5.3 Hashing

- Hashing is a technique used for performing insertion, deletions, and finds in constant average time.
- The Hash table data structure is an array of some fixed size, containing the keys.
- A key is a value associated with each record.

5.3.1 Hash Functions

A Hashing function is a key – to – address transformation, which acts upon a given key to compute the relative position of the key in an array.

A simple Hash function

HASH (KEYVALUE) = KEYVALUE MOD TABLESIZE

Example:

Hash (92)

Hash (92) = 92 mod 10 =2

The key value 92 is placed in the relative location 2

Properties of good Hash function A good Hash Function should

– Minimize collisions

– Be easy and quick to compute

– Distribute keys evenly in the hash table

Some of the methods of Hashing Function

1. **Module Division**
2. **Mid – square Method**
3. **Folding Method**
4. **PSEUDO Random Method**

5. **Radix Transformation**

Applications of Hash tables

- Database systems
- Symbol tables
- Network processing algorithms
- Browse caches

COLLISIONS

▶ Collision occurs when a hash value of a record being inserted hashes to an address that already contain a different record. (ie) When two key values hash to the same position.

Example : 37, 24 , 7

Index	Slot
0	
1	
2	37
3	
4	24

- 37 is placed in index 2
- 24 is placed in index 4
- Now inserting 7
- Hash (7) = 7 mod 5 = 2
  - 2 collides

Collision Resolution strategies

The process of finding another position for the collide record is called Collision Resolution strategy.

Two categories

1. Open hashing - separate chaining

- Each bucket in the hash table is the head of a linked list.
- All elements that hash to same value are linked together.

2. Closed hashing - Open addressing, rehashing and extendible hashing.

- Collide elements are stored at another slot in the table.
- It ensures that all elements are stored directly into the hash table.

5.12 Separate Chaining

- ▶ Separate chaining is an open hashing technique.
- ▶ A pointer field is added to each record location.
- ▶ When an overflow occurs this pointer is set to point to overflow blocks making a linked list.
- ▶ In this method, the table can never overflow, since the linked list are only extended upon the arrival of new keys.

Insertion

- ▶ To perform the insertion of an element, traverse down the appropriate list to check whether the element is already in place.
- ▶ If the element turns to be a new one, it is inserted either at the front of the list or at the end of the list. If it is duplicate element, an extra field is kept.

Advantages and Disadvantages

Advantages

- ▶ More number of elements can be inserted as it uses array of linked lists.
- ▶ Collision resolution is simple and efficient.

Disadvantages

- ▶ It requires pointers, that occupies more space.
- ▶ It takes more effort to perform search, since it takes time to

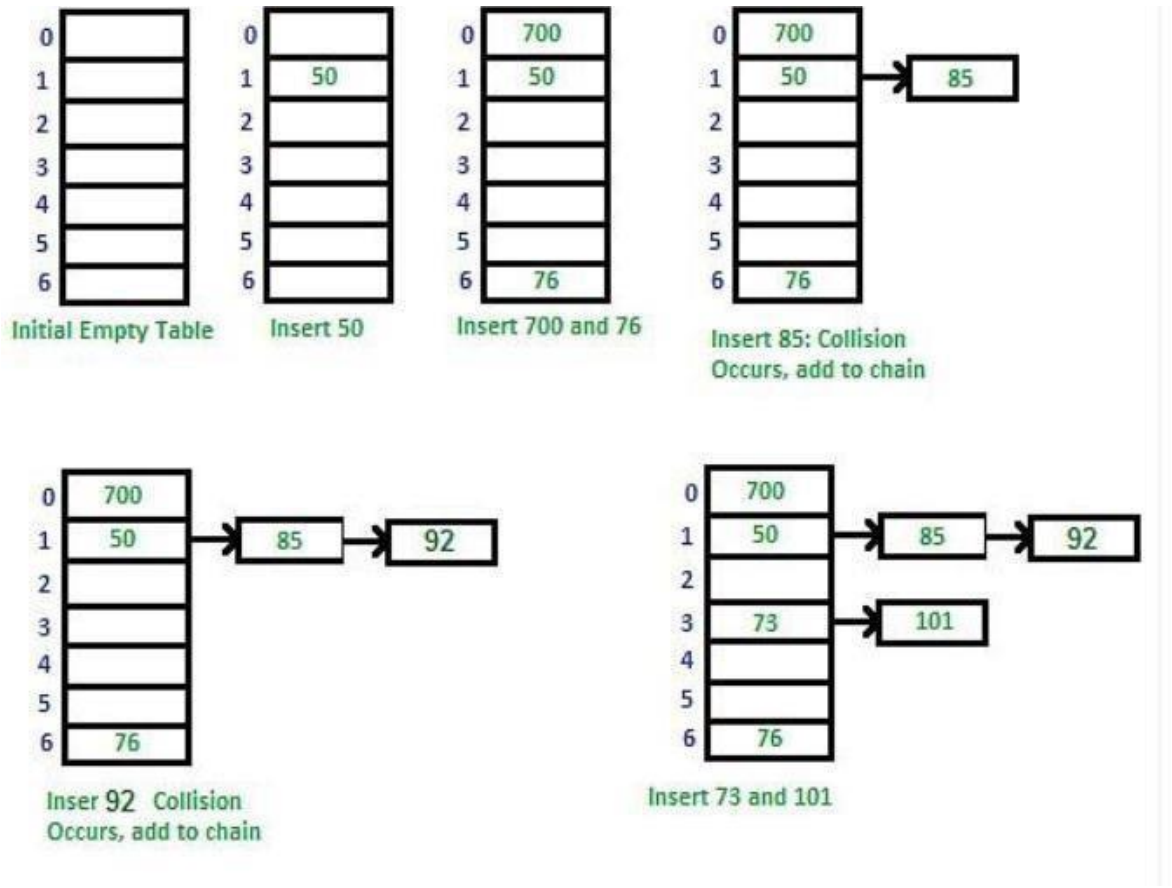
evaluate the hash function and also to traverse the list

Program

```
void insert(int value)
{
   //create a newnode with value
   struct node *newNode = malloc(sizeof(struct
   node)); newNode->data = value;
   newNode->next = NULL;
   //calculate hash key  int
   key = value % size;
   //check if chain[key] is empty
   if(chain[key] == NULL)
      chain[key] = newNode;
   //collision
   else
   {
      //add the node at the end of chain[key].
      struct node *temp = chain[key];
      while(temp->next)
      {
         temp = temp->next;
      }
      temp->next = newNode;
   }
}
```

Example:

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



5.13 Open Addressing

- Open addressing is a closed hashing technique.
- All elements are stored directly into the hash table
- In this method, if collision occurs, alternative cells are tried until an empty cell is found.
- There are three common methods
  1. Linear probing
  2. Quadratic probing
  3. Double hashing

LINEAR PROBING:

- In linear probing, we linearly probe for next slot.
- Let 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 Algorithm for linear probing:

1. Apply hash function on the key value and get the address of the location.
2. If the location is free, then

   i) Store the key value at this location, else

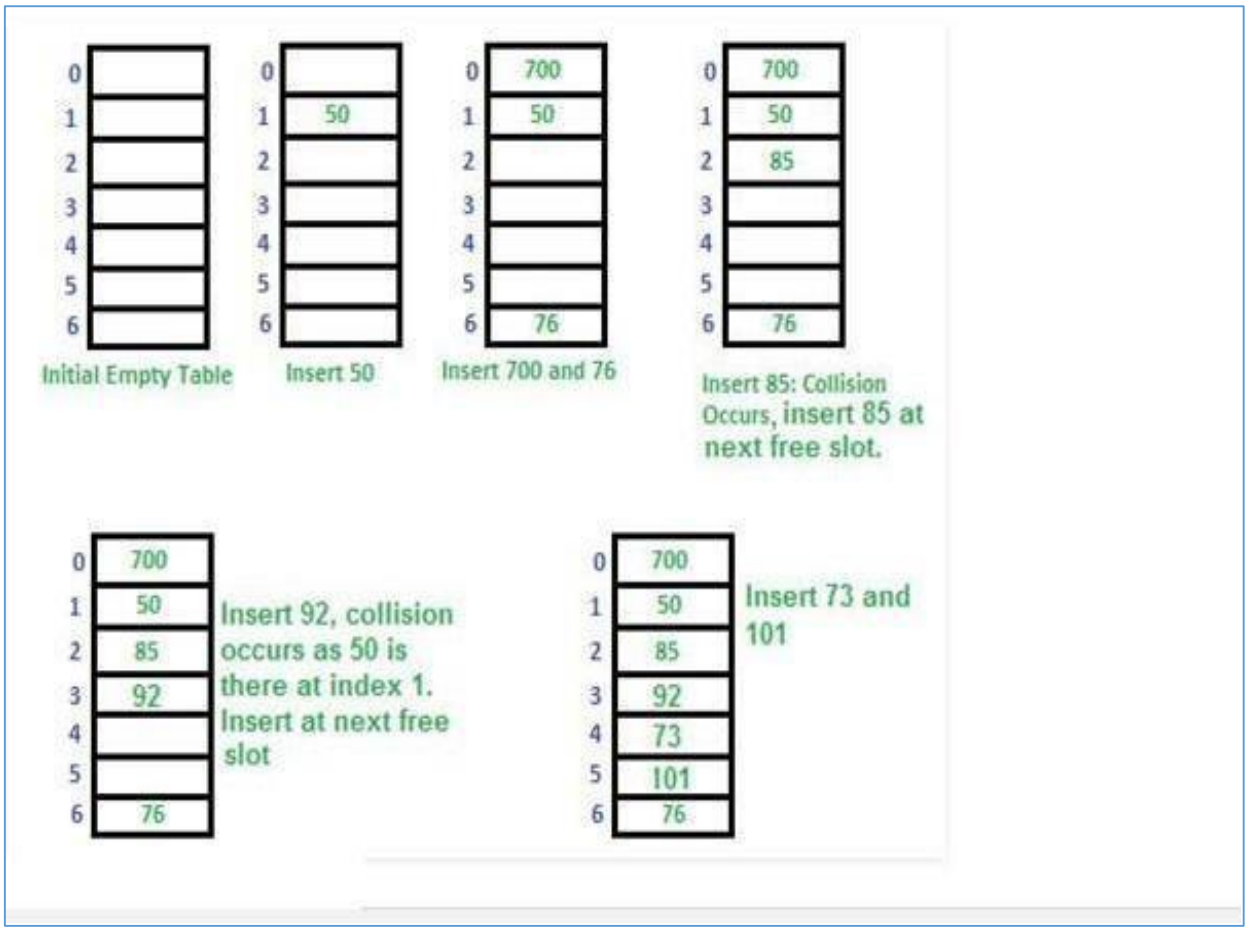
   ii) Check the remaining locations of the table one after the other till an empty location is reached. Wrap around on the table can be used. When we reach the end of the table, start looking again from the beginning.

   iii) Store the key in this empty location.

3. End

Example

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



Program

```
void insert(intkey,int data)

{

struct DataItem *item = (structDataItem*)
malloc(sizeof(structDataItem)); item->data = data;

item->key = key;

//get the hash

int hashIndex = hashCode(key);

//move in array until an empty or deleted cell

while(hashArray[hashIndex] != NULL &&hashArray[hashIndex]->key != -1)

{
```

```
//go to next cell

++hashIndex;

//wrap around the table hashIndex %= SIZE;

}

hashArray[hashIndex] = item;

}
```

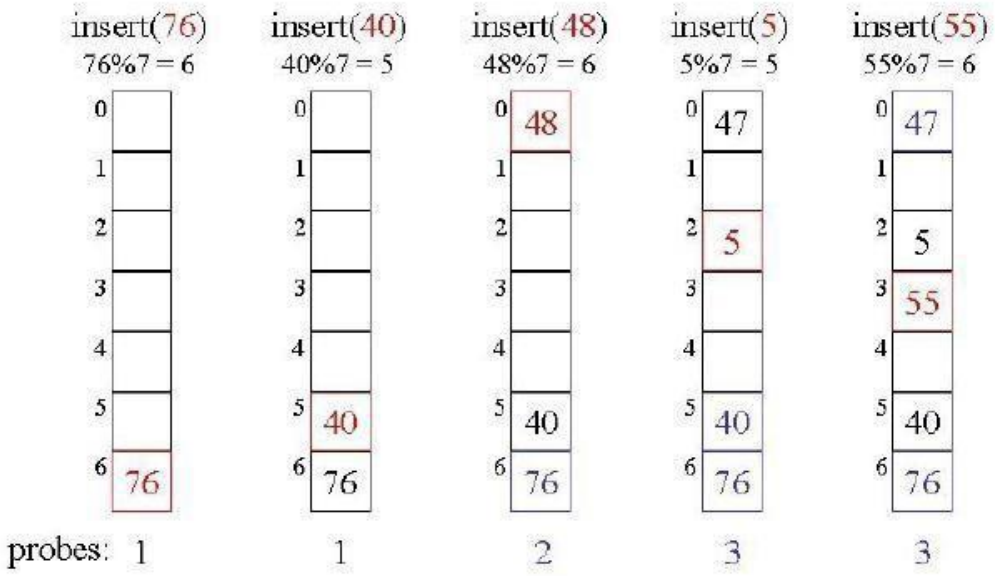
QUADRATIC PROBING

- Quadratic probing is similar to linear probing and the only difference is the interval between successive probes or entry slots.
- Here, when the slot at a hashed index for an entry record is already occupied, you must start traversing until you find an unoccupied slot.
- The interval between slots is computed by adding the successive value of an arbitrary polynomial in the original hashed index.

Let us assume that the hashed index for an entry is index and at index there is an occupied slot. The probe sequence will be as follows:

index = index % hashTableSize
index = (index + 1) % hashTableSize
index = (index + 4) % hashTableSize
index = (index + 9) % hashTableSize

Quadratic Probing Example



Double Hashing

- Double hashing is a collision resolving technique in Open Addressed Hash tables. Double hashing uses the idea of applying a second hash function to key when a collision occurs.

Double hashing can be done using :

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

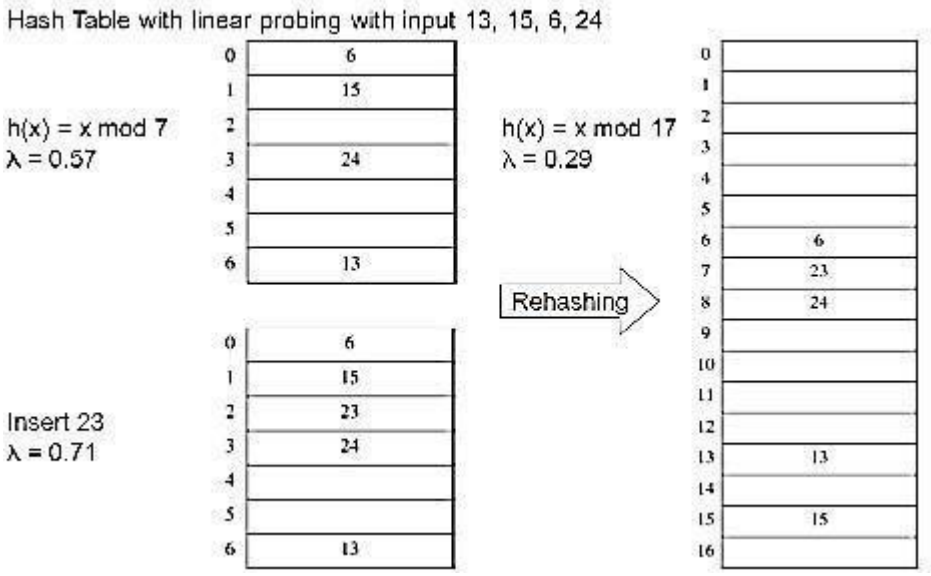
Here hash1() and hash2() are hash functions and TABLE_SIZE is size of hash table. (We repeat by increasing i when collision occurs)

5.14 Rehashing

- If the table is close to full, the search time grows and may become equal to the table size.
- When the load factor exceeds a certain value (e.g. greater than 0.5) we do rehashing :
- Build a second table twice as large as the original and rehash there

all the keys of the original table.

- Rehashing is expensive operation, with running time O(N)
- However, once done, the new hash table will have good performance.



5.15 Extendible Hashing

- Used when the amount of data is too large to fit in main memory and external storage is used.
- N records in total to store, M records in one disk block
- The problem: in ordinary hashing several disk blocks may be examined to find an element - a time consuming process.

Extendible hashing: no more than two blocks are examined. Idea:

- Keys are grouped according to the first m bits in their code. Each group is stored in one disk block.
- If the block becomes full and no more records can be inserted, each group is split into two, and m+1 bits are considered to determine the location of a record.

Extendible Hashing Example

• Suppose that g=2 and bucket size = 4.

• Suppose that we have records with these keys and hash function h(key) = key mod 64:

key	h(key) = key mod 64	bit pattern
288	32	100000
8	8	001000
1064	40	101000
120	56	111000
148	20	010100
204	12	001100
641	1	000001
700	60	111100
258	2	000010
1586	50	110010
44	44	101010