OXFORD
HIGHER EDUCATION

CHAPTER 15

HASHING AND COLLISION

INTRODUCTION

- In chapter 12, we have studied about two search algorithms- linear search and binary search. While linear search algorithm has running time proportional to O(n), binary search takes time proportional to O(log n), where n is the number of elements in the array.
- is there any way in which searching can be done in constant time, irrespective of the number of elements in the array?
- There are two solutions to this problem. To anaiyze the first solution let us take an example. In a small company of 100 employees, each employee is assigned an Emp_ID number in the range 0 – 99. To store the employee's records in an array, each employee's Emp_ID number act as an index in to the array where this employee's record will be stored as shown in figure

KEY	ARRAY OF EMPLOYEE'S RECORD
Key 0 [0]	Record of employee having Emp_ID 0
Key 1 [1]	Record of employee having Emp_ID 1
Key 2 [2]	Record of employee having Emp_ID 2
……	……
……	……
Key 98 [98]	Record of employee having Emp_ID 98
Key 99 [99]	Record of employee having Emp_ID 99

- In this case we can directly access the record of any employee, once we know his Emp_ID, because array index is same as that of Emp_ID number. But practically, this implementation is hardly feasible.
- Let us assume that the same company use a five digit Emp_ID number as the primary key. In this case, key values will range from 00000 to 99999. If we want to use the same technique as above, we will need an array of size 100,000, of which only 100 elements will be used. This is illustrated in figure

KEY		ARRAY OF EMPLOYEE'S RECORD
Key 00000	[0]	Record of employee having Emp_ID 00000
……		……
Key n	[n]	Record of employee having Emp_ID n
……		……
Key 99998	[99998]	Record of employee having Emp_ID 99998
Key 99999	[99999]	Record of employee having Emp_ID 99999

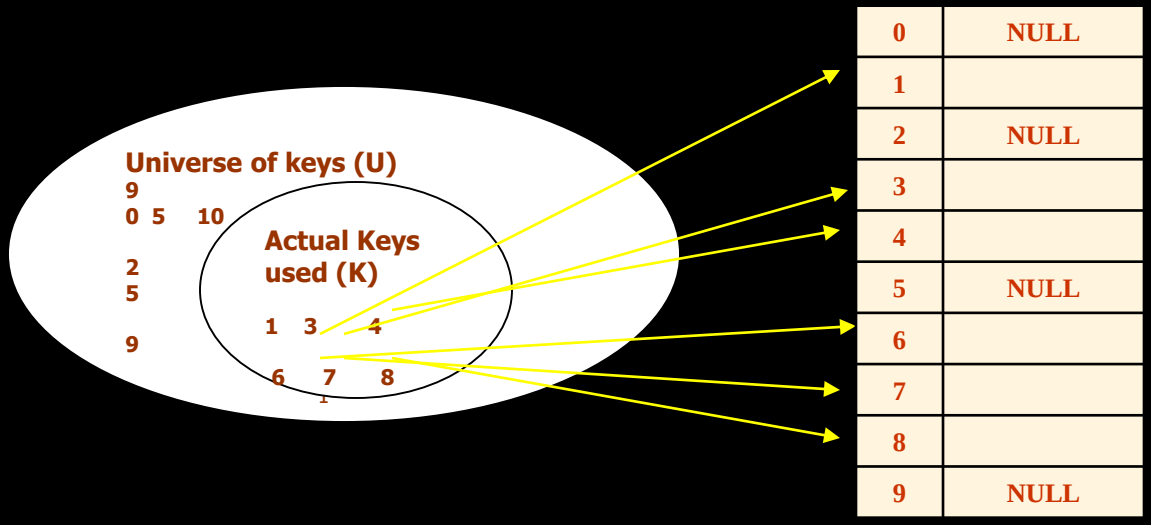
It is impractical it is to waste that much storage just to ensure that each employee' record is in a unique and predictable location.

Whether we use a two digit primary key (Emp_ID) or a five digit key, there are just 100 employees in the company. Thus, we will be using only 100 locations in the array. Therefore, in order to keep the array size down to the size that we will actually be using (100 elements), another good option is to use just the last two digits of key to identify each employee. For example, the employee with Emp_ID number 79439 will be stored in the element of the array with index 39. Similarly, employee with Emp_ID 12345 will have its record stored in the array at the 45th location.

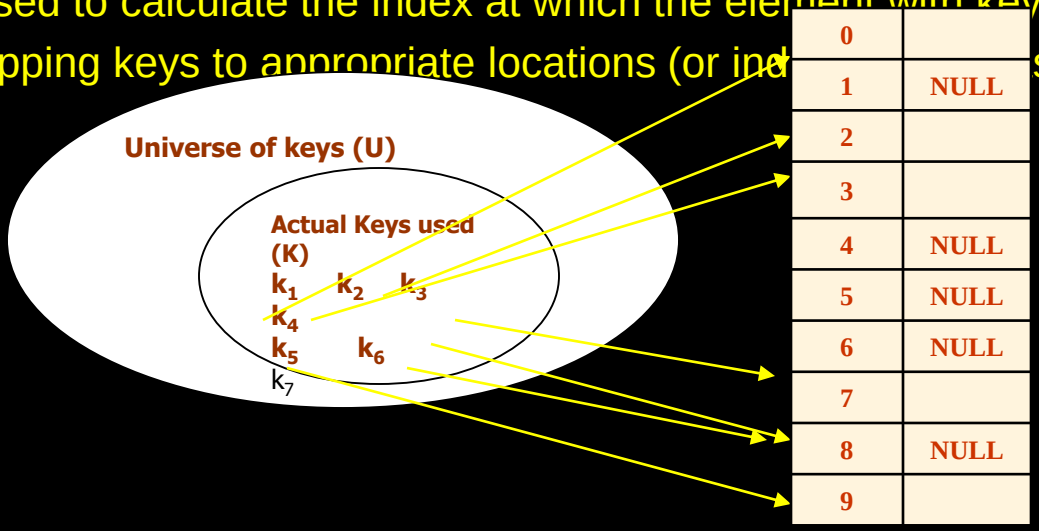
So, in the second solution we see that the elements are not stored according to the *value* of the **key**. So in this situation, we need a way to convert a five-digit key number to two-digit array index. We need some *function* that will do the transformation. In this case, we will use the term Hash Table for an array and the function that will carry out the transformation will be called a Hash Function.

HASH TABLE

- Hash Table is a data structure in which keys are mapped to array positions by a hash function.
- A value stored in the Hash Table can be searched in O(1) time using a hash function to generate an address from the key (by producing the index of the array where the value is stored).
- Look at the figure which shows a direct correspondence between the key and the index of the array. This concept is useful when the total universe of keys is small and when most of the keys are actually used from the whole set of keys. This is equivalent to our first example, where there are 100 keys for 100 employees.



- However, when the set K of keys that are actually used is much smaller than that of U, a hash table consumes much less storage space. The storage requirement for a hash table is just O(k), where k is the number of keys actually used.
- In a hash table, an element with key k is stored at index h(k) not k. This means, a hash function h is used to calculate the index at which the element with key k will be stored. Thus, the process of mapping keys to appropriate locations (or indices) in a hash table is called ***hashing***.



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

HASH FUNCTION

- Hash Function, h is simply a mathematical formula 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. Hash function produces a unique set of integers within some suitable range. Such function produces no collisions. But practically speaking, there is no hash function that eliminates collision completely. A good hash function can only minimize the number of collisions by spreading the elements uniformly throughout the array

Division Method

- Division method is the most simple method of hashing an integer *x*. The method divides *x* by *M* and then use the remainder thus obtained. In this case, the hash function can be given as

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

- The division method is quite good for just about any value of *M* and since it requires only a single division operation, the method works very fast. However, extra care should be taken to select a suitable value for *M*.
- For example, *M* is an even number, then *h*(*x*) is even if *x* is even; and *h*(*x*) is odd if *x* is odd. If all possible keys are equi-probable, then this is not a problem. But if even keys are more likely than odd keys, then the division method will not spread hashed values uniformly.
- Generally, it is best to choose *M* to be a prime number because making *M* a prime increases the likelihood that the keys are mapped with a uniformity in the output range of values. Then M should also be not too close to exact powers of 2. if we have,

  $h(k) = x \bmod 2k$

- then the function will simply extract the lowest *k* bits of the binary representation of *x*

- A potential drawback of the division method is that using this method, consecutive keys map to consecutive hash values. While on one hand this is good as it ensures that consecutive keys do not collide, but on the other hand it also means that consecutive array locations will be occupied. This may lead to degradation in performance.
- **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**

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 *kA*
- **Step 4:** Multiply the result of Step 3 by *m* and take the floor.
- Hence, the hash function can be given as,

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

- where, *kA* mod 1 gives the fractional part of *kA* and m is the total number of indices in the hash table
- The greatest advantage of the multiplication method is that it works practically with any value of *A*. Although the algorithm works better with some values than the others but the optimal choice depends on the characteristics of the data being hashed. Knuth has suggested that the best choice of A is
- **» (sqrt5 - 1) /2 = 0.6180339887**

- **Example: Given a hash table of size 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**

Mid Square Method

- Mid square method is a good hash function which works in two steps.
- **Step 1:** Square the value of the key. That is, find *k2*
- **Step 2:** Extract the middle *r* bits of the result obtained in Step 1.
- The algorithm works well because most or all bits of the key value contribute to the result. This is because all the digits in the original key value contribute to produce the middle two digits of the squared value. Therefore, the result is not dominated by the distribution of the bottom digit or the top digit of the original key value.
- In the mid square method, the same *r* bits must be chosen from all the keys. Therefore, the hash function can be given as,

$$h(k) = s$$

- where, *s* is obtained by selecting r bits from *k2*

- **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.***
- **When k = 1234, k2 = 1522756, h (k) = 27**
- **When k = 5642, k2 = 31832164, h (k) = 21**
- **Observe that 3rd and 4th digits starting from the right are chosen.**

Folding Method

The folding method works in two steps.

- **Step 1:** Divide the key value into a number of parts. That is divide 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 that the number of digits in each part of the key will vary depending upon the size of the hash table. For example, if the hash table has a size of 1000. Then it means there are 1000 locations in the hash table. To address these 1000 locations, we will need at least three digits, therefore, each part of the key must have three digits except the last part which may have lesser digits.

- **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 locat part (except the last) will contain two di**
- **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. Obviously, two records can not be stored in the same location. Therefore, a method used to solve the problem of collision also called collision resolution technique is applied. The two most popular method of resolving collision are:
- Collision resolution by open addressing
- Collision resolution by chaining

- **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. In this technique of collision resolution, all the values are stored in the hash table. The hash table will contain two types of values- either sentinel value (for example, -1) or a data value. The presence of sentinel value indicates that the location contains no data value at present but can be used to hold a value.
- The process of examining memory locations in the hash table is called probing. Open addressing technique can be implemented using- linear probing, quadratic probing and double hashing. We will discuss all these techniques in this section.

- ***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, $h'(k) = k \bmod m$ and *i* is the probe number and varies from *0* to *m-1*.

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,

Step1: Key = 72

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

= (2) mod 10

= 2

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

Step2: Key = 27

h(27, 0) = (27 mod 10 + 0) mod 10

= (7) mod 10

= 7

Since, T[7] is vacant, insert key 27 at this location

Step3: Key = 36

h(36, 0) = (36 mod 10 + 0) mod 10

= (6) mod 10

= 6

Since, T[6] is vacant, insert key 36 at this location

Step4: Key = 24

h(24, 0) = (24 mod 10 + 0) mod 10
= (4) mod 10
= 4

Since, T[4] is vacant, insert key 24 at this location

-1	-1	72	-1	24	-1	36	27	-1	-1

Step5: Key = 63

h(63, 0) = (63 mod 10 + 0) mod 10
= (3) mod 10
= 3

Since, T[3] is vacant, insert key 63 at this location

-1	-1	72	63	24	-1	36	27	-1	-1

Step6: Key = 81

h(81, 0) = (81 mod 10 + 0) mod 10
= (1) mod 10
= 1

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

-1	81	72	63	24	-1	36	27	-1	-1

Step7: Key = 92

h(92, 0) = (92 mod 10 + 0) mod 10
= (2) mod 10
= 2

Now, T[2] is occupied, so we cannot store the key 92 in T[2]. Therefore, try again for next location. Thus probe, i =
Key = 92

h(92, 1) = (92 mod 10 + 1) mod 10
= (2 + 1) mod 10
= 3

Now, T[3] is occupied, so we cannot store the key 92 in T[3]. Therefore, try again for next location. Thus probe, i = 2, this time.

Key = 92

h(92, 2) = (92 mod 10 + 2) mod 10
= (2 + 2) mod 10
= 4

Now, T[4] is occupied, so we cannot store the key 92 in T[4]. Therefore, try again for next location. Thus probe, i = 3, this time.

Key = 92

h(92, 3) = (92 mod 10 + 3) mod 10
= (2 + 3) mod 10
= 5

Since, T[5] is vacant, insert key 92 at this location

-1	-1	72	63	24	92	36	27	-1	-1

Searching a value using Linear Probing

- When searching a value in the hash table, the array index is re-computed and the key of the element stored at that location is checked with the value that has to be searched. If a match is found, then the search operation is successful. The search time in this case is given as O(1). Otherwise, if the key does not match, then the search function begins a sequential search of the array that continues until:
- the value is found

- the search function encounters a vacant location in the array, indicating that the value is not present
- 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-1) comparison, and the running time of the search algorithm may take time given as O(n). The worst case will be encountered when the table is full and after scanning all n-1 elements, the value is either present at the last location or not present in the table.

- Thus, we see that with increase in the number of collisions, the distance from the array index computed by the hash function and the actual location of the element increases, thereby increasing the search time.

Pros and Cons

- **Linear probing finds an empty location by doing a linear search in the array beginning from position *h(k).* Although, the algorithm provides good memory caching, through good locality of reference, but the drawback of this algorithm is that it results in clustering, and thus a higher risk that where there has been one collision there will be more. The performance of linear probing is sensitive to the distribution of input values.**

- **In linear probing as the hash table fills, clusters of consecutive cells are formed and the time required for a search increases with the size of the cluster. In addition to this, when a new value has to be inserted in to the table at a position which is already occupied, that value is inserted at the end of the cluster, which all the more increases the length of the cluster. Generally, an insertion is made between two clusters that are separated by one vacant location.**

- **But with linear probing there are more chances that subsequent insertions will also end up in one of the clusters, thereby potentially increasing the cluster length by an amount much greater than one. More the number of collisions, higher the probes that are required to find a free location and lesser is the performance. This phenomenon is called *primary clustering*. To avoid primary clustering, other techniques like quadratic probing and double hashing are used.**

- ***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.
- **$h(k, i) = [h'(k) + c1i + c2i2] \bmod m$**
- 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 eliminates the primary clustering phenomenon of linear probing because instead of doing a linear search, it does a quadratic search. For a given key *k,* first the location generated by *h'(k)* mod *m* is probed. If the location is free, the value is stored in it else, subsequent locations probed are offset by factors that depend in a quadratic manner on the probe number *i.* Although quadratic probing performs better than linear probing but to maximize the utilization of the hash table, the values of *c1, c2* and m needs to be constrained.

Example: Consider a hash table with size = 10. Using quadratic probing insert the keys 72, 27, 36, 24, 63, 81 and 101 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,

-1	-1	-1	-1	-1	-1	-1	-1	-1	-1

We have,

$h(k, i) = [h'(k) + c1i + c2i2] \bmod m$

Step1: Key = 72

$h(72) = [72 \bmod 10 + 1 \times 0 + 3 \times 0] \bmod 10$

$= [72 \bmod 10] \bmod 10$

$= 2 \bmod 10$

$= 2$

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

-1	-1	72	-1	-1	-1	-1	-1	-1	-1

Step2: Key = 27

h(27) = [27 mod 10 + 1 X 0 + 3 X 0] mod 10
= [27 mod 10] mod 10
= 7 mod 10
= 7

Since, T[7] is vacant, insert the key 27 in T[7]. The hash table now becomes,

0	1	2	3	4	5	6	7	8	9
-1	-1	72	-1	-1	-1	-1	27	-1	-1

Step3: Key = 36

h(36) = [36 mod 10 + 1 X 0 + 3 X 0] mod 10
= [36 mod 10] mod 10
= 6 mod 10
= 6

Since, T[6] is vacant, insert the key 36 in T[6]. The hash table now becomes,

-1	-1	72	-1	-1	-1	36	27	-1	-1

Step4: Key = 24

h(24) = [24 mod 10 + 1 X 0 + 3 X 0] mod 10
= [24 mod 10] mod 10
= 4 mod 10
= 4

Since, T[4] is vacant, insert the key 24 in T[4]. The hash table now becomes

-1	-1	72	-1	24	-1	36	27	-1	-1

Step5: Key = 63

h(63) = [63 mod 10 + 1 X 0 + 3 X 0] mod 10
= [63 mod 10] mod 10
= 3 mod 10
= 3

Since, T[3] is vacant, insert the key 63 in T[3]. The hash table now becomes,

-1	-1	72	63	24	-1	36	27	-1	-1

Step6: Key = 81

h(81) = [81 mod 10 + 1 X 0 + 3 X 0] mod 10
= [81 mod 10] mod 10
= 81 mod 10
= 1

Since, T[1] is vacant, insert the key 81 in T[1]. The hash table now becomes,

-1	81	72	63	24	-1	36	27	-1	-1

Step7: Key = 101

h(101) = [101 mod 10 + 1 X 0 + 3 X 0] mod 10
= [101 mod 10 + 0] mod 10
= 1 mod 10
= 1

Since, T[1] is already occupied, the key 101 can not be stored in T[1]. Therefore, try again for next location. Thus probe, i = 1, this time.

Key = 101

h(101) = [101 mod 10 + 1 X 1 + 3 X 1] mod 10
= [101 mod 10 + 1 + 3] mod 10
= [101 mod 10 + 4] mod 10
= [1 + 4] mod 10
= 5 mod 10
= 5

Since, T[5] is vacant, insert the key 101 in T[5]. The hash table now becomes,

-1	81	72	63	24	101	36	27	-1	-1

- ***Pros and Cons***

- Quadratic probing caters to the primary clustering problem that exists in linear probing technique. Quadratic probing provides good memory caching because it preserves some locality of reference. But linear probing does this task better and gives better cache performance.

- One of the major drawbacks with quadratic probing is that a sequence of successive probes may only explore a fraction of the table, and this fraction may be quite small. If this happens then we will not be able to find an empty location in the table despite the fact that the table is by no means full.

- Although quadratic probing is free from primary clustering, but it is still liable to what is known as *secondary clustering*. This means that if there is a collision between two keys then the same probe sequence will be followed for both. (Try to insert key 92 and you will see how this happens). With quadratic probing, potential for multiple collisions increases as the table becomes full. This situation is usually encountered when the hash table is more than full.
- Quadratic probing is widely applied in the Berkeley Fast File System to allocate free blocks.

- ***Searching a value using Quadratic Probing***
- While searching for a value using quadratic probing technique, the array index is re-computed and the key of the element stored at that location is checked with the value that has to be searched. If the desired key value matches the key value at that location, then the element is present in the hash table and the search is said to be successful.

- In this case the search time is given as *O(1).* However, if the value does not match then, the search function begins a sequential search of the array that continues until:
- the value is found

- the search function encounters a vacant location in the array, indicating that the value is not present

- the search function terminates because the table is full and the value is not present

Double Hashing

- To start with double hashing uses one hash value and then repeatedly steps forward an interval until an empty location is reached. The interval is decided using a second, independent hash function, hence the name double hashing.

- Therefore, in double hashing we use two hash functions rather a single function. The hash function in case of double hashing can be given as,

  h(k, i) = [h1(k) + ih2(k)] mod m

- where, m is the size of the hash table, h1(k) and h2(k) are two hash functions given as, h1(k) = k mod m, h2(k) = k mod m', i is the probe number that varies from 0 to m-1 and m' is chosen to be less than m. we can choose m' = m-1 or m-2.

- When we have to insert a key k in the hash table, we first probe the location given by applying h1(k) mod m because during the first probe, i = 0. If the location is vacant the key is inserted into it else subsequent probes generate locations that are at an offset of h2(k) mod m from the previous location. Since the offset may vary with every probe depending on the value generated by second hash function, the performance of double hashing is very close to the performance of the ideal scheme of uniform hashing.

.

Pros and Cons

- Double hashing minimizes repeated collisions and the effects of clustering. That is, double hashing is free from problems associated with primary clustering as well secondary clustering.

Example: Consider a hash table with size = 11. Using double hashing insert the keys 72, 27, 36, 24, 63, 81, 92 and 101 into the table. Take h1 = k mod 10 and h2 = k mod 8.

Let m = 11

Initially the hash table can be given as,

-1	-1	-1	-1	-1	-1	-1	-1	-1	-1

We have,

h(k, i) = [h1(k) + ih2(k)] mod m

Step1: Key = 72

h(72, 0) = [72 mod 10 + (0 X 72 mod 8] mod 11
= [2 + (0 X 0)] mod 10
= 2 mod 11
= 2

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

-1	-1	72	-1	-1	-1	-1	-1	-1	-1

Step2: Key = 27

h(27, 0) = [27 mod 10 + (0 X 27 mod 8)] mod 10
= [7 + (0 X 3)] mod 10
= 7 mod 10
= 7

Since, T[7] is vacant, insert the key 27 in T[7]. The hash table now becomes,

-1	-1	72	-1	-1	-1	-1	27	-1	-1

Step3: Key = 36

h(36, 0) = [36 mod 10 + (0 X 36 mod 8)] mod 10
= [6 + (0 X 4)] mod 10
= 6 mod 10
= 6

Since, T[6] is vacant, insert the key 36 in T[6]. The hash table now becomes,

-1	-1	72	-1	-1	-1	36	27	-1	-1

Step4: Key = 24

h(24, 0) = [24 mod 10 + (0 X 24 mod 8)] mod 10
= [4 + (0 X 0)] mod 10
= 4 mod 10
= 4

Since, T[4] is vacant, insert the key 24 in T[4]. The hash table now becomes,

-1	-1	72	-1	24	-1	36	27	-1	-1

Step5: Key = 63

h(63, 0) = [63 mod 10 + (0 X 63 mod 8)] mod 10
= [3 + (0 X 7)] mod 10
= 3 mod 10
= 3

Since, T[3] is vacant, insert the key 63 in T[3]. The hash table now becomes,

-1	-1	72	63	24	-1	36	27	-1	-1

Step6: Key = 81

h(81, 0) = [81 mod 10 + (0 X 81 mod 8)] mod 10
= [1 + (0 X 1)] mod 10
= 1 mod 10
= 1

Since, T[1] is vacant, insert the key 81 in T[1]. The hash table now becomes,

-1	81	72	63	24	-1	36	27	-1	-1

Step7: Key = 92

h(92, 0) = [92 mod 10 + (0 X 92 mod 8)] mod 10
= [2 + (0 X 4)] mod 10
= 2 mod 10
= 2

Now, T[2] is occupied, so we cannot store the key 92 in T[2]. Therefore, try again for next location.
Thus probe, i = 1, this time.
Key = 92

h(92, 1) = [92 mod 10 + (1 X 92 mod 8)] mod 10
= [2 + (1 X 4)] mod 10
= (2 + 4) mod 10
= 6 mod 10
= 6

Now, T[6] is occupied, so we cannot store the key 92 in T[6]. Therefore, try again for next location.
Thus probe, i = 2, this time.
Key = 92
Key = 92

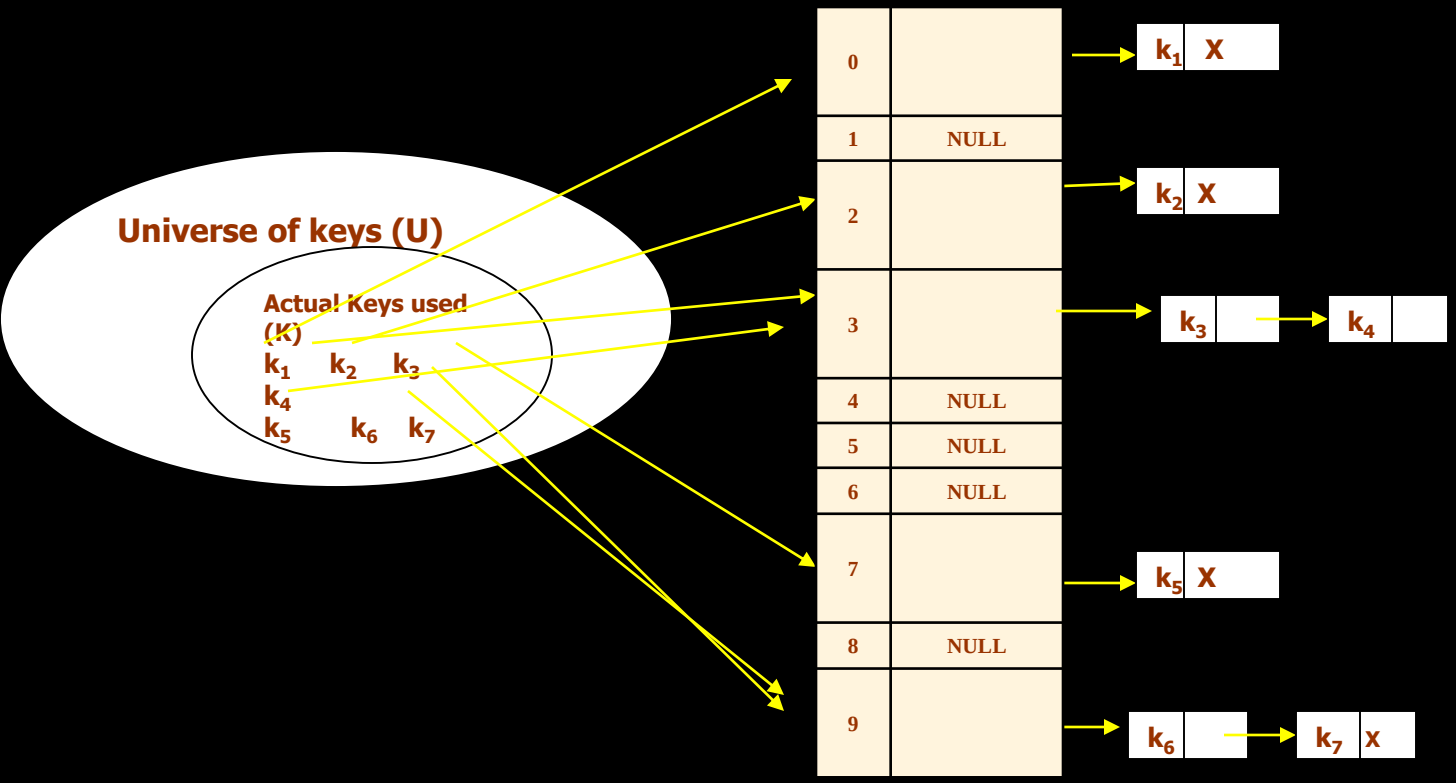
h(92) = [92 mod 10 + (2 X 92 mod 8)] mod 10
= [2 + (2 X 4)] mod 10
= [2 + 8] mod 10
= 10 mod 10
= 0

Since, T[1] is vacant, insert the key 81 in T[1]. The hash table now becomes,

90	81	72	63	24	-1	36	27	-1	-1

COLLISION RESOLUTION BY CHAINING

- In chaining, each location in the hash table stores a pointer to a linked list that contains the all key values that were hashed to the same location. That is, location l in the hash table points to the head of the linked list of all the key values that hashed to l. however, if no key value hashes to l, then location l in the hash table contains NULL. Figure shows how the key values are mapped to a location l in the hash table and stored in a linked list that corresponds to l.



Example: Insert the keys 7, 24, 18, and 52 in a chained hash table of 9 memory locations. Use h(k) = k mod m

In this case, m=9. Initially, the hash table can be given as

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

Step 1: Key = 7

h(k) = 7 mod 9

= 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 = 24

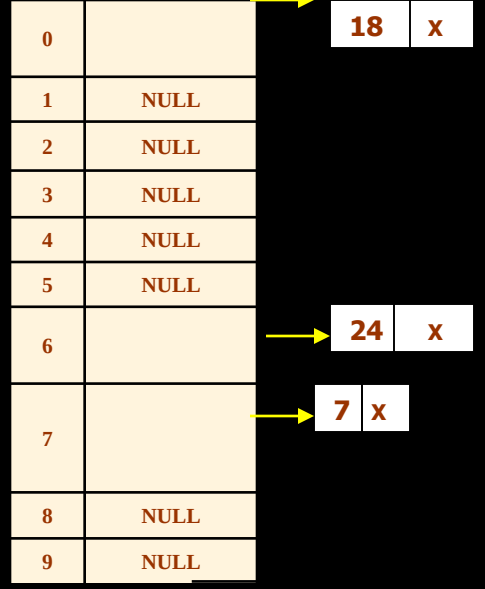
h(k) = 24 mod 9

= 6

0	NULL
1	NULL
2	NULL
3	NULL
4	NULL
5	NULL
6	→ 24 x
7	→ 7 x
8	NULL
9	NULL

Step 3: Key = 18

h(k) = 18 mod = 0

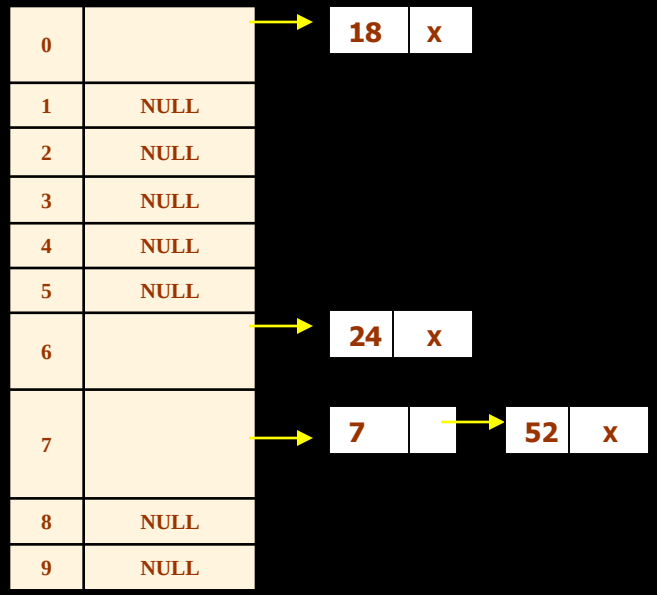


Step 4: Key = 52

h(k) = 52 mod 9

= 7

Insert 52 in the beginning of the linked list of location 7



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).

- For example, a chained hash table with 1000 memory locations and 10,000 stored keys will give 5 to 10 times less performance as compared to the performance of chained hash table having 10,000 locations. But the conclusion is that a chained hash table is still 1000 times faster than a simple hash table.

- The other advantage of using chaining for collision resolution is that unlike in quadratic probing, the performance does not degrades when the table is more than half full. This technique is absolutely free from clustering problems and thus provides an efficient mechanism to handle collisions.

- However, chained hash tables inherit the disadvantages of linked lists. First, to store even a key value, the space overhead of the next pointer in each entry can be significant. Second, traversing a linked list has poor cache performance, making the processor cache ineffective.

PROS AND CONS OF CHAINED HASH TABLE

PROS AND CONS OF HASHING

- One advantage of hashing is that no extra space is required to store the index as in case of other data structures. In addition, a hash table provides fast data access and an added advantage of rapid updates.

- On the other hand, the primary drawback of using hashing technique for inserting and retrieving data values is that it usually lacks locality and sequential retrieval by key. This makes insertion and retrieval of data values even more random.

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

APPLICATIONS OF HASHING

- Hash tables are widely used in situations where enormous amounts of data have to be accessed to quickly search and retrieve information. 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 indexes. When data has to be retrieved from a file, 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 key information in the index file is often stored as a hashed value.

- Hashing is used as symbol tables, for example, in Fortran language to store variable names. Hash tables speeds up execution of the program as the references to variables can be looked up quickly.

- Hashing is also widely being used for internet search engines.

APPLICATIONS OF HASHING

- .
- In many database systems, File and Directory hashing is used in high performance file systems. Such systems use two complementary techniques to improve the performance of file access. While one of theses techniques is caching which saves information in memory, the other is hashing which makes looking up the file location in memory much quicker than most other methods.

- Hash tables can be used to store massive amount of information for example, to store driver's license records. Given the driver's license number, hash tables help to quickly get information about the driver (i.e. name, address, age)

- Hashing technique is for compiler symbol tables in C++. The compiler uses a symbol table to keep a record of the user-defined symbols in a C++ program.

- Hashing facilitates the compiler to quickly look up variable names and other attributes associated with symbols
- Hashing is also widely being used for internet search engines.