Hashing and Collision

Introduction.

- Linear Search and Binary Search. Linear Search → has a running time proportional to O(n), while Binary Search takes time proportional to O(log n), where n is the number of element in the array.

- Both linear and Binary Search trees are efficient algorithms to search for an elements. But what if want to perform the search operation in time proportional to O(1)? In other words, is there a way to search an array in constant time, irrespective of its size?

  ↳ There are two solution to this problem →

  Let take an example → to explain the first solution :→

  In a company of 100 employees, each employee is assigned an Emp_ID in the range 0-99. To store the records in an array, each employee's EMP-ID act as an index into the array where the employee's record will be stored.

  In this case → we can directly access the record of any employee, once we know his Emp-ID, because the array index is the same as the EMP-ID

  But practically this is not feasible.

Let us assume that the same company uses a five digits Emp-ID as the primary key. In this case, key values will range from 00000 to 99999.

If we want to use same technique as above, we need an array of size 100,000 of which only 100 elements will be used.

Key	Array of employee record
Key 00000 → [0]	Employee record with Emp-ID

Key n → [n]	Employee record with Emp-ID n
----------	----------
----------	----------
Key 99998 → [99998]	Employee record with Emp-ID 99998
Key 99999 → [99999]	Employee record with Emp-ID 99999

Record of employees with a five-digit Emp-ID.

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

Whether we use 2-digit primary key (Emp-ID) or a five-digit key, there are just 100 employee in the company. Thus, we will be using only 100 location in the array. Therefore in order to keep the array size down to the size that we will actually be using (100) element, another good option is to use the last two digits of the key to identify each employee.

For ex → Emp-ID 79439 → array with index 39.
Emp-ID 12345 → array with index 45.

→ In the second solution → the elements are not stored according to the value of the key. → So in this case, we need a function which 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 Tables

→ Hash table is a data structure in which keys are mapped to array positions by a hash function.

→ We will use a hash function that extracts the last two digits of the key. ⟶ Therefore we map the keys to array locations or array indicies.

→ A value stored in a hash table can be searched in O(1) time by using function which generates an address from the key.

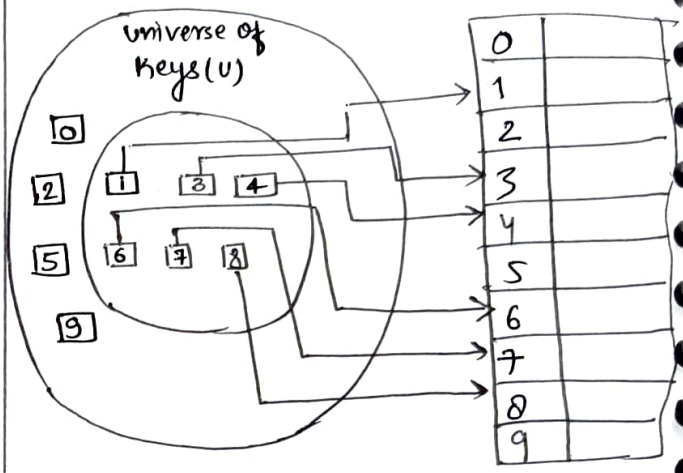
→ When the set K of keys that are actually used is smaller than the universe of keys (U), a hash table consume less storage space.

→ The storage requirement for a hash table is O(K), where K is the number of keys actually used.

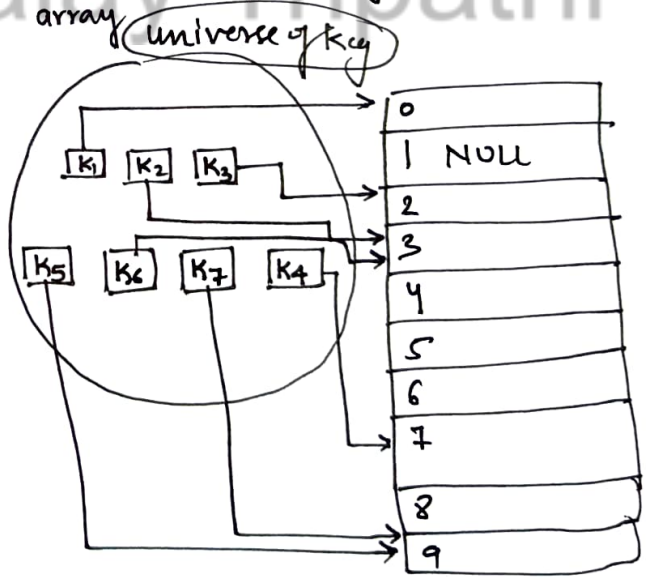
→ In a hash table, an element with key k is stored at index h(k) and not k ⟶ it means a hash function h is used to calculate the index at which the element with key k will be stored. This process of mapping the keys to approprate locations (or indices) in a hash table is called hashing.

→ In a hash table in which each key from the set K is mapped to locations generated by using a hash function. Note that keys, K point to the same memory location is known as Collision.

The main goal of hash function is to reduce the range of array indices that have to be handled. Thus, instead of having U values, we just need K values, thereby reducing the amount of storage space required.



Direct Relationship b/w key and Index in the array



Relationship b/w keys and hash table Index.

HASH FUNCTION.

→ A hash function → mathematical formula. → when applied to a key, produces an integer → which can be used as an index for the key in the hash table.

→ aim of Hash function → is that elements should be relatively, randomly and uniformly distributed.

→ it produces a unique ~~set~~ of integers within some suitable range ~~of~~ in order to reduce the number of collision.

→ NO HASH FUNCTION → CAN ELIMINATE COLLISION
A GOOD HASH FUNCTION → CAN ONLY MINIMIZE THE NUMBER OF COLLISIONS BY SPREADING THE ELEMENTS UNIFORMLY THROUGHOUT THE ARRAY.

PROPERTIES OF GOOD HASH FUNCTION

- Low Cost → Ex → if Binary Search Algo. can search an element from a sorted table of n items with $\log_2 n$ key comparison → then the hash function must cost less than performing $\log_2 n$ key comparison.
- Determinism
  - it means same value must be generated for a given input value. However this criteria excludes hash functions that depend on external variable parameters (such as the time of day) and on the memory address of the object being hashed.
- Uniformity
  - A good function must map the key as evenly as possible over its o/p range.
  - This means that the probability of generating every hash value in the o/p range should roughly be the same.
  - Property of uniformity also minimizes the no. of collisions.

DIFFERENT HASH FUNCTION

We discuss the hash functions which uses numeric keys. However, there can be cases in real-world applications where we can have alphanumeric keys rather than simple numeric keys.

Division Method.

- It is the most simple method of hashing an integer x.
- This method divides x by M and then uses the remainder obtained.
- The hash function can be given as

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

- method works very fast as it requires only a single division operation. However, extra care should be taken to select a suitable value for M.

- for ex → suppose M is 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, they this is not a problem. But if even keys are more likely than odd keys, then the division method will not spread the hashed values uniformly.

→ Generally → it is best to choose M to be prime number because making M a prime number increases the likelihood that the keys are mapped with uniformity in the output range of values.

M → should also be not too close to the exact powers of 2.

If we have $h(x) = x \bmod 2^k$

then the function will simply extract the lowest k bits of the Binary representation of x.

→ Potential Drawback of the division method →

Using this method, consecutive keys map to consecutive hash values.

On one hand, this is good as it ensures that consecutive keys do not collide, but on the other, it also means that consecutive array locations will be occupied.

This may lead to performance degradation.

Ques Calculate the hash values of keys 1234 and 5462

Solution M=97, hash value can be calculated as

$h(1234) = 1234 \% 97 = 70$

$h(5642) = 5642 \% 97 = 16.$

Multiplication Method.

The steps involved in the multiplication method are as follows:—

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 the size of hash table (m).

Hence, the hash function can be given as :→

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

where $kA \bmod 1$ gives the fractional part of kA and m is the total number of indicies in the hash table.

Greatest advantage of this method is that it works practically with any value of A ⟶ Its optimal value or choice depends on the characteristics of the data being hashed.

Knuth has suggested that the best choice of A is = 0.6180339887

Q→ 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 $K = 12345$

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

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

$$= \lfloor 1000 (.617385) \rfloor$$

$$= \lfloor 617.385 \rfloor$$

$$= 617.$$

Mid Square Method.

— It works in two steps :—

Step 1 : Square the value of the key. That is, find k^2.

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

→ The algorithm works well because most or all digits of the key values contribute to the result. → This is because all digits in the original keys value contribute to produce the middle 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 digits must be choosen from all the keys. Therefore, the hash function can be given as :

$h(K) = s$

where s is obtained by selecting r digits from k^2.

Ques → Calculate the hash value for keys 1234 and 5642 using mid-square method. The hash table has 100 memory locations?

Ans → Note that the hash ~~function~~ table has 100 memory locations whose indices vary from 0 to 99. This means that only two digits are needed to map the key to a location in the hash table, so r=2.

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

$k = 5642$, $k^2 = 31832164$, $h(5642) = 21$

Observe that 3rd & 4th digits starting from the right are choosen.

Folding Method.

This method works in the following two steps :→

Step 1 :→ Divide the key value into a number of parts. That is, divide K into parts $K_1, K_2, \dots, K_n$, where each part has the same no. 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

$$K_1 + K_2 + \dots + K_n$$

The hash value is produced by ignoring the last carry

Ques → Given a hashtable of 100 location, calculate the hash value using folding method for keys 5678, 321, and 34567.

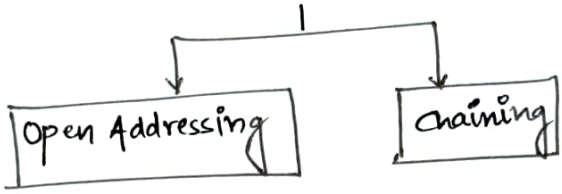
Soln → Since there are 100 memory locations to address, we will break the key into parts where each part (except the last) will contain two digits. The hash values can be obtained as shown below :→

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 the same location.
- Obviously, two records cannot be stored in the same location.
- Therefore → a method used to solve the problem of collision, also called COLLISION RESOLUTION TECHNIQUE, is applied.

TWO MOST POPULAR METHODS OF RESOLVING COLLISION ARE



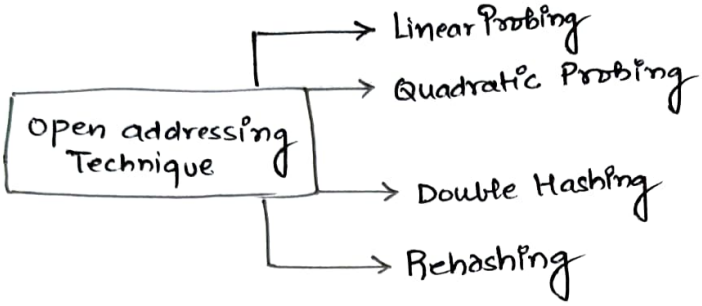
Collision Resolution by Open Addressing

- Once a collision takes place, open addressing or closed hashing computes new positions using a probe sequence and the next record is stored in that position.

- The hash table contains two types of values:
  (a). Sentinel values (e.g -1) → this value indicates that the location contains no data value at present but can be used to hold a value.
  (b). Data values

- however, if the location → already has some data value stored in it, then other slots are examined systematically in the forward direction to find a free slot. → If NO SINGLE FREE LOCATION PRESENT → than the "OVERFLOW CONDITION".

- The process of examining memory location in the hash table is called Probing.



(1). Linear Probing

→ In this technique, if a value is already stored at a 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 that varies from 0 to $m-1$.

← Therefore, for a given key k, first the location generated by $[h'(k) \bmod m]$ is probed because for the first time $i=0$.

— If location is free → value is stored. else,

— second probe generates the address of the location given by $[h'(k) + 1] \bmod m$. Similarly, if the location is occupied, then subsequent probes, generate the address as $[h'(k) + 2] \bmod m$, $[h'(k) + 3] \bmod m$, $[h'(k) + 4] \bmod m$, $[h'(k) + 5] \bmod m$ and so on, until a free location is found.

Ques → Consider a hash table of size 10. Using Linear Probing, insert the key 72, 27, 36, 24, 63, 81, 92 and 101 into the table?

Let $h'(k) = k \bmod m$, $m = 10$

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

Step 1: → key = 72

$h(72,0) = (72 \bmod 10 + 0) \bmod 10$

$= 2 \bmod 10$

$= 2$

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

Step 2: → key = 27

$h(27,0) = (27 \bmod 10 + 0) \bmod 10$

$= (7) \bmod 10$

$= 7$

2							7		
	72						27		

Step 3:- key = 36

$h(36,0) = (36 \bmod 10 + 0) \bmod 10$

$= (6 + 0) \bmod 10 \Rightarrow 6$

		2		4			6	7		
		72					36	27		

Step 4:- key 24

$h(24,0) = (24 \bmod 10 + 0) \bmod 10$

$= (4) \bmod 10$

$= 4$

		2		4			6	7		
		72		24			36	27		

Step 5:- key = 63

		2	3	4			6	7		
		72	63	24			36	27		

Step6:- Key = 81

$h'(k) = (81 \bmod 10 + 0) \bmod 10$

$= 1$

1	2	3	4	5	6	7	8	9
81	72	63	24		36	27		

Step7:- Key = 92

$h(92,0) = (92 \bmod 10 + 0) \bmod 10$

$= (2) \bmod 10$

$= 2$

Now T[2] is occupied, so we cannot store the key 92 in T[2]

$h(92,1) = (92 \bmod 10 + 1) \bmod 10$

$= 3$

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

key 92

$h(92,2) = (92 \bmod 10 + 2) \bmod 10$

$= 4.$

T[4] occupied.

Thus probe i = 3

$h(92,3) = (92 \bmod 10 + 3) \bmod 10$

$= (2+3) \bmod 10$

$= 5$

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

0	1	2	3	4	5	6	7	8	9
	81	72	63	24	92	36	27		

Searching a value using Linear Probing.

→ The procedure for searching a value in a hash table is same as for storing a value in a hash table.

→ While searching for a value in a hash table, the array index is recomputed and key of the element stored at that location is compared 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).

– If the key does not match → search function begin sequential search of the array that continue until.

↳ value is found.

↳ search func. encounters a vacant location in the array – indicate value not present.

↳ seach func. terminates → reaches end of the table & value is not present.

In worst case → the search operation may have to make (n-1) comparisons, and the running time of the search algorithm may take O(n) time

NOTE →

With the INCREASE IN THE NUMBER OF COLLISIONS, The DISTANCE b/w the array index computed by the hash function and actual location of the element increases thereby increasing the search time.

PROS AND CONS

– Linear Probing finds empty location by doing a linear search in the array. beginning from position h(k).

– Algo provide good memory caching through good locality of reference, the drawback of this algorithm results in clustering, thus there is higher risk of more collision where one collision has already taken place.

→ Performance of linear Probing is sensitive to the distribution of i/p values.

→ As the hash table fills, clusters of consecutive cells are formed and time for searching increases with the size of cluster. ——→ and as new value added it is also added at the end of cluster, which again increases the length of the cluster.

More the no. of collision, higher the probes that are required to find a free location and lessey is the peyprmance. This phenomena called PRIMARY CLUSTERING.

To AVOID PRIMARY CLUSTERING, other techniques such as QUADRATIC PROBING & DOUBLE HASHING USED.

QUADRATIC PROBING.

$h(K, i) = [h'(K) + c_1 i + c_2 i^2] \bmod m$

m = size of hash table.

$h'(K) = (K \bmod m)$

i = probe number that varies from 0 to $m-1$, and c_1 and c_2 are constants such that c_1 and $c_2 \neq 0$.

Quadratic Probing eliminates the primary clustering phenomena 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 $h'(K) \bmod m$ is probed. If the location is free, the value is stored in it, else subsequent location probed are offset by factors that depends in a Quadratic manner on the probe number i.

Although Quadratic Probing, performs better than linear probing, in order to maximize the utilization of the hash table, the value of c_1, c_2 & m needs to be constrained.

Ques:- Consider a hash table of size 10. Using Quadratic probing, insert the keys 72, 27, 36, 24, 63, 81 and 101 into the table. Take $c_1 = 1$ and $c_2 = 3$.

Let $h'(k) = k \bmod m$, $m = 10$

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

We have $h(k,i) = [h'(k) + c_1 i + c_2 i^2] \bmod m$

Step 1:- Key = 72

$h(72,0) = [72 \bmod 10 + 1\times0 + 3\times0] \bmod 10$

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

$= 2 \bmod 10$

$= 2$

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

2
72

Step 2:→ Key = 27

$h(27,0) = [27 \bmod 10 + 1\times0 + 3\times0] \bmod 10$

$= 7 \bmod 10$

$= 7$

2	7
72	27

Step 3:→ Key = 36.

$h(36,0) = [36 \bmod 10 + 1\times0 + 3\times0] \bmod 10$

$= 6 \bmod 10 = 6$

2	6	7
72	36	27

Step 4:- Key = 24

$h(24,0) = [24 \bmod 10 + 1\times0 + 3\times0] \bmod 10$

$= 4 \bmod 10 \Rightarrow 4$

2	4	6	7
72	24	36	27

Step 5:- Key = 63

$h(63,0) =$

2	3	4	6	7
72	63	24	36	27

Step 6:- Key = 81

0	1	2	3	4	5	6	7	8	9
-1	81	72	-1	24	-1	36	27	-1	-1

Step 7: Key = 101

$h(101,0) = [101 \bmod 10 + 1\times0 + 3\times0] \bmod 10$

$= 1 \bmod 10$

$= 1$

T[1] already occupied, the key 101 cannot be stored in T[1].

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

key = 101

$h(101,0) = [101 \bmod 10 + 1\times1 + 3\times1] \bmod 10$

$= [1 + 4] \bmod 10$

$= 5 \bmod 10$

$= 5$

Since T[5] is vacant, insert the key 101 in T[5]. The hash table now becomes 8→

0	1	2	3	4	5	6	7	8	9	
-1	81	72	63	24	101	86	27	-1	-1	

PROS AND CONS.

→ Quadratic probing resolves the primary clustering problem that exists in Linear probing.

→ Quadratic Probing provides good memory caching because it preserves some locality of reference. But linear probing does this task better and gives a better cache performance.

→ One of the major drawback of Quadratic Probing is that a sequence of successive probes may only explore a fraction of the table & 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 is full.

Although Quadratic Probing is free from primary clustering, it is still liable to what is known as "SECONDARY CLUSTERING".

It means that if there is a collision b/w two keys, then the same probe sequence will be followed for both.

With Quadratic Probing, the probability for multiple collisions increases as the table becomes full.

Quadratic Probing is widely applied in the Berkeley Fast File System to allocate free blocks.

Double Hashing

- To start with, double hashing uses one hash and then repeatedly steps forward an interval until an empty location is reached.

The interval is decided using a second, independent hash function, hence named DOUBLE HASHING.

→ In Double Hashing we use two function rather than a single function.

→ The hash function

$$h(k,i) = [h_1(k) + i\,h_2(k)] \bmod m$$

- m = size of the hash table.
- $h_1(k) = k \bmod m$
- $h_2(k) = k \bmod m'$
- i = probe number that varies from
- 0 to $m-1$
- m' is choosen 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 $[h_1(k) \bmod m]$ because during the first probe, $i=0$. If the location is vacant value inserted, else subsequent probes generate location that are at an offset of $[h_2(k) \bmod m]$

Ques→ hash table of size = 10, using double hashing insert the key 72, 27, 36, 24, 63, 81, 92 and 101 into the table.

$h_1 = (k \bmod 10)$

$h_2 = (k \bmod 8)$

⇒ Let $m = 10$

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

We have

$h(k,i) = [h_1(k) + i\,h_2(k)] \bmod m$

Step 1: Key = 72

$h(72,0) = [72 \bmod 10 + (0 \times 72 \bmod 8)] \bmod 10$

$= [2 + (0 \times 0)] \bmod 10$

$= 2 \bmod 10$

$= 2$

Step 2: Key = 27

$h(27,0) = [27 \bmod 10 + (0 \times 27 \bmod 8)] \bmod 10$

$= [7 + (0 \times 3)] \bmod 10$

$= 7.$

⋮

Step 7 = Key = 92

$h(92,0) = [92 \bmod 10 + (0 \times 92 \bmod 8)] \bmod 10$

$= [2 + (0 \times 4)] \bmod 10$

$= 2 \bmod 10$

$= 2$

Now T[2] is occupied, so we cannot store the key 92 in T[2]. Thus probe, $i=1$, this time

Key = 92

$h(92,1) = [92 \bmod 10 + (1 \times 92 \bmod 8)] \bmod 10$

$= (2+4) \bmod 10$

$= 6$

T[6] is occupied →

$h(92,2) = [92 \bmod 10 + (2 \times 92 \bmod 8)] \bmod 10$

$= 10 \bmod 10$

$= 0.$ 92 stored at 0.

Step 8 :→ key = 101

$h(101, 0) = [101 \bmod 10 + (0 \times 101 \bmod 8)] \bmod 10$

$= 1 \bmod 10$

$= 1$

Now T[1] is occupied, so we cannot stored the key 101 in T[1]. Therefore, try again for the next location. Thus probe, i=1, this time.

key = 101

$h(101, 1) = [101 \bmod 10 + (1 \times 101 \bmod 8)] \bmod 10$

$= [1 + (1 \times 5)] \bmod 10$

$= [1+5] \bmod 10$

$= 6$

Now T[6] is occupied → so 101 cannot be store the key in T[6] → Therefore try again for the next location with probe i=2. Repeat the entire process until a vacant location is found → you have probe many times to insert 101 key.

Although Doublehashing is a very efficient algorithm → it always require m to be a prime number. In our case m=10 which is not a prime number, hence the degradation in performance. Had m been equal to 11, the algorithm would have worked very efficiently.

Thus we say that the performance of the technique is sensitive to the value of m.

Pros And Cons

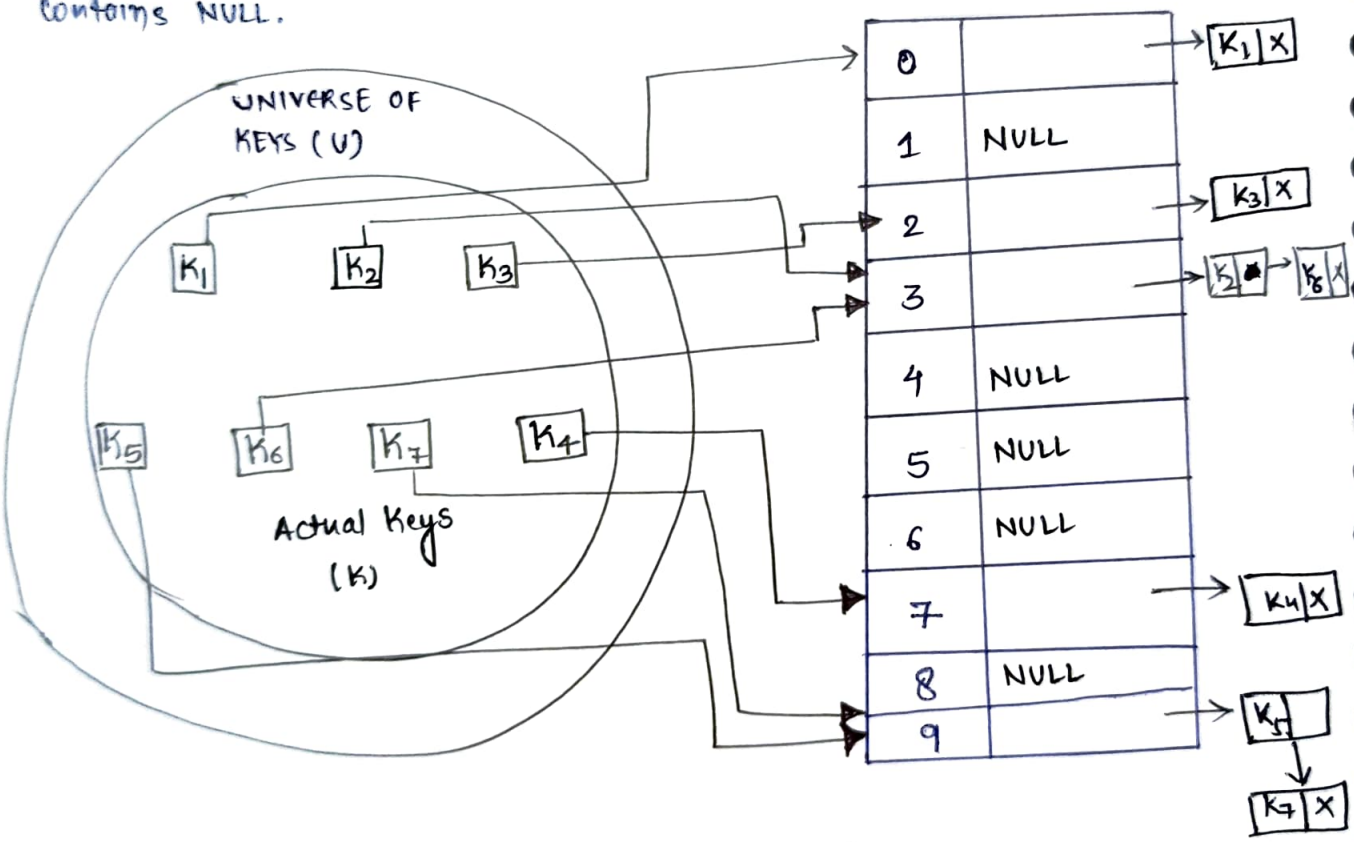
Double hashing minimizes repeated collision and the effect of clustering That is double hashing is free from problems associated with primary clustering as well as secondary clustering.

Collision Resolution by Chaining

→ In chaining, each location in a hash table stores a pointer to a linked list that contain all the key values that were hashed to that location.

That is location 1 in the hash table pointer to the head of the linked list of all the key values that hashed to 1.

However, if no key value hashes to 1, then location 1 in the hash table contains NULL.



Keys being hashed to a chained hash table.

Operations on a Chained Hash Table.

→ Searching for a value in a chained hash table is as simple as scanning a linked list for an entry with the given key.

→ Insertion operation appends the key to the end of the linked list pointed by the hashed location.

→ Deleting a key requires searching the list and removing the element.

→ Chained hash table with linked lists are widely used due to the simplicity of the algorithms to insert, delete, and search a key.

→ Code for these algorithms is exactly the same as that for inserting, deleting and searching a value in a single linked list

→ Cost of Inserting a key in a chained hash table is $O(1)$.

→ Cost of Deleting and searching a value is given as $O(m)$ where m is the number of elements in the list of that location.

→ Searching and Deleting takes more time because these operations scan the entries of the selected location for the desired key.

→ In worst case → searching a value may take a running time of $O(n)$, where n is the number of key values stored in the chained hash table.

PROS AND CONS.

→ main adv. of using a chained hash table → remain effective even when the number of key values to be stored is much higher than the no. of location in the hash table.

→ with increase in the number of keys to be stored, the performance of a chained hash table does degrade gradually (linearly).

→ for ex → a chained hash table with 1000 memory location and 10,000 stored keys will gives 5 to 10 times less performance as compared to a chained hash table with 10,000 location.

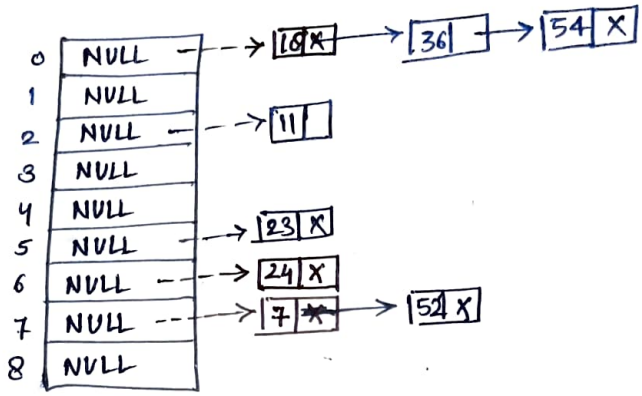
→ Chain hash free from clustering problem.

→ Chained hash table is still 1000 times faster than a simple hash table.

→ In this performance does not degrade when the table is more than half full. unlike QUADRATIC PROBING.

Ques → Insert the keys 7, 24, 18, 52, 36, 54, 11 and 23 in a chained hash table of 9 memory locations. Use $h(K) = k \bmod m$.

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



Step 1: key = 7
$h(K) = 7 \bmod 9$
$= 7$

Step 2: key = 24
$h(K) = 24 \bmod 9$
$= 6$

Step 3: key = 18
$h(K) = 18 \bmod 9$
$= 0$

Step 4: key = 52
$h(K) = 52 \bmod 9$
$= 52 \bmod 9 = 7.$

Step 5:- key = 36
$= 36 \bmod 9 = 0$

Step 6:- 54 = key
$= 54 \bmod 9 = 0$

Step 7: 11
$11 \bmod 9 = 2$

Step 8: key = 23
$23 \bmod 9 = 5$

Unordered Map is based on the Concept of Hashing

//1. Functions of the unordered map

```
/*
#include <iostream>
#include <unordered_map>
#include <string>
int main() {
   // Create and initialize an unordered_map
   std::unordered_map<int, std::string> malay;
   // Insert elements
   malay [1] = "one";
   malay [2] = "two";
   malay [3] = "three";
   // Access elements
   std::cout << "Key 1 has value: "<< malay [1] << std::endl;
   std::cout << "Key 2 has value: " <<malay [2] << std::endl;
   // Check if a key exists
   int key = 4;
   auto it = malay.find(key);
   if (it != malay.end()) {
      std::cout << "Found key " << key << " with value: " << it->second << std::endl;
   } else {
      std::cout << "Key " << key << " not found" << std::endl;
   }
   // Erase an element
   malay.erase(2);
   std::cout << "After erasing key 2, size of map: " << malay.size() << std::endl;
   // Check if map is empty
   if (malay.empty()) {
      std::cout << "Map is empty" << std::endl;
   } else {
```

```
        std::cout << "Map is not empty" << std::endl;
    }
    // Clear the map
    malay.clear();
    std::cout << "After clearing, size of map: " << malay.size() << std::endl;
    return 0;
}
*/
```

//2. Demonstrate that the Vectors are dynamic in nature

```
/*
#include <iostream>
#include <vector>
int main() {
    std::vector<int> vec;
    // Display initial size and capacity
    std::cout << "Initial size: " << vec.size() << ", capacity: " << vec.capacity() << std::endl;
    // Add elements to the vector and observe capacity changes
    for (int i = 0; i < 20; ++i) {
        vec.push_back(i);
        std::cout << "After adding element " << i << ", size: " << vec.size() << ", capacity: " <<
vec.capacity() << std::endl;
    }
    // Remove elements from the vector
    for (int i = 0; i < 10; ++i) {
        vec.pop_back();
        std::cout << "After removing element, size: " << vec.size() << ", capacity: " << vec.capacity() <<
std::endl;
    }
    // Clear the vector
    vec.clear();
    std::cout << "After clearing, size: " << vec.size() << ", capacity: " << vec.capacity() << std::endl;
    return 0;
```

```
}
*/
// Program showing the concept of chain hashing in C for collision removal
/*
#include <stdio.h>
#include <stdlib.h>
#define BUCKET_SIZE 10
// Node structure for the linked list in each bucket
struct Node {
   int key;
   struct Node* next;
};
// Hash table structure containing an array of linked list nodes
struct HashTable {
   struct Node* table[BUCKET_SIZE];
};
// Initialize the hash table
void initHashTable(struct HashTable* ht) {
   for (int i = 0; i < BUCKET_SIZE; i++) {
      ht->table[i] = NULL;
   }
}
// Hash function (k mod 10)
int hashFunction(int key) {
   return key % BUCKET_SIZE;
}
// Insert a key into the hash table
void insert(struct HashTable* ht, int key) {
   int index = hashFunction(key);
   struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
   newNode->key = key;
```

```
    newNode->next = ht->table[index];
    ht->table[index] = newNode;
}
// Print the hash table
void printHashTable(struct HashTable* ht) {
    for (int i = 0; i < BUCKET_SIZE; i++) {
        printf("Bucket %d: ", i);
        struct Node* current = ht->table[i];
        while (current != NULL) {
            printf("%d -> ", current->key);
            current = current->next;
        }
        printf("NULL\n");
    }
}
int main() {
    struct HashTable ht;
    initHashTable(&ht);
    insert(&ht, 1);
    insert(&ht, 11);
    insert(&ht, 21);
    insert(&ht, 3);
    insert(&ht, 13);
    insert(&ht, 23);
    printHashTable(&ht);
    return 0;
}
*/
```

// 4. Program to show chain hashing in c++ for collision removal

```
/*
#include <iostream>
#include <vector>
using namespace std;
// Node structure for the linked list in each bucket
struct Node {
   int key;
   Node* next;
   Node(int k) : key(k), next(nullptr) {}
};

// Hash table using chain hashing
class HashTable {
private:
   vector<Node*> table;
   int tableSize;
public:
   // Constructor
   HashTable(int size) : tableSize(size) {
      table.resize(tableSize, nullptr);
   }
   // Hash function (k mod 10)
   int hashFunction(int key) {
      return key % tableSize;
   }
   // Insert a key into the hash table
   void insert(int key) {
      int index = hashFunction(key);
      Node* newNode = new Node(key);
      newNode->next = table[index];
```

```
        table[index] = newNode;
    }
    // Print the hash table
    void printHashTable() {
        for (int i = 0; i < tableSize; ++i) {
            cout << "Bucket " << i << ": ";
            Node* current = table[i];
            while (current) {
                cout << current->key << " -> ";
                current = current->next;
            }
            cout << "NULL" << endl;
        }
    }
};
int main() {
    HashTable ht(10); // Create hash table with bucket size 10

    // Insert keys into the hash table
    ht.insert(1);
    ht.insert(11);
    ht.insert(21);
    ht.insert(3);
    ht.insert(13);
    ht.insert(23);
    ht.insert(24);
    ht.insert(25);
    ht.insert(26);
    // Print the hash table
    ht.printHashTable();
    return 0; } */
```

Codes to initialize, insert, delete & search a value in a chained hash table

while the cost of inserting in a chained hash table is O(1), but the cost of deleting and searching a value is given as O(n) where n is the number of elements in the list of that location.

Searching and deleting takes more time because these operations scan the entries of the selected location for the desired key.

In the WORST CASE, Searching a value may take a running time of O(n), where n is the number of key values stored in the chained hash table.
This cases arises when all the key values are inserted into the linked list of the same location (of the hash table).
In this case, the hash table is ineffective.

```
// STRUCTURE OF THE NODE

typedef struct node-MT
  {
    int value;
    struct node *next;
  }node;
```

Code to initialize a chained hash table

```
/* Initializes m location in the chained hash table.
   The operation takes a running time of O(m) */

void initializeHashTable (node * hash_table[], int m)
     {
       int i;
       for (i=0; i<=m; i++)
           {
             hash_table[i] = NULL;
```

Code to Insert a value

```
/* The element is inserted at the beginning of the linked
   list whose pointer to its head is stored in the
   location given by h(K). The running time of the insert
   operation is O(1), as the new key value is always added
   as the first element of the list irrespective of the size
   of the linked list as well as that of the chained hash table */

node  *insert_value (node *hash_table[], int value)
     {
       node *new_node;
       new_node = (node*) malloc (sizeof (node));
       new_node->value = val;
       new_node->next = hash_table [h(x)];
       hash_table [h(x)] = new_node;
     }
```

Code to Search a value

```
/* The element is searched in the linked list whose
   pointer to its head is stored in the location
   given by h(K). If the search is successful, the
   function returns a pointer to the node in the linked
   list; otherwise it returns NULL.
   The WORST CASE running time of the SEARCH OPERATION
   is given as order of size of the linked list. */

node * search_value (node * hash_table [], int val)
    {
      node * ptr;
      ptr = hash_table [h(x)];
      While ( ( ptr != NULL ) && ( ptr -> value != val ))
              ptr = ptr -> next;
          if (ptr -> value == val)
              return ptr;
          else
              return NULL;
    }
```

Code to delete a value

/* To delete the node from the linked list whose head is stored at the location given by h(k) in the hash table, we need to know the address of the node's predecessor. We do this using a pointer save.

The Running Time complexity of the delete operation is same as that of the search operation because we need to search the predecessor of the node so that the node can be removed w/o affecting other nodes in the list */

```
void delete_value (node * hash_table [], int val)
{
 node  * save, * ptr;
  save = NULL;
  ptr = hash_table [h(x)];
  while ((ptr != NULL) && (ptr->value != val))
    {
      save = ptr;
       ptr = ptr -> next;
    }
  if ( ptr != NULL)
    {
     save -> next = ptr -> next;
       free(ptr);
    }
   else
       printf ("\n VALUE NOT FOUND ");
}
```