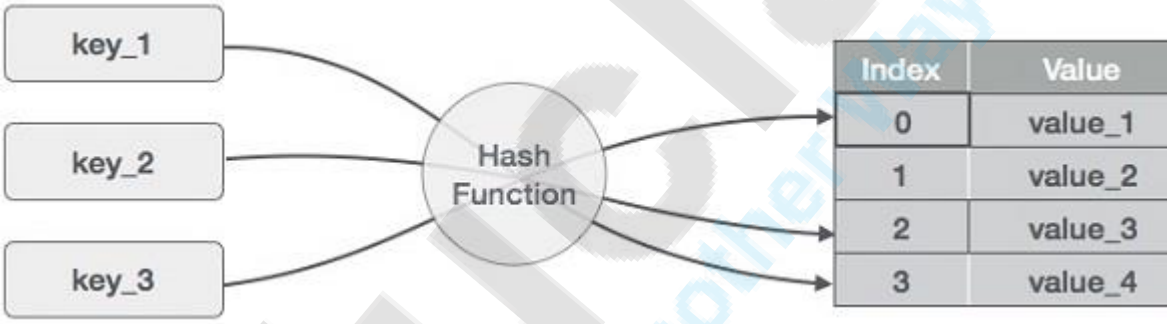
Q.1	Explain hashing
Ans.	Hash Table is a data structure which stores data in an associative manner. In a hash table, data is stored in an array format, where each data value has its own unique index value. Access of data becomes very fast if we know the index of the desired data. Thus, it becomes a data structure in which insertion and search operations are very fast irrespective of the size of the data. Hash Table uses an array as a storage medium and uses hash technique to generate an index where an element is to be inserted or is to be located from. **Hashing** Hashing is a technique to convert a range of key values into a range of indexes of an array. We're going to use modulo operator to get a range of key values. Consider an example of hash table of size 20, and the following items are to be stored. Item are in the (key,value) format.  • (1,20) • (2,70) • (42,80) • (4,25) • (12,44) • (14,32) • (17,11) • (13,78) • (37,98)

Sr. No.	Key	Hash	Array Index
1	1	1 % 20 = 1	1
2	2	2 % 20 = 2	2
3	42	42 % 20 = 2	2
4	4	4 % 20 = 4	4
5	12	12 % 20 = 12	12
6	14	14 % 20 = 14	14
7	17	17 % 20 = 17	17
8	13	13 % 20 = 13	13
9	37	37 % 20 = 17	17

Q2 Linear Probing

Ans. Linear Probing

As we can see, it may happen that the hashing technique is used to create an already used index of the array. In such a case, we can search the next empty location in the array by looking into the next cell until we find an empty cell. This technique is called linear probing.

Sr. No.	Key	Hash	Array Index	After Linear Probing, Array Index

1	1	1 % 20 = 1	1	1
2	2	2 % 20 = 2	2	2
3	42	42 % 20 = 2	2	3
4	4	4 % 20 = 4	4	4
5	12	12 % 20 = 12	12	12
6	14	14 % 20 = 14	14	14
7	17	17 % 20 = 17	17	17
8	13	13 % 20 = 13	13	13
9	37	37 % 20 = 17	17	18

Basic Operations

Following are the basic primary operations of a hash table.

- **Search** − Searches an element in a hash table.
- **Insert** − inserts an element in a hash table.
- **delete** − Deletes an element from a hash table.

DataItem

Define a data item having some data and key, based on which the search is to be conducted in a hash table.

Hash Method

Define a hashing method to compute the hash code of the key of the data item.

Search Operation

Whenever an element is to be searched, compute the hash code of the key passed and

	locate the element using that hash code as index in the array. Use linear probing to get the element ahead if the element is not found at the computed hash code. Insert Operation Whenever an element is to be inserted, compute the hash code of the key passed and locate the index using that hash code as an index in the array. Use linear probing for empty location, if an element is found at the computed hash code. Delete Operation Whenever an element is to be deleted, compute the hash code of the key passed and locate the index using that hash code as an index in the array. Use linear probing to get the element ahead if an element is not found at the computed hash code. When found, store a dummy item there to keep the performance of the hash table intact.
Q.3	Some hashing technique (functions)
Ans.	**Common hashing technique** There are around seven common methods used in hashing, or the seven ways to insert values into a key accessed table. These are listed in no particular order that is given below. • Division Method • Multiplication Method • Folding Method • Length-dependent Method • Midsquare Method • Digit-Analysis • Addition Method This report discusses each of these with emphasis on the division, multiplication and folding methods. Also one method is there called as random method generator. The random number produced can be transformed to produce a valid hash value. **The Division Method** A hash function must guarantee that the number it returns is a valid index to one of the table cells. The simplest way to accomplish this is division method. In this an integer key is divided by the table size and the remainder is taken as the hash value. It has been found that the best result with the division method is achieved when the table size is prime. Algorithm: H (x) = x mod m + 1 Where: m is some predetermined divisor integer (i.e., the table size), x is the preconditioned key and mod stands for modulo. Note that adding 1 is only necessary if the table starts at key 1 (if it starts at 0, the algorithm simplifies to (H (x) = x mod m). In the applet, we did not add 1. So, in other words: given an item, divide the preconditioned key of that item by the table size (+1). The remainder is the hash key. **Example:** Given a hash table with 10 buckets, what is the hash key for 'Cat'?

Since 'Cat' = 131130 when converted to ASCII, then x = 131130.We are given the table size (i.e., m = 10, starting at 0 and ending at 9).

H(x) = x mod m

H (131130) = 131130 mod 10

= 0 (there is no remainder)

'Cat' is inserted into the table at address 0.

The Division method is distribution-independent.

The Multiplication Method

A different hash method is multiplicative method. This method is used in the applet. It multiplies of all the every single digits in the key jointly, and takes the remainder after dividing the resulting number by the table size.

In practical notation, the algorithm is:

H(x) = (a * b * c * d *....) mod m

Where: m is the table size, a, b, c, d, etc. are the individual digits of the item, and mod stands for modulo. this can clearly understood this algorithm by applying it to an example.

Example:

Given a hash table of ten buckets (0 through 9), what is the hash key for 'Cat'?

Since 'Cat' = 131130 when converted to ASCII, then x = 131130

We are given the table size (i.e., m = 10).

The constant can be any number we want, let's use five (i.e., c = 5).

H (x) = (a * b * c * d *....) mod m

H (131130) = (1 * 3 * 1 * 1 * 3 * 0) mod 10

= 0 mod 10

= 0

'Cat' is inserted into the table at address 0.

Both of these Multiplication methods are distribution-independent.

The Folding Method

The folding method breaks up a key into precise segments that are added to form a hash value. And still another technique is to apply a multiplicative hash function to each segment individually before folding. Algorithm: H (x) = (a + b + c) mod m

Where: a, b, and c represent the preconditioned key broken down into three parts, „m‟ is the table size, and mod stands for modulo.

In other words: the sum of three parts of the preconditioned key is divided by the table size.

The remainder is the hash key.

Example:

Fold the key 123456789 into a hash table of ten spaces (0 through 9).

We are given x = 123456789 and the table size (i.e., m = 10).

Since we can break x into three parts any way we want to, we will break it up evenly.

Thus a = 123, b = 456, and c = 789.

H (x) = (a + b + c) mod m

H (123456789) = (123+456+789) mod 10

= 1368 mod 10

= 8

123456789 are inserted into the table at address 8.

The Folding method is distribution-independent.

Random-Number Generator

This is a scheme used for generating a pseudo-random numbers. Primarily, in digitized the state of a chaotic system is used to form a binary string. This binary string is

	subsequently hashed to construct a second binary string. This second binary string is now used to seed a pseudo-random number generator. The output of pseudorandom number generator is used in constructing a password or cryptographic key which is used in a security system. The algorithm must ensure that: • It always generates the same random value for a given key. • It is unlikely for two keys to yield the same random value. The random number produced can be transformed to produce a valid hash value. Pseudo random number generators mix up a state like hash functions, but they don't take any input. They have a state of their own, and they just keep churning it. They produce random-looking sequences, which can be used as fake data. **Other Hashing Methods** A few of the many hashing methods that are also used for hashing are given below- • Length-dependent Method • Midsquare Method • Digit-Analysis • Addition Method **Midsquare** In Midsquare method, the key is multiply x by itself (i.e., x2) and select a number of digits from the middle of the result. How many digits you select will depend on your table size and the size of your hash key. If the square is considered as the decimal number, the table size must be a power of 10, whereas if it is considered as the decimal number, the table size must be a power of 2.Unforunately the midsquare. Method does not yield uniform hash values and does not perform as well as the multiplicative or the division method. **For Example:** To map the key 3121 into a hash table of size 1000, we square it 31212 = 9740641 and extract 406 as the hash value. It can be more efficient with powers of 2 as hash table size. Works well if the keys do not contain a lot of leading or zeros. On-integer keys have to be pre-processed to obtain corresponding integer values. **Addition** The sum of the digits of the preconditioned key is divided by the table size. The remainder is the hash key. **Digit Analysis** Certain digits of the preconditioned key are selected in a certain predetermined and consistent pattern. There are many other hash functions each with its own advantages and disadvantages depending upon on the set of keys to be hashed. One consideration in choosing function is efficiency of calculation. It does not good to be able to find an object on the first try if that try takes longer than several trials in an alternative method. If keys are not integer they must be converted to the integer before applying one of foregoing hash function.
Q. 4	Collision resolution techniques: Linear probe, Quadratic probe
Ans	**Collision Resolution Technique** **Linear Probing** In linear probing, F is a linear function of i, typically F(i) = i. This amounts to trying cells sequentially (with wraparound) in search of an empty cell. Figure 2.11 shows the result of inserting keys {89, 18, 49, 58, 69} into a hash table using the same hash function as before and the collision resolution strategy, F(i) = i. The first collision occurs

	when 49 is inserted; it is put in spot 0 which is open. key 58 collides with 18, 89, and then 49 before an empty cell is found three away. The collision for 69 is handled in a similar manner. As long as the table is big enough, a free cell can always be found, but the time to do so can get quite large. Worse, even if the table is relatively empty, blocks of occupied cells start forming. This effect is known as **Primary Clustering.** We will assume a very large table and that each probe is independent of the previous probes. These assumptions are satisfied by a random collision resolution strategy and are reasonable unless λ is very close to 1. First, we derive the expected number of probes in an unsuccessful search. This is just the expected number of probes until we find an empty cell. Since the fraction of empty cells is 1 - λ, the number of cells we expect to probe is 1/(1 - λ). The number of probes for a successful search is equal to the number of probes required when the particular element was inserted. When an element is inserted, it is done as a result of an unsuccessful search. Thus we can use the cost of an unsuccessful search to compute the average cost of a successful search. The caveat is that λ changes from 0 to its current value, so that earlier insertions are cheaper and should bring the average down. For instance, in the table above, λ= 0.5, but the cost of accessing 18 is determined when 18 is inserted. At that point, λ = 0.2. Since 18, was inserted into a relatively empty table, accessing it should be easier than accessing a recently inserted element such as 69. We can estimate the average by using an integral to calculate the mean value of the insertion time, obtaining I(λ) = 1/ λ ∫ λ 1/ 1 – x .dx = 1 / λ ln 1 / 1 – λ These formulas are clearly better than the corresponding formulas for linear probing. Clustering is not only a theoretical problem but actually occurs in real implementations. Figure 2.12 compares the performance of linear probing (dashed curves) with what would be expected from more random collision resolution. Successful searches are indicated by an S, and unsuccessful searches and insertions are marked with U and I, respectively.

$$h(x) + i^2 = h(x) + j^2 \quad (mod\ H\_SIZE)$$

$$i^2 = j^2 \quad (mod\ H\_SIZE)$$

$$i^2 - j^2 = 0 \quad (mod\ H\_SIZE)$$

$$(i - j)(i + j) = 0 \quad (mod\ H\_SIZE)$$

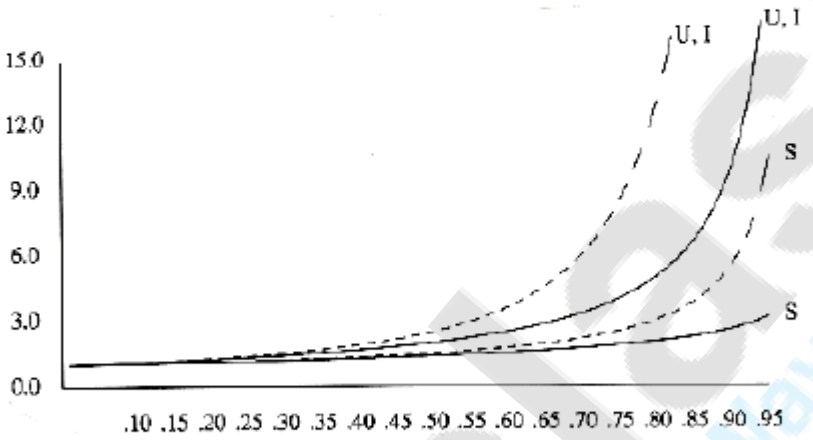


Figure 2.12 Number of probes plotted against load factor for linear probing (dashed) and random strategy. S is successful search,U is unsuccessful search, I is insertion

If $\lambda = 0.75$, then the formula above indicates that 8.5 probes are expected for an insertion in linear probing. If $\lambda = 0.9$, then 50 probes are expected, which is unreasonable. This compares with 4 and 10 probes for the respective load factors if clustering were not a problem. We see from these formulas that linear probing can be a bad idea if the table is expected to be more than half full. If $\lambda = 0.5$, however, only 2.5 probes are required on average for insertion and only 1.5 probes are required, on average, for a successful search.

Quadratic Probing

Quadratic probing is a collision resolution method that eliminates the primary clustering problem of linear probing. Quadratic probing is what you would expect-the collision function is quadratic. The popular choice is F(i) = i2. Figure 2.13 shows the resulting closed table with this collision function on the same input used in the linear probing example.

	Empty Table	After 89	After 18	After 49	After 58	After 69
0				49	49	49
1						
2					58	58
3						69
4						
5						
6						
7						
8			18	18	18	18
9		89	89	89	89	89

Figure 2.13 Closed hash table with quadratic probing, after each insertion

When 49 collides with 89, the next position attempted is one cell away. This cell is empty, so 49 is placed there. Next 58 collides at position 8. Then the cell one away is tried but another collision occurs. A vacant cell is found at the next cell tried, which is 22 = 4 away. 58 is thus placed in cell 2. The same thing happens for 69. For quadratic probing, the situation is even more drastic: There is no guarantee of finding an empty cell once the table gets more than half full, or even before the table gets half full if the table
size is not prime. This is because at most half of the table can be used as alternate locations to resolve collisions. Indeed, we prove now that if the table is half empty and the table size is prime, then we are always guaranteed to be able to insert a new element.

Q.5 Double Hashing

Ans.

Double Hashing

The last collision resolution method we will examine is double hashing. For double hashing, one popular choice is F(i) = i.h2(x). This formula says that we apply a second hash function to x and probe at a distance h2(x), 2h2(x), . . ., and so on. For instance, the obvious choice h2(x) = x mod 9 would not help if 99 were inserted into the input in the previous examples. Thus, the function must never evaluate to zero. It is also important to make sure all cells can be probed (this is not possible in the example below, because the table size is not prime). A function such as h2(x) = R - (x mod R), with R a prime smaller than H_SIZE, will work well. If we choose R = 7, then Figure 2.18 shows the results of inserting the same keys as before.

```
void insert( element_type key, HASH_TABLE H )
{
position pos;
```

```
pos = find( key, H );
if( H->the_cells[pos].info != legitimate )
{ /* ok to insert here */
H->the_cells[pos].info = legitimate;
H->the_cells[pos].element = key;
/* Probably need strcpy!! */
}
}
```

	Empty Table	After 89	After 18	After 49	After 58	After 69
0						69
1						
2						
3					58	58
4						
5						
6				49	49	49
7						
8			18	18	18	18
9		89	89	89	89	89

Figure 2.18 Closed hash table with double hashing, after each insertion

The first collision occurs when 49 is inserted. h2(49) = 7 – 0 = 7, so 49 is inserted in position 6. h2(58) = 7 - 2 = 5, so 58 is inserted at location 3. Finally, 69 collides and is inserted at a distance h2(69) = 7 - 6 = 1 away. If we tried to insert 60 in position 0, we would have a collision. Since h2(60) = 7 - 4 = 3, we would
then try positions 3, 6, 9, and then 2 until an empty spot is found. It is generally possible to find some bad case, but there are not too many here. As we have said before, the size of our sample hash table is not prime. We have done this for convenience in computing the hash function, but it is worth seeing why it is important to make sure the table size is prime when double hashing is used. If we attempt to insert 23 into the table, it would collide with 58. Since h2(23) = 7 - 2 = 5, and the table size is 10, we essentially have only one alternate location, and it is already taken. Thus, if the table size is not prime, it is possible to run out of alternate locations prematurely. However, if double hashing is correctly implemented, simulations imply that the expected number of probes is almost the same as for a random collision resolution strategy. This makes double hashing theoretically interesting. Quadratic probing, however, does not require the use of a second hash function and is thus likely to be simpler and faster in practice.

Q. 6	Rehashing
Ans.	**Rehashing** If the table gets too full, the running time for the operations will start taking too long and inserts might fail for closed hashing with quadratic resolution. This can happen if there are too many deletions intermixed with insertions. A solution for this is to build another table that is about twice as big and scan down the entire original hash table, computing the new hash value for each (non-deleted) element and inserting it in the new table. For example, suppose the elements 13, 15, 24, and 6 are inserted into a closed hash table of size 7. The hash function is h(x) = x mod 7. Suppose linear probing is used to resolve collisions. The resulting hash table appears in Figure 2.19. If 23 is inserted into the table, the resulting table in Figure 2.20 will be over 70 percent full. Because the table is so full, a new table is created. The size of this table is 17, because this is the first prime which is twice as large as the old table size. The new hash function is then h(x) = x mod 17. The old table is scanned, and elements 6, 15, 23, 24, and 13 are inserted into the new table. The resulting table appears in Figure 2.21. This entire operation is called rehashing. This is obviously a very expensive operation -- the running time is O(n), since there are n elements to rehash and the table size is roughly 2n, but it is actually not all that bad, because it happens very infrequently. In particular, there must have been n/2 inserts prior to the last rehash, so it essentially adds a constant cost to each insertion.

0	6
1	15
2	
3	24
4	
5	
6	13

Figure 2.19 Closed hash table with linear probing with input 13,15, 6, 24

0	6
1	15
2	23
3	24
4	
5	
6	13

Figure 2.20 Closed hash table with linear probing after 23 is inserted

0	
1	
2	
3	
4	
5	
6	6
7	23
8	24
9	
10	
11	
12	
13	13
14	
15	15
16	

Figure 2.21 Closed hash table after rehashing

Rehashing can be implemented in several ways with Quadratic probing. One alternative is to rehash as soon as the table is half full. The other extreme is to rehash only when an insertion fails. A third, middle of the road, strategy is to rehash when the table reaches a certain load factor. Rehashing frees the programmer from worrying about the table size and is important because hash tables cannot be made arbitrarily large in complex programs.

```
HASH_TABLE
rehash( HASH_TABLE H )
{
unsigned int i, old_size;
cell *old_cells;
 old_cells = H->the_cells;
old_size = H->table_size;
/* Get a new, empty table */
H = initialize_table( 2*old_size );
/* Scan through old table, reinserting into new */
for( i=0; i<old_size; i++ )
if( old_cells[i].info == legitimate )
insert( old_cells[i].element, H );
free( old_cells );
return H;
}
```

Figure 2.22 Shows that rehashing is simple to implement.