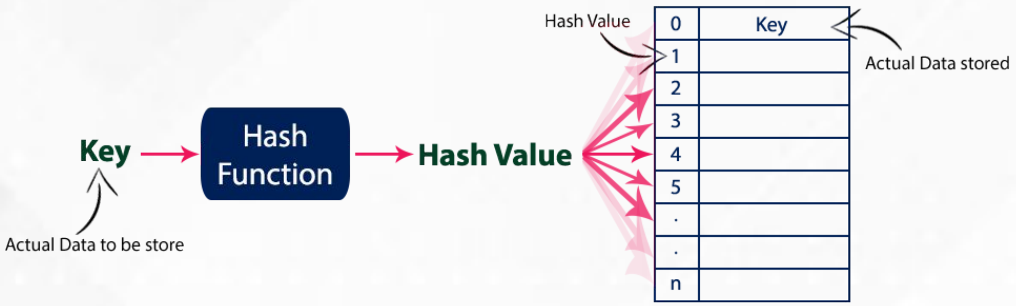
Data Structures (DS)
GTU # 3130702

Unit-4

Hashing & File Structure (Hashing)

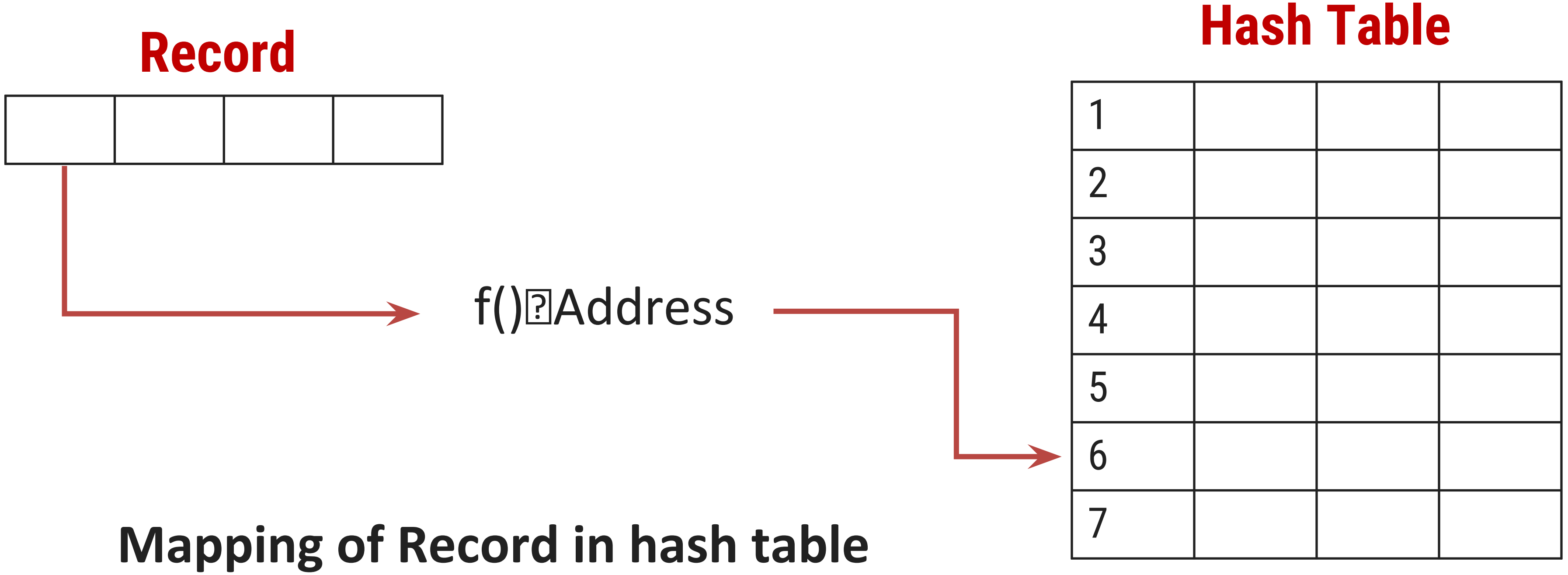


What is Hashing?

- **Sequential search** requires, on the average **O(n) comparisons** to **locate an element**, so many comparisons are not desirable for a large database of elements.
- **Binary search** requires much fewer comparisons on the average **O (log n)** but there is an additional requirement that the **data should be sorted**. Even with best sorting algorithm, sorting of elements require O(n log n) comparisons.
- There is **another** widely used **technique** for **storing of data** called **hashing**. It does away with the requirement of keeping data sorted (as in binary search) and its best case timing complexity is of constant order O(1). In its worst case, hashing algorithm starts behaving like linear search.
- **Best case** timing **behavior** of searching using hashing = **O(1)**
- **Worst case** timing Behavior of searching using hashing = **O(n)**

What is Hashing?

- In hashing, **the record** for a key value "**key**", is **directly referred** by **calculating** the **address** from the key value.
- **Address** or location of an element or record x, is **obtained** by **computing** some arithmetic **function** f.
- **f(key)** gives the address of x in the table.



1			
2			
3			
4			
5			
6			
7			

Mapping of Record in hash table

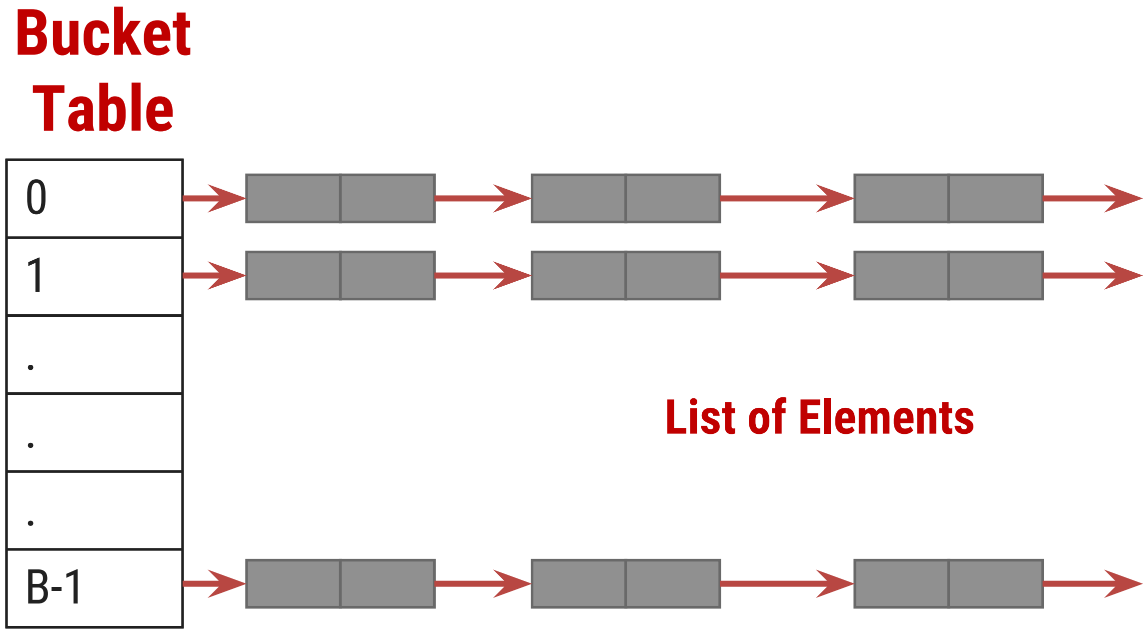
Hash Table Data Structure

- There are two different forms of hashing.

1. **Open hashing or external hashing**
    - Open or external hashing, allows records to be stored in unlimited space (could be a hard disk).
    - It places no limitation on the size of the tables.
2. **Close hashing or internal hashing**
    - Closed or internal hashing, uses a fixed space for storage and thus limits the size of hash table.

Open Hashing Data Structure

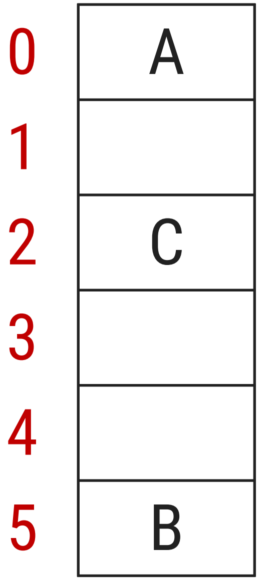
- The basic idea is that the **records [elements]** are **partitioned** into **B classes**, numbered 0,1,2 … B-1
- A Hashing function **f(x)** maps a record with **key x** to an integer value between **0 and B-1**
- Each **bucket** in the **bucket table** is the **head** of the **linked list** of records mapped to that bucket



The open hashing data organization

Close Hashing Data Structure

- A closed hash table **keeps the elements in the bucket** itself.
- Only **one element can be put** in the bucket.
- If we **try to place an element** in the bucket and find **it already holds** an element, then we say that a **collision** has **occurred**.
- In **case of collision**, the element should be **rehashed** to alternate empty location within the bucket table.
- In closed hashing, collision handling is a very important issue.



0	A
1	
2	C
3	
4	
5	B

Hashing Functions

Characteristics of a Good Hash Function

- A good hash function avoids collisions.
- A good hash function tends to spread keys evenly in the array.
- A good hash function is easy to compute.

Different hashing functions

1. Division-Method
2. Midsquare Methods
3. Folding Method
4. Digit Analysis
5. Length Dependent Method
6. Algebraic Coding
7. Multiplicative Hashing

Division-Method

- In this method we use **modular arithmetic system** to **divide** the **key value** by **some integer** divisor **m** (may be table size).
- It gives us the location value, where the element can be placed.
- We can write, **L = (K mod m) + 1**,
  - **L** = location in table/file
  - **K** = key value
  - **m** = table size/number of slots in file
- Suppose, **k = 23, m = 10** then
  - L = (23 mod 10) + 1= 3 + 1=4
  - The key whose **value is 23** is placed in **4th location**.

Midsquare Methods

- In this case, we **square the value of a key** and take the **number of digits required** to form an address, from the **middle position** of squared value.
- Suppose a **key** value is **16**
  - Its **square is 256**
  - Now if we want **address of two digits**
  - We select the address as **56** (i.e. two digits starting from middle of 256)

Folding Method

- Most machines have a **small number of primitive data types** for which there are arithmetic instructions
- Frequently **key** to be used will **not fit** easily in to one of these **data types**
- It is **not possible** to **discard** the **portion** of the **key** that does not fit into such an arithmetic data type
- The **solution** is to **combine** the various **parts of the key** in such a way that all parts of the key affect for final result such an operation is termed folding of the key
- That is the **key** is actually **partitioned** into number of parts, **each part** having the **same length** as that of the required address
- **Add** the **value** of each parts, **ignoring** the final **carry** to get the **required address**

Folding Method

- This is done in two ways
- **Fold-shifting:** Here **actual values** of **each parts** of key are **added**
    - Suppose, the **key** is : **12345678**, and the required address is of two digits,
    - Break the key into: **12, 34, 56, 78**
    - Add these, we get 12 + 34 + 56 + 78 : **180**, ignore first 1 we get **80 as location**
- **Fold-boundary:** Here the **reversed values of outer parts** of key are added
    - Suppose, the **key** is : **12345678**, and the required address is of two digits,
    - Beak the key into: **21, 34, 56, 87**
    - Add these, we get 21 + 34 + 56 + 87 : **198**, ignore first 1 we get **98 as location**

Digit Analysis

- This hashing function is a **distribution-dependent**
- Here we make a **statistical analysis** of **digits** of the **key**, and **select** those **digits** (of fixed position) which **occur** quite **frequently**
- Then reverse or **shifts the digits** to get the **address**
- For example,
  - The key is : **9861234**
  - If the statistical analysis has revealed the fact that the **third** and **fifth** position digits occur quite frequently,
  - We **choose** the **digits** in **these positions** from the key
  - So we get, **62**. **Reversing** it we get **26 as the address**

Length Dependent Method

- In this type of hashing function **we use the length of the key** along **with some portion of the key** to produce the address, directly.
- In the **indirect method**, the **length of the key** along with some portion of the key is **used** to obtain **intermediate value**.

Algebraic Coding

- Here a **n bit key** value is **represented as a polynomial**.
- The **divisor polynomial** is then **constructed** based on the **address range** required.
- The **modular division** of **key-polynomial** by **divisor polynomial**, to get the address-polynomial.
    - Let **f(x)** = polynomial of **n bit key** = $a_1 + a_2x + \ldots\ldots. + a_nx^{n-1}$
    - **d(x)** = **divisor polynomial** = $d_1 + d_2x + \ldots. + d_nx^{n-1}$
    - Required **address** polynomial will be **f(x) mod d(x)**

Multiplicative Hashing

- This method is based on obtaining an **address** of a **key**, **based on the multiplication value**.
- If **k** is the **non-negative key**, and a **constant c**, (**0 < c < 1**)
  - Compute **kc mod 1**, which is a fractional part of kc.
  - **Multiply** this fractional part **by m** and **take a floor value** to get the **address**

$$\lfloor m\ (kc \bmod 1) \rfloor$$

$$0 < h\ (k) < m$$

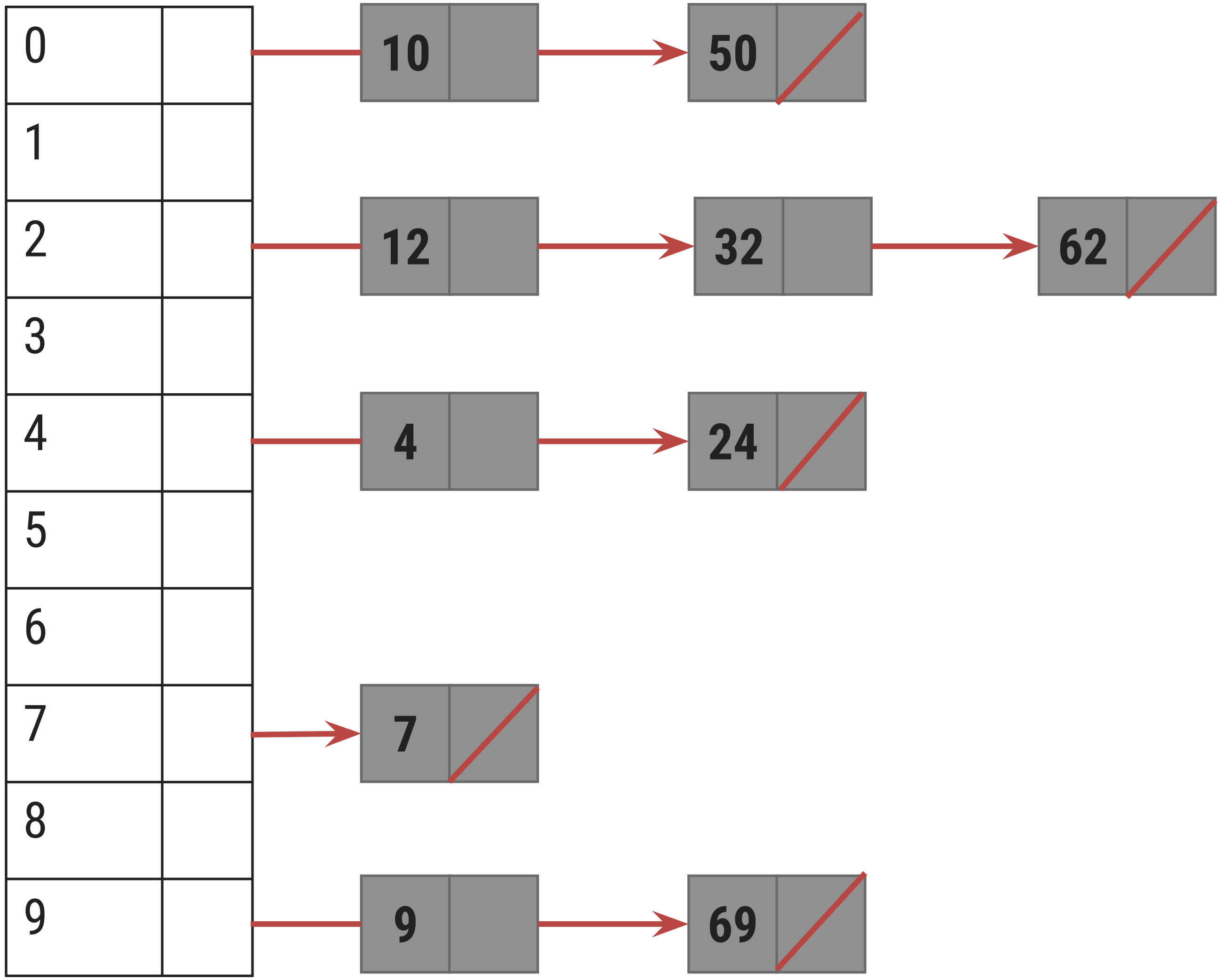
Collision Resolution Strategies

- Collision resolution is the main problem in hashing.
- If the element to be inserted is mapped to the same location, where an element is already inserted then we have a **collision** and it must be resolved.
- There are several strategies for collision resolution. The most commonly used are :
  - **Separate chaining** - used with open hashing
  - **Open addressing** - used with closed hashing

Separate chaining

- In this strategy, a **separate list** of all elements mapped to the same value is maintained.
- Separate chaining is based on **collision avoidance**.
- If memory space is tight, separate chaining should be avoided.
- Additional memory space for links is wasted in storing address of linked elements.
- **Hashing function** should **ensure even distribution** of elements among buckets; otherwise the **timing behaviour** of most operations on hash table **will deteriorate**.

Separate chaining



A Separate Chaining

Hash Table

Example - Separate chaining

Example : The integers given below are to be **inserted** in a **hash table** with **5 locations** using chaining to resolve collisions.

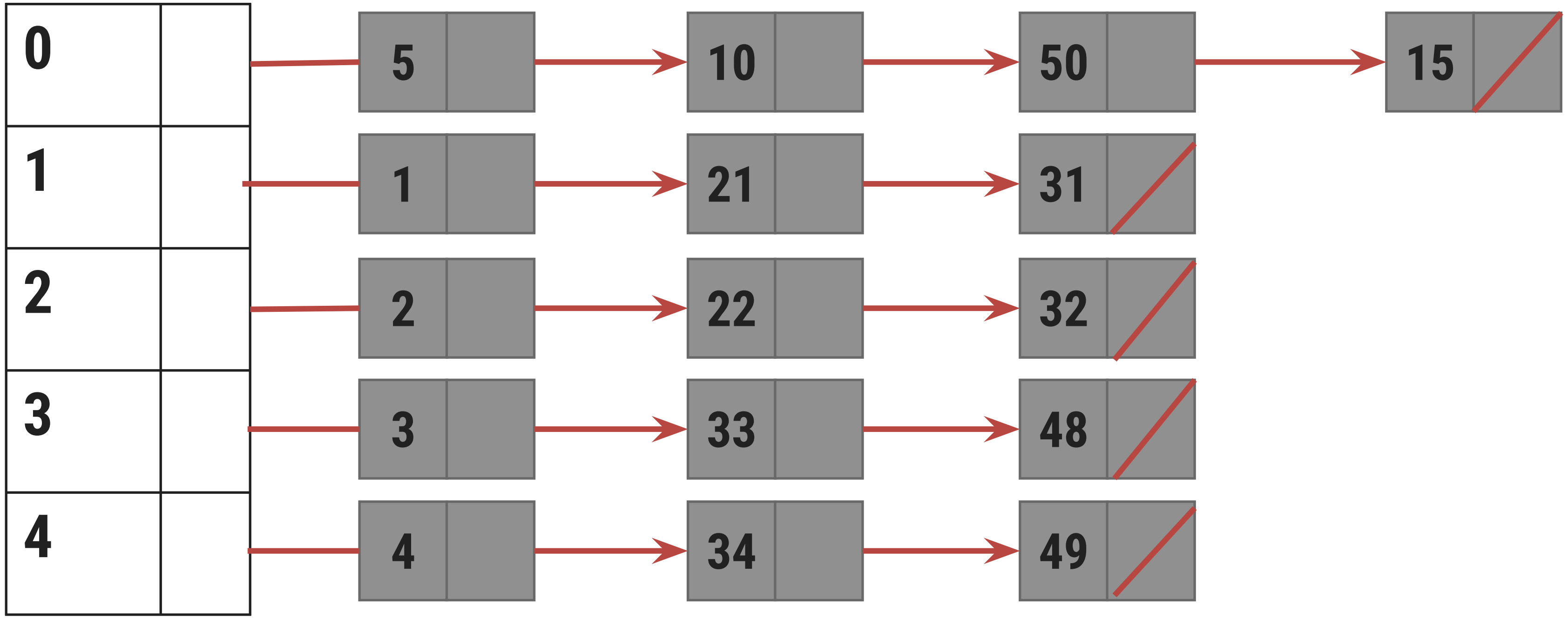
Construct hash table and use simplest hash function.

1, 2, 3, 4, 5, 10, 21, 22, 33, 34, 15, 32, 31, 48, 49, 50

An **element** can be **mapped** to a location in the hash table using the mapping **function key % 10**

Hash Table Location	Mapped elements
0	5, 10, 15, 50
1	1 , 21 , 31
2	2 , 22 , 32
3	3 , 33 , 48
4	4 , 34 , 49

Example - Separate chaining



Hash Table

Open Addressing

- Separate chaining requires additional memory space for pointers.
- Open addressing hashing is an alternate method of handling collision.
- In **open addressing**, if a **collision** occurs, **alternate cells are tried** until an empty cell is found.
  a. Linear probing
  b. Quadratic probing
  c. Double hashing.

Linear Probing

- In **linear probing**, whenever there is a **collision**, **cells are searched sequentially** (with wraparound) **for an empty cell**.
- Fig. shows the result of inserting keys **{5,18,55,78,35,15}** using the hash function (f(key)= **key%10**) and linear probing strategy.

	Empty Table	After 5	After 18	After 55	After 78	After 35	After 15
0							**15**
1							
2							
3							
4							
5		**5**	5	5	5	5	5
6				**55**	55	55	55
7						**35**	35
8			**18**	18	18	18	18
9					**78**	78	78

Linear Probing

- Linear probing **is easy to implement** but it suffers from "**primary clustering**"
- When many **keys** are **mapped** to the **same location** (clustering), linear probing **will not distribute** these keys **evenly** in the hash table.
- These **keys** will be **stored** in **neighbourhood** of the location where they are mapped.
- This will **lead to clustering** of keys around the point of collision

Quadratic probing

- One way of **reducing** "**primary clustering**" is to use quadratic probing to resolve collision.
- Suppose the "**key**" is mapped to the location **j** and the cell **j** is already **occupied**.
- In quadratic probing, the **location j, (j+1), (j+4), (j+9),** ... are examined to find the first empty cell where the key is to be inserted.
- This table **reduces primary clustering**.
- It **does not ensure** that all cells in the table will be examined to **find an empty cell**.
- Thus, it may be **possible** that **key** will **not be inserted** even **if there is an empty cell** in the table.

Double Hashing

- This method requires **two hashing functions** f1 (key) and f2 (key).
- Problem of **clustering** can **easily** be **handled** through double hashing.
- Function **f1 (key)** is known as **primary hash function**.
- In case the address obtained by f1 (key) is already occupied by a key, the function f2 (key) is evaluated.
- The second function **f2 (key) is used** to **compute** the **increment** to be added to the address obtained by the first hash function f1 (key) in case of collision.
- The search for an empty location is made successively at the addresses
    - f1(key) + f2(key),
    - f1(key) + 2 * f2(key),
    - f1 (key) + 3 * f2(key),...

Thank You