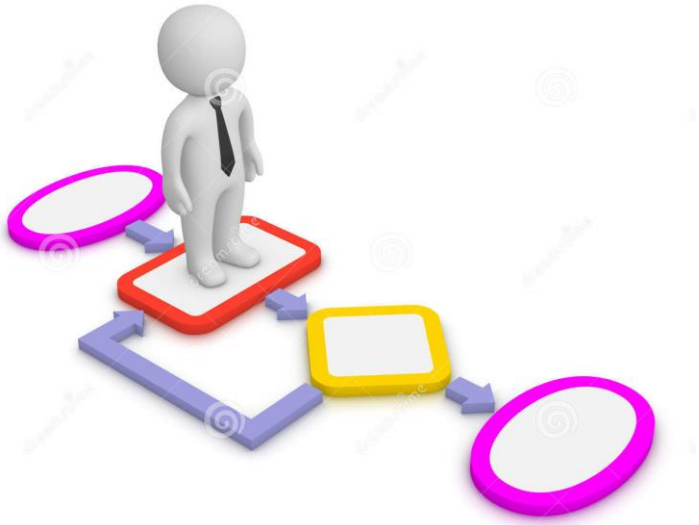
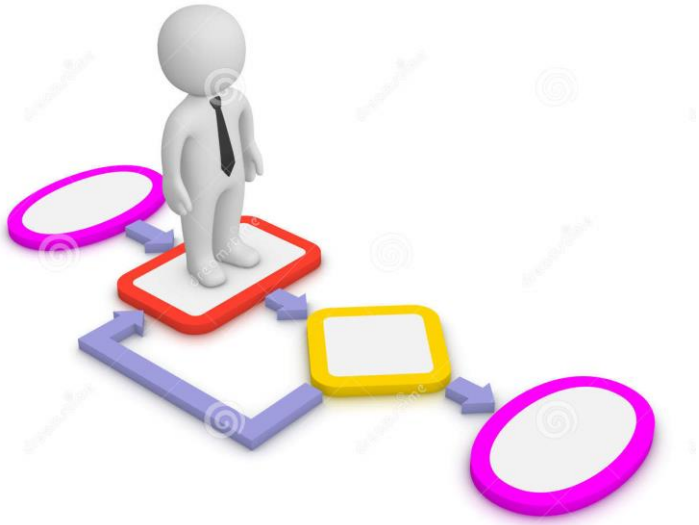


Design & Analysis of Algorithms

Saman Riaz (PhD)
Associate Professor
RSCI, Lahore

Lecture # 08

ANALYSIS OF SEARCHING ALGORITHMS



Do we need searching?

- Computers can search for us
- World wide web
  - Different searching mechanisms, chat gpt, deep seek, yahoo.com, ask.co.uk, google.com„ Spreadsheet
  - list of names
  - searching mechanism to find a name„
- Databases
  - use to search for a record
  - select * from ..
- Large records –1000s takes time –many comparison slow system –user won’t wait long time

Searching Algorithms

- Different types
  - Sequential Search, Binary Search
- „Discuss the search algorithms analyze them
- „Analysis of the algorithms enables programmers to decide which algorithm to use for a specific application

Sequential Search

- Consider the problem of searching for a given element in linked list or array
- We may not know whether the target element occurs in the list in advance of the search.
- Sequential search works the same for both linked list and arrays
- For simplicity here we shall assume an array of integers.

Linear Search

ARRAY-SEARCH(A, n, key)

```
1. for (i=1; i<=n; i++)
2.         if A[i]=key
3.                 return i
4. return -1
```

Analysis

- Count the number of key comparisons because this number gives us the most useful information, this criterion for counting the number of key comparisons can be applied equally well to other search algorithms
- Then number of key comparisons depends on where in the list the search item is located.
- „If search item is first element of L make one key comparison best case
- „Worst case search item is the last item algorithm makes n comparisons

Linear Search – Average and Worst Case

	Worst case
Array	$\Theta(n)$
Linked List	$\Theta(n)$

Average case running time for search in an array can be computed by assuming a key has equal probability of being in any position i.e. *1/n*

Then

$$T(n)=1c*1/n+2c*1/n+....+nc*1/n$$

$$= c/n(1+2+3+...+n)=cn(n+1)/2n =\theta(n)$$

Probabilistic Linear Search

- In Probabilistic Linear Search, the probability of finding a key in an array is assumed to be known
- Let this probability be *p*
- Then probability that key is not found in array is *1-p*
- Probability of key being in a slot is *p/n*

- Expected running time

$$T(n) = \sum_{k=1}^{n} c.k.p/n \quad + c.n(1-p)$$

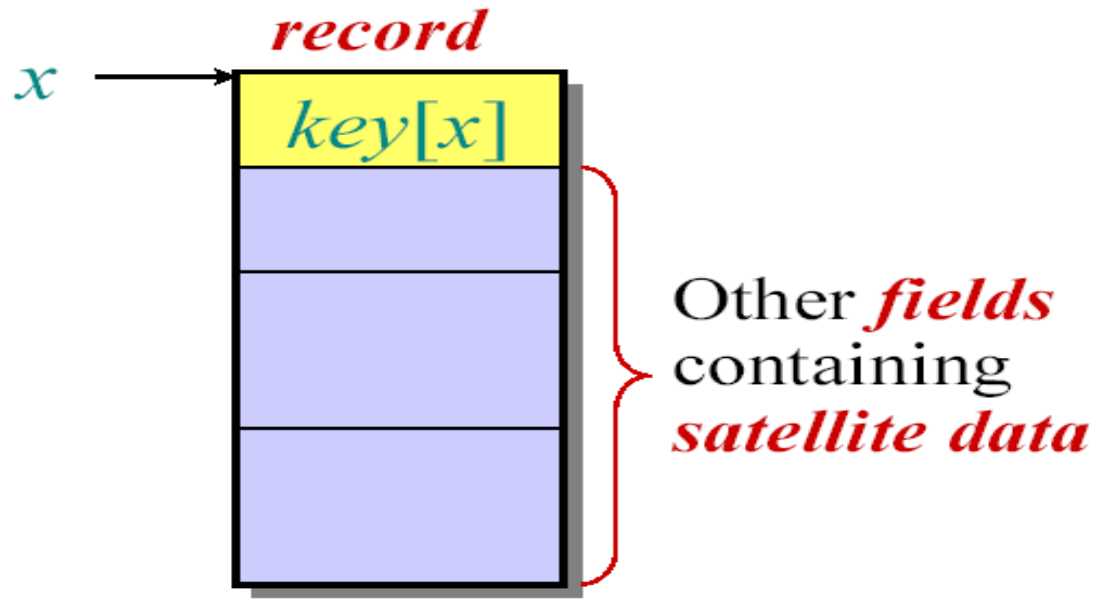
$$=cpn(n+1)/2n+cn(1-p)=cn(1-p/2)+cp/2$$

- For *p=1*, $T(n)=cn/2+c/2=c/2(n+1)= \theta(n)$

HASHING

Dictionary

- Holds *n* records



Operations on T:

- INSERT(T, x)
- DELETE(T, x)
- SEARCH(T, k)

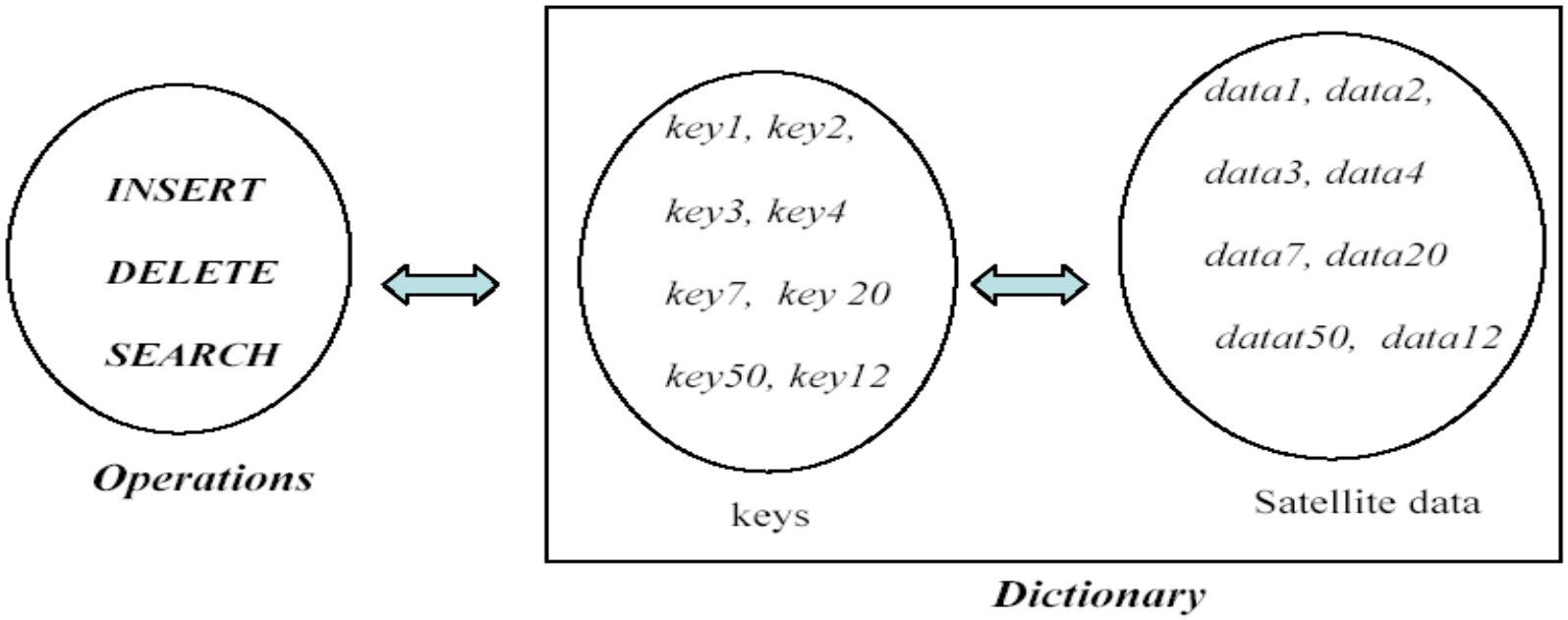
- What data structure should be used to implement *T*?

Hashing

The ***hashing algorithms*** are often used on a special data structure called ***dictionary.***

- A dictionary is a dynamic data structure consisting of a ***set of keys.*** It supports three basic operations: ***insertion***, ***deletion***, and ***search.***

- Generally, the keys in a dictionary can have additional related elements, called ***satellite data,*** as illustrated in the diagram .



Direct Addressing

- Assumptions
  - The set of keys
  - Keys are distinct

$$K \subseteq U = \{0, 1, \ldots, u-1\}$$

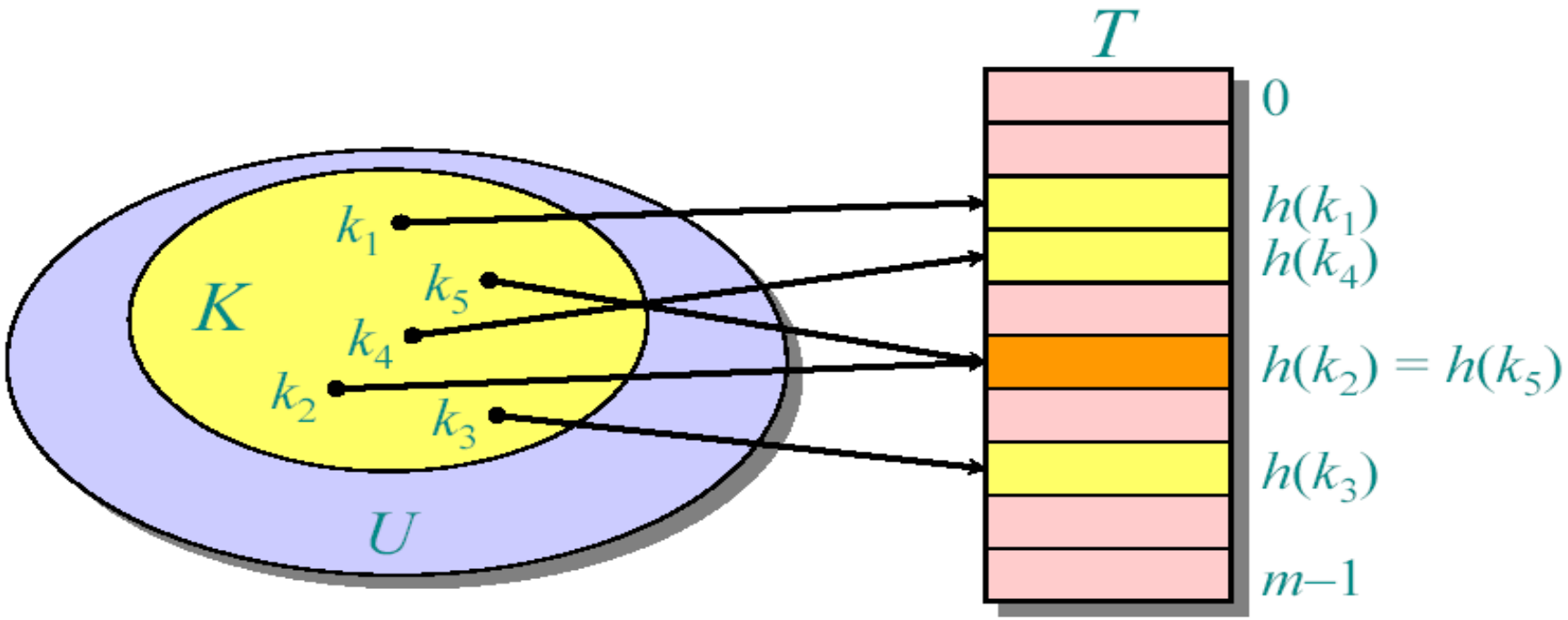
  - Create a table T[0..u-1]

$$T[k] = \begin{cases} x & \text{if } k \in K \text{ and } key[x] = k, \\ \text{NIL} & \text{otherwise.} \end{cases}$$

  - Benefit
    - Each operation takes constant time
  - Drawbacks
    - The range of keys can be large

Hashing

- Solution
  - Use a ***hash function*** *h* to map the universe *U* of all keys into {0, 1, …, *m*–1}



Hash Table

- The mapped keys are stored into table called ***hash table***
- The table consists of *m* cells
- A hash table requires much less storage than a direct address table
- With direct addressing, an element in key *k* is stored in slot *k*, with hashing, this element is stored in slot *h(k)*
- So the hash function $h : U \rightarrow \{0, 1, \ldots, m-1\}$
- *h(k)* is also called hash value of key *k*

Hash Table

- Two or more than two keys may hash to the same slot
- When a record to be inserted maps to an already occupied slot in T, a ***collision*** occurs
- Can we avoid collisions altogether?
- Not if $|U| > m$
- We need a method to resolve collisions that occur

Open hashing/Closed addressing

- A key is always stored in the bucket it's hashed to. Collisions are dealt with using **separate data structures** on a per-bucket basis.

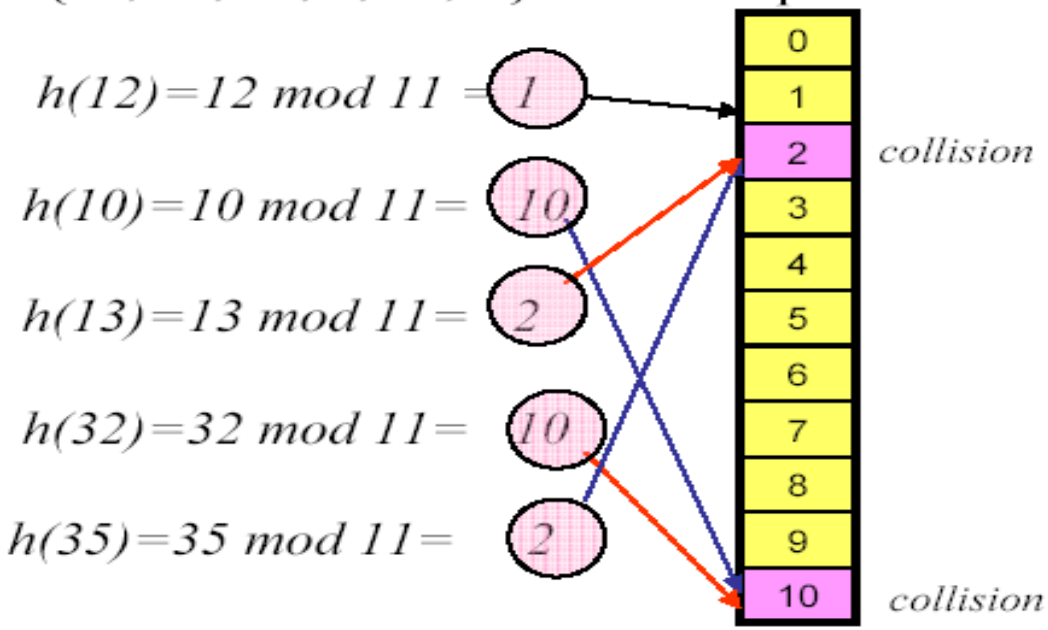
- Arbitrary number of keys per bucket.

Collisions

- Consider the hash function :

$$h(k) = k \bmod 11$$

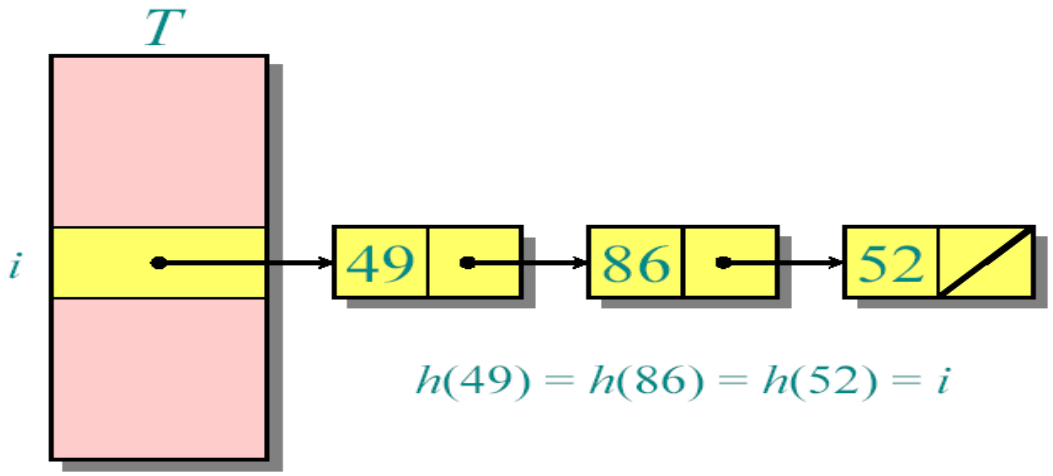
The keys {*12, 10, 13, 2, 14, 3*} would map as follows



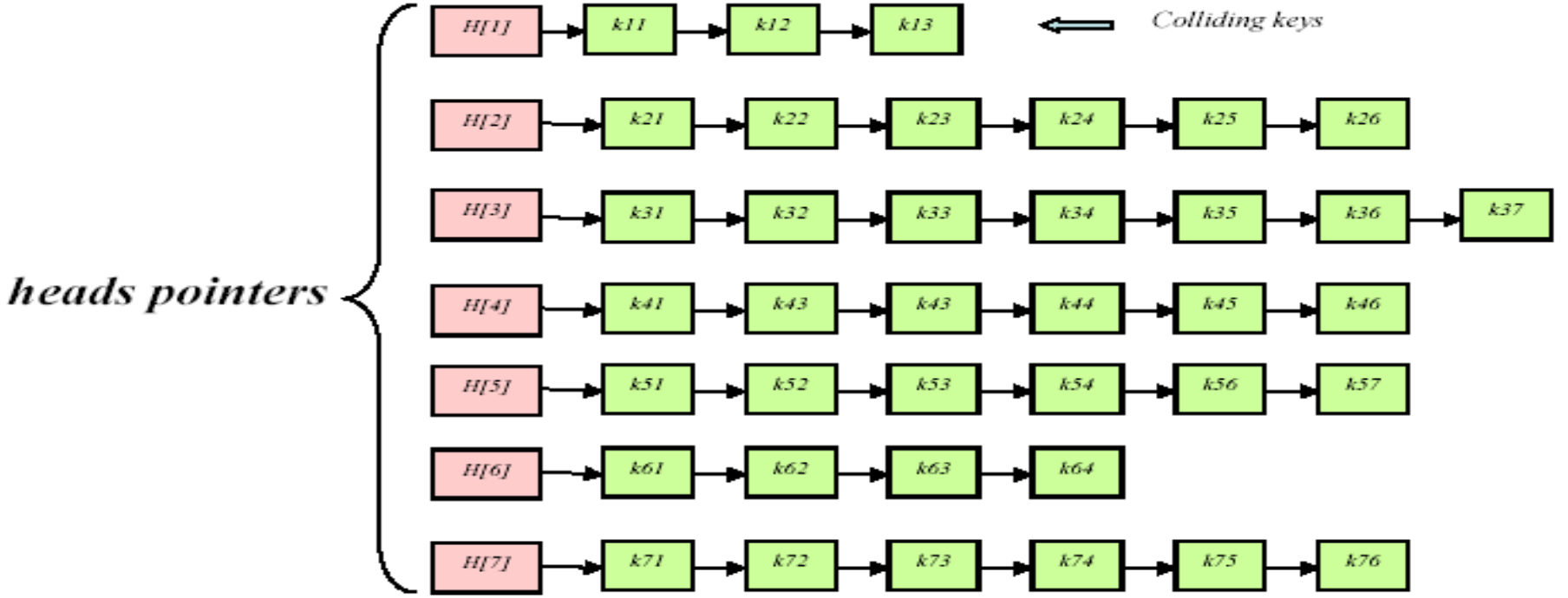
In this example, the keys {*10 , 32*} map to cell 10, and the keys { *13, 35*} map to cell 2. Thus, the keys *10* and *32* have ***collisions***. Same way, the keys *13* and *35* have collisions.

Collision Resolution by Chaining(Close addressing)

- Records in the same slot are linked into a list



Collision Resolution by Chaining (contd…)



- The chained hashing is ***flexible***. The new keys can be inserted easily. Searching is fast. However, there is ***storage overhead*** for pointers. Also, deletion is inefficient.

Open addressing

- Collisions are dealt with by searching for **another empty buckets** within the hash table array itself.
- At most one key per bucket

1. Linear Probing

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

1. Quadratic Probing
- $h(k)=(h(k) + i^2) \bmod m$

Open addressing

1. 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.

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

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

Closed hashing/ open addressing

1. Quadratic Probing
   If you observe carefully, then you will understand that the interval between probes will
   increase proportionally to the hash value. Quadratic probing is a method with the help of
   which we can solve the problem of clustering that was discussed above. This method is also
   known as the **mid-square** method. In this method, we look for the **i^2'th** slot in
   the **i^{th}** iteration. We always start from the original hash location. If only the location is
   occupied then we check the other slots.

- h(k)=(h(k) + i^2) mod m