HASHING

Hashing

- In all search techniques like linear search, binary search and search trees, the time required to search an element depends on the total number of elements present in that data structure.
- In all these search techniques, as the number of elements increases the time required to search an element also increases linearly.
- Hashing is another approach in which time required to search an element doesn't depend on the total number of elements.
- Using hashing data structure, a given element is searched with **constant time complexity**. Hashing is an effective way to reduce the number of comparisons to search an element in a data structure.

Hashing

- Hashing is defined as follows...

Hashing is the process of indexing and retrieving element (data) in a data structure to provide a faster way of finding the element using a hash key.

- Here, the hash key is a value which provides the index value where the actual data is likely to be stored in the data structure.
- In this data structure, we use a concept called **Hash table** to store data. All the data values are inserted into the hash table based on the hash key value.
- The hash key value is used to map the data with an index in the hash table. And the hash key is generated for every data using a **hash function**. That means every entry in the hash table is based on the hash key value generated using the hash function.

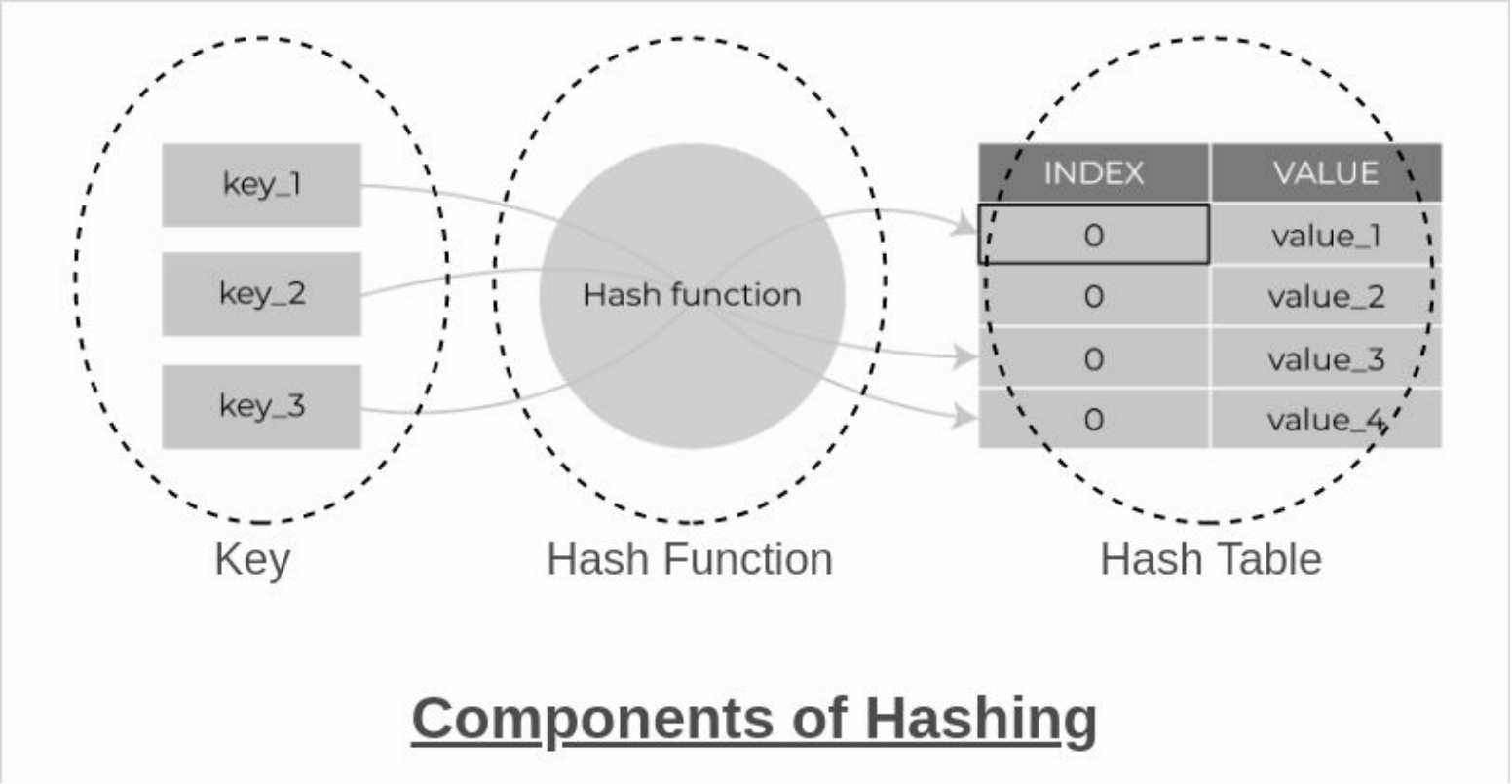
Hashing

- Hash Table is defined as follows...

Hash table is just an array which maps a key (data) into the data structure with the help of hash function such that insertion, deletion and search operations are performed with constant time complexity (i.e. O(1)).

- Hash tables are used to perform insertion, deletion and search operations very quickly in a data structure.
- Using hash table concept, insertion, deletion, and search operations are accomplished in constant time complexity.
- Generally, every hash table makes use of a function called **hash function** to map the data into the hash table.
- A hash function is defined as follows...
- Hash function is a function which takes a piece of data (i.e. key) as input and produces an integer (i.e. hash value) as output which maps the data to a particular index in the hash table.

Hashing Concept-Components of Hashing



Components of Hashing

There are majorly three components of hashing:

1. **Key:** A **Key** can be anything string or integer which is fed as input in the hash function the technique that determines an index or location for storage of an item in a data structure.
2. **Hash Function:** The **hash function** receives the input key and returns the index of an element in an array called a hash table. The index is known as the **hash index**.
3. **Hash Table:** Hash table is a data structure that maps keys to values using a special function called a hash function. Hash stores the data in an associative manner in an array where each data value has its own unique index.

Types of Hash functions

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

1. Division Method.
2. Mid Square Method.
3. Folding Method.
4. Multiplication Method.

Division Method

This is the most simple and easiest method to generate a hash value. The hash function divides the value k by M and then uses the remainder obtained.

Formula:

h(K) = k mod M

Here,

***k** is the key value, and*

***M** is the size of the hash table.*

It is best suited that **M** is a prime number as that can make sure the keys are more uniformly distributed. The hash function is dependent upon the remainder of a division.

Example:

k = 1276

M = 11

h(1276) = 1276 mod 11

= 0

Division Method

Pros:

1. This method is quite good for any value of M.
2. The division method is very fast since it requires only a single division operation.

Cons:

1. This method leads to poor performance since consecutive keys map to consecutive hash values in the hash table.
2. Sometimes extra care should be taken to choose the value of M.

Mid Square Method

The mid-square method is a very good hashing method. It involves two steps to compute the hash value-

1. Square the value of the key k i.e. k^2
2. Extract the middle **r** digits as the hash value.

Formula:

h(K) = h(k x k)

Here,

***k** is the key value.*

The value of **r** can be decided based on the size of the table.

Example:

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

k = 60

k x k = 60 x 60

= 3600

h(60) = 60

The hash value obtained is 60

Mid Square Method

Pros:

1. The performance of this method is good as most or all digits of the key value contribute to the result. This is because all digits in the key contribute to generating the middle digits of the squared result.
2. The result is not dominated by the distribution of the top digit or bottom digit of the original key value.

Cons:

1. The size of the key is one of the limitations of this method, as the key is of big size then its square will double the number of digits.
2. Another disadvantage is that there will be collisions but we can try to reduce collisions.

Digit Folding Method

This method involves two steps:

1. Divide the key-value **k** into a number of parts i.e. **k1, k2, k3,….,kn**, where each part has the same number of digits except for the last part that can have lesser digits than the other parts.
2. Add the individual parts. The hash value is obtained by ignoring the last carry if any.

Formula:

k = k1, k2, k3, k4, ….., kn

s = k1+ k2 + k3 + k4 +….+ kn

h(K)= s

Here,

s** is obtained by adding the parts of the key **k

Example:

k = 12345

k1 = 12, k2 = 34, k3 = 5

s = k1 + k2 + k3

= 12 + 34 + 5

= 51

h(K) = 51

Multiplication Method

This method involves the following steps:

- Choose a constant value A such that $0 < A < 1$.
- Multiply the key value with A.
- Extract the fractional part of kA.
- Multiply the result of the above step by the size of the hash table i.e. M.
- The resulting hash value is obtained by taking the floor of the result obtained in step 4.

Formula:

h(K) = floor (M (kA mod 1))

Here,

***M** is the size of the hash table.*

***k** is the key value.*

***A** is a constant value.*

Multiplication Method

Example:

k = 12345

A = 0.357840

M = 100

*h(12345) = floor[100 (12345*0.357840 mod 1)]*

= floor[100 (4417.5348 mod 1)]

= floor[100 (0.5348)]

= floor[53.48]

= 53

Multiplication Method

Pros:

The advantage of the multiplication method is that it can work with any value between 0 and 1, although there are some values that tend to give better results than the rest.

Cons:

The multiplication method is generally suitable when the table size is the power of two, then the whole process of computing the index by the key using multiplication hashing is very fast.

Complexity

Complexity of calculating hash value using the hash function

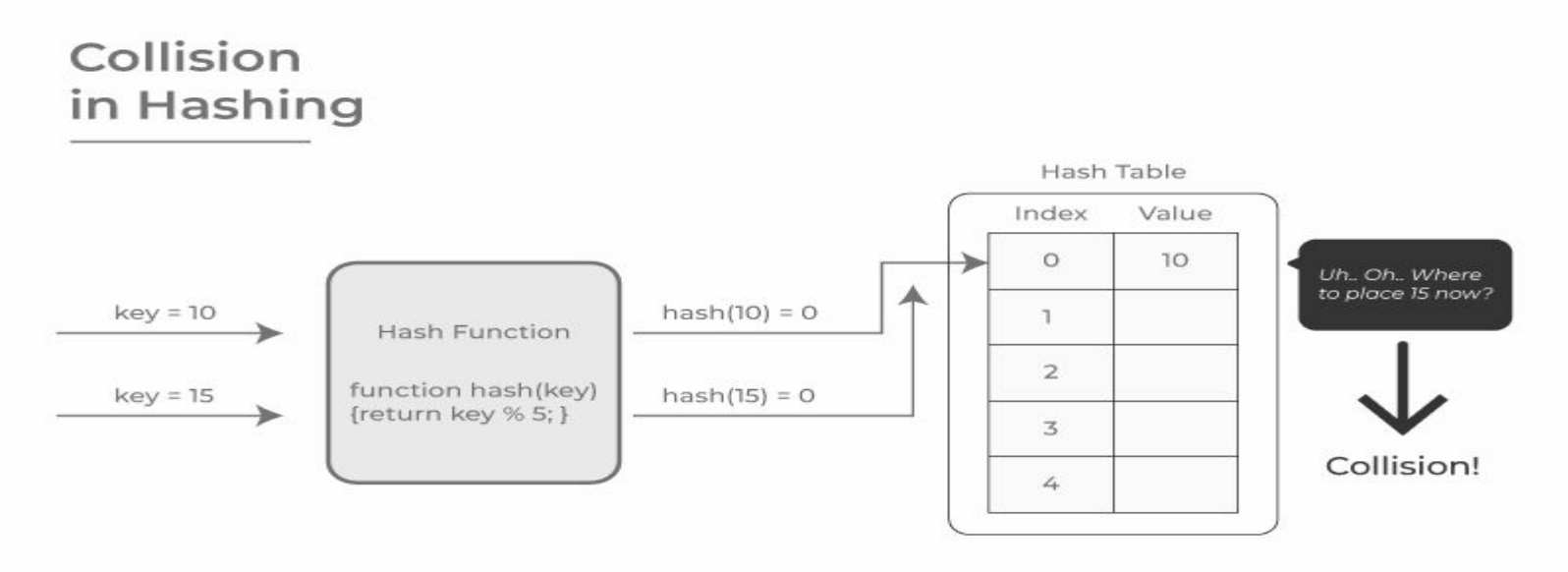
- Time complexity: O(n)
- Space complexity: O(1)

Problem with Hashing

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

Collision

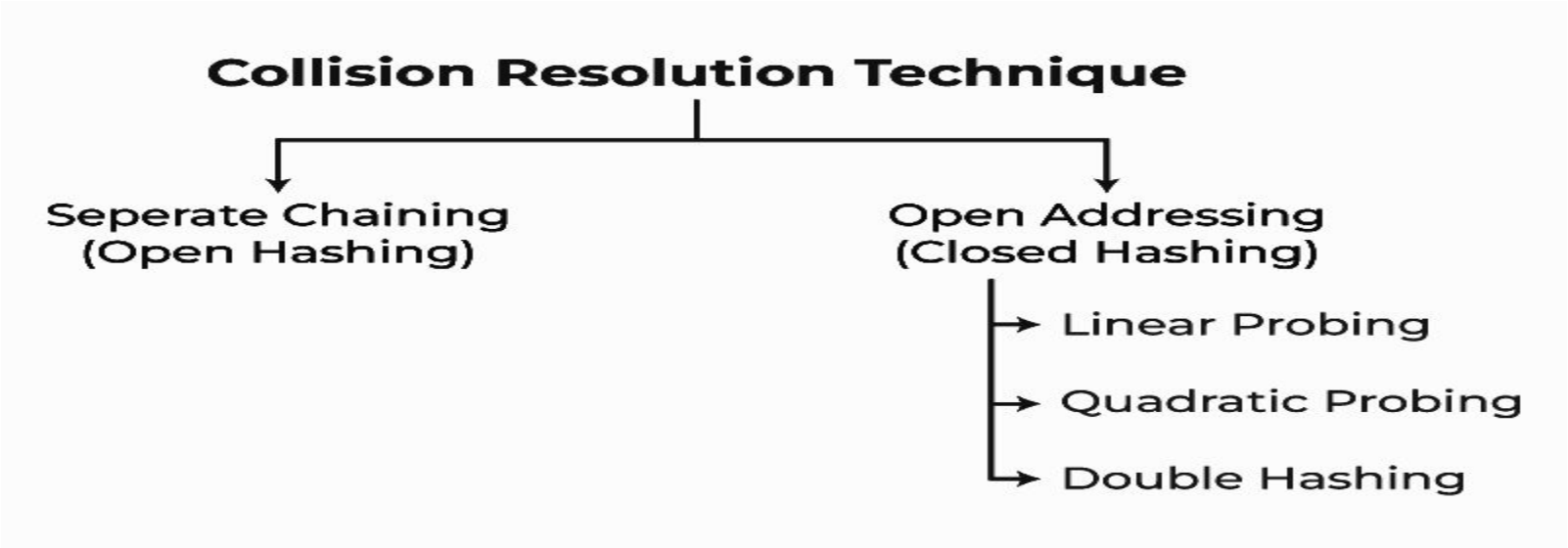
- The hashing process generates a small number for a big key, so there is a possibility that two keys could produce the same value. The situation where the newly inserted key maps to an already occupied, and it must be handled using some collision handling technology.



How to handle Collisions?

There are mainly two methods to handle collision:

1. Separate Chaining:
2. Open Addressing:



Separate Chaining

- The idea is to make each cell of the hash table point to a linked list of records that have the same hash function value. Chaining is simple but requires additional memory outside the table.
- Example: We have given a hash function and we have to insert some elements in the hash table using a separate chaining method for collision resolution technique.

Hash function = key % 5,

Elements = 12, 15, 22, 25 and 37.

Let’s see step by step approach to how to solve the above problem:

- **Step 1:** First draw the empty hash table which will have a possible range of hash values from 0 to 4 according to the hash function provided.
- **Step 2:** Now insert all the keys in the hash table one by one. The first key to be inserted is 12 which is mapped to bucket number 2 which is calculated by using the hash function 12%5=2.

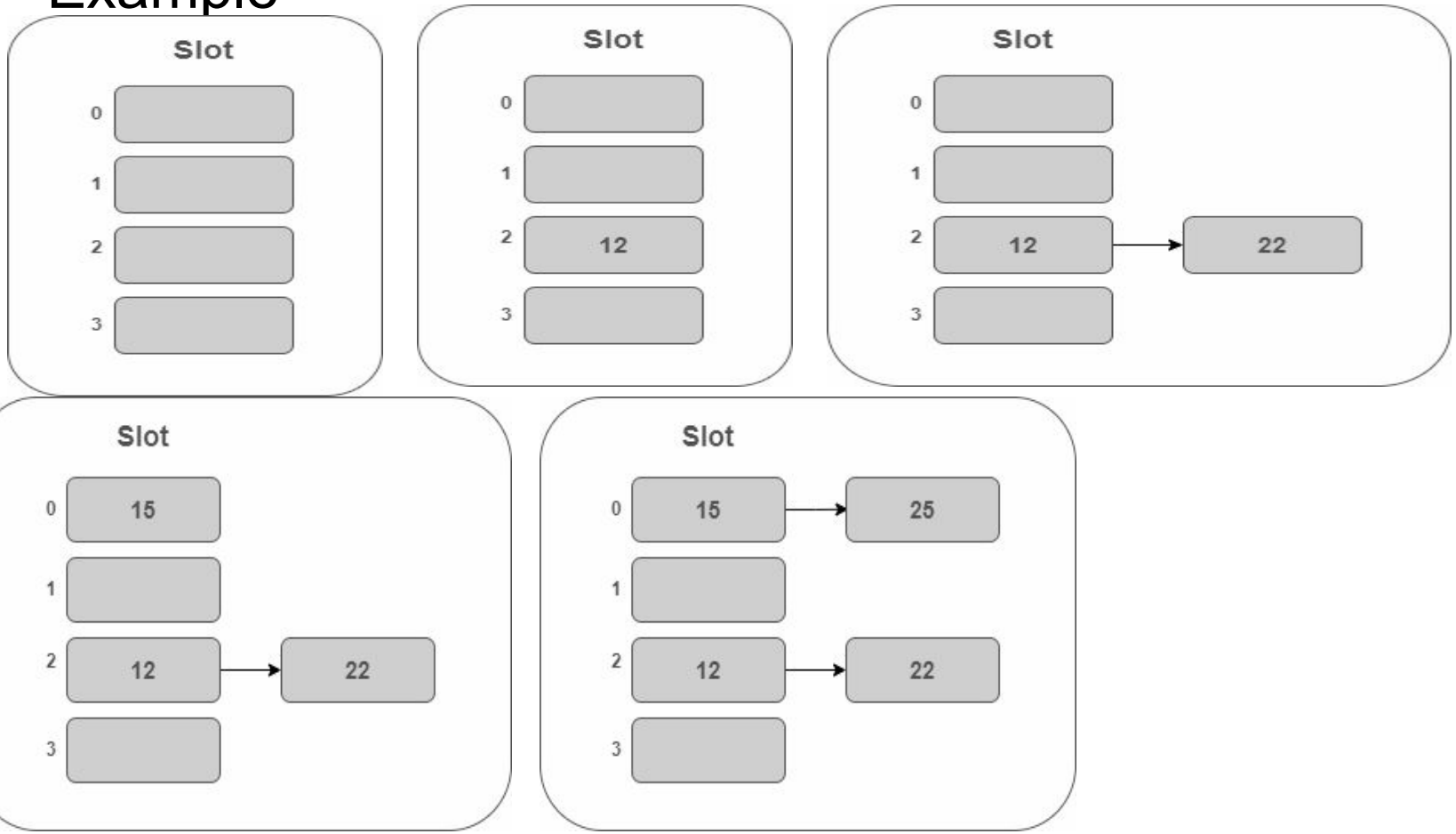
Separate Chaining

Step 3: The next key is 15. It will map to slot number 0 because 15%5=0. So insert it into bucket number 5.

Step 4: Now the next key is 22. It will map to bucket number 2 because 22%5=2. But bucket 2 is already occupied by key 12. So separate chaining method will handle this collision by creating a linked list to bucket 2

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

Example


Slot
0
1
2
3
Slot
0
1
2
12
3
Slot
0
1
2
12
22
3
Slot
0
15
1
2
12
22
3
Slot
0
15
25
1
2
12
22
3

Open Addressing

- 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). This approach is also known as closed hashing. This entire procedure is based upon probing. We will understand the types of probing ahead:
  - ***Insert(k):*** *Keep probing until an empty slot is found. Once an empty slot is found, insert k.*
  - ***Search(k):*** *Keep probing until the 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".*

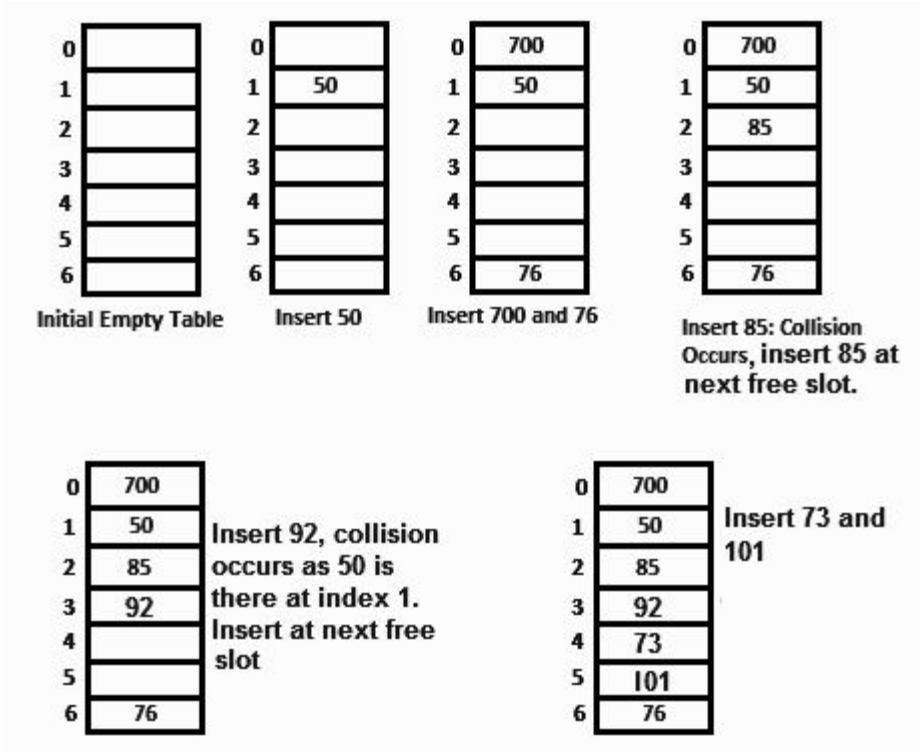
    The insert can insert an item in a deleted slot, but the search doesn't stop at a deleted slot.

Linear Probing

- In linear probing, the hash table is searched sequentially that starts from the original location of the hash. If in case the location that we get is already occupied, then we check for the next location.
- *The function used for rehashing is as follows: rehash(key) = (n+1)%table-size.*
  For example, The typical gap between two probes is 1 as seen in the example below:
- *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

Example

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



Quadratic Probing

- Quadratic probing is a method with the help of which we can solve the problem of clustering. This method is also known as the **mid-square** method. In this method, we look for the **i2‘th** slot in the **ith** iteration.
- We always start from the original hash location. If only the location is occupied then we check the other slots.

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

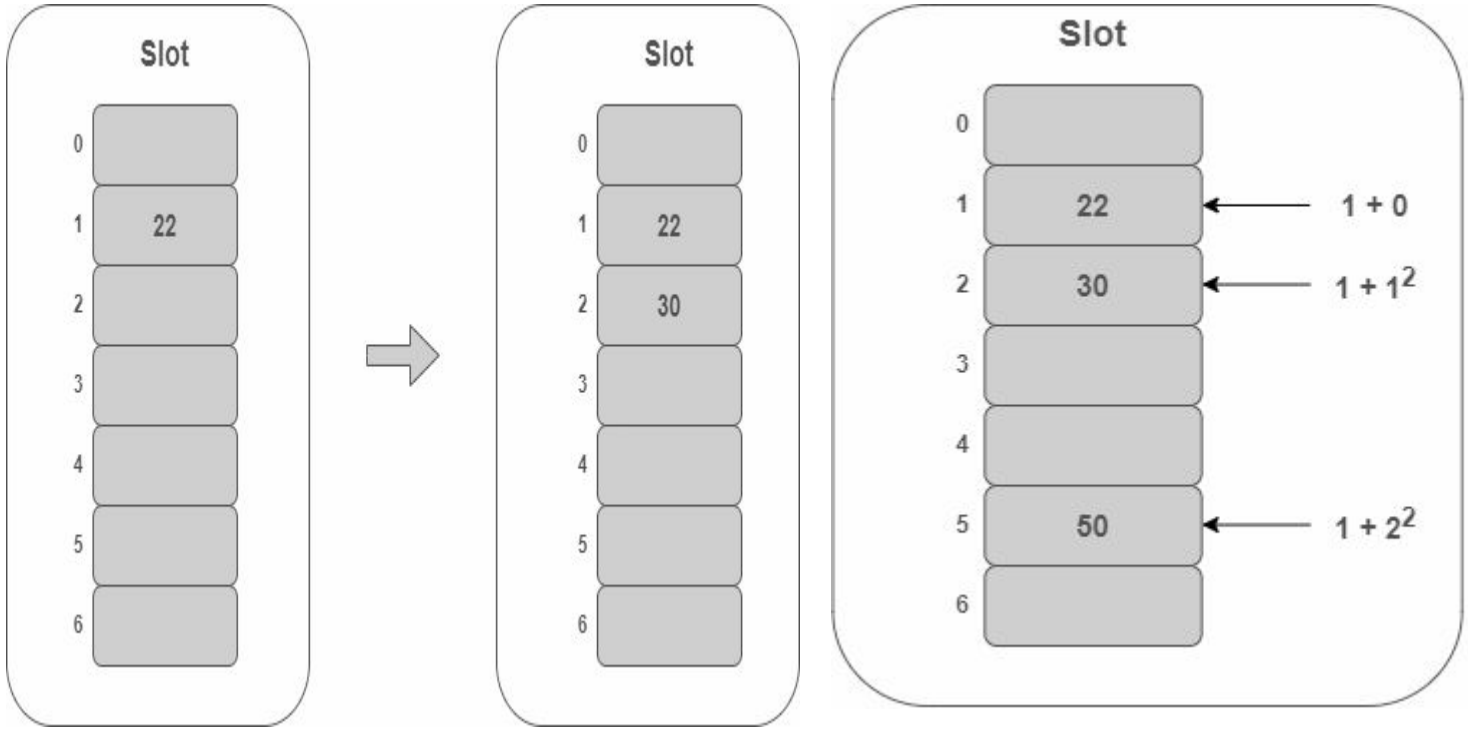
If (hash(x) + 2*2) % S is also full, then we try (hash(x) + 3*3) % S

Example

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

- **Step 1:** Create a table of size 7.
- **Step 2** – Insert 22 and 30
    i. Hash(22) = 22 % 7 = 1, Since the cell at index 1 is empty, we can easily insert 22 at slot 1.
    ii. Hash(30) = 30 % 7 = 2, Since the cell at index 2 is empty, we can easily insert 30 at slot 2.
- **Step 3:** Inserting 50
    a. Hash(50) = 50 % 7 = 1
    b. In our hash table slot 1 is already occupied. So, we will search for slot 1+12, i.e. 1+1 = 2,
    c. Again slot 2 is found occupied, so we will search for cell 1+22, i.e.1+4 = 5,
    d. Now, cell 5 is not occupied so we will place 50 in slot 5.

Example


Slot
0
1
22
2
3
4
5
6
Slot
0
1
22
2
30
3
4
5
6
Slot
0
1
22
1 + 0
2
30
1 + 1^2
3
4
5
50
1 + 2^2
6

Double Hashing

- The intervals that lie between probes are computed by another hash function. Double hashing is a technique that reduces clustering in an optimized way. In this technique, the increments for the probing sequence are computed by using another hash function. We use another hash function hash2(x) and look for the i*hash2(x) slot in the **ith** 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

Insert the keys 27, 43, 692, 72 into the Hash Table of size 7. where first hash-function is **h1(k) = k mod 7** and second hash-function is **h2(k) = 1 + (k mod 5)**

- **Step 1:** Insert 27
  - 27 % 7 = 6, location 6 is empty so insert 27 into 6 slot.
- **Step 2:** Insert 43
  - 43 % 7 = 1, location 1 is empty so insert 43 into 1 slot.
- **Step 3:** Insert 692
  - 692 % 7 = 6, but location 6 is already being occupied and this is a collision
  - So we need to resolve this collision using double hashing.

$$h_{new} = [h1(692) + i * (h2(692)] \% 7$$

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

$$= 9\% 7$$

$$= 2$$

Now, as 2 is an empty slot,

so we can insert 692 into 2nd slot.

- **Step 4:** Insert 72
    - 72 % 7 = 2, but location 2 is already being occupied and this is a collision.
    - So we need to resolve this collision using double hashing.

*h_{new} = [h1(72) + i * (h2(72)] % 7*

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

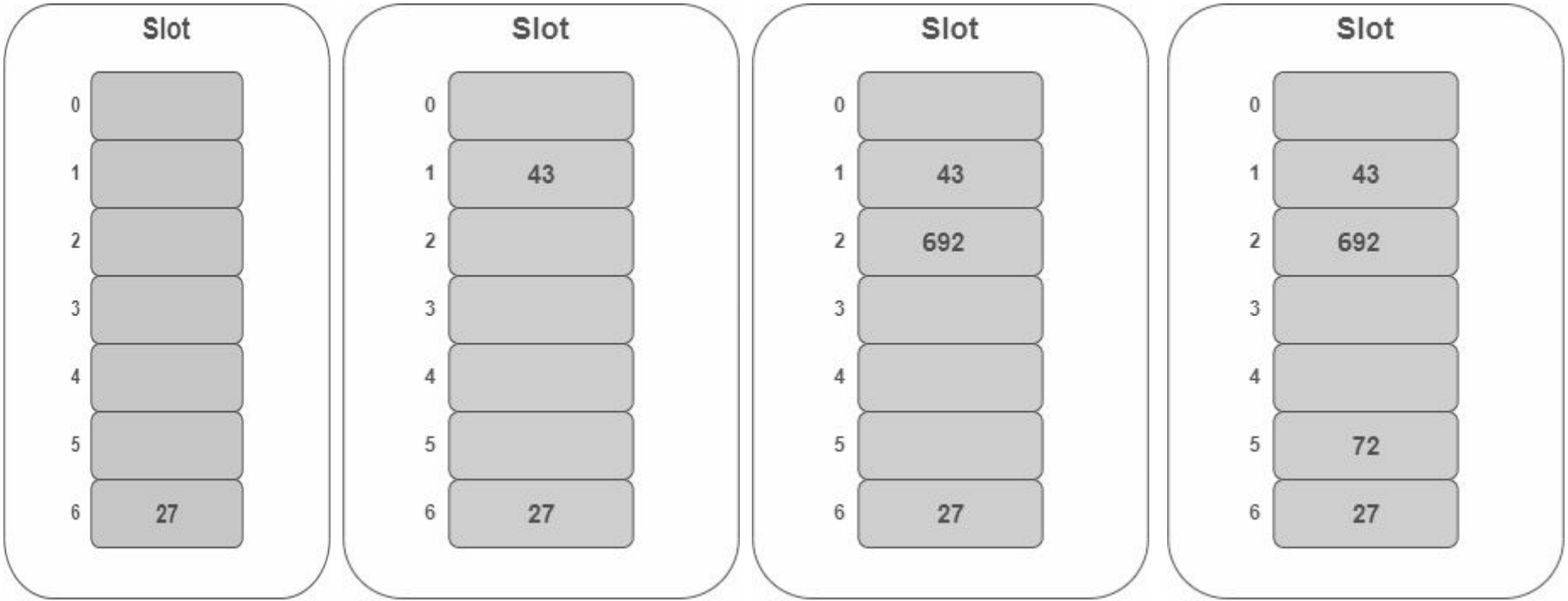
= 5 % 7

= 5,

Now, as 5 is an empty slot,

so we can insert 72 into 5th slot.

Example


Slot
0
1
2
3
4
5
6
27
Slot
0
1
43
2
3
4
5
6
27
Slot
0
1
43
2
692
3
4
5
6
27
Slot
0
1
43
2
692
3
4
5
72
6
27

Comparison

- 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 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