Unit 5: Part 2

Hash Tables: Hashing and Collision

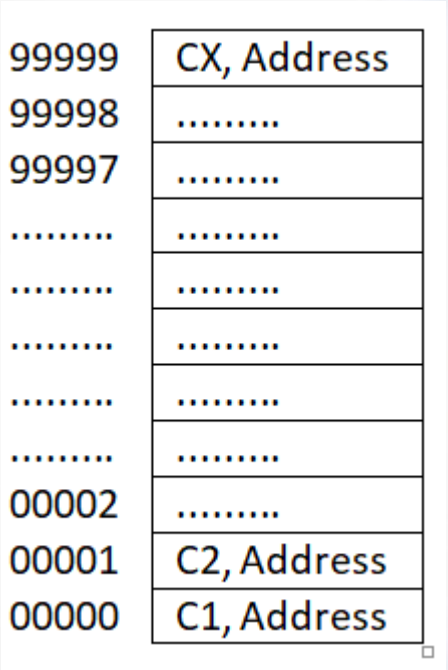
Introduction

- **Computational complexity of linear search: O(n)**
- **Computational complexity of binary search: O(log n)**
- Is there any way in which searching can be done in **constant time** , irrespective of the number of elements in the array? E.g. O(1) or O(k) where **k** is small value not dependent on **n**.

Introduction

- Consider Telephone directory example, using 5 digit phone numbers.
- Total possible phone numbers(customers): 10^5 = 1,00,000 (Numbers: 00000-99999)
- Want to access customer details in time O(1)
- **Potential solution:**
- Maintain an array of 1,00,000 elements
- Phone number can be used as array index to access in time complexity of O(1).

```
struct customer{
char cname[20];
char addr[50];
}
struct customer custDB[100000];
```



custDB

Introduction

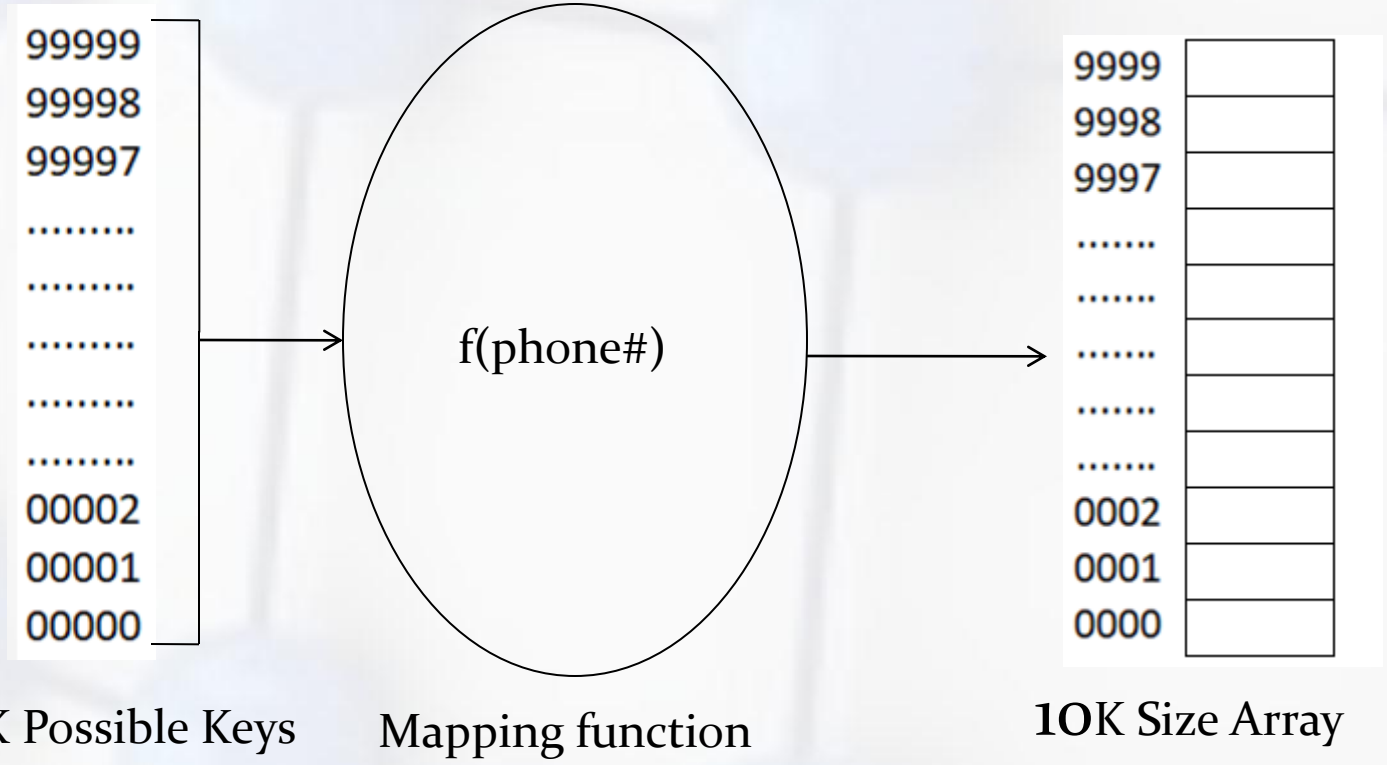
Problem with the proposed solution:

- Assume actual telephone users are only: 10,000
- Is it worth maintaining an array of 100K elements to store only 10K customer details, as done in direct mapping of key(phone number) to array index?

A more practical solution:

- If there is a **function f()** that can map each possible 5 digit phone number to a 4 digit index values, array of 10,000 only needed and search time O(1) can be obtained.
- y = f(key) key ➔ A five-digit phone number

  y ➔ A four-digit number that can be used as array index of a 10K size array.



1OOK Possible Keys

Mapping function

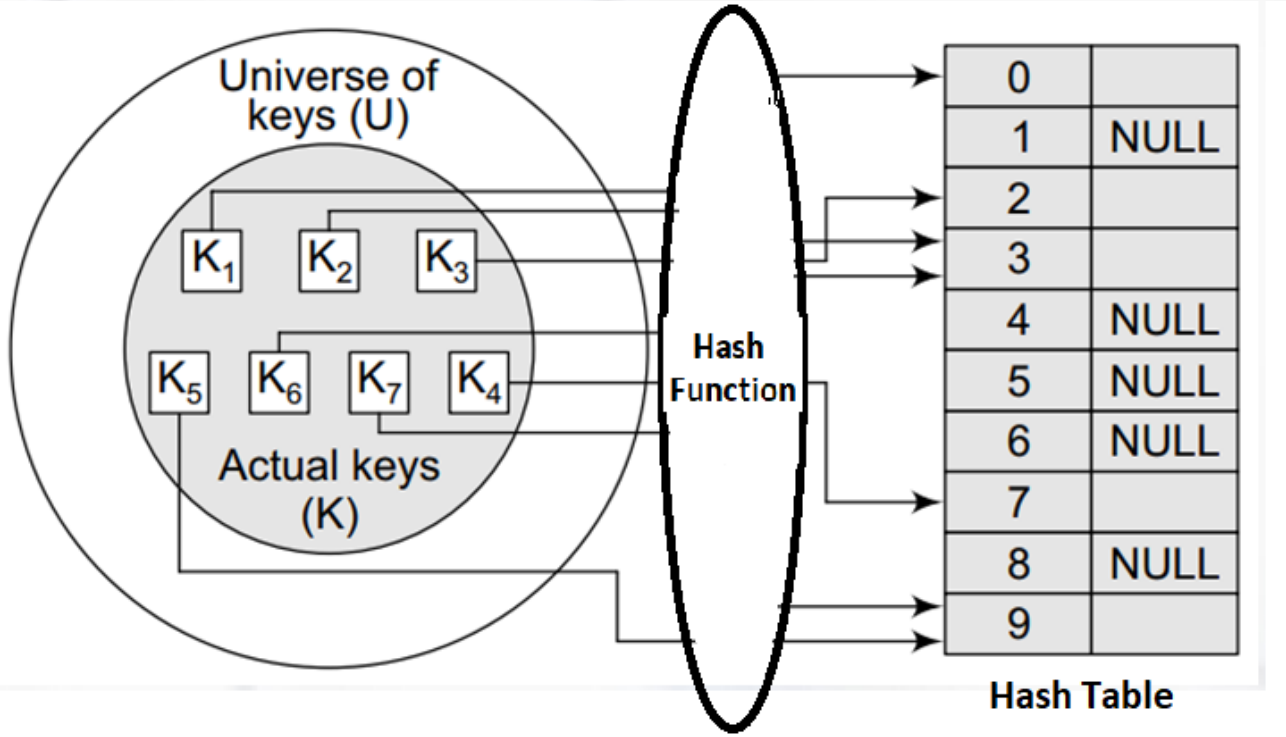
1OK Size Array

Hash Table

- **Hash Table** is a data structure in which **keys** are mapped to **array positions** by a **hash function.**
- A value/record stored in the Hash Table can be searched in O(1) time using a **hash function** to generate an address(array index) from the key.

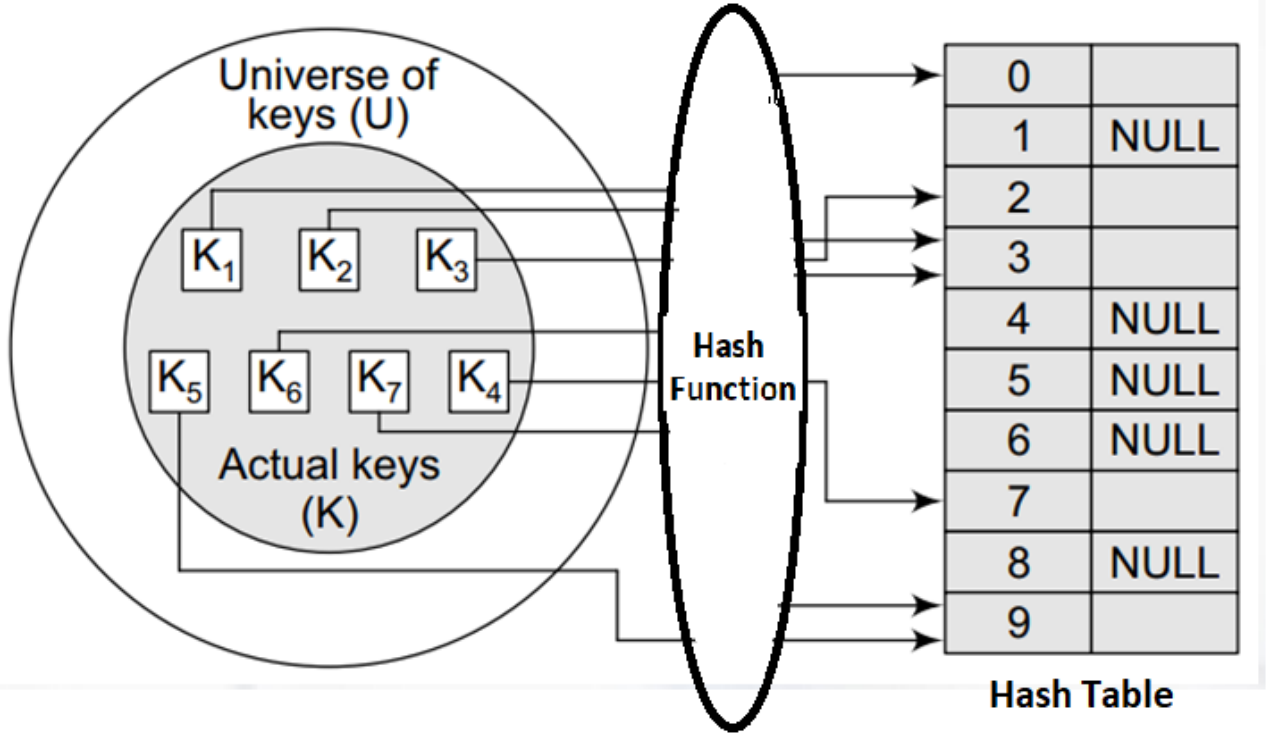
Universe of keys = Set of all possible key values

K_n = Actual keys used (a subset of Universe of keys)

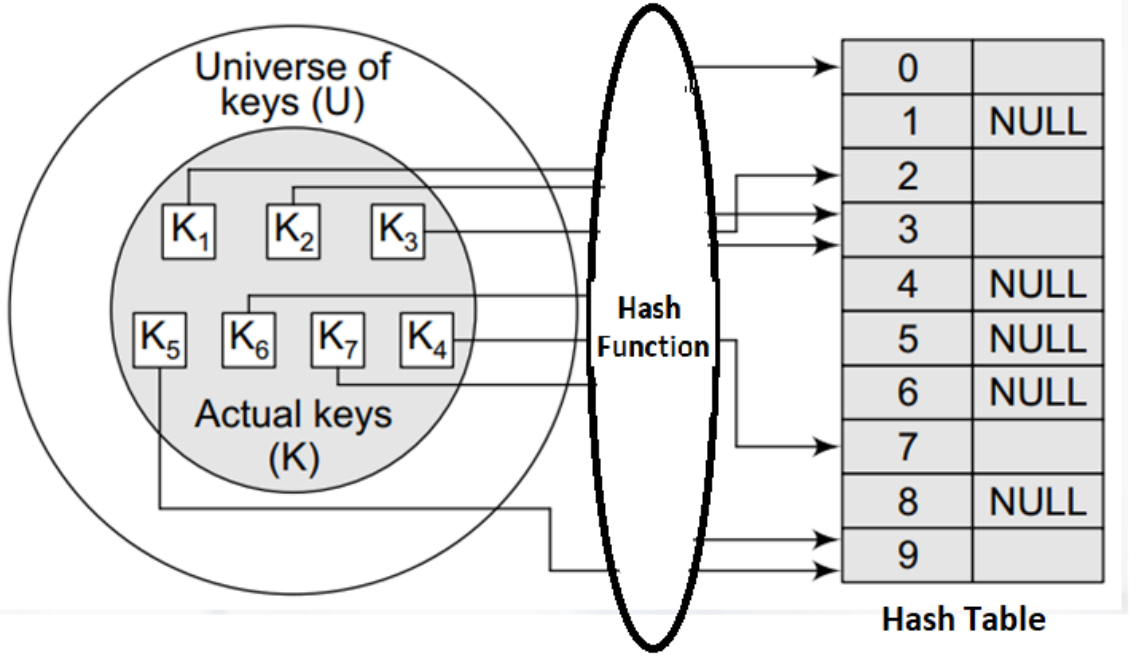


Hashing

- The process of mapping **keys** to appropriate locations (or indexes) in a **hash table** is called ***hashing*.**



Collision



When **two or more keys** maps to the same memory location, **a collision** is said to occur.

Collision resolution techniques are used to resolve collision scenarios.

Hash Function

- **Hash Function, h** is simply a mathematical formula/algorithm which when applied to the key, produces an integer which can be used as an **index** for the key in the hash table.
- The main aim of a hash function is that **elements should be relatively randomly and uniformly distributed in the array.**
- Hash function produces a unique set of integers within some suitable range. Such function produces no collisions in **ideal case**. Reality is different.
- In practice, there is no hash function that eliminates collisions completely. ***A good hash function can only minimize the number of collisions*** by spreading the elements uniformly across the hash table.

Properties of a good hash function

- ***Low cost*** *– Computation time for computing hash value should be low.*
- ***Determinism*** - A hash procedure must be deterministic. This means that the **same hash value** must be **generated every time** for a given **key**.
- ***Uniformity*** - A good hash function must map the keys as evenly distributed as possible over its output range. This means that the probability of generating every hash value in the output range should roughly be the same.

Hash Functions

Some common hashing Techniques

1. **Division Method**
2. **Multiplication Method**
3. **Mid square Method**
4. **Folding Method**

Hash Functions-Division Method

1. Division Method

$$h(x) = x \bmod M$$

•Division method is the most simple method of hashing an integer key ***x***. The method divides ***x*** by *M* and then use the remainder thus obtained.

• The division method is fast since it requires just one operation to find the hash value.

• However, care should be taken to select a suitable value for *M*.

Hash Functions-Division Method

Selecting a good value for M

- Generally, a prime number for M increases the likelihood that the keys are mapped with uniformly .
- M should not be too close to or exact powers of 2.

M=8, a power of 2 ➔Collision
13%8 = 5 (1101 %1000 = 101)
21%8 = 5(10101%1000 = 101)

M= 5, A prime ➔ No collision
13%5 = 3(1101 %101 = 11)
21%5 = 1(10101%101 = 01)

Hash Functions-Division Method

- Example: Calculate hash values of keys 1234 and 5462.

  Setting m = 97, hash values can be calculated as

  h(1234) = 1234 % 97 = 70

  h(5642) = 5642 % 97 = 16

Hash Functions-Multiplication Method

2. Multiplication Method

The steps involved in the multiplication method can be given as below:

Step 1: Choose a constant *A* such that *0 < A < 1.*

Step 2: Multiply the key *k* by *A*

Step 3: Extract the fractional part of *k*A* (using K*A % 1)

Step 4: Multiply the result of Step 3 by *m* and take the floor.

Hence, the hash function can be given as,

$$h(k) = \lfloor m\ (kA \bmod 1) \rfloor$$

Assume *kA* mod 1 ➔ gives the fractional part of *kA* and

m ➔ the total number of indices in the hash table (Size of hash table)

Hash Functions-Multiplication Method

Example: Given a hash table of size m = 1000, map the key 12345 to an appropriate location in the hash table

We will use A = 0.618033, m = 1000 and k = 12345

h(12345) = $\lfloor 1000 (12345 \times 0.618033 \bmod 1) \rfloor$

= $\lfloor 1000 (7629.617385 \bmod 1) \rfloor$

= $\lfloor 1000 (0.617385) \rfloor$

= 617.385

= 617

Hash Functions-Mid Square Method

3. Mid Square Method

Mid square method is a good hash function which works in two steps.

Step 1: Square the key value, i.e., find k^2

Step 2: Extract the middle *r* digits of the result obtained in Step 1.

h (k) = s

where, ***s*** is obtained by selecting r middle digits from k^2

Hash Functions-Mid Square Method

Example: Calculate the hash value for keys 1234 and 5642 using the mid square method. The hash table has 100 memory locations.

Note the hash table has 100 memory locations whose indices vary from 0-99. this means, only two digits are needed to map the key to a location in the hash table, so ***r = 2***.
** What is basis for deciding r?*

When k = 1234, k^2 = 1522756, h (k) = 27

When k = 5642, k^2 = 31832164, h (k) = 32

Hash Functions-Folding Method

4. Folding Method

The folding method works in two steps.

Step 1: Divide the key value into a number of parts.

That is divide key k into parts, *k1, k2, …, kn,* where each part has the same number of digits except the last part which may have lesser digits than the other parts.

Step 2: Add the individual parts.

That is obtain the sum of *k1 + k2 + .. + kn*. Hash value is produced by ignoring the last carry, if any.

Note: Number of digits in each part of the key will depend on size of hash table. If size is 100 (i.e., index values 0-99, each part/fold will be of size 2)

Hash Functions-Folding Method

Example: Given a hash table of 100 locations, calculate the hash value using folding method for keys- 5678, 321 and 34567.

Here, since there are 100 memory locations to address, we will break the key into parts where each part (except the last) will contain two digits.

Therefore,

Key	**5678**	**321**	**34567**
Parts	**56 and 78**	**32 and 1**	**34, 56 and 7**
Sum	**134**	**33**	**97**
Hash Value	**34 (ignore the last carry)**	**33**	**97**

Collisions

Collision occurs when the hash function maps two different keys to same location in hash table.

The two most popular method of resolving collision are:

1. Open addressing (also called **Closed hashing**)
2. Chaining (also called **Open Hashing**)

Collision Resolution by Open Addressing

- Once a collision takes place, open addressing computes new positions using a **probe sequence** and the next record is stored in that position.
- The process of examining memory locations in the hash table is called probing.

Open addressing technique can be implemented using:

- Linear probing
- Quadratic probing
- Double hashing.

Collisions

Collision Resolution by Open Addressing

- Once a collision takes place, open addressing computes new positions using a **probe sequence** and the next record is stored in that position.
- The process of examining memory locations in the hash table is called probing.

Open addressing can be implemented using following techniques:

- Linear probing
- Quadratic probing
- Double hashing.

Linear Probing

- The simplest approach to resolve a collision is linear probing.
- In this technique, if a value is already stored at location generated by *h(k),*

then the following hash function is used to resolve the collision.

$$h(k, i) = [h'(k) + i] \bmod m$$

where, *m* is the size of the hash table, ***h'(k) = k mod m*** and ***i*** is the probe number and varies from *0* to *m-1*.

Linear Probing

Example: Consider a hash table with size = 10. Using linear probing insert the keys 72, 27, 36, 24, 63, 81 and 92 into the table.

Let h'(k) = k mod m, m = 10

Initially the hash table can be given as,

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

Step1: Key = 72, i=0

h(72, 0) = (72 mod 10 + 0) mod 10

= 2 mod 10

= 2

Since, T[2] is vacant, insert key 72 at this location

0	1	2	3	4	5	6	7	8	9
-1	-1	72	-1	-1	-1	-1	-1	-1	-1

Linear Probing

Step2: Key = 81, i=0

h(81, 0) = (81 mod 10 + 0) mod 10

= 1 mod 10

= 1

Since, T[1] is vacant, insert key 81 at this location

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

Step 3 : Key = 71 , i= 0

h(71, 0) = (71 mod 10 + 0)mod 10

= 1 mod 10

= 1

Since, T[1] is not vacant, Check the next location using: i=1

h(71, 1) = (71 mod 10 + 1) mod 10

= (1 +1) mod 10

= 2

Since, T[2] is not vacant, Check the next location using i = 2

Linear Probing

Since, T[2] is not vacant, Check the next location using i = 2

h(71, 2) = (71 mod 10 + 2) mod 10

= (1+2) mod 10

= 3

Since, T[3] is vacant, Put 71 at T[3]

0	1	2	3	4	5	6	7	8	9
-1	81	72	71	-1	-1	-1	-1	-1	-1

Searching a value using Linear Probing

- Similar to insertion, prob the adjacent locations using the index generated by Hash function.
- The search function terminates because the table is full and the value is not present.
- In worst case, the search operation may have to make n comparison for hash table of size n, and the running time of the search algorithm may take time given as O(n).

Quadratic Probing

In this technique, if a value is already stored at location generated by h(k), then the following hash function is used to resolve the collision.

$$h(k, i) = [h'(k) + c1*i + c2*i^2] \bmod m$$

where, *m* is the size of the hash table, *h'(k) = k mod m* and *i* is the probe number that varies from *0* to *m-1* and *c1* and *c2* are constants such that *c1* and *c2* ≠ *0.*

Quadratic Probing

Example: Consider a hash table with size = 10. Using quadratic probing insert the keys 72, 62, 32..into the table. Take c1 = 1 and c2 = 3.

Let h'(k) = k mod m, m = 10

Initially the hash table can be given as,

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

We have,

h(k, i) = [h'(k) + c1*i + c2*i²] mod m

I. Insert Key = 72

Step1: i = 0

h(72,0) = [72 mod 10 + 1 X 0 + 3 X 0] mod 10

= [72 mod 10] mod 10

= 2 mod 10

= 2

Since, T[2] is vacant, insert the key 72 in T[2]. The hash table now becomes,

0	1	2	3	4	5	6	7	8	9
-1	-1	72	-1	-1	-1	-1	-1	-1	-1

Quadratic Probing

II: Insert Key = 62

Step 1: i=0

h(62,0) = [62 mod 10 + 1 X 0 + 3 X 0] mod 10

= [62 mod 10] mod 10

= 2 mod 10

= 2

Since, T[2] is occupied, the key 62 can not be stored in T[2]. Therefore, try again for next location. Thus probe, i = 1, this time.

Step 2: i=1

h(62,1) = [62 mod 10 + 1 X 1 + 3 X 1^2] mod 10

= [62 mod 10 + 1 + 3] mod 10

= [2+1+3] mod 10

= 6

Since, T[6] is vacant, insert the key 62 in T[6]. The hash table now becomes,

0	1	2	3	4	5	6	7	8	9
-1	-1	72	-1	-1	-1	62	-1	-1	-1

Quadratic probing leads to better distribution(spread) of records in the table compared to linear probing.

Double Hashing

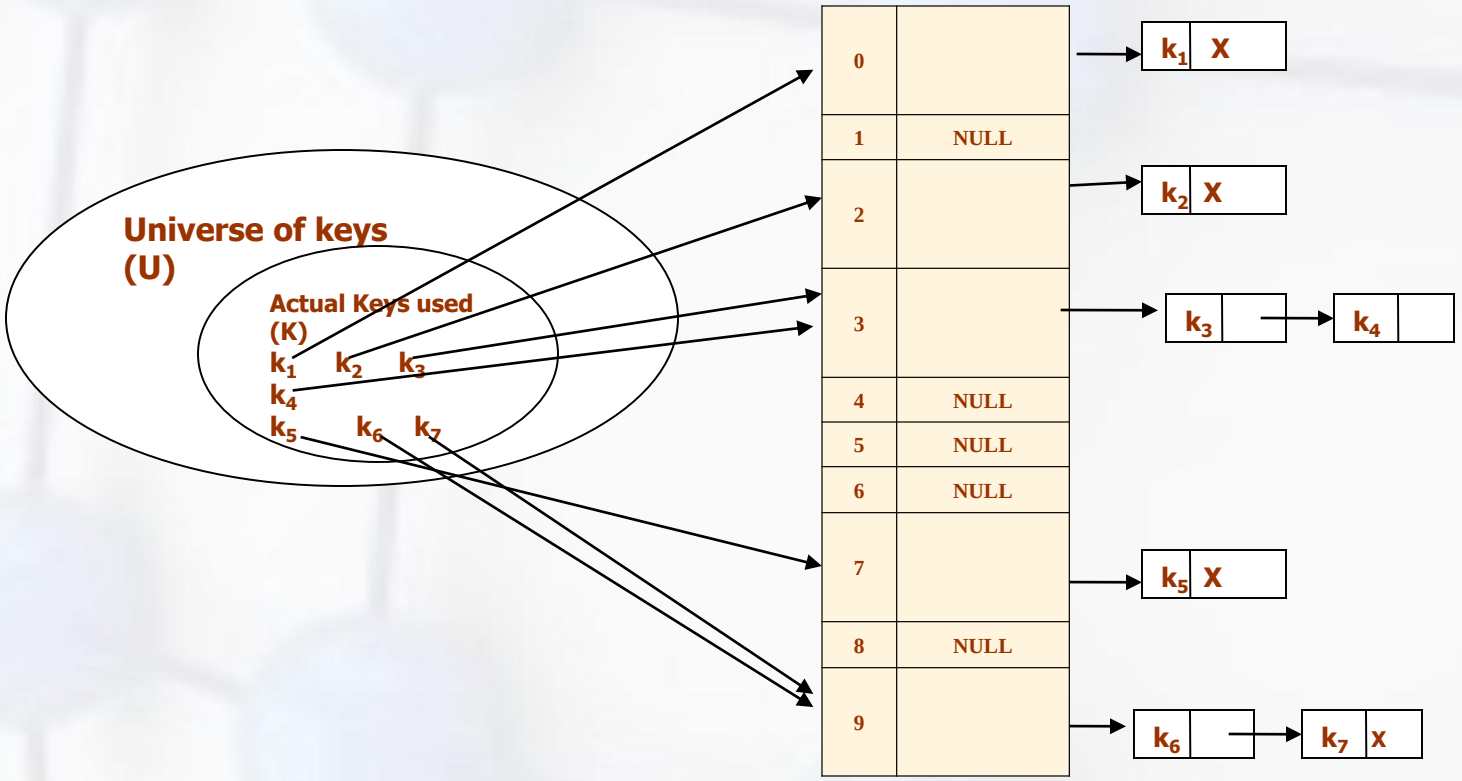
h(k, i) = [h1(k) + i*h2(k)] mod m

- Where, m is the size of the hash table, h1(k) and h2(k) are two hash functions e.g. , h1(k) = k mod m, h2(k) = k^2 mod m.

Collision Resolution by Chaining

- In chaining, each location in the hash table stores a pointer to a linked list that contains the all the key values that were hashed to the same location.
- That is, location i in the hash table points to the head of the linked list of all the key values that hashed to i.
- If no key value hashes to location i, then location i in the hash table contains NULL.
- **Chaining** is also known as **Open hashing** or **Separate chaining**

Collision Resolution by Chaining



Collision Resolution by Chaining

Example: Insert the keys 7, 26, 20, and 53 in a chained hash table of 10 memory locations(m=10). Use h(k) = k mod m

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

In this case, m=10. Initially, the hash table can be given as

Step 1: Key = 7 h(k) = 7%10 = 7

0	NULL
1	NULL
2	NULL
3	NULL
4	NULL
5	NULL
6	NULL
7	→ 7 | x
8	NULL
9	NULL

Step 2: Key = 26

h(k) = 26 mod 10

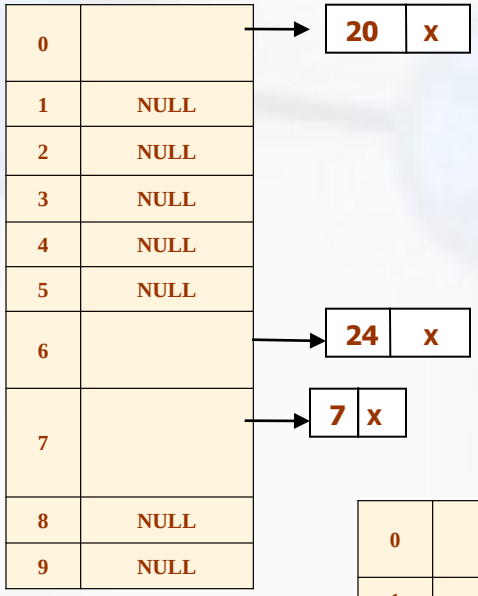
= 6

0	NULL
1	NULL
2	NULL
3	NULL
4	NULL
5	NULL
6	→ 26 | x
7	→ 7 | x
8	NULL
9	NULL

Collision Resolution by Chaining

Step 3: Key = 20

h(k) = 20 mod 10= 0



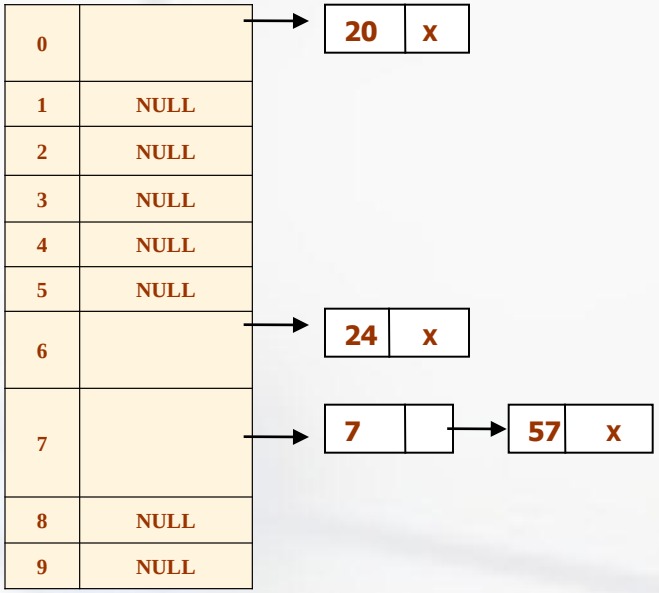
Step 4: Key = 57

h(k) = 57 mod 10

= 7

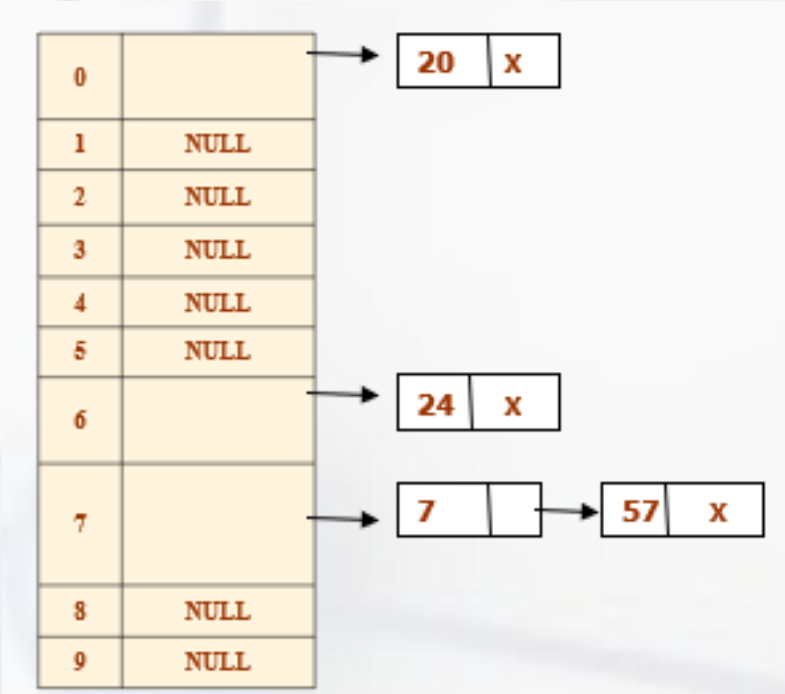
Insert 57 in the after 7

(Storing as an **ordered list** helps while searching for record)



Pros and Cons of Chained Hash Table

- The main advantage of using a chained hash table is that it remains effective even when the number of key values to be stored is much higher than the number of locations in the hash table.



- However, with the increase in number of keys to be stored, the performance of chained hash table does degrade gracefully (linearly).

Pros and Cons of Hashing

- One advantage of hashing is that no extra space is required to store the index (due to use of arrays) as in case of other data structures like linked lists.
- A properly designed hash table provides fast access to data.
- A primary drawback of using hashing technique for storing and retrieving data values is that it usually lacks locality of reference[3] in sequential retrieval by key.

- Choosing an effective hash function is more an art than a science. It is not uncommon (in open-addressed hash tables) to create a poor hash function.

Applications of Hashing

- Hash tables are widely used in situations where amounts of data need to be accessed to quickly. A few typical examples where hashing is used are given below.

- **Hashing is used for database indexing:** Some DBMSs store a separate file known as index file. When data is to be retrieved, the key information is first found in the appropriate index file which references the exact record location of the data in the database file. This index file is often stored as a hashed table.

- **Hash table is used as symbol tables in compilers:** Symbol tables are created by compilers during compilation process to store details of variables, functions, constants etc. Using hash table for this purpose improves the efficiency of compilers by giving faster access to the symbol table contents.

Applications of Hashing

- In Operating systems**, File and Directory hashing** is used for high performance file systems. Hashing makes looking up the file location much quicker than most other methods.
- Hashing is used in **internet search engines**.

References

1. What is hashing: https://www.geeksforgeeks.org/what-is-hashing/
2. https://en.wikipedia.org/wiki/Hash_function
3. Locality of reference: https://www.geeksforgeeks.org/locality-of-reference-and-cache-operation-in-cache-memory/