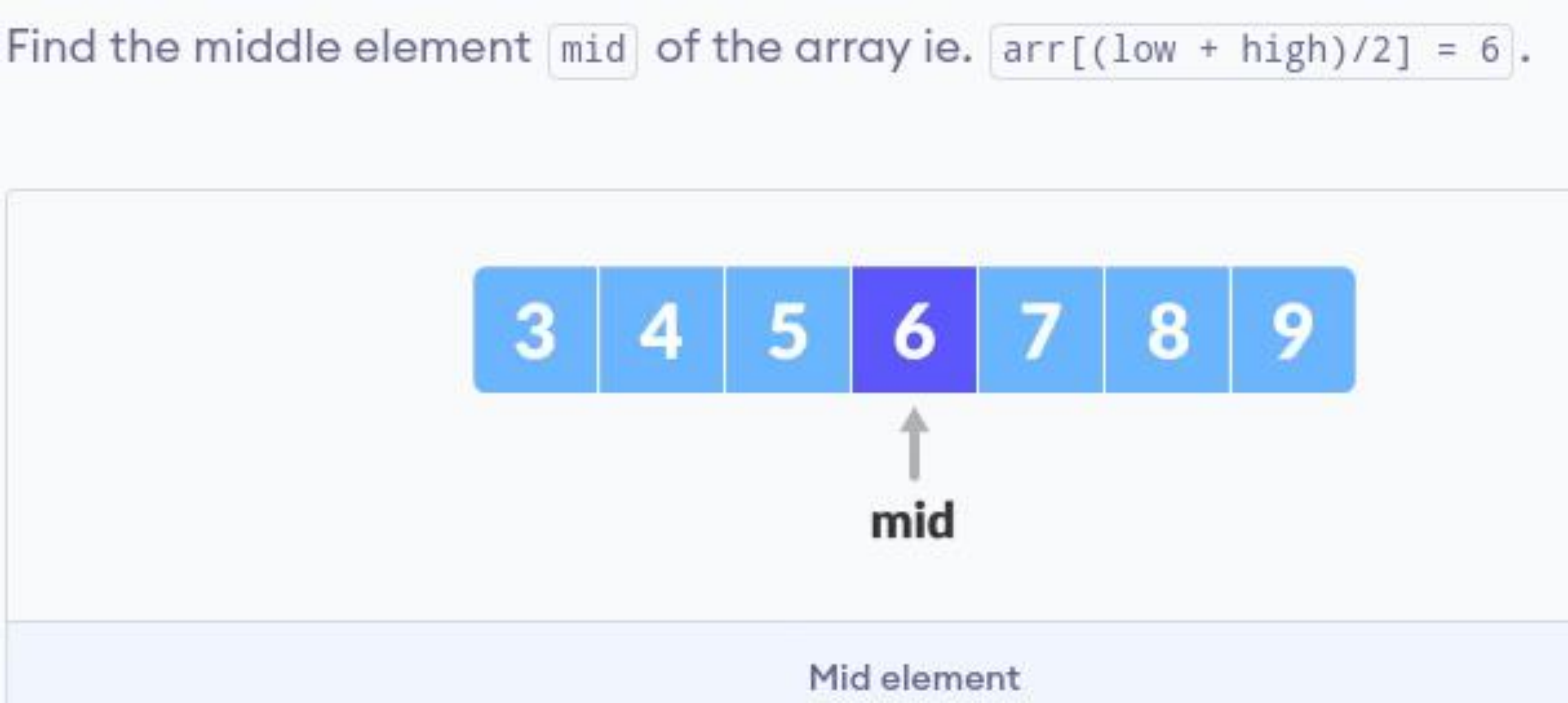


Let `x = 4` be the element to be searched.

Set two pointers low and high at the lowest and the highest positions respectively.

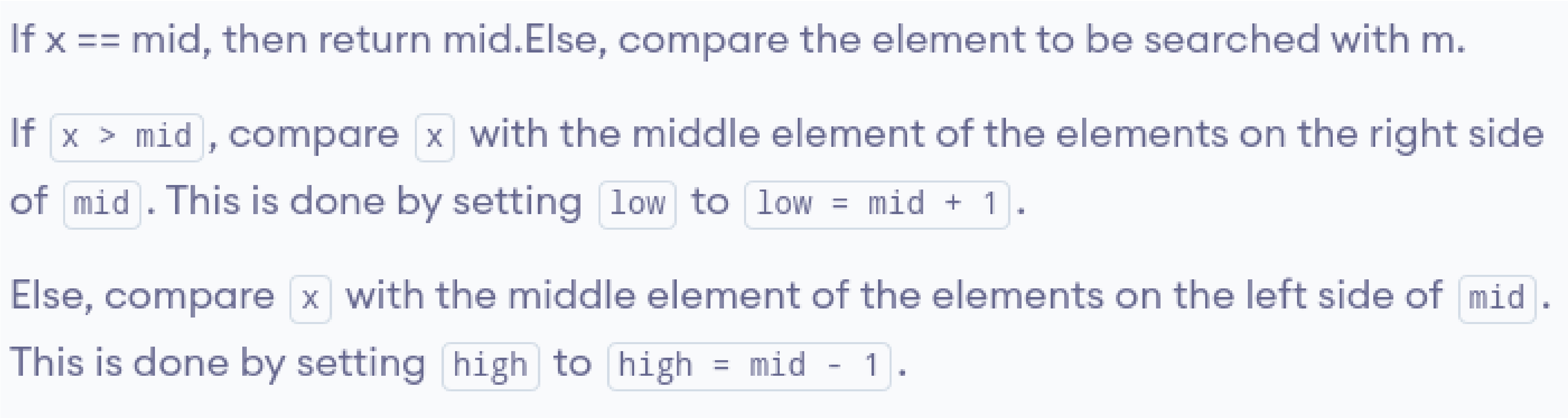
3	4	5	6	7	8	9

Setting pointers



Find the middle element `mid` of the array ie. `arr[(low + high)/2] = 6`.

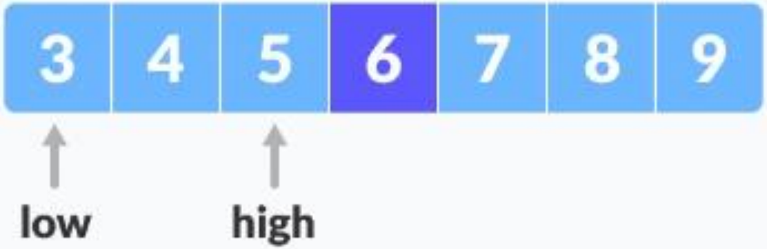
Mid element



If x == mid, then return mid.Else, compare the element to be searched with m.

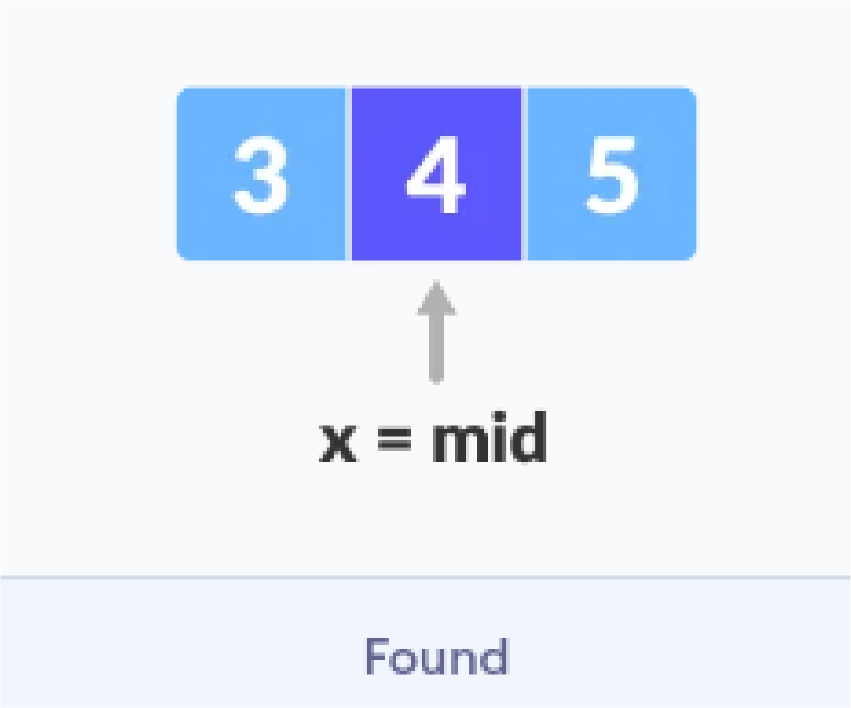
If `x > mid`, compare `x` with the middle element of the elements on the right side of `mid`. This is done by setting `low` to `low = mid + 1`.

Else, compare `x` with the middle element of the elements on the left side of `mid`. This is done by setting `high` to `high = mid - 1`.



Finding mid element

Repeat process until low meets high



Binary Search Iteration Algorithm

```
do until the pointers low and high meet each other.
    mid = (low + high)/2
    if (x == arr[mid])
        return mid
    else if (x > arr[mid]) // x is on the right side
        low = mid + 1
    else                   // x is on the left side
        high = mid - 1
```

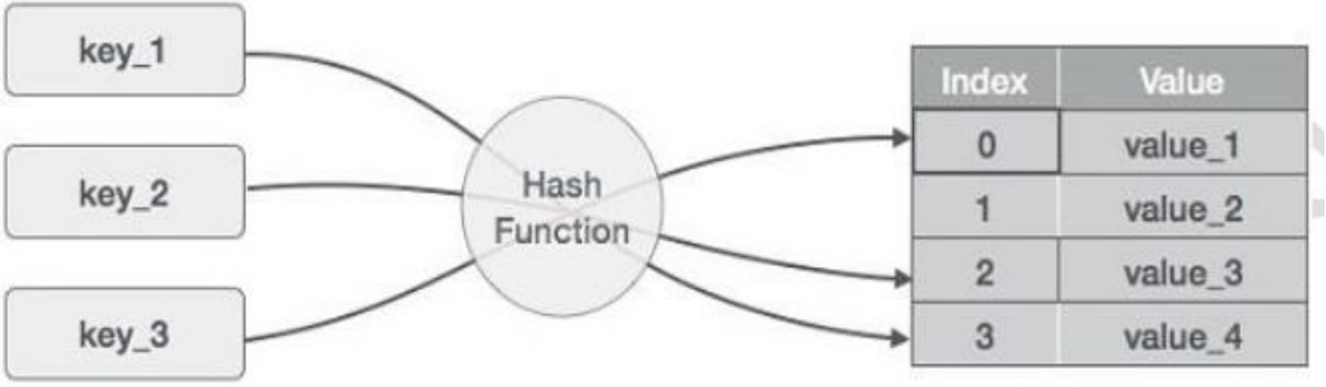
Binary Search using Recursive Algorithm

```
binarySearch(arr, x, low, high)
    if low > high
        return False
    else
        mid = (low + high) / 2
        if x == arr[mid]
            return mid
        else if x > arr[mid]        // x is on the right side
            return binarySearch(arr, x, mid + 1, high)
        else                        // x is on the right side
            return binarySearch(arr, x, low, mid - 1)
```

Hashing

- “Derive a number from the key information given to you and use that number to access all information related to the key”. This is the basic principle of hashing.

- This way, we can access any record in a specific time without making any comparison, irrespective of the number of records in the file.

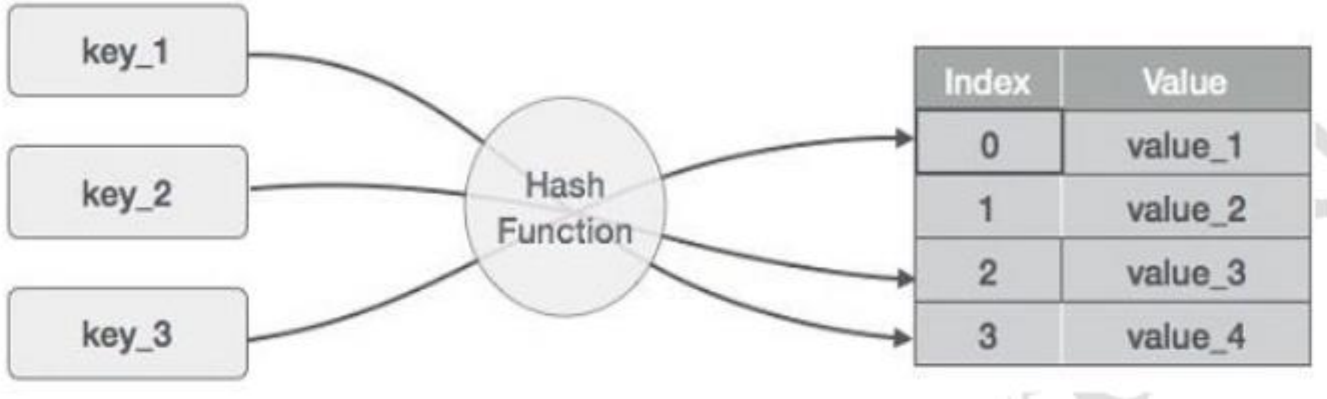


Index	Value
0	value_1
1	value_2
2	value_3
3	value_4

Hashing

- Hashing is a scheme of searching that will use some function say hash to directly find out the location of the record in a constant search time, no matter where the record is in the file.

- In other words, the process of mapping large amount of data into a smaller table is called hashing.



Index	Value
0	value_1
1	value_2
2	value_3
3	value_4

Hash Table

A hash table is simply an array that is address via a hash function. It is a data structure made up of

- **A table of some fixed size to hold a collection of records each uniquely identified by some key.**

- **A function called hash function that is used to generate index values in the table.**

Some Hash Functions

- The basic idea in hashing is the transformation of a key into the corresponding location in the hash table. This is done by a hash function.

- A hash function can be defined as a function that takes key as input and transforms it into a hash table index usually denoted by H.

We have, in general following methods.

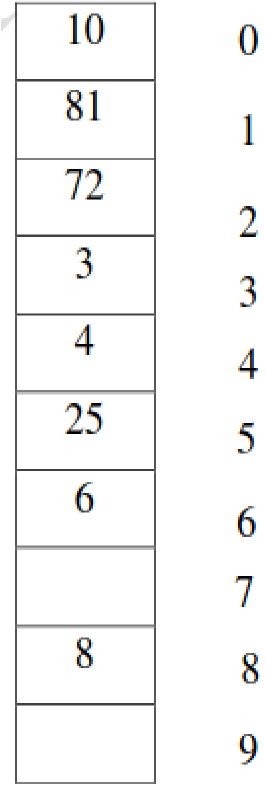
i. **Folding method**

ii. **Mid square method**

iii. **Division method**

Let us take a hash function h(x)=x % 10

If we want to store keys 8,3,6,4,10,81,72,25



10	0
81	1
72	2
3	3
4	4
25	5
6	6
	7
8	8
	9

We have, in general following methods.

i. **Folding method**

ii. **Mid square method**

iii. **Division method**

Some Hash Functions

- Hash of Key Let H be a hash function and k is a key then H(k) is called hash-of-key. The hash-of-key is the index at which a record with the key values k must be kept.

What is collision?

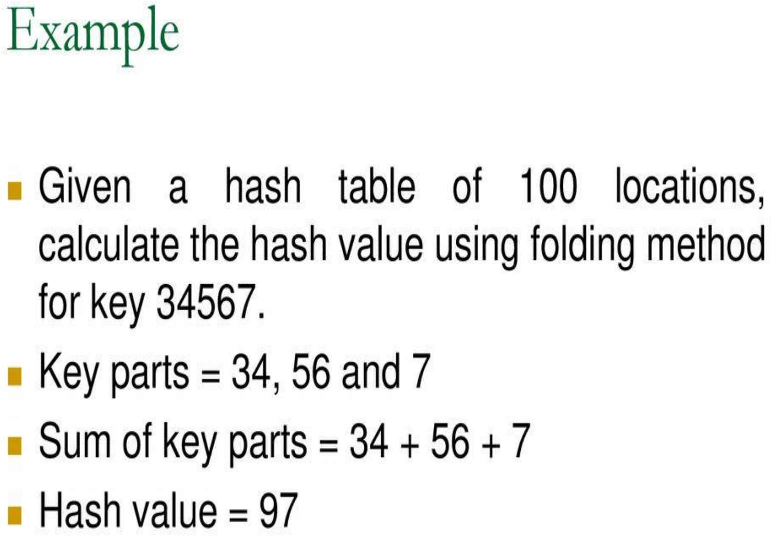
- There are a finite number of indices in a table. But there are large numbers of keys so it is clearly impossible to get two different indexes for two distinct keys.

- A situation in which two different keys k1 and k2 hash to the same index of the table i.e. h(k1) = h(k2), it is called collision or hash clash.

Some Hash Functions

Folding method

- Eg. Let the keys be of four digits, chopping the key into two parts and adding yields Hash (5421) = 54 + 21 = 75 So,75 is the index at which we should store or retrieve record with key 5421.



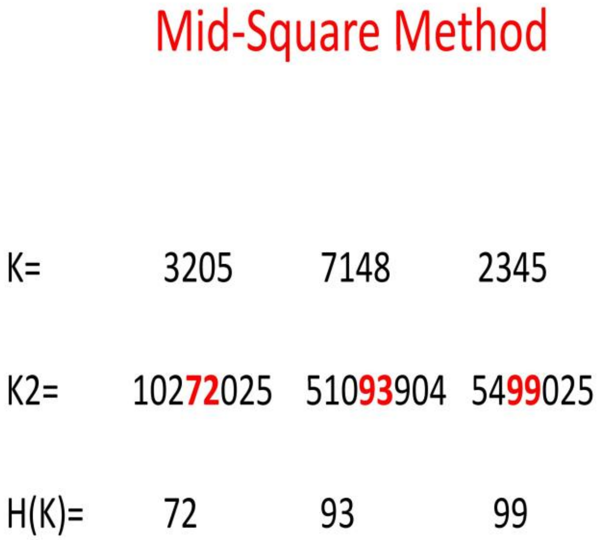
Example

- Given a hash table of 100 locations, calculate the hash value using folding method for key 34567.
- Key parts = 34, 56 and 7
- Sum of key parts = 34 + 56 + 7
- Hash value = 97

Some Hash Functions

Mid square method

- The key is squared.
- We defined the hash function in this case as Hash (key) = p; Where p is obtained by deleting digits from both ends of (key)^2 .



Mid-Square Method

K=	3205	7148	2345
K2=	10272025	51093904	5499025
H(K)=	72	93	99

Some Hash Functions

Division Remainder method

- Convert the key to an integer, divide by the size of the index range and take the remainder as the result.

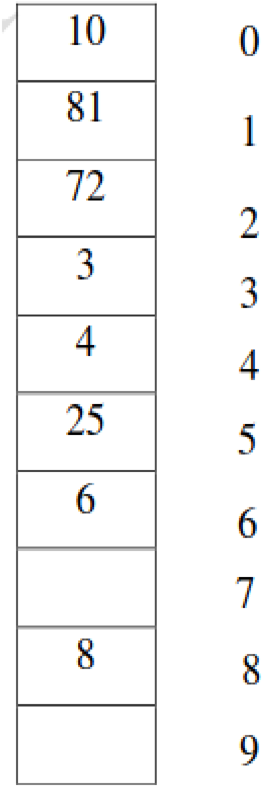
- Eg, hash (key) = hash (1029) Let 100 be the table size (index range) Then, Hash (1029) = 1029%100 = 29. 29 is the hash value (i.e. index of the array).

- To get a good distribution of indices, prime number makes the best table size.

Division Remainder Method

Let us take a hash function h(x)=x % 10

If we want to store keys 8,3,6,4,10,81,72,25



10	0
81	1
72	2
3	3
4	4
25	5
6	6
	7
8	8
	9

Some Hash Functions

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.

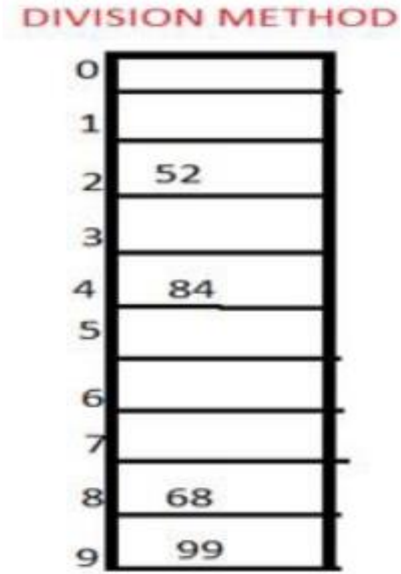
Then: h(key)= record% table size.

52%10 =2

68%10 =8

99%10 =9

84%10 =4



Why Hashing?

Suppose we want to design a system for storing employee records keyed using phone numbers. And we want following queries to be performed efficiently:

- Insert a phone number and corresponding information.
- Search a phone number and fetch the information.
- Delete a phone number and related information.

Following data structures to maintain information about different phone numbers.

- Array of phone numbers and records.
- Linked List of phone numbers and records.

Hash Functions and Hash table

Collision Resolution Techniques

- When a collision occurs, alternative locations in the table are tried until an empty location is found. Looking for the next available position is called probing.
- Techniques are as follows:
  i. Linear probing
  ii. Quadratic probing
  iii. Double hashing
  iv. Chaining

Collision Resolving Techniques

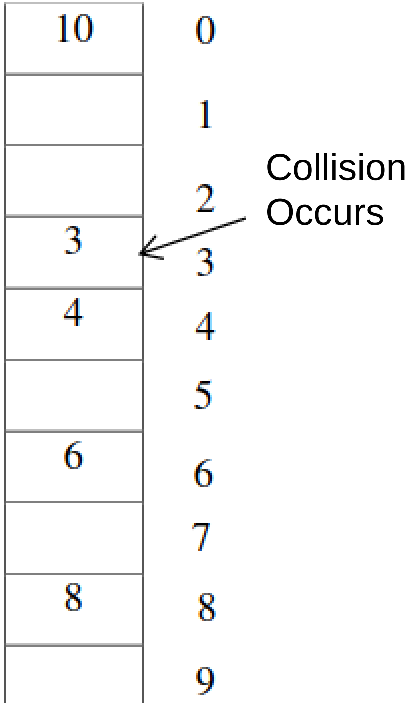
Two keys mapping to the same location in the hash table is called “Collision”

Collisions can be reduced with a selection of a good hash function

Let us consider the following key 8,3,13,6,4,10

and hash size of 10.

h(x)=x % 10



10	0
	1
	2
3	3
4	4
	5
6	6
	7
8	8
	9

Collision Occurs

Collision resolving techniques

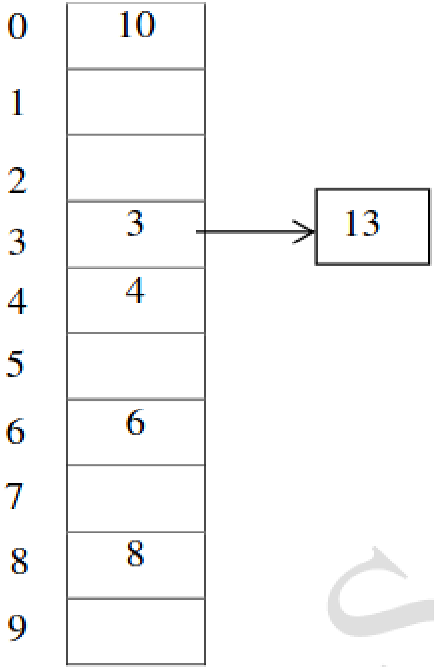
- Closed Addressing or chaining (Open hashing)
- Open Addressing (Closed Hashing)
    - Linear Probing
    - Quadratic Probing
    - Double Hashing

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 the following key 8,3,13,6,4,10 and hash size of S=10.

h(x)=x % 10



0	10	
1		
2		
3	3	→ 13
4	4	
5		
6	6	
7		
8	8	
9		

Let's assume table size as 3.

Then the array of linked list will be,

```
-------------          -------------
|           |          |           |
| chain[0]  |------->  |    NULL   |
|           |          |           |
-------------          -------------
-------------          -------------
|           |          |           |
| chain[1]  |------->  |    NULL   |
|           |          |           |
-------------          -------------
-------------          -------------
|           |          |           |
| chain[2]  |------->  |    NULL   |
|           |          |           |
-------------          -------------
```

i)Insert 6

Hash key = 6 % 3 = 0.

Hence add the node with data 6 in the chain[0].

```
-------------         ----------------
|           |         |     |        |
| chain[0]  |------->  |  6  | NULL   |
|           |         |     |        |
-------------         ----------------
-------------         -------------
|           |         |           |
| chain[1]  |------->  |    NULL   |
|           |         |           |
-------------         -------------
-------------         -------------
|           |         |           |
| chain[2]  |------->  |    NULL   |
|           |         |           |
-------------         -------------
```

ii)Insert 12

Hash key = 12 % 3 = 0

Collision! Both 6 and 12 points to the hash index 0.

We can avoid the collision by adding data 12 at the end of the chain[0].

This how we can avoid the collision in separate chaining method.

```
-------------          ----------------        ----------------
|           |          |     |        |        |     |         |
| chain[0]  |-------> |  6  |        |------> |  12 | NULL    |
|           |          |     |        |        |     |         |
-------------          ----------------        ----------------
-------------          -------------
|           |          |           |
| chain[1]  |-------> |    NULL   |
|           |          |           |
-------------          -------------
-------------          -------------
|           |          |           |
| chain[2]  |-------> |    NULL   |
|           |          |           |
-------------          -------------
```

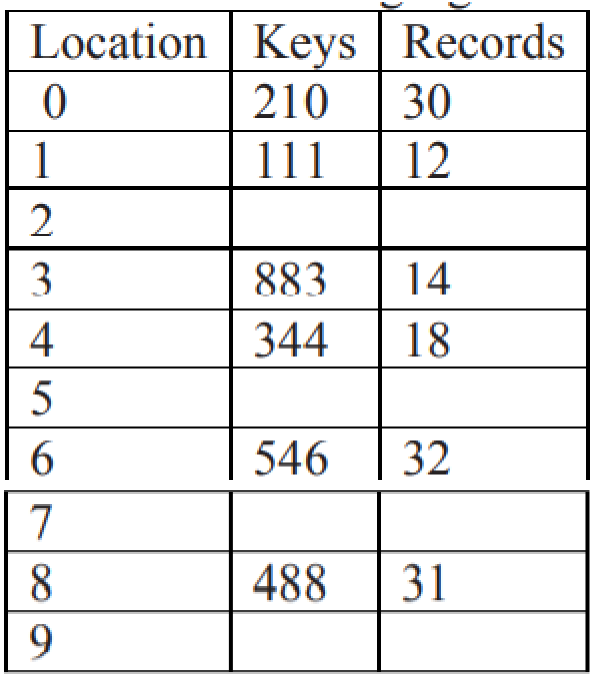
iii)Insert 10

Hash key = 10 % 3 = 1.

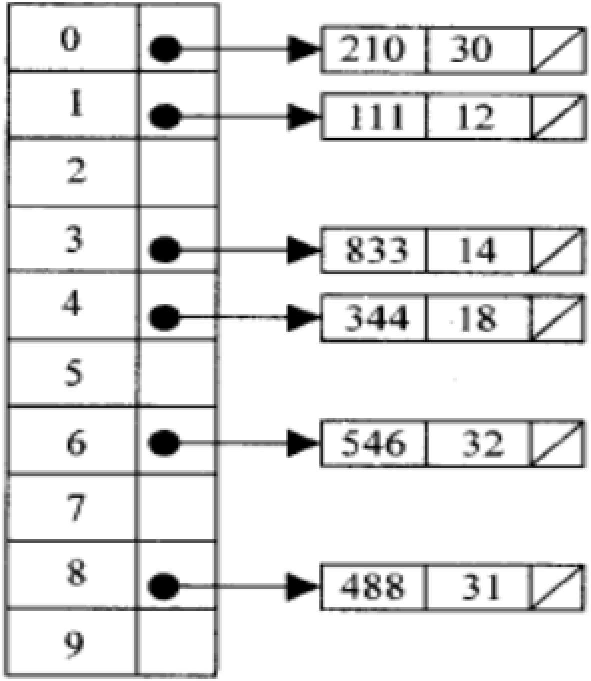
Hence add node with data 10 in the chain[1].

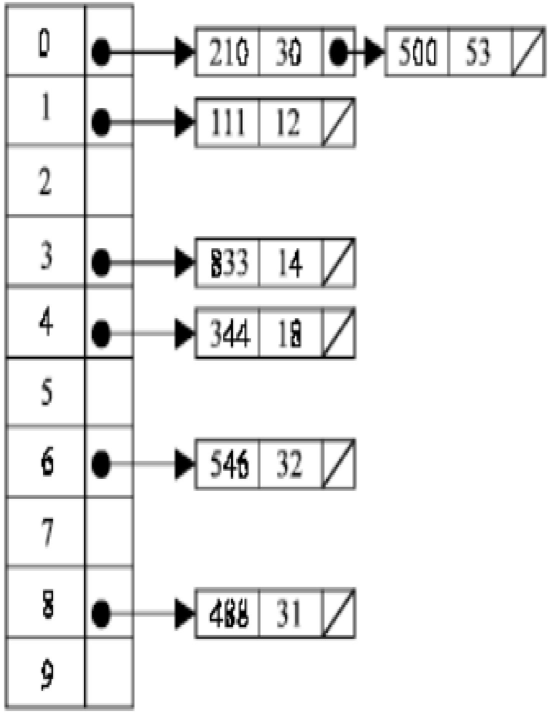
```
-------------          -----------------        -----------------

|           |          |     |         |        |     |         |
| chain[0]  |--------> |  6  |         |------> |  12 | NULL    |
|           |          |     |         |        |     |         |

-------------          -----------------        -----------------
-------------          -----------------

|           |          |     |         |
| chain[1]  |--------> |  10 |  NULL   |
|           |          |     |         |

-------------          -----------------
-------------          -------------

|           |          |           |
| chain[2]  |--------> |    NULL   |
|           |          |           |

-------------          -------------
```

Chaining



Location	Keys	Records
0	210	30
1	111	12
2		
3	883	14
4	344	18
5		
6	546	32
7		
8	488	31
9		





Algorithm to Insert data into the separate chain

```
1. Declare an array of a linked list with the hash table size.
2. Initialize an array of a linked list to NULL.
3. Find hash key.
4. If chain[key] == NULL
        Make chain[key] points to the key node.
5. Otherwise(collision),
        Insert the key node at the end of the chain[key].
```

Searching a value from the hash table

1. Get the value

2. Compute the hash key.

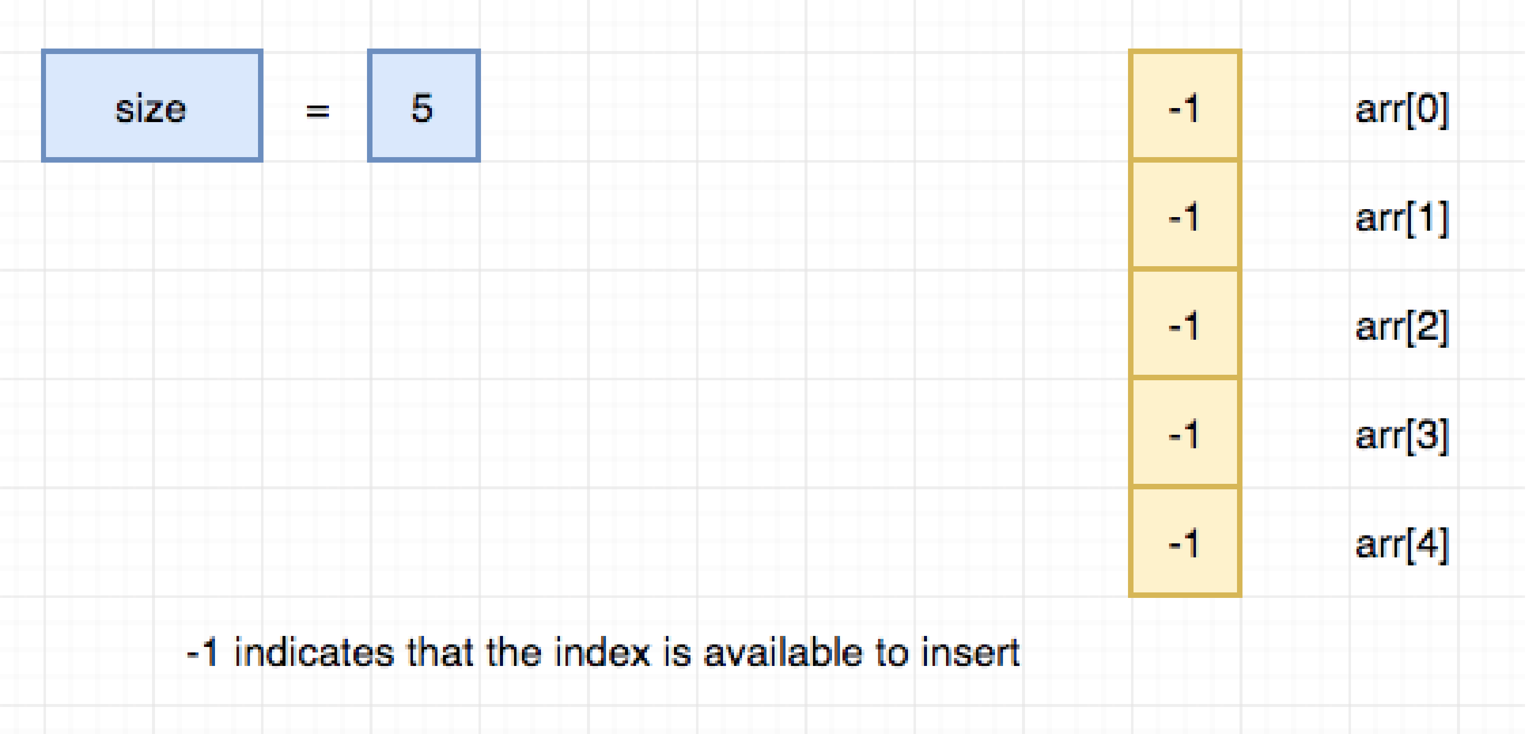
3. Search the value in the entire chain. i.e. chain[key].

4. If found, print "Search Found"

5. Otherwise, print "Search Not Found"

Open addressing (Closed Hashing)

Linear Probing: h i(X) = (Hash(X) + i) mod TableSize

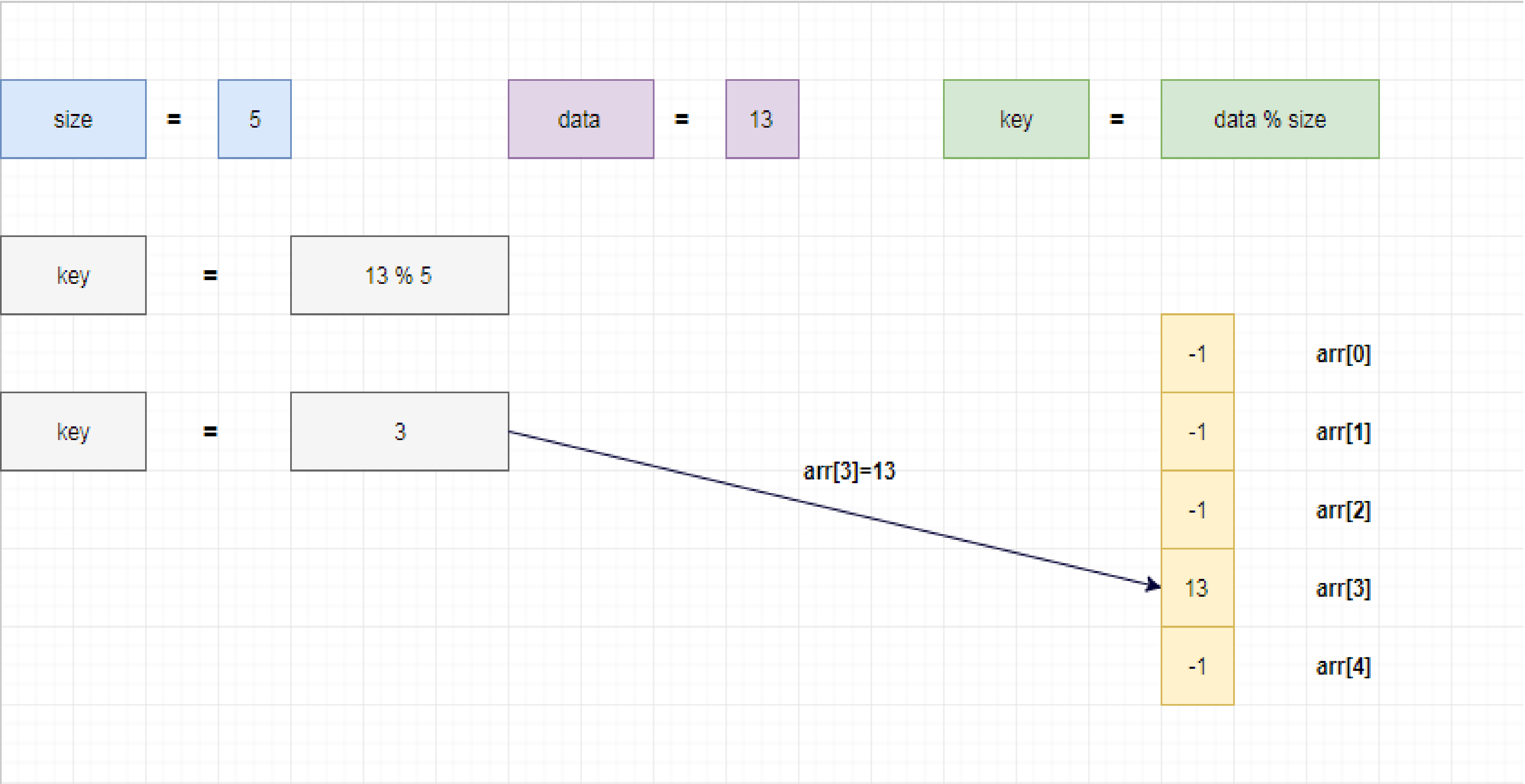


size = 5

-1	arr[0]
-1	arr[1]
-1	arr[2]
-1	arr[3]
-1	arr[4]

-1 indicates that the index is available to insert

Insert 13



size = 5

data = 13

key = data % size

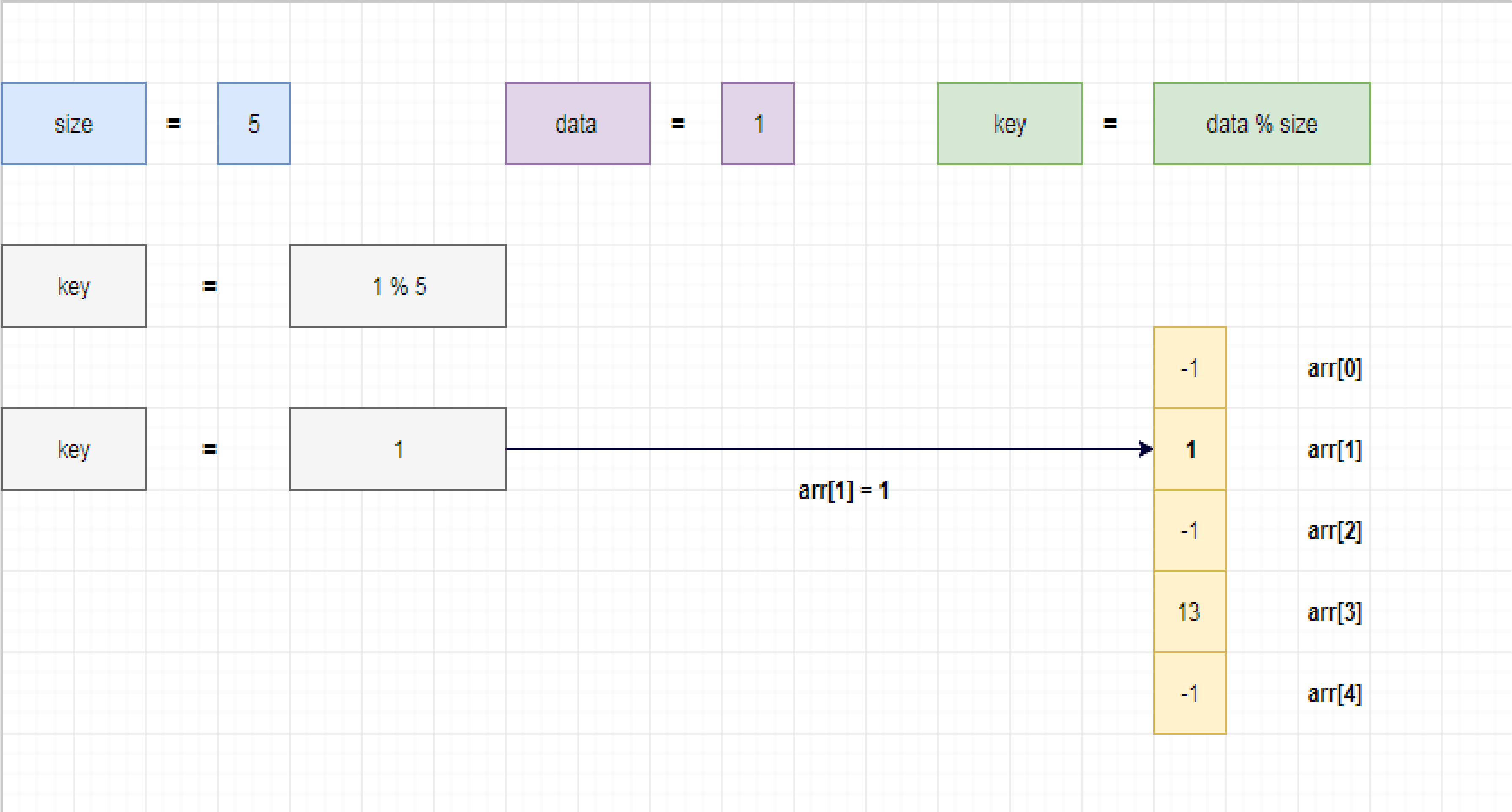
key = 13 % 5

key = 3

arr[3]=13

-1	arr[0]
-1	arr[1]
-1	arr[2]
13	arr[3]
-1	arr[4]

Insert 1



size = 5

data = 1

key = data % size

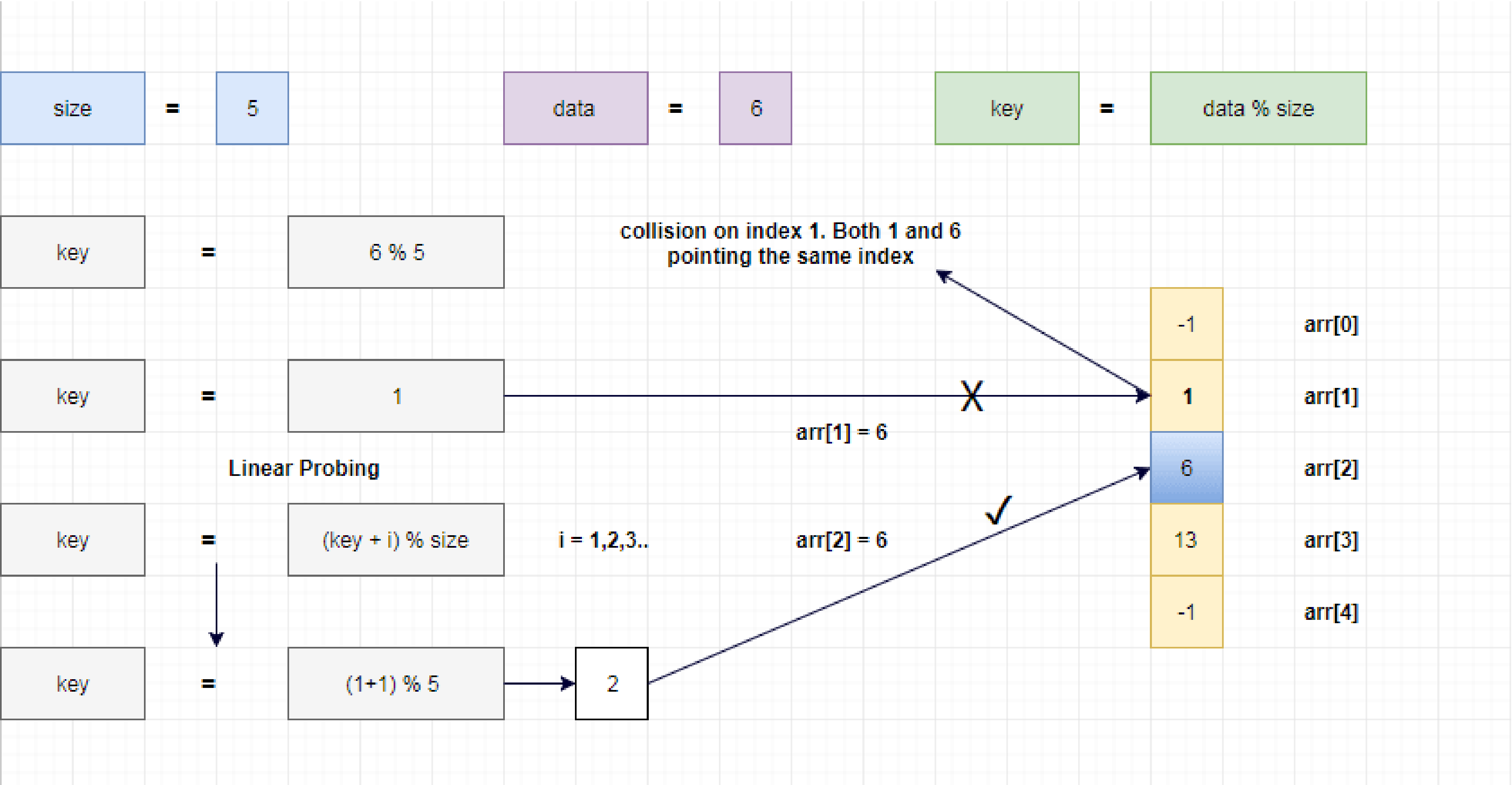
key = 1 % 5

key = 1

arr[1] = 1

-1	arr[0]
1	arr[1]
-1	arr[2]
13	arr[3]
-1	arr[4]

Insert 6



size = 5

data = 6

key = data % size

key = 6 % 5

collision on index 1. Both 1 and 6 pointing the same index

key = 1

X

arr[1] = 6

Linear Probing

key = (key + i) % size

i = 1,2,3..

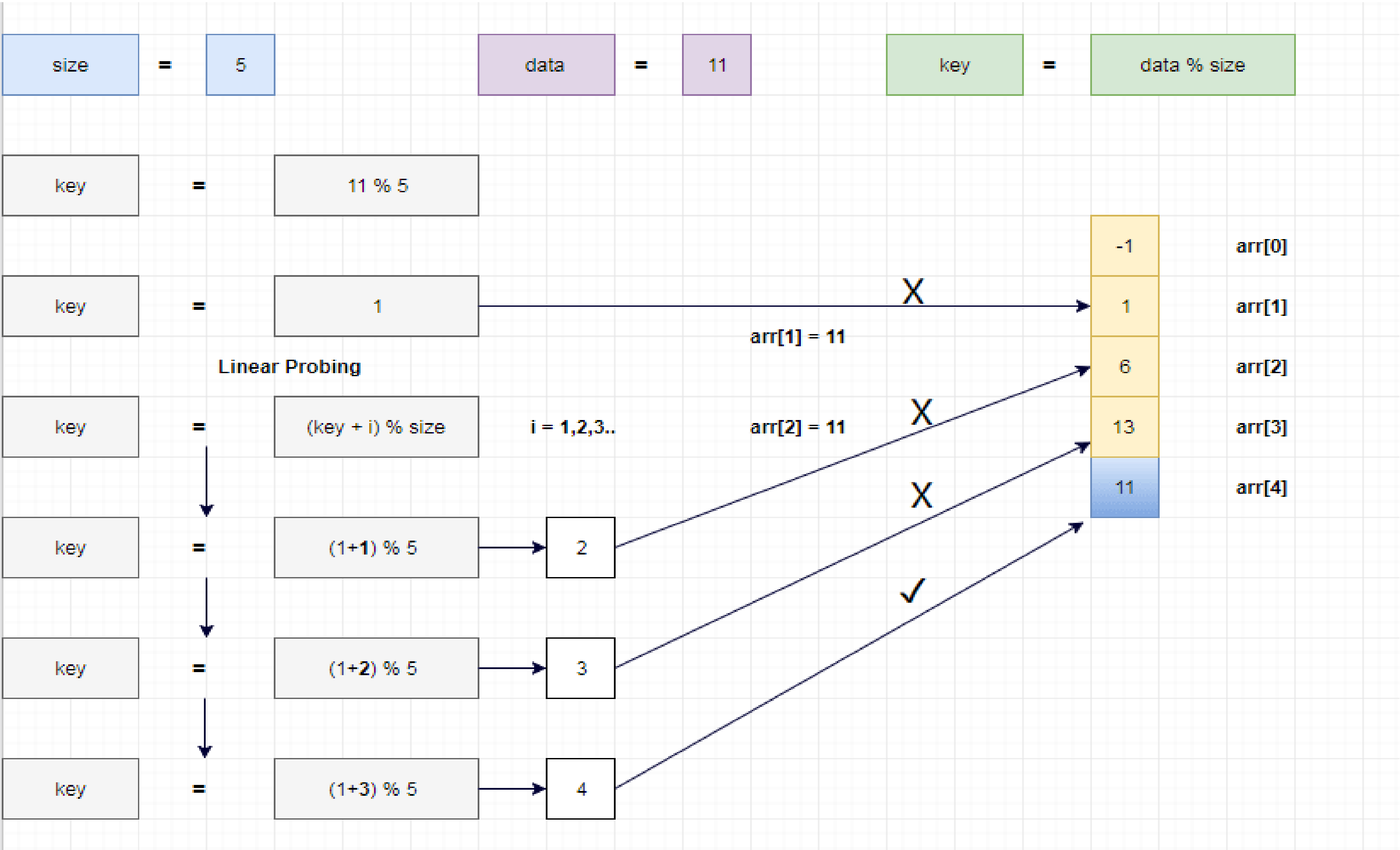
arr[2] = 6

✓

key = (1+1) % 5 → 2

-1	arr[0]
1	arr[1]
6	arr[2]
13	arr[3]
-1	arr[4]

Insert 11



size = 5

data = 11

key = data % size

key = 11 % 5

key = 1

arr[1] = 11

Linear Probing

key = (key + i) % size

i = 1,2,3..

arr[2] = 11

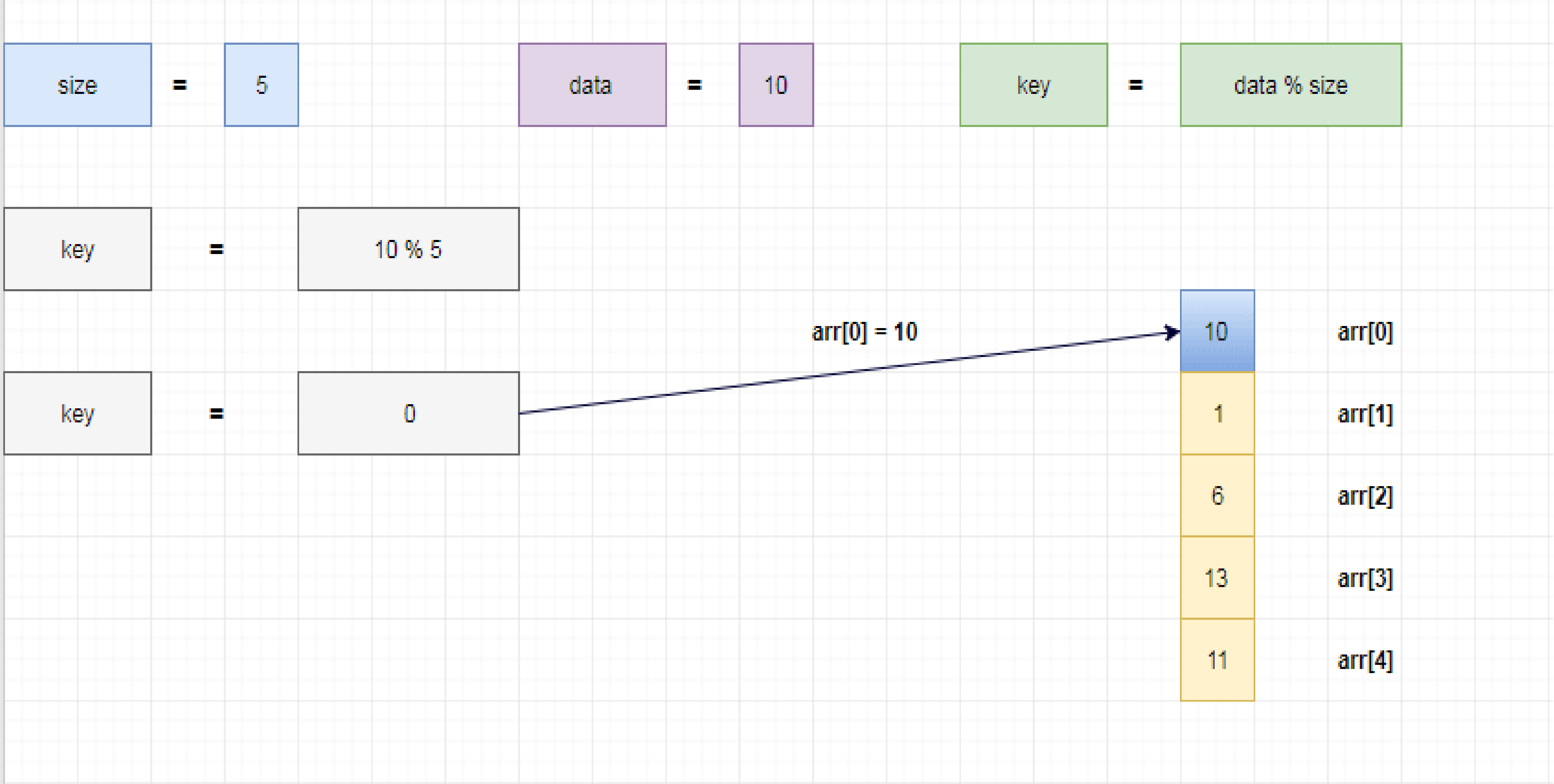
key = (1+1) % 5 → 2

key = (1+2) % 5 → 3

key = (1+3) % 5 → 4

-1	arr[0]
1	arr[1]
6	arr[2]
13	arr[3]
11	arr[4]

Insert 10



size = 5

data = 10

key = data % size

key = 10 % 5

key = 0

arr[0] = 10

10	arr[0]
1	arr[1]
6	arr[2]
13	arr[3]
11	arr[4]

Compute the following key using modulo division in the hash table

Key: 3, 2, 9, 6, 11, 13, 7, 12

Use the hash function: 2k+3

And size of hash table be 10.

Use Linear probing to avoid collision.

Algorithm

Calculate the hash key. key = data % size;

If hashTable[key] is empty, store the value directly. hashTable[key] = data.

If the hash index already has some value, check for next index.

key = (key+1) % size;

If the next index is available hashTable[key], store the value. Otherwise try for next index.

Do the above process till we find the space.

Quadratic Probing

h i(X) = (Hash(X) + i^2) mod TableSize

Calculate the hash key. key = data % size;

If hashTable[key] is empty, store the value directly. hashTable[key] = data.

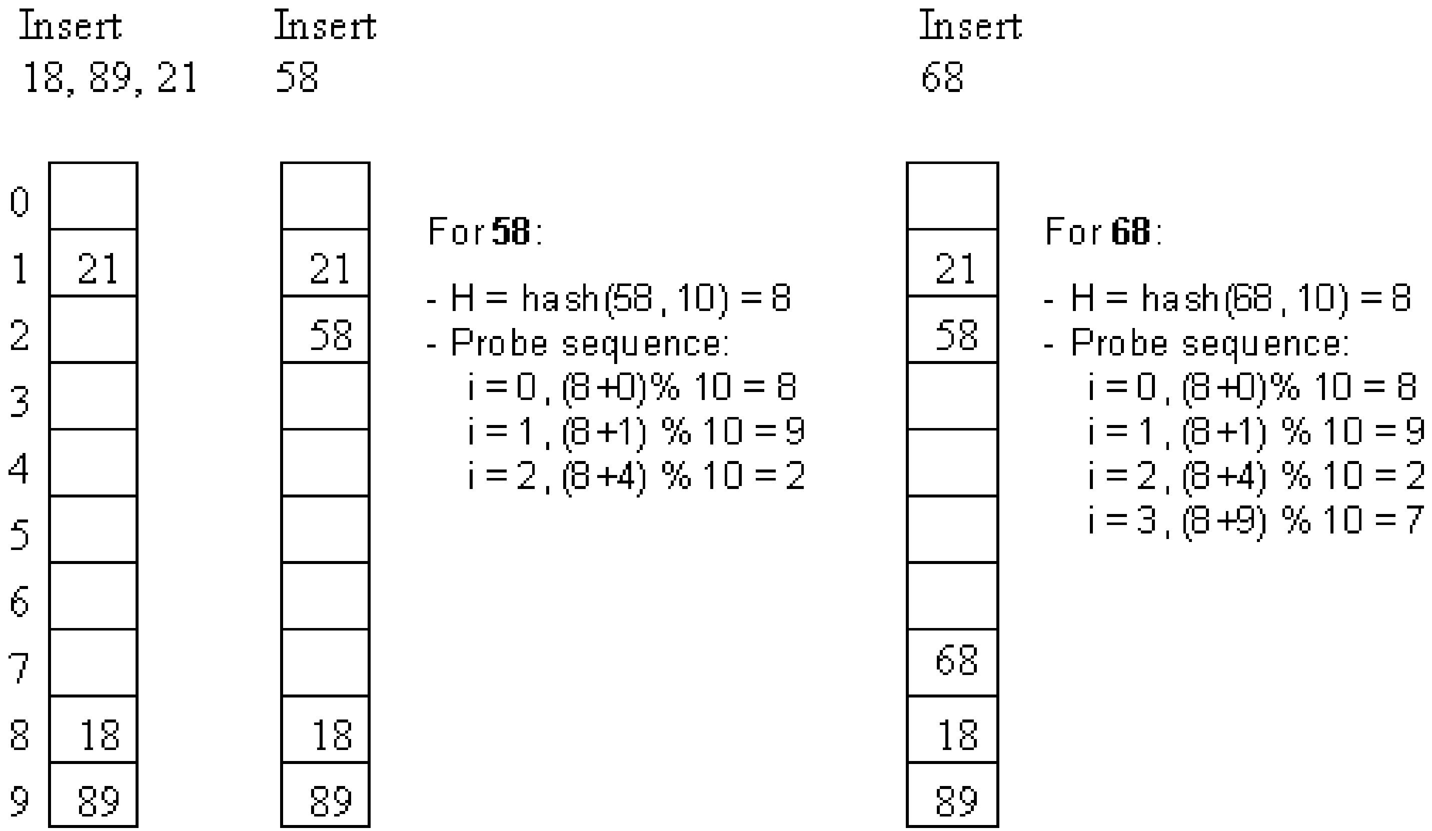
If the hash index already has some value, check for next index.

key = (key+i^2) % size;

If the next index is available hashTable[key], store the value. Otherwise try for next index.

Do the above process till we find the space.

Quadratic Probing



	Insert 18, 89, 21	Insert 58	Insert 68
0			
1	21	21	21
2		58	58
3			
4			
5			
6			
7			68
8	18	18	18
9	89	89	89

For **58**:

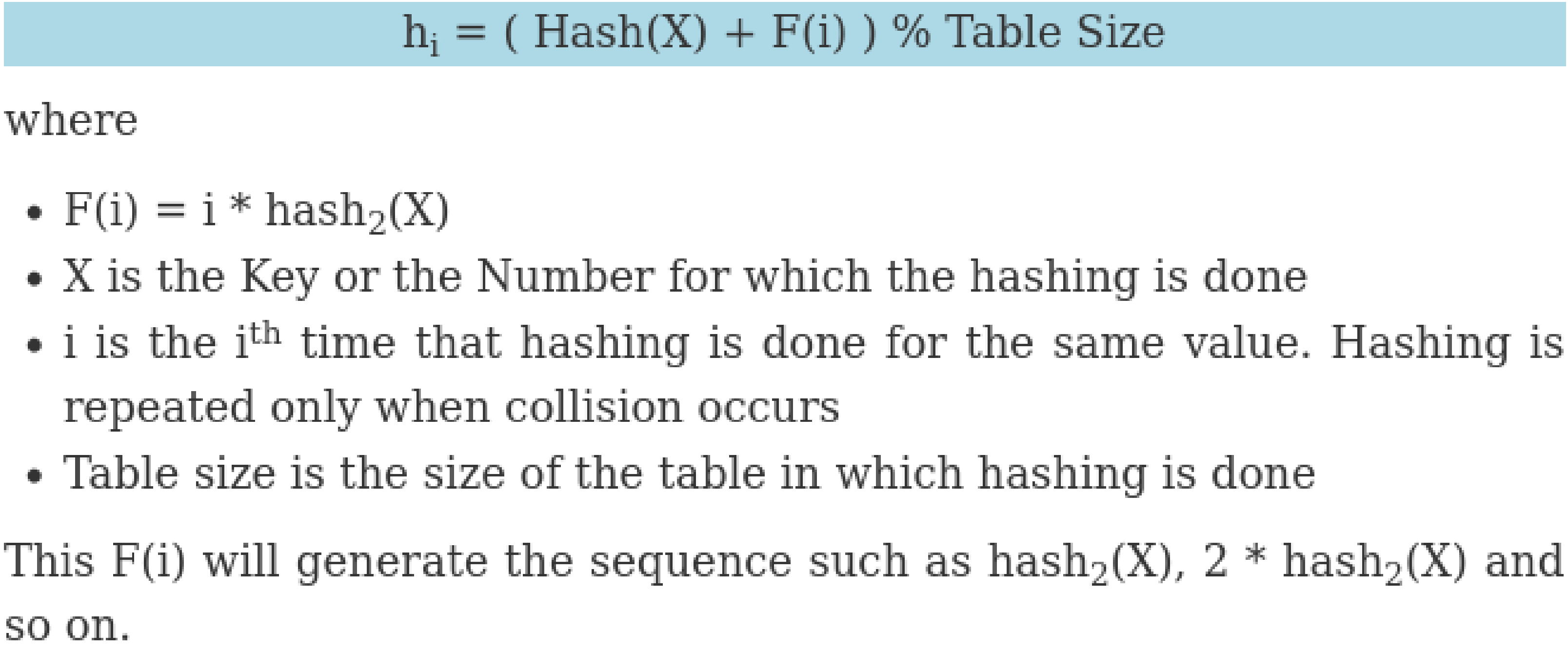
- H = hash(58, 10) = 8
- Probe sequence:
  i = 0, (8 +0)% 10 = 8
  i = 1, (8 +1) % 10 = 9
  i = 2, (8 +4) % 10 = 2

For **68**:

- H = hash(68, 10) = 8
- Probe sequence:
  i = 0, (8 +0)% 10 = 8
  i = 1, (8 +1) % 10 = 9
  i = 2, (8 +4) % 10 = 2
  i = 3, (8 +9) % 10 = 7

Double Hashing

Double Hashing is a hashing collision resolution technique where we use 2 hash functions.

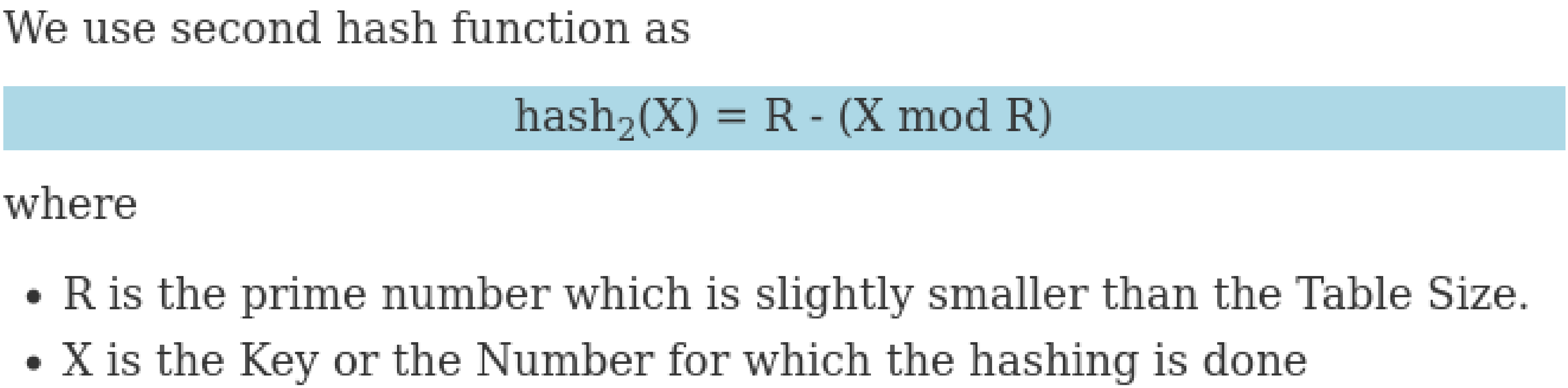


$$h_i = (\text{Hash}(X) + F(i)) \% \text{ Table Size}$$

where

- $F(i) = i * \text{hash}_2(X)$
- X is the Key or the Number for which the hashing is done
- i is the i^{th} time that hashing is done for the same value. Hashing is repeated only when collision occurs
- Table size is the size of the table in which hashing is done

This F(i) will generate the sequence such as $\text{hash}_2(X)$, $2 * \text{hash}_2(X)$ and so on.



We use second hash function as

$$\text{hash}_2(X) = R - (X \bmod R)$$

where

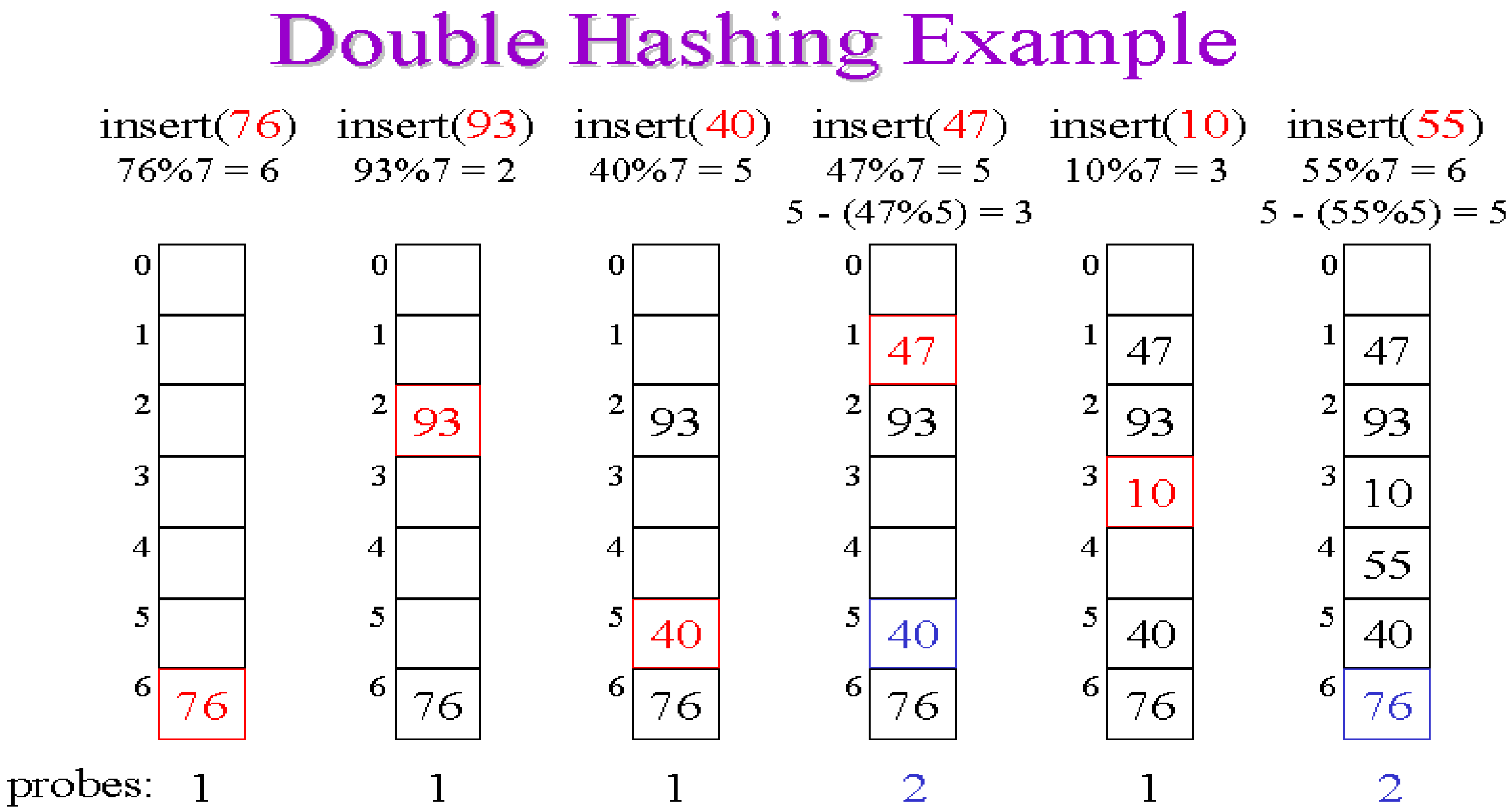
- R is the prime number which is slightly smaller than the Table Size.
- X is the Key or the Number for which the hashing is done

Use double hashing for the key values: 79, 28, 39, 68, 89

Hash table size: 10

R will be 7

Double Hashing



Double Hashing Example

	insert(76) 76%7 = 6	insert(93) 93%7 = 2	insert(40) 40%7 = 5	insert(47) 47%7 = 5 5 - (47%5) = 3	insert(10) 10%7 = 3	insert(55) 55%7 = 6 5 - (55%5) = 5
0						
1				47	47	47
2		93	93	93	93	93
3					10	10
4						55
5			40	40	40	40
6	76	76	76	76	76	76
probes:	1	1	1	2	1	2

Double Hashing

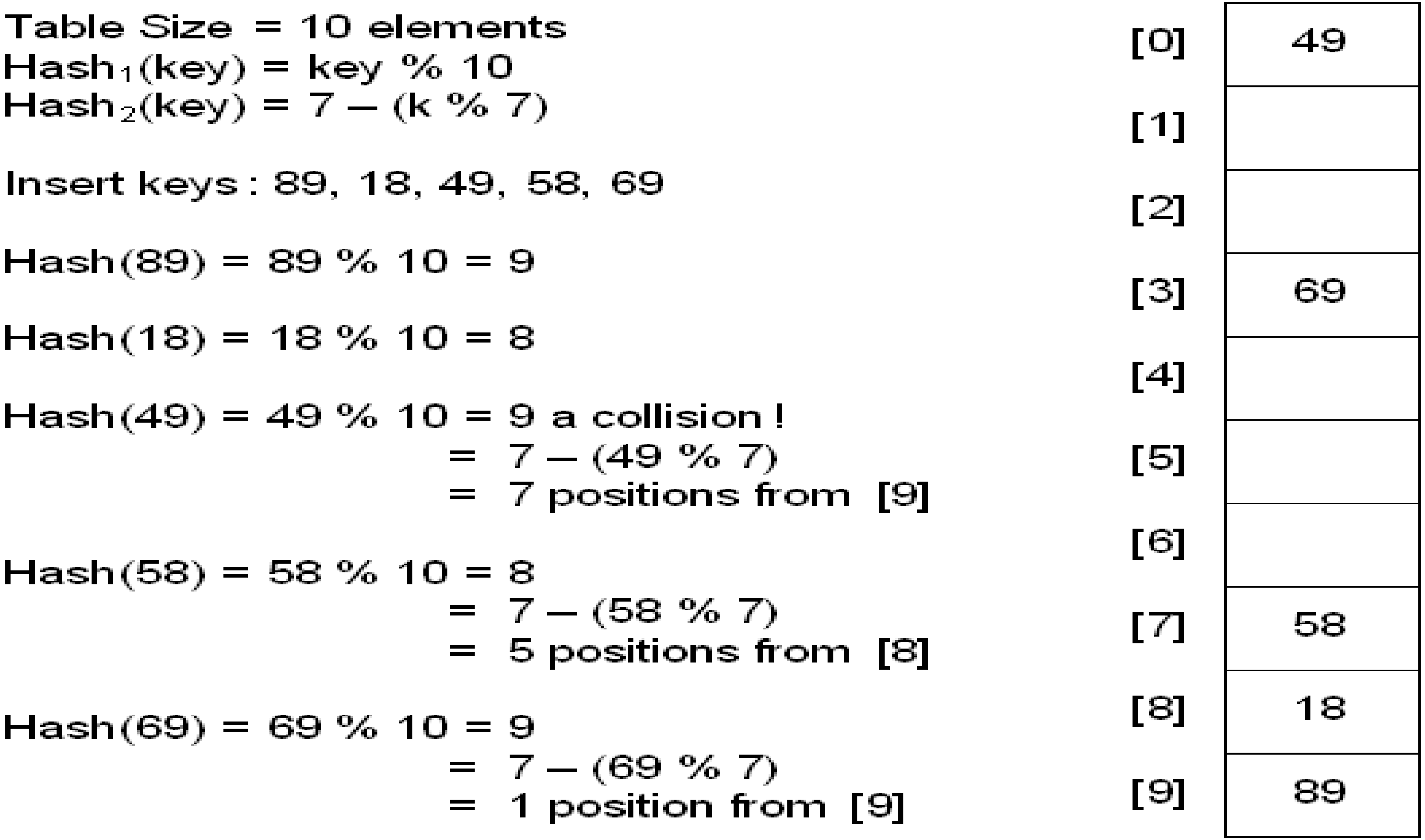


Table Size = 10 elements
Hash$_1$(key) = key % 10
Hash$_2$(key) = 7 – (k % 7)

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]

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

Key	Hash Function h(X)	Index	Collision
79	$h_0(79)$ = (Hash(79) + F(0)) % 10 = ((79 % 10) + 0) % 10 = 9	9	
28	$h_0(28)$ = (Hash(28) + F(0)) % 10 = ((28 % 10) + 0) % 10 = 8	8	
39	$h_0(39)$ = (Hash(39) + F(0)) % 10 = ((39 % 10) + 0) % 10 = 9	9	first collision occurs
	$h_1(39)$ = (Hash(39) + F(1)) % 10 = ((39 % 10) + 1(7-(39 % 7))) % 10 = (9 + 3) % 10 =12 % 10 = 2	2	
68	$h_0(68)$ = (Hash(68) + F(0)) % 10 = ((68 % 10) + 0) % 10 = 8	8	collision occurs
	$h_1(68)$ = (Hash(68) + F(1)) % 10 = ((68 % 10) + 1(7-(68 % 7))) % 10 = (8 + 2) % 10 =10 % 10 =0	0	
89	$h_0(89)$ = (Hash(89) + F(0)) % 10 = ((89 % 10) + 0) % 10 = 9	9	collision occurs
	$h_1(89)$ = (Hash(89) + F(1)) % 10 = ((89 % 10) + 1(7-(89 % 7))) % 10 = (9 + 2) % 10 =10 % 10 = 0	0	Again collision occurs
	$h_2(89)$ = (Hash(89) + F(2)) % 10 = ((89 % 10) + 2(7-(89 % 7))) % 10 = (9 + 4) % 10=13 % 10 =3	3	

Double hashing can be done using :

(hash1(key) + i * hash2(key)) % TABLE_SIZE

Here hash1() and hash2() are hash functions and TABLE_SIZE

is size of hash table.

(We repeat by increasing i when collision occurs)

First hash function is typically hash1(key) = key % TABLE_SIZE

A popular second hash function is : **hash2(key) = PRIME – (key % PRIME)** where PRIME is a prime smaller than the TABLE_SIZE.

A good second Hash function is:

- It must never evaluate to zero
- Must make sure that all cells can be probed.

Thank you