Hashing

Hashing is the transformation of a string of character into a usually shorter fixed-length value or key that represents the original string.

Hashing is used to index and retrieve items in a database because it is faster to find the item using the shortest hashed key than to find it using the original value. It is also used in many encryption algorithms.

A hash code is generated by using a key, which is a unique value.

Hashing is a technique in which given key field value is converted into the address of storage location of the record by applying the same operation on it.

The advantage of hashing is that allows the execution time of basic operation to remain constant even for the larger side.

Why we need Hashing?

Suppose we have 50 employees, and we have to give 4 digit key to each employee (as for security), and we want after entering a key, direct user map to a particular position where data is stored.

If we give the location number according to 4 digits, we will have to reserve 0000 to 9999 addresses because anybody can use anyone as a key. There is a lot of wastage.

In order to solve this problem, we use hashing which will produce a smaller value of the index of the hash table corresponding to the key of the user.

Universal Hashing

Let H be a finite collection of hash functions that map a given universe U of keys into the range {0, 1..... m-1}. Such a collection is said to be universal if for each pair of distinct keys k,l∈U, the number of hash functions h∈ H for which h(k)= h(l) is at most |H|/m. In other words, with a hash function randomly chosen from H, the chance of a collision between distinct keys k and l is no more than the chance 1/m of a collision if h(k) and h(l)were randomly and independently chosen from the set {0,1,...m-1}.

Rehashing

If any stage the hash table becomes nearly full, the running time for the operations of will start taking too much time, insert operation may fail in such situation, the best possible solution is as follows:

1. Create a new hash table double in size.
2. Scan the original hash table, compute new hash value and insert into the new hash table.
3. Free the memory occupied by the original hash table.

Example: Consider inserting the keys 10, 22, 31,4,15,28,17,88 and 59 into a hash table of length m = 11 using open addressing with the primary hash function h' (k) = k mod m .Illustrate the result of inserting these keys using linear probing, using quadratic probing with c_1=1 and c_2=3, and using double hashing with $h_2(k)$ = 1 + (k mod (m-1)).

Solution: Using Linear Probing the final state of hash table would be:

0	22
1	88
2	/
3	/
4	4
5	15
6	28
7	17
8	59
9	31
10	10

Using Quadratic Probing with $c_1=1$, $c_2=3$, the final state of hash table would be h (k, i) = (h' (k) $+c_1*i+ c_2 *i^2$) mod m where m=11 and h' (k) = k mod m.

0	22
1	88
2	/
3	17
4	4
5	/
6	28
7	59
8	15
9	31
10	10

Using Double Hashing, the final state of the hash table would be:

0	22
1	/
2	59
3	17
4	4
5	15
6	28
7	88
8	/
9	31
10	10

Hash Tables

It is a collection of items which are stored in such a way as to make it easy to find them later.

Each position in the hash table is called slot, can hold an item and is named by an integer value starting at 0.

The mapping between an item and a slot where the item belongs in a hash table is called a Hash Function. A hash Function accepts a key and returns its hash coding, or hash value.

Assume we have a set of integers 54, 26, 93, 17, 77, 31. Our first hash function required to be as "remainder method" simply takes the item and divide it by table size, returning remainder as its hash value i.e.

1. h item = item % (size of table)
2. Let us say the size of table = 11, then
3. 54 % 11 = 10 26 % 11 = 4 93 % 11 = 5
4. 17 % 11 = 6 77 % 11 = 0 31 % 11 = 9

ITEM	HASH VALUE
54	10
26	4
93	5
17	6
77	0
31	9

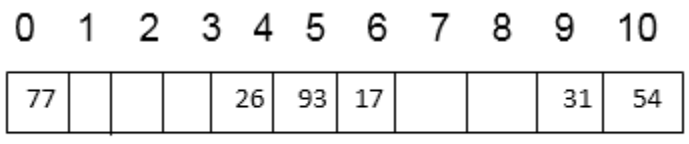


Fig: Hash Table

Now when we need to search any element, we just need to divide it by the table size, and we get the hash value. So we get the O (1) search time.

Now taking one more element 44 when we apply the hash function on 44, we get (44 % 11 = 0), But 0 hash value already has an element 77. This Problem is called as Collision.

Collision: According to the Hash Function, two or more item would need in the same slot. This is said to be called as Collision.

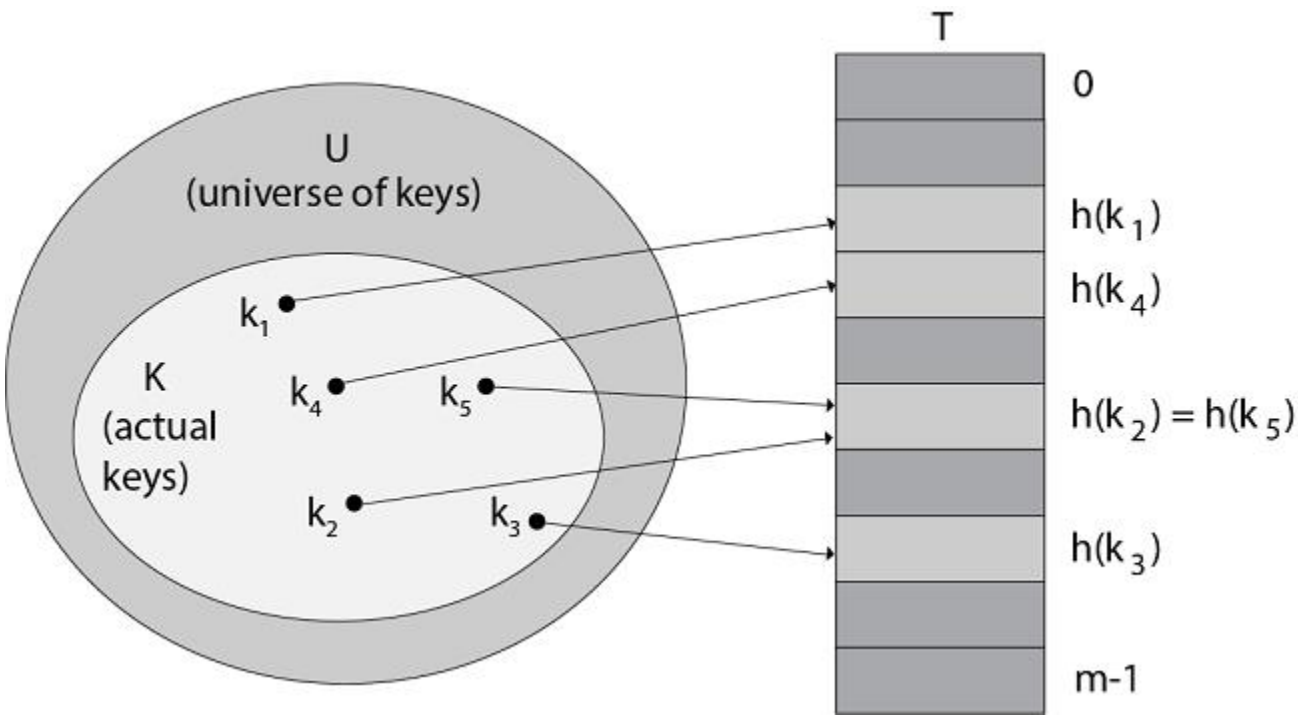


Figure: using a hash function h to map keys to hash-table slots. Because keys K2 and k5 map to the same slot, they collide.

Why use HashTable?

1. If **U** (Universe of keys) is large, storing a table T of size [U] may be impossible.
2. Set **k** of keys may be small relative to **U** so space allocated for **T** will waste.

So Hash Table requires less storage. Indirect addressing element with key k is stored in slot k with hashing it is stored in h (k) where h is a hash f^n and hash (k) is the value of key k. Hash f^n required array range.

Application of Hash Tables:

Some application of Hash Tables are:

1. **Database System:** Specifically, those that are required efficient random access. Usually, database systems try to develop between two types of access methods: sequential and random. Hash Table is an integral part of efficient random access because they provide a way to locate data in a constant amount of time.
2. **Symbol Tables:** The tables utilized by compilers to maintain data about symbols from a program. Compilers access information about symbols frequently. Therefore, it is essential that symbol tables be implemented very efficiently.
3. **Data Dictionaries:** Data Structure that supports adding, deleting, and searching for data. Although the operation of hash tables and a data dictionary are similar, other Data Structures may be used to implement data dictionaries.
4. **Associative Arrays:** Associative Arrays consist of data arranged so that n^{th} elements of one array correspond to the n^{th} element of another. Associative Arrays are helpful for indexing a logical grouping of data by several key fields.

Methods of Hashing

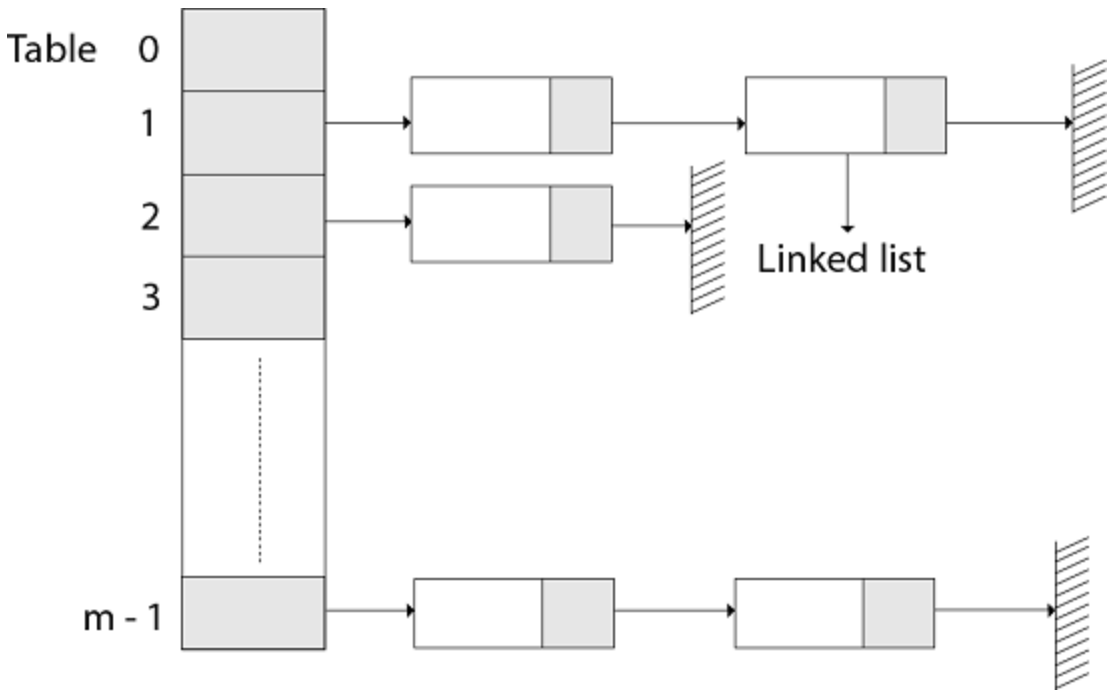
There are two main methods used to implement hashing:

1. Hashing with Chaining
2. Hashing with open addressing

1. Hashing with Chaining

In Hashing with Chaining, the element in S is stored in Hash table T [0...m-1] of size m, where m is somewhat larger than n, the size of S. The hash table is said to have m slots. Associated with the hashing scheme is a hash function h which is mapping from U to {0...m-1}.Each key k ∈S is stored in location T [h (k)], and we say that k is hashed into slot h (k). If more than one key in S hashed into the same slot then we have a **collision**.

In such case, all keys that hash into the same slot are placed in a linked list associated with that slot, this linked list is called the chain at slot. The load factor of a hash table is defined to be ∝=n/m it represents the average number of keys per slot. We typically operate in the range m=θ(n), so ∝ is usually a constant generally ∝<1.



Collision Resolution by Chaining:

In chaining, we place all the elements that hash to the same slot into the same linked list, As fig shows that Slot j contains a pointer to the head of the list of all stored elements that hash to j ; if there are no such elements, slot j contains NIL.

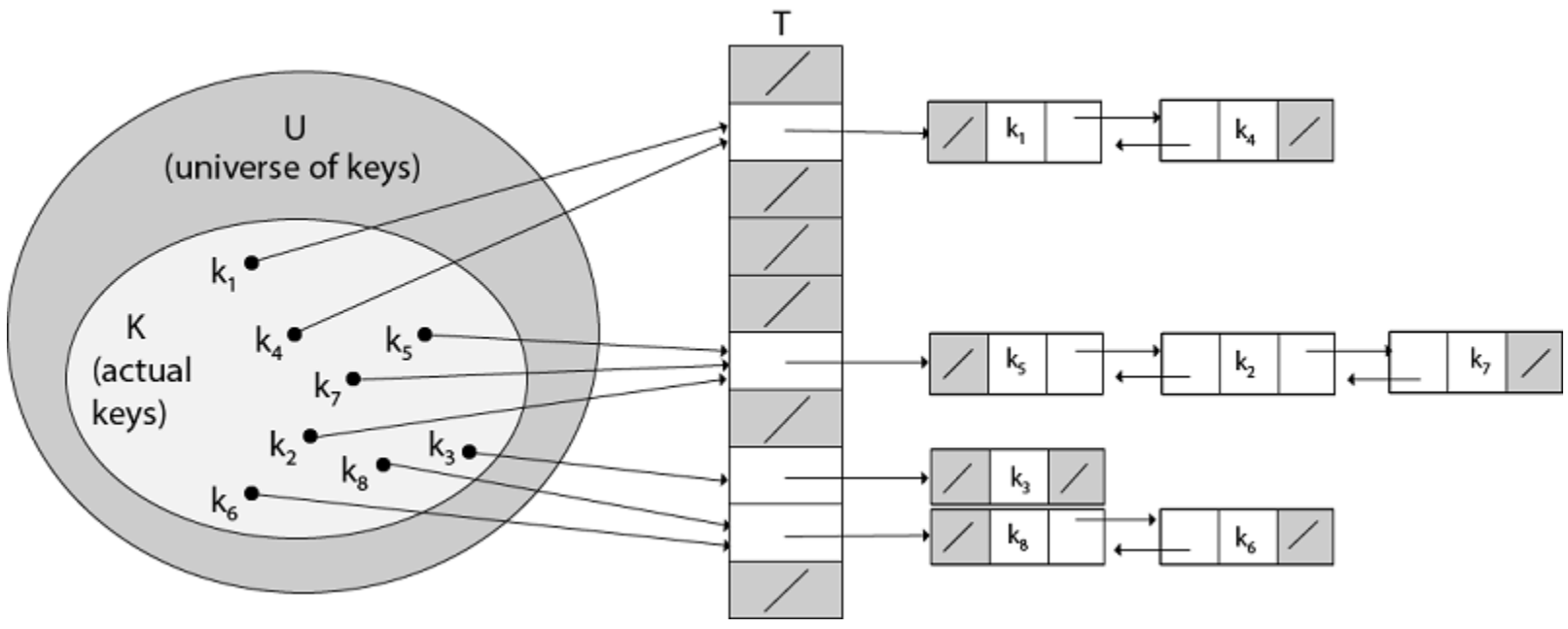


Fig: Collision resolution by chaining.

Each hash-table slot T [j] contains a linked list of all the keys whose hash value is j.

For example, h (k1) = h (k4) and h (k5) = h (k7) =h (K2). The linked list can be either singly or doubly linked; we show it as doubly linked because deletion is faster that way.

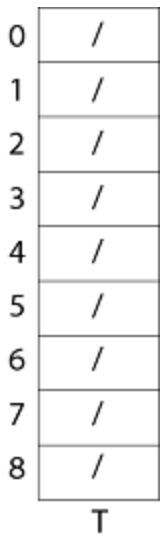
Analysis of Hashing with Chaining:

Given a hash table T with m slots that stores n elements, we define the load factors α for T as n/m that is the average number of elements stored in a chain. The worst case running time for searching is thus θ(n) plus the time to compute the hash function- no better than if we used one linked list for all the elements. Clearly, hash tables are not used for their worst-case performance.

The average performance of hashing depends on how well the hash function h distributes the set of keys to be stored among the m slots, on the average.

Example: let us consider the insertion of elements 5, 28, 19,15,20,33,12,17,10 into a chained hash table. Let us suppose the hash table has 9 slots and the hash function be h (k) =k mod 9.

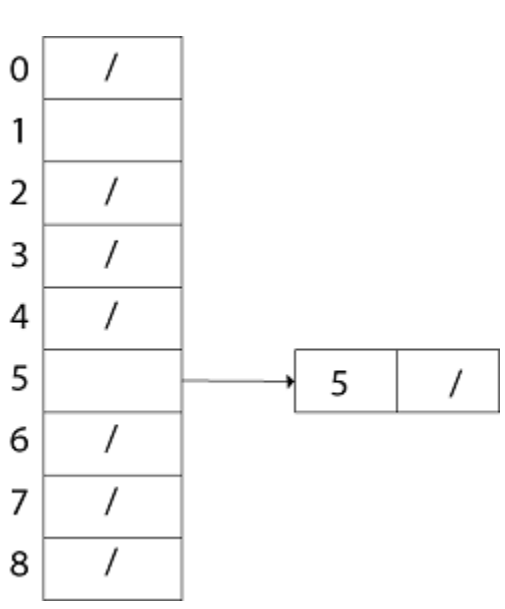
Solution: The initial state of chained-hash table



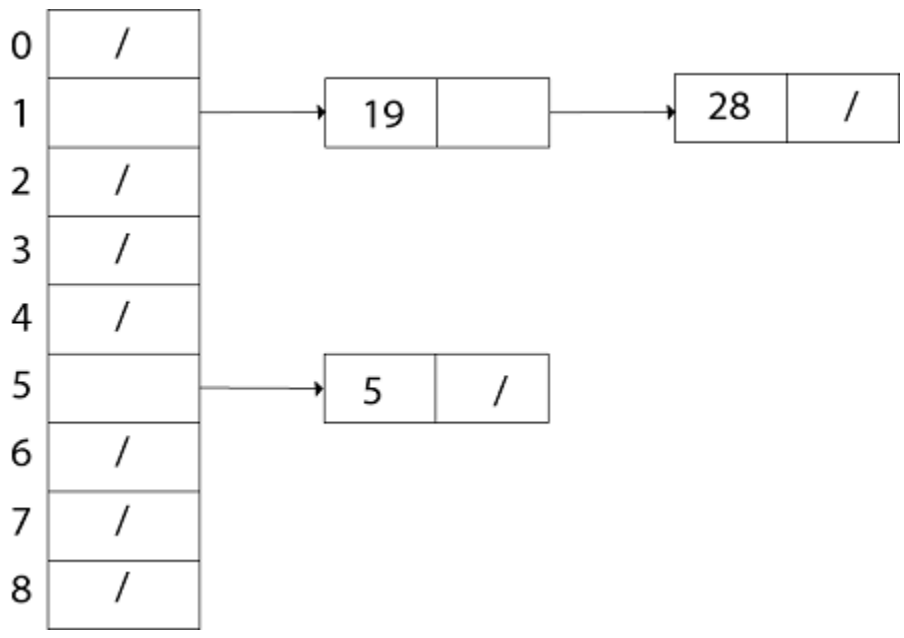
Insert 5:

1. h (5) = 5 mod 9 =5

Create a linked list for T [5] and store value 5 in it.

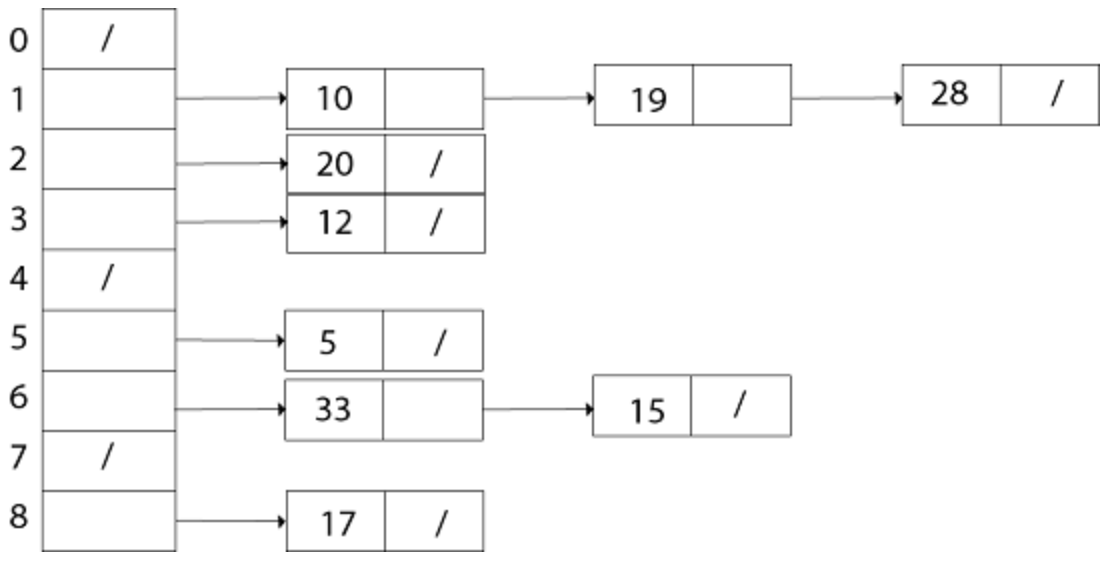


Similarly, insert 28. h (28) = 28 mod 9 =1. Create a Linked List for T [1] and store value 28 in it. Now insert 19 h (19) = 19 mod 9 = 1. Insert value 19 in the slot T [1] at the beginning of the linked-list.



1. Now insert h 15, h (15) = 15 mod 9 = 6. Create a link list **for** T [6] and store value 15 in it.
2. Similarly, insert 20, h (20) = 20 mod 9 = 2 in T [2].
3. Insert 33, h (33) = 33 mod 9 = 6
4. In the beginning of the linked list T [6]. Then,
5. Insert 12, h (12) = 12 mod 9 = 3 in T [3].
6. Insert 17, h (17) = 17 mod 9 = 8 in T [8].
7. Insert 10, h (10) = 10 mod 9 = 1 in T [1].

Thus the chained- hash- table after inserting key 10 is



2. Hashing with Open Addressing

In Open Addressing, all elements are stored in hash table itself. That is, each table entry consists of a component of the dynamic set or NIL. When searching for an item, we consistently examine table slots until either we find the desired object or we have determined that the element is not in the table. Thus, in open addressing, the load factor α can never exceed 1.

The advantage of open addressing is that it avoids Pointer. In this, we compute the sequence of slots to be examined. The extra memory freed by not sharing pointers provides the hash table with a larger number of slots for the same amount of memory, potentially yielding fewer collision and faster retrieval.

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

Thus, the hash function becomes

1. h : U x {0,1,....m-1} → {0,1,....,m-1}.

With open addressing, we require that for every key k, the probe sequence

1. {h, (k, 0), h (k, 1)....h (k, m-1)}
2. Be a Permutation of (0, 1..... m-1)

The HASH-INSERT procedure takes as input a hash table T and a key k

HASH-INSERT (T, k)

```
1. i ← 0
2. repeat j ← h (k, i)
3. if T [j] = NIL
4. then T [j] ← k
5. return j
6. else ← i= i +1
7. until i=m
8. error "hash table overflow"
```

The procedure HASH-SEARCH takes as input a hash table T and a key k, returning j if it finds that slot j contains key k or NIL if key k is not present in table T.

HASH-SEARCH.T (k)

```
1. i ← 0
2. repeat j ← h (k, i)
3. if T [j] =j
4. then return j
5. i ← i+1
6. until T [j] = NIL or i=m
7. return NIL
```

Open Addressing Techniques

Three techniques are commonly used to compute the probe sequence required for open addressing:

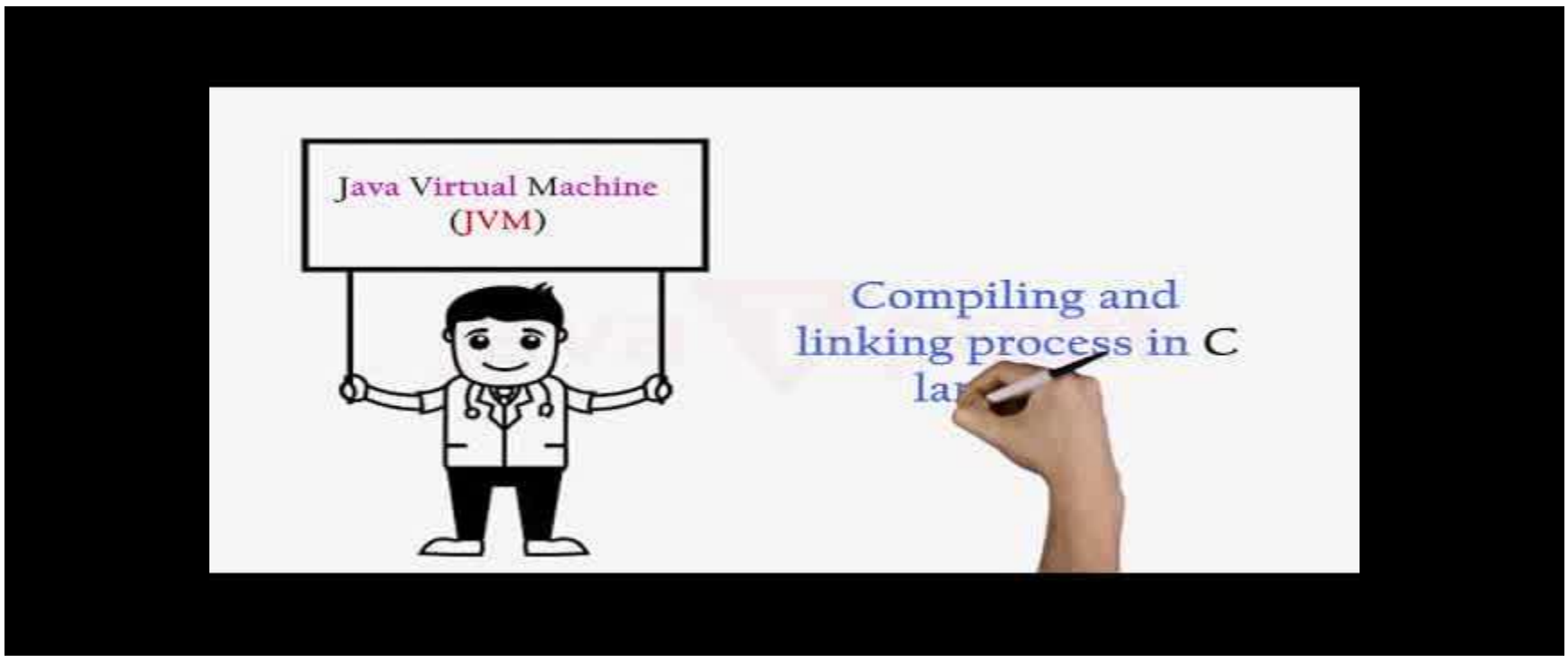
1. Linear Probing.
2. Quadratic Probing.
3. Double Hashing.

1. Linear Probing:

It is a Scheme in Computer Programming for resolving collision in hash tables.

Suppose a new record R with key k is to be added to the memory table T but that the memory locations with the hash address H (k). H is already filled.

Our natural key to resolve the collision is to crossing R to the first available location following T (h). We assume that the table T with m location is circular, so that T [i] comes after T [m].



The above collision resolution is called "Linear Probing".

Linear probing is simple to implement, but it suffers from an issue known as primary clustering. Long runs of occupied slots build up, increasing the average search time. Clusters arise because an empty slot proceeded by i full slots gets filled next with probability (i + 1)/m. Long runs of occupied slots tend to get longer, and the average search time increases.

Given an ordinary hash function h': U {0, 1...m-1}, the method of linear probing uses the hash function.

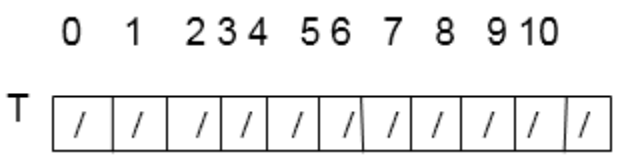
1. h (k, i) = (h' (k) + i) mod m

Where 'm' is the size of hash table and h' (k) = k mod m. for i=0, 1....m-1.

Given key k, the first slot is T [h' (k)]. We next slot T [h' (k) +1] and so on up to the slot T [m-1]. Then we wrap around to slots T [0], T [1]....until finally slot T [h' (k)-1]. Since the initial probe position dispose of the entire probe sequence, only m distinct probe sequences are used with linear probing.

Example: Consider inserting the keys 24, 36, 58,65,62,86 into a hash table of size m=11 using linear probing, consider the primary hash function is h' (k) = k mod m.

Solution: Initial state of hash table



Insert 24. We know h (k, i) = [h' (k) + i] mod m
Now h (24, 0) = [24 mod 11 + 0] mod 11
= (2+0) mod 11 = 2 mod 11 = 2
Since T [2] is free, insert key 24 at this place.

Insert 36. Now h (36, 0) = [36 mod 11 + 0] mod 11
= [3+0] mod 11 = 3
Since T [3] is free, insert key 36 at this place.

Insert 58. Now h (58, 0) = [58 mod 11 +0] mod 11
= [3+0] mod 11 =3
Since T [3] is not free, so the next sequence is
h (58, 1) = [58 mod 11 +1] mod 11
= [3+1] mod 11= 4 mod 11=4

T [4] is free; Insert key 58 at this place.

Insert 65. Now h (65, 0) = [65 mod 11 +0] mod 11
= (10 +0) mod 11= 10
T [10] is free. Insert key 65 at this place.

Insert 62. Now h (62, 0) = [62 mod 11 +0] mod 11
= [7 + 0] mod 11 = 7
T [7] is free. Insert key 62 at this place.

Insert 86. Now h (86, 0) = [86 mod 11 + 0] mod 11
= [9 + 0] mod 11 = 9
T [9] is free. Insert key 86 at this place.
Thus,

	0	1	2	3	4	5	6	7	8	9	10
T	/	/	24	36	58	/	/	62	/	86	65

2. Quadratic Probing:

Suppose a record R with key k has the hash address H (k) = h then instead of searching the location with addresses h, h+1, and h+ 2...We linearly search the locations with addresses

$h, h+1, h+4, h+9...h+i^2$

Quadratic Probing uses a hash function of the form

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

Where (as in linear probing) h' is an auxiliary hash function c_1 and $c_2 \neq 0$ are auxiliary constants and i=0, 1...m-1. The initial position is T [h' (k)]; later position probed is offset by the amount that depend in a quadratic manner on the probe number i.

Example: Consider inserting the keys 74, 28, 36,58,21,64 into a hash table of size m =11 using quadratic probing with c_1=1 and c_2=3. Further consider that the primary hash function is h' (k) = k mod m.

Solution: For Quadratic Probing, we have

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

0	1	2	3	4	5	6	7	8	9	10
/	/	/	/	/	/	/	/	/	/	/

This is the initial state of hash table

Here c_1= 1 and c_2=3
h (k, i) = [k mod m + i + $3i^2$] mod m

Insert 74.

h (74,0)= (74 mod 11+0+3x0) mod 11
= (8 +0+0) mod 11 = 8
T [8] is free; insert the key 74 at this place.

Insert 28.

h (28, 0) = (28 mod 11 + 0 + 3 x 0) mod 11
= (6 +0 + 0) mod 11 = 6.
T [6] is free; insert key 28 at this place.

Insert 36.

h (36, 0) = (36 mod 11 + 0 + 3 x 0) mod 11
= (3 + 0+0) mod 11=3
T [3] is free; insert key 36 at this place.

Insert 58.

h (58, 0) = (58 mod 11 + 0 + 3 x 0) mod 11

```
                = (3 + 0 + 0) mod 11 = 3
T [3] is not free, so next probe sequence is computed as
      h (59, 1) = (58 mod 11 + 1 + 3 x1²) mod 11
                  = (3 + 1 + 3) mod 11
                =7 mod 11= 7
T [7] is free; insert key 58 at this place.
```

Insert 21.

```
      h (21, 0) = (21 mod 11 + 0 + 3 x 0)
                = (10 + 0) mod 11 = 10
T [10] is free; insert key 21 at this place.

      Insert 64.

      h (64, 0) = (64 mod 11 + 0 + 3 x 0)
                = (9 + 0+ 0) mod 11 = 9.
T [9] is free; insert key 64 at this place.
```

Thus, after inserting all keys, the hash table is

0	1	2	3	4	5	6	7	8	9	10
/	/	/	36	/	/	28	58	74	64	21

3. Double Hashing:

Double Hashing is one of the best techniques available for open addressing because the permutations produced have many of the characteristics of randomly chosen permutations.

Double hashing uses a hash function of the form

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

Where h_1 and h_2 are auxiliary hash functions and m is the size of the hash table.

$h_1(k) = k \bmod m$ or $h_2(k) = k \bmod m'$. Here m' is slightly less than m (say m-1 or m-2).

Example: Consider inserting the keys 76, 26, 37,59,21,65 into a hash table of size m = 11 using double hashing. Consider that the auxiliary hash functions are $h_1(k)=k \bmod 11$ and $h_2(k) = k \bmod 9$.

Solution: Initial state of Hash table is

	0	1	2	3	4	5	6	7	8	9	10
T	/	/	/	/	/	/	/	/	/	/	/

1. Insert 76.

```
          h₁(76) = 76 mod 11 = 10
          h₂(76) = 76 mod 9 = 4
                    h (76, 0) = (10 + 0 x 4) mod 11
                              = 10 mod 11 = 10
T [10] is free, so insert key 76 at this place.
```

2. Insert 26.

```
       h₁(26) = 26 mod 11 = 4
           h₂(26) = 26 mod 9 = 8
                     h (26, 0) = (4 + 0 x 8) mod 11
                               = 4 mod 11 = 4
T [4] is free, so insert key 26 at this place.
```

3. Insert 37.

```
       h₁(37) = 37 mod 11 = 4
           h₂(37) = 37 mod 9 = 1
                      h (37, 0) = (4 + 0 x 1) mod 11 = 4 mod 11 = 4
T [4] is not free, the next probe sequence is
                   h (37, 1) = (4 + 1 x 1) mod 11 = 5 mod 11 = 5
T [5] is free, so insert key 37 at this place.
```

4. Insert 59.

$h_1(59) = 59 \bmod 11 = 4$

$h_2(59) = 59 \bmod 9 = 5$

$h\ (59, 0) = (4 + 0 \times 5) \bmod 11 = 4 \bmod 11 = 4$

Since, T [4] is not free, the next probe sequence is

$h\ (59, 1) = (4 + 1 \times 5) \bmod 11 = 9 \bmod 11 = 9$

T [9] is free, so insert key 59 at this place.

5. Insert 21.

$h_1(21) = 21 \bmod 11 = 10$

$h_2(21) = 21 \bmod 9 = 3$

$h\ (21, 0) = (10 + 0 \times 3) \bmod 11 = 10 \bmod 11 = 10$

T [10] is not free, the next probe sequence is

$h\ (21, 1) = (10 + 1 \times 3) \bmod 11 = 13 \bmod 11 = 2$

T [2] is free, so insert key 21 at this place.

6. Insert 65.

$h_1(65) = 65 \bmod 11 = 10$

$h_2(65) = 65 \bmod 9 = 2$

$h\ (65, 0) = (10 + 0 \times 2) \bmod 11 = 10 \bmod 11 = 10$

T [10] is not free, the next probe sequence is

$h\ (65, 1) = (10 + 1 \times 2) \bmod 11 = 12 \bmod 11 = 1$

T [1] is free, so insert key 65 at this place.

Thus, after insertion of all keys the final hash table is

0	1	2	3	4	5	6	7	8	9	10
/	65	21	/	26	37	/	/	/	59	76

Hash Function

Hash Function is used to index the original value or key and then used later each time the data associated with the value or key is to be retrieved. Thus, hashing is always a one-way operation. There is no need to "reverse engineer" the hash function by analyzing the hashed values.

Characteristics of Good Hash Function:

1. The hash value is fully determined by the data being hashed.
2. The hash Function uses all the input data.
3. The hash function "uniformly" distributes the data across the entire set of possible hash values.
4. The hash function generates complicated hash values for similar strings.

Some Popular Hash Function is:

1. Division Method:

Choose a number m smaller than the number of n of keys in k (The number m is usually chosen to be a prime number or a number without small divisors, since this frequently a minimum number of collisions).

The hash function is:

$$h(k) = k \bmod m \qquad h(k) = k \bmod m + 1$$

For Example: if the hash table has size m = 12 and the key is k = 100, then h (k) = 4. Since it requires only a single division operation, hashing by division is quite fast.

2. Multiplication Method:

The multiplication method for creating hash functions operates in two steps. First, we multiply the key k by a constant A in the range 0 < A < 1 and extract the fractional part of kA. Then, we increase this value by m and take the floor of the result.

The hash function is:

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

Where "k A mod 1" means the fractional part of k A, that is, k A -⌊k A⌋.

3. Mid Square Method:

The key k is squared. Then function H is defined by

1. H (k) = L

Where L is obtained by deleting digits from both ends of k^2. We emphasize that the same position of k^2 must be used for all of the keys.

4. Folding Method:

The key k is partitioned into a number of parts k_1, k_2.... k_n where each part except possibly the last, has the same number of digits as the required address.

Then the parts are added together, ignoring the last carry.

$$H\ (k)\ =\ k^1+\ k^2+.....+k^n$$

Example: Company has 68 employees, and each is assigned a unique four- digit employee number. Suppose L consist of 2-digit addresses: 00, 01, and 02....99. We apply the above hash functions to each of the following employee numbers:

1. 3205, 7148, 2345

(a) Division Method: Choose a Prime number m close to 99, such as m =97, Then

1. H (3205) = 4, H (7148) = 67, H (2345) = 17.

That is dividing 3205 by 17 gives a remainder of 4, dividing 7148 by 97 gives a remainder of 67, dividing 2345 by 97 gives a remainder of 17.

(b) Mid-Square Method:

k =	3205	7148	2345
k^2=	10272025	51093904	5499025
h (k) =	72	93	99

Observe that fourth & fifth digits, counting from right are chosen for hash address.

(c) Folding Method: Divide the key k into 2 parts and adding yields the following hash address:

1. H (3205) = 32 + 50 = 82 H (7148) = 71 + 84 = 55
2. H (2345) = 23 + 45 = 68