Unit 1

Introduction

* **Data** → Anything which gives information. Data is defined as facts or info. that is stored or used by a computer.

* **Data Types** → Types ① Composite ② atomic

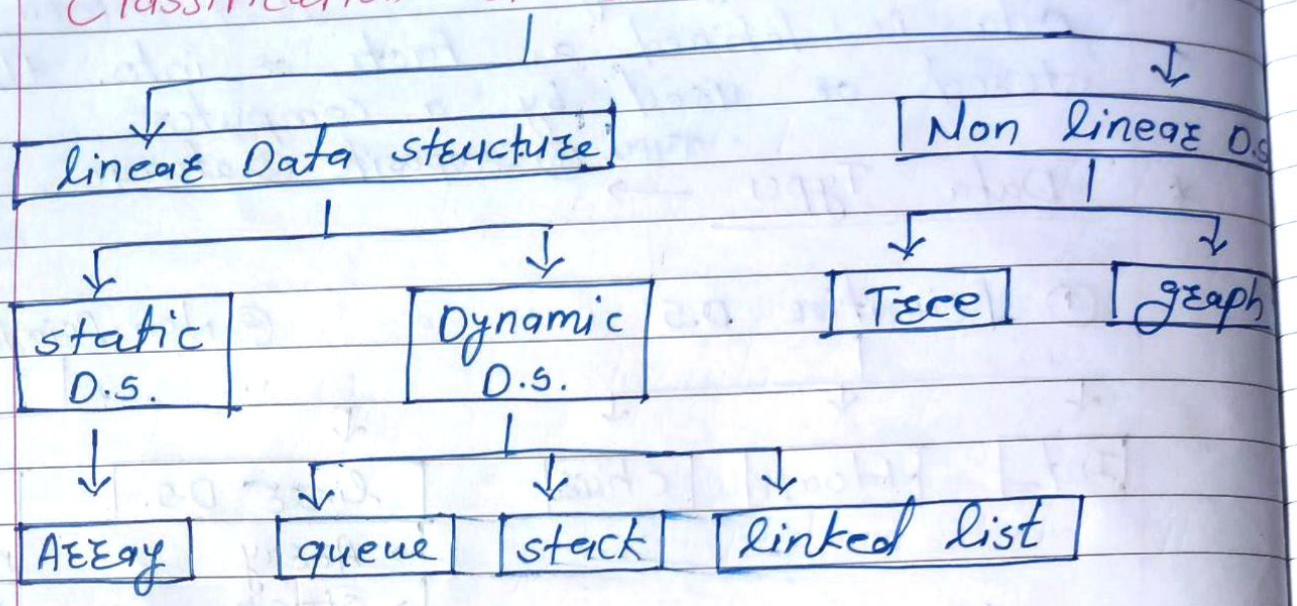
① Primitive D.S.
- Int
- Float
- char

② Non Primitive D.S
- linear D.S.
  - ↳ Array
  - ↳ stack
  - ↳ queue
  - ↳ Linked list
- Non linear DS
  - ↳ tree
  - ↳ graph

* **Data Structure** →

- A data structure is a specialized format for organizing, processing, retrieving and storing data.
- Data structures make it easy for users to access and work with the data they need in appropriate ways.
- Data structure provides efficiency, reusability and abstraction.

* Classification of Data structure →



① Linear Data structure →

Data structure in which data elements are arranged sequentially or linearly, where each element is attached to its previous and next adjacent elements is called a linear data structure.

ex :- array, stack, queue, linked list

* Static Data structure →

static Data structure has a fixed memory size. It is easier to access the elements in a static data structure

ex : Array

* Dynamic Data structure →

In dynamic D.S., the size is not fixed. It can be randomly updated

during the runtime which may be considered efficient concerning the memory (space) complexity of the code.
ex: queue, stack

② Non linear Data structure →

- Data structure where data elements are not placed sequentially are called non-linear D.S.
- In Non linear D.S. we can't traverse all the elements in a single run only.
- This structure is mostly used for representing data that contains a hierarchical relationship among various elements.
  ex: Graphs, Family of tree and table of contents.

* Abstract Data Types (ADT) →

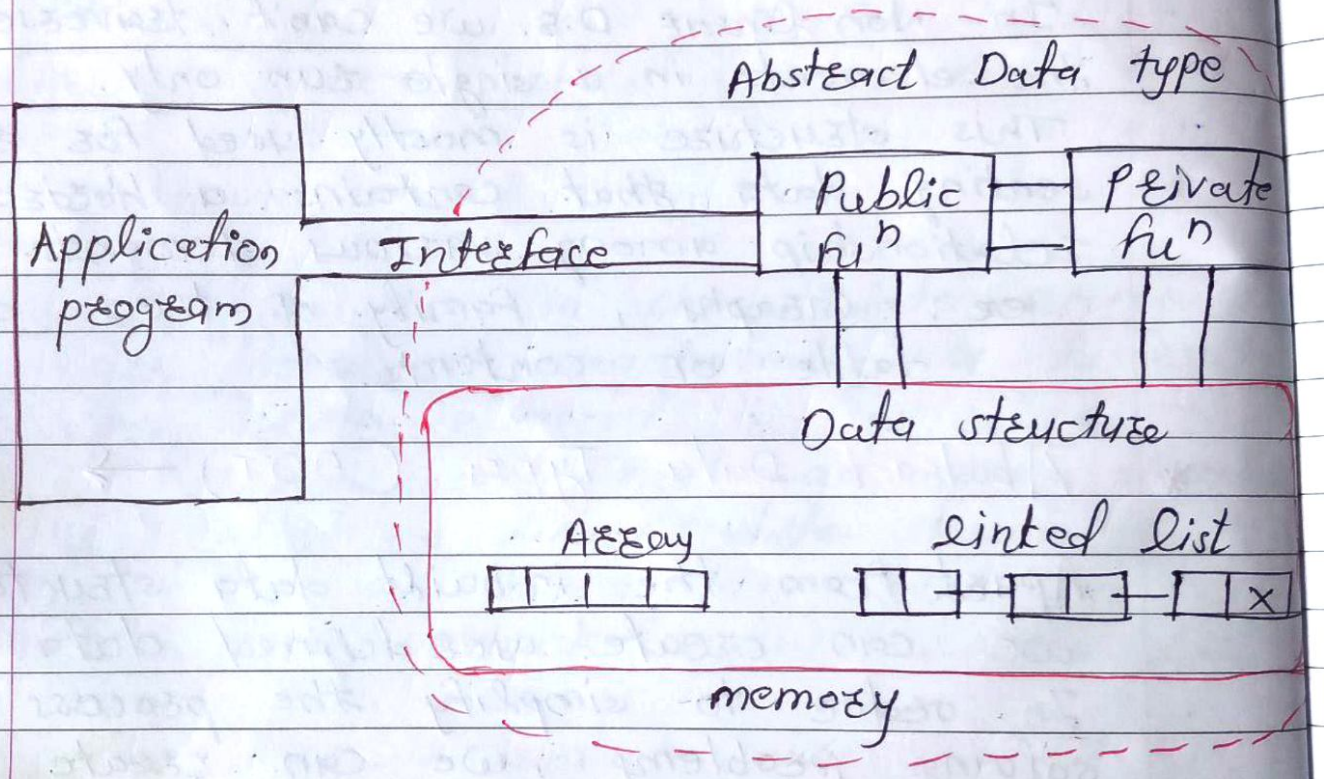
- Apart from the in built data structures, we can create user defined data types.
- In order to simplify the process of solving problems, we can create data structures along with their operations.
  and such data structures that are not in-built are known as abstract data type (ADT).

- ADT is a type or class for objects whose behaviour is defined by a set of values and a set of operations.
- It does not specify how data will be

organized in memory and what algorithms will be used for implementing the operations.

- It is called "abstract" because it gives an implementation - independent view.

- The process of providing only the essentials and hiding the details is known as abstraction.



- Think of ADT as a black box which hides the inner structure and design the data type.

- There are 3 ADTs Namely
① List ADT ② stack ADT
③ queue ADT.

* List ADT →

- The data is generally stored in key sequence in a list which has a head structure consisting of count, pointers and address of compare function needed to compare the data in the list.
- The data node contains the pointer to a d.s. and a self-referential pointer which points to the next node in the list.
- The List ADT functions is given below:
- get () → Return an element from the list at any given position.
- insert () → Insert an element at any position of the list.
- remove () → Remove the first occurrence of any element from a non-empty list.
- removeAt () → Remove the element at a specified location.
- replace () → Replace an element at any position by another element.
- size () → Return the number of element in the list.
- isEmpty () → Return true if the list is empty, otherwise return false.
- isFull () → Return true, if the list is full, otherwise return false.

* Stack ADT →

- In stack ADT Implementation instead of data being stored in each node, the pointer to data is stored.
- The program allocates memory for the data and address is passed to the stack ADT.
- The head node and the data nodes are encapsulated in the ADT. The calling fun can only see the pointer to the stack.
- The stack head structure also contains a pointer to top and count no. of entries currently in stack.

* push () → Insert an element at one end of the stack called top.

* pop () → Remove & return the element at the top of the stack if it is not empty.

* peek () → Return the element at the top of the stack without removing it, if the stack is not empty.

* size () → Return the no. of elements in the stack.

* isEmpty () → Return true if the stack is empty otherwise return false.

* isFull () → Return true if the stack is full, otherwise return false.

* Queque ADT →

- The queue abstract data type follows the basic design of the stack abstract DT.
- Each node contains a void pointer to the data and the link pointer to the next element in the queue. The program responsibility is to allocate memory for storing the data.

* enqueue () → Insert an element at the end of the queue.

* dequeue () → Remove and return the first element of the queue, if the queue is not empty.

* peek () → Return the element of the queue without removing it, if the queue is not empty.

* size () → Return the no. of elements in the queue.

* isEmpty () → Return true, if the queue is empty otherwise return false.

* isFull () → Return true, if the queue is full, otherwise return false.

* Features of ADT →

① Abstraction : The user doesnot need to know the implementation of the D.S.

② Better Conceptualization : ADT gives us better understanding of the real world

③ Robust : The program is robust & has the ability to catch errors.

* Representation of Information (Data) →

- Computer doesn't understand the language of human. any data viz - letters, symbols, pictures, audio, videos etc.
  This is to be converted to machine language first.
  Computer represents data in following 3 ways

① Number System →

- In computer, everything is a number i.e alphabets, pictures, sounds etc are numbers.
- No. system is categorized into 4 types:

① Binary no. system consists of only two values i.e. 0 & 1.

② Octal No. system represents values in 8 digit

③ Decimal No. system represents values in 10 digit.

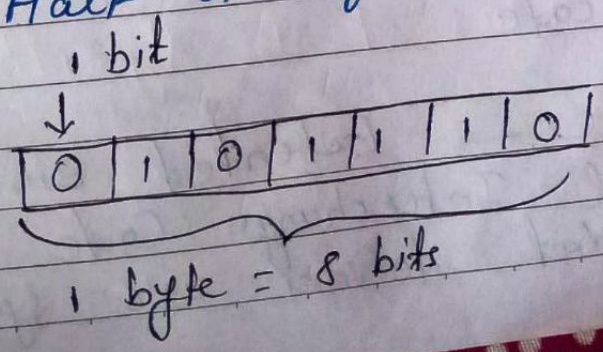
④ Hexadecimal no. system represents values in 16 digits.

No. system	Base	digits.
Binary	2	0 1
octal	8	0 1 2 3 4 5 6 7
Decimal	10	0 1 2 3 4 5 6 7 8 9
Hexadecimal	16	0 1 2 3 4 5 6 7 8 9 A B C D E F

② Bits and Bytes →

- Bits → A bit is a smallest possible unit of data that a computer can recognize or use. Computer usually uses bits in groups.

- Bytes → group of eight bits is called byte. Half of byte is called a nibble.



* Conversion of Bits and Bytes :

Byte value	Bit value
1 Byte	8 bits
1024 byte	1 KB
1024 KB	1 MB
1024 MB	1 GB
1024 GB	1 TB
1024 TB	1 petabyte [PB]
1024 PB	1 exabyte [EB]
1024 EB	1 zettabyte [ZB]
1024 ZB	1 yottabyte [YB]
1024 YB	1 Brontobyte [BB]
1024 BB	1 Geopbytes.

③ Text Code →

Text code is format used commonly to represent alphabets, punctuation mark and other symbols.
4 most popular text code s/m →

① EBCDIC
② ASCII
③ Extendend ASCII
④ Unicode

① EBCDIC → Extended Binary Coded Decimal Interchange Code is an 8 bit code that defines 256 symbols.

② ASCII → American Standard Code for Information Interchange is an 8 bit code that specifies character values from 0 to 127.

ASCII Code	Decimal value	Character
00000000	0	Null prompt
00000001	1	start of heading
00000010	2	start of text
00000011	3	End of text
00000100	4	End of transmit
00000101	5	Enquiry
00000110	6	Acknowledge
00000111	7	Audible bell
00001000	8	Backspace
00001001	9	Horizontal tab
00001010	10	Line feed

③ Extended ASCII → Extended American standard Code for Information Interchange is an 8 bit code that specifies character values from 128 to 255.

④ Unicode → Unicode Worldwide Character standard uses 4 to 32 bits to represent letters, numbers & symbols.

* Algorithm :→

- Algorithm is a step by step procedure, which defines a set of instructions to be executed in a certain order to get the desired output.
- Algorithms are generally created independent of underlying languages i.e. an algorithm can be implemented in more than one programming language.
- In data structure some imp categories of algorithms are :-

① Search → Algo. to search an item in a data structure.

② Sort → Algo. to sort items in certain order

③ Insert → Algo. to insert item in D.S.

④ Update → Algo. to update an existing item in a data structure

⑤ Delete → Algo. to delete an existing item from D.S.

* Characteristics of an Algorithm :→

① Unambiguous :- Algorithm should be clear and unambiguous. Each of its steps and their inputs/outputs should be clear

and must lead to only one meaning

② Input :- An algo. should have 0 or more well defined inputs.

③ Output :- An algorithm should have 1 or more well defined outputs. and should match the desired output.

④ Finiteness :- Algo. must terminate after a finite number of steps.

⑤ Feasibility :- Should be feasible with the available resources.

⑥ Independent :- An algorithm should have step-by-step directions, which should be independent of any programming code.

* Algorithm Complexity :-

Space Complexity :-
Space complexity of an algo. represents the amount of memory space required by the algo. in its life cycle.

Time Complexity :-
Time complexity of an algo. represents the amount of time required by the algo. to run to completion.
Time requirements can be defined as a numerical function T(n), where T(n) can be

measured as the no. of steps, provided each step consumes constant time.

* Asymptotic Analysis :-

Asymptotic analysis of an algo. refers to defining the mathematical boundation of its run-time performance.

According to that we can define the cases of algo. i.e. best case, worst case & avg. case.

Best case :- Minimum time required for program execution.

Average case :- Average case time require for the program execution.

Worst case :- Maximum time required for program execution.

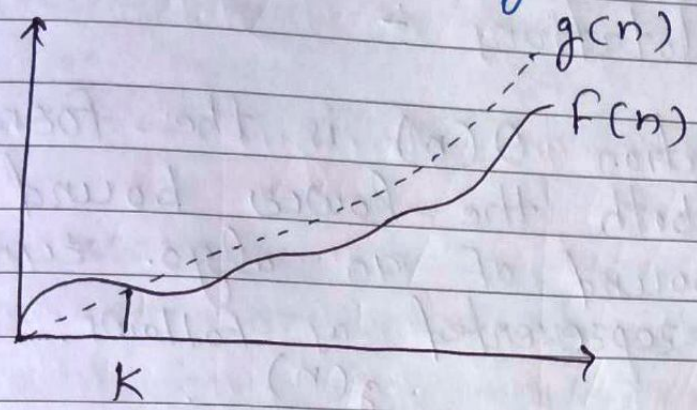
* Following are the commonly used asymptotic notations to calculate the running time complexity of an algo.

O Notation, Ω notation, Θ notation

* Big O Notation :-

- The notation O(n) is the formal way to express the upper bound of an algorithm running time.

It measures the worst case time complexity or the longest amount of time an algo. can possibly take to complete.

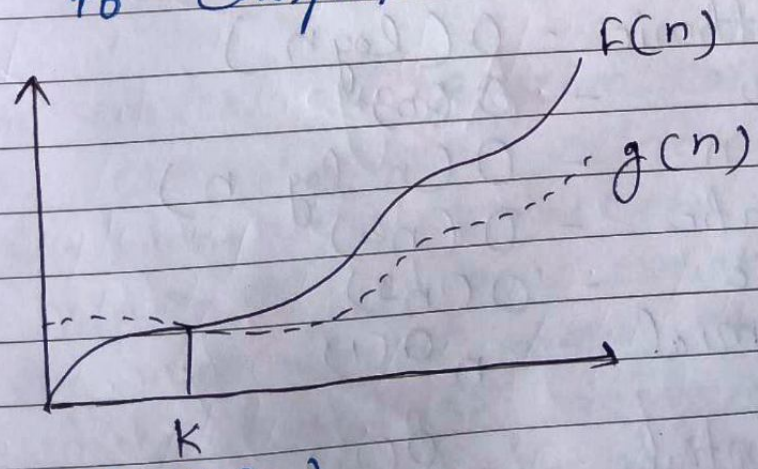


For a Fu^n f(n)

$O(f(n)) = \{g(n)$: there exists $c > 0$ and n_0 such that $f(n) \leq c \cdot g(n)$ for all $n > n_0$. $\}$

* Omega Notations :-

The notation $\Omega(n)$ is the formal way to express the lower bound of an algo. running time. It measures the best case time complexity or the best amount of time an algo. can possibly take to complete.



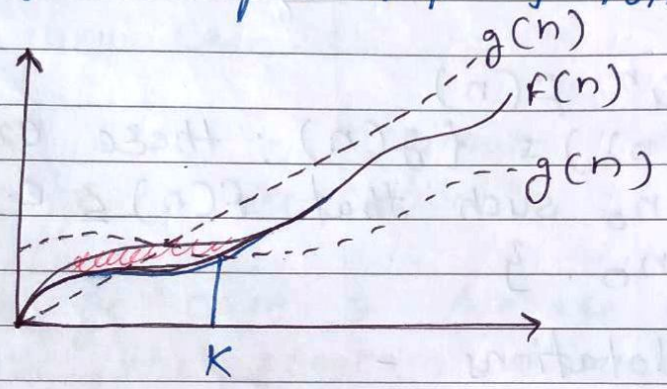
For a Fu^n f(n)

$\Omega(f(n)) \geq \{g(n)$: there exists $c > 0$

and n_0 such that $g(n) \leq c \cdot f(n)$ for all $n \geq n_0$. }

* Theta Notations :-

The notation $\Theta(n)$ is the formal way to express both the lower bound & the upper bound of an algo. running time. It is represented as follows:



$\Theta(f(n)) = \{ g(n)$ if & only if $g(n) = O(f(n))$ and $g(n) = \Omega(f(n))$ for all $n > n_0 \}$

* Common Used Asymptotic Notations :-

Constant - $O(1)$
logarithmic - $O(\log n)$
linear - $O(n)$
n log n - $O(n \log n)$
quadratic - $O(n^2)$
cubic - $O(n^3)$
polynomial - $n^{O(1)}$
exponential - $2^{O(n)}$

* Program :-

A program is a set of instructions that a computer follows in order to perform a particular task.

* Program Analyzing :-

- In computer science, program analysis is the process of automatically analyzing the behavior of computer programs regarding a property such as correctness, robustness, safety and liveness.

- Program analysis focuses on two major areas: Program optimization & program correctness.

- Program analysis can be performed without executing the program i.e static program analysis. And during runtime it will be the dynamic program analysis.

* Static Program Analysis :-

- In the context of program correctness, static analysis can discover vulnerabilities during the development phase of the program. These vulnerabilities are easier to correct than the ones found during the testing phase since the static analysis leads to the root of the vulnerability.

Static program analysis includes ① Control Flow analysis ② Data Flow analysis ③ Abstract interpretation ④ Type system ⑤ Effect system ⑥ model checking.

* Dynamic Program Analysis :-

- Dynamic analysis can use runtime knowledge of the program. to increase the precision of the analysis. While also providing runtime protection. But it can only analyze a single execution of the problem & might degrade the programs performance due to runtime check.

- Dynamic Program Analysis includes Testing, Monitoring, Program slicing.

- Dynamic program analysis is the analysis of computer software that is performed by executing programs on a real or virtual processor.

* Array

- Array is a container which can hold a fix number of items & these items should be of the same data type.

In array 2 imp terms are :-

① Element → Each item stored in an array is called an element.

② Index :→ Each index location of an element in an array has a numerical index. which is used to identify the element.

* Array – Advantage & Disadvantage

Advantages →

① Array allow random access to elements. This makes accessing elements by position faster.

② Arrays have better cache locality. which makes a pretty big diff. in performance.

③ Array represent multiple data items of the same type using a single name.

Disadvantages →

① Once you declared the array you can't change its size becoz of static memory allocation.

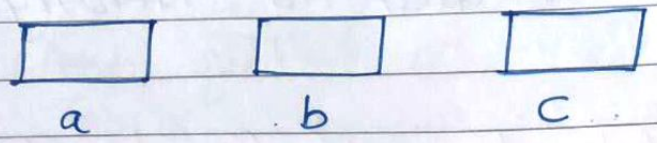
② Insertion & deletion are difficult as the elements are stored in consecutive memory locations & the shifting operation is costly too.

① Array [One Dimensional] →

- When we declare variables as:

```
int a, b, c;
```

- Compiler reserve 3 memory locations to store 3 int. numbers.



memory allocation for variables

- In case, we want to store more numb. say 1000 we have those many numbers variables declared.
But this is going to be inconvenient for the programmer.

* Defining an Array →

Array is declared as

data-type array-name [size] ;

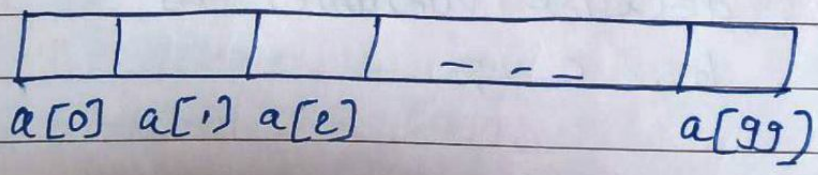
ex

int a [100] ;

- It means we declared 100 variables of integer type.

- This variables have the same name. a.

- The above declaration reserve 100 locations in the memory.



These location a[0], a[1] --- [99] are element

of array.

* Reading into Array →

To read elements into a array we can use for loop as,

```
for (i=0 ; i<n ; i++)
    scanf("%d", &a[i]);
```

where n is the number of elements in the array, n can be any no. less than size of array.

where i=0, first element entered from keyword gets stored in the location a[0], when i=1 second element is stored in location a[1] & so on.

If array no. is float we can use x[100] -

```
for (i=0 ; i<n ; i++)
    scanf("%f", &x[i])
```

* Printing of Array →

The element in the array can be displayed using for loop.

```
for (i=0 ; i<n ; i++)
    printf("%d" a[i])
```

* Processing of array →

Array element can be processed the same way we process other variables. i.e just like we write c = a+b we can write

a[2] = a[0] + a[1];

Numbers stored at location a[0] & a[1] in the array are added & the sum is stored in location a[2].
Similarly we can have -

sum = a[0] + a[1];
avg = (a[0]+a[1])/2;
a[10] = 20;
a[5] = 4;
a[9] = a[8];

It means we can take any element in an array randomly.

* ex of Array →

C++ program to store & disay Array

```
# include < iostream.h >
using namespace std;
int main ()
{ int num [5] ;
cout << " Enter array element : \n " ;
```

```
for ( int i=0 ; i<5 ; i++)
  {
      cin >> num [i] ;
  }
  }
```

* To print single element of array.

```
int arr [5] = { 1 , 3 , 5 , 7 , 9 }

cout << arr [3] ;
```

o/p → array will print the value 7 ; i.e. index arr[3] .

* Initializing an Array :-

int x = 10 , array elements can also be initialized as

```
int a [] = { 10 , 20 , 30 , 40 } ;
```

It will initialize the elements of array as a[0] = 10 , a[1] = 20 , a[2] = 30 , a[3] = 40 ;

② Two dimensional Array :-

- Sometimes we have a group of numbers represented in terms of rows & column.

	item 1	item 2
S1	10	40
S2	20	30
S3	30	50

- In math this table of values is represented in terms of matrix.

- Each element is represented as Arr [row] [column], where Arr [ ] [ ] in the 2D array.

* Representation of Array →

	col 1	col 2	col 3
Row 1	a[0][0]	a[0][1]	a[0][2]
Row 2	a[1][0]	a[1][1]	a[1][2]
Row 3	a[2][0]	a[2][1]	a[2][2]

* Initializing a 2D array in C++

```
int arr[4][2] = {
        {1234, 56},
        {1212, 33},
```

```
{1434 , 80 },
{ 1312 , 78 }.
};
```

* Printing 2 D Array

```
# include < iostream>        # include <conio.h>
using namespace std ;
main ()
{
    int arr [4] [2] = {
            { 10, 11 },
            { 20, 21 },
            { 30, 31 } ,
            { 40, 41 } } ;

    int i , j ;

  cout <<" Printing a 2 D Array : \n " ;
  For ( i = 0 ; i < 4 ; i++)
  {
      For ( j = 0 ; j < 2 ; j++)

    {
      cout <<"\t " << arr [ i ] [ j ] ;
    }
      cout <<endl ;
  } getch () ;
}
```

O/P →	10	11
	20	21
	30	31
	40	41

* Taking 2D Array as a user Input

```
# include < iostream. h>
# include < conio. h >
  main ()
{
    int s [2] [2] ;
    int i , j ;

  cout <<" \n 2D Array Input : \n " ;
      For (i = 0 ; i < 2 ; i++)
    {
        For (j = 0 ; j < 2 ; j++)
    {
      cout <<" \n s [" << i << " ] [" << j << " ] = " ;
      cin >> s [i] [j] ;
    }
    }

  cout <<" \n the 2-D array is : \n " ;
  For (i = 0 ; i < 2 ; i++)
    {
      For (j = 0 ; j < 2 ; j++)
    {
      cout <<" \t " << s [i] [j] ;
    }
      cout << endl ;
    }
    getch () ;
}
```

o/p → 2 D Array Input :

S[0][0] = 1
S[0][1] = 2
S[1][0] = 3
S[1][1] = 4

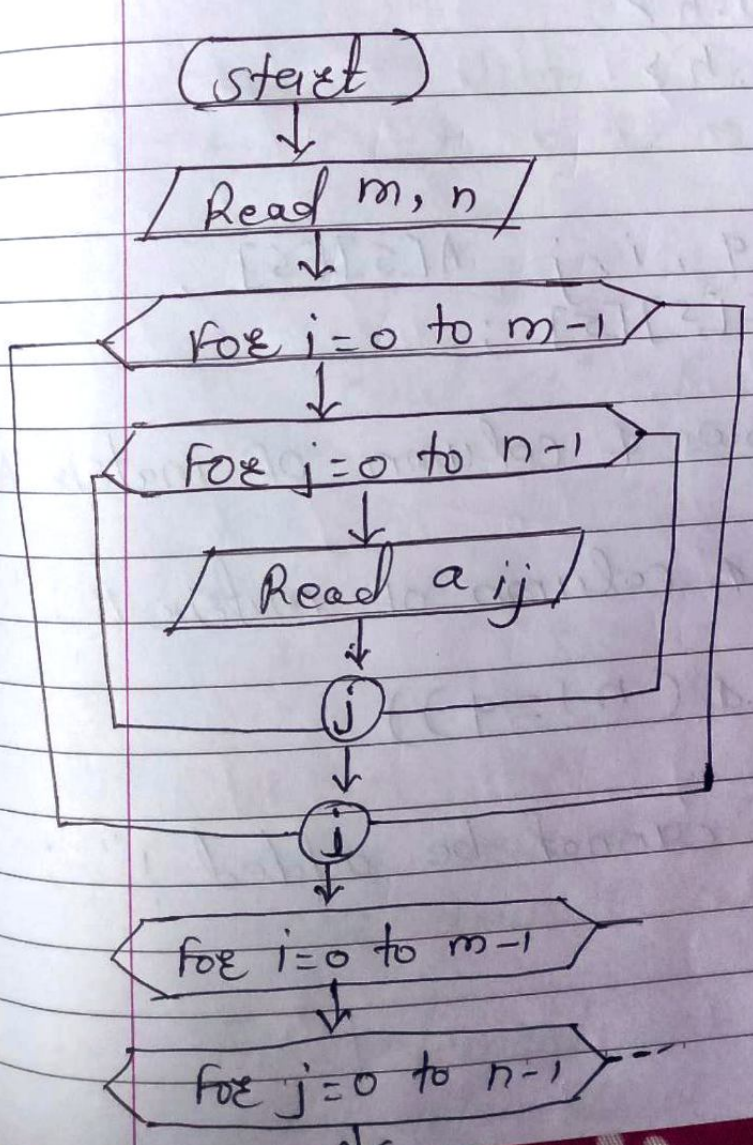
The 2 D Array is :

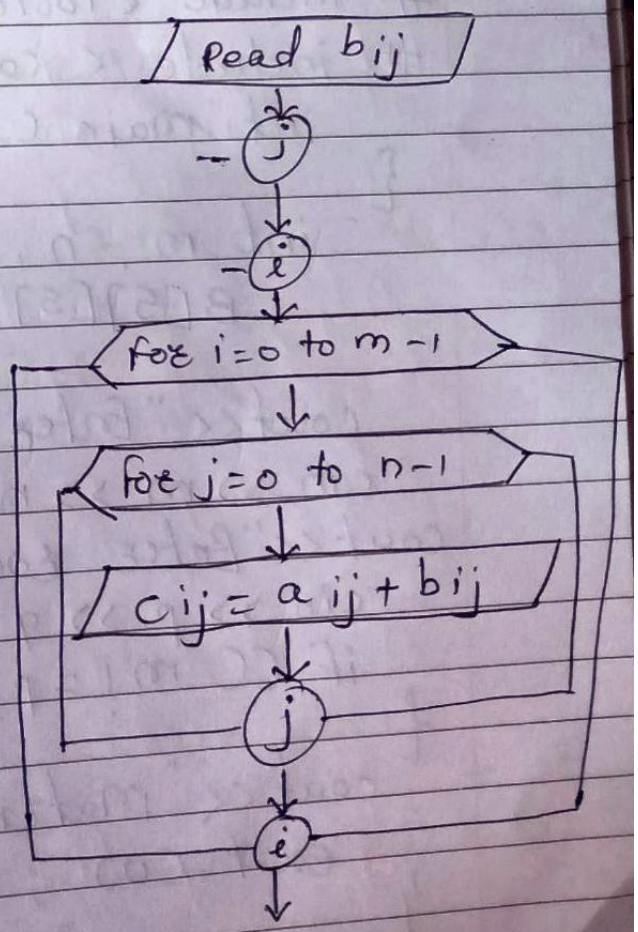
1	2
3	4

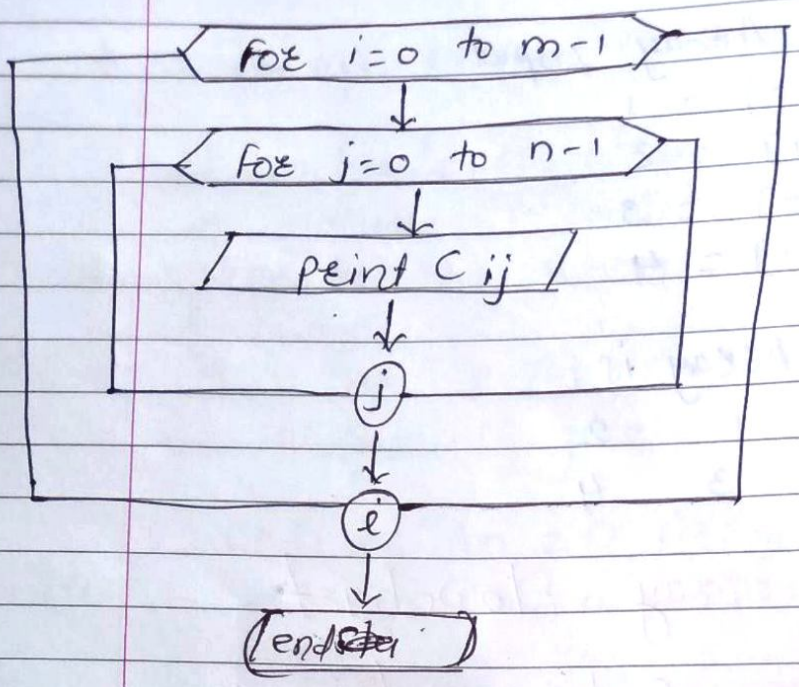
* Ex. For 2D array Flowchart

To find sum of two matrix :

c[i][j] = a[i][j] + b[i][j]







* Program for addition of two matrix.

```
# include <iostream.h>
# include < conio.h>
    int main ()
 {
    int m, n, p, q, i, j, A[5][5],
      B[5][5], C[5][5];

   cout<< "Enter rows & column of matrix A:"
   cin >> m >> n ;
cout<<"Enter rows & column of matrix B:";
   cin >> p >> q ;
   if ((m!=p) && (n!=q))
 {
  cout<< "Matrices cannot be added!";
    exit (0);
 }
```

```
cout << "Enter elements of matrix A : ";
    for ( i=0 ; i<m ; i++)
        for (j=0 ; j<m ; j++)
            cin >> A[i][j];

cout << "Enter elements of matrix B : ";
    for (i=0 ; i<p ; i++)
        for (j=0 ; j<q ; j++)

            cin >> B[i][j] ;

for ( i=0 ; i<m ; i++)
        for (j=0 ; j<n ; j++)

    C [i][j] = A[i][j] + B[i][j] ;

cout << "Sum of matrices\n" ;
for (i=0 ; i<m ; i++)
  {
        for (j=0 ; j<n ; j++)

            cout << C[i][j] << " " ;
            cout << "\n" ;
  }
        return 0 ;
}
```

O/P →

```
Enter rows & column for matrix A : 2 2
Enter rows & column for matrix B : 2 2

Enter elements of matrix A : 1 2 3 4
Enter elements of matrix B : 4 3 2 1
```

Sum of matrices $\begin{bmatrix} 5 & 5 \\ 5 & 5 \end{bmatrix}$

* Storage Representation :-

two most commonly used representations
① Adjacency matrix
② Adjacency List

① Adjacency Matrix →

- It is a 2D Array of size V × V where V is the number of vertices in a graph.
- let 2D array be adj [ ][ ] a slot adj[i][j] = 1 indicates that there is an edge from vertex i to vertex j.
- Adjacency Matrix is for undirected graph is always symmetric.
- Adjacency Matrix is also used to represent weighted graph.
- If adj [i] [j] = w then there is an edge from vertex i to j with weight w.

② Adjancency List →

An array of linked list is used.

- Size of the array is equal to the number of vertices.

- An entry array[i] represents the linked list of vertices adjacent to the ith vertex.

- This representation is used to represent weighted graph.

- The weight of edge can be stored in nodes of linked list.

* Representation of Sparse Matrix using array

Sparse Matrix →

- Sparse matrix are those matrices that have the majority of their elements equal to zero.

- Sparse matrix contains lesser non zero elements than zero, so less memory can be used to store elements.
  It evaluates only the non zero elements.

- In Computing Time → In case of searching in sparse matrix we need to traverse only the non zero elements rather than traversing all the sparse matrix elements.

- It saves computing time by logically designing a data structure traversing non zero elements.

* Representation →

Two types of representing sparse matrix
① Array representation
② Linked List representation.

① Array Representation →

In 2D array sparse matrix, there are three fields that are -

Row	Col	Value

Row → It is the index of a row where a non zero element is located in the matrix.

Col → It is the index of the column where a non zero element is located in the matrix.

Value → It is the value of the non zero element that is located at the index (row, column).

ex:

	0	1	2	3
0	0	4	0	5
1	0	0	3	6
2	0	0	2	0
3	2	0	0	0
4	1	0	0	0

← sparse matrix

In the above fig. we can observe 5x4 sparse matrix containing 7 non zero elements & 13 zero elements.

Above matrix occupies 5x4=20 memory space.

Tabular representation of above matrix :-

Row	Colum	Value
0	1	4
0	3	5
1	2	3
1	3	6
2	2	2
3	0	2
4	0	1

* Implemetation of sparse matrix →

```
# include < ~~stdiostr~~ iostream.h >
# include < conio.h >

int main ()
{
  int a [10] [10] = { {0, 0, 9} ,
```

```
{5, 0, 8} , {7, 0, 0} } ;

int i, j, count = 0 ;

int row=3 , col = 3 ;
for ( i=0 ; i< row ; ++i)
  {
    for (j=0 ; j< col ; ++j)
  {
      if (a [i] [j] ==0)
        count ++ ;
  }
  }

  cout << "The matrix is : " << endl ;
  for ( i=0 ; i < row ; ++i)
  {
    for ( j=0 ; j< col ; ++j)
  {
    cout << a [i] [j] << "  " ;
  }
    cout << endl ;
  }

cout << "The number of zeros in the matrix are " << count << endl ;

if ( count > ((row * col) /2))

cout << " This is a sparse matrix "<< endl ;
else
```

```
cout << " This is not sparse matrix " << endl;
    return 0;
}
```

O/P → The matrix is : 0 0 9
5 0 8
7 0 0

The number of zeros in the matrix are 5.
This is sparse matrix.

* Transpose of Sparse Matrix →

- Transpose of matrix is obtained by interchanging rows & columns.
- In another way we can say that element in the i, j position gets put in the j, i position.

ex:

Simple transpose of matrix :-

```
# include <stdio.h>
# include <conio.h>
    void main ()
    {
        int a[10][10], b[10][3], r, c, s=0,
        i, j;
        clrscr();
```

```
printf("\n enter the order of the sparse
        matrix ");
scanf("%d %d" &r, &c);

Printf("\n enter the elements in the sparse
    matrix (mostly zeros) ");

for (i=0 ; i<r ; i++)
{

    for (j=0 ; j<c ; j++)
{

  Printf("\n %d row and %d column",
            i,j);
  scanf("%d", &a[i][j]);
  if(a[i][j]!=0)
    {

      b[s][0] = a[i][j] ;
      b[s][1] = i ;
      b[s][2] = j ;
      s++ ;
  }
  }
}

    printf("\n the sparse matrix is
            given by ");

  printf("\n");
    for( i=0 ; i<s ; i++)
  {
```

```
for (j=0 ; j<3 ; j++)
{
  printf(" %d ", b[i][j]);
}
  printf("\n");
}
  getch();
}
```

* Time complexity of simple transpose : O (nc)
  n = No. of non zero elements
  c = No. of columns.

* Hash Tables →

- Hash tables are the data structure which favor efficient storage & retrieval of data elements which are linear in nature

- Access of data becomes very fast if we know the index of the desired data.

- Using Hash tables, insertion & deletion becomes very fast irrespective of the size of the data.

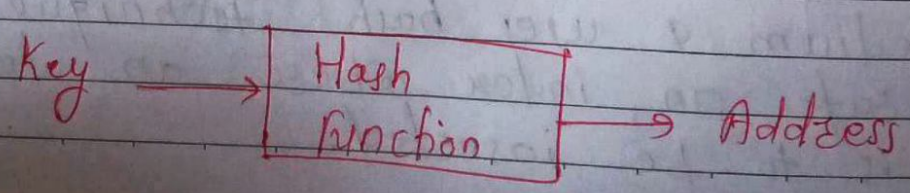
- Hash tables uses an array as a storage medium & uses hash technique to generate an index where an element is to be inserted.

* Dictionaries →

- It is a collection of data elements uniquely identified by a field called key.
- A dictionary supports operations of search, insert, & delete.
- A dictionary supports both sequential & Random access.
- A sequential access is the process in which the data elements of the dictionary are ordered & accessed according to the order of keys. [ascending & descending]
- Hash tables are ideal data structures for dictionaries.

* Hash Search →

- Hash search is a search in which the key, through an algorithmic fun, determines the location of data.
- Hashing is a key to address transformation in which the key map to address in a list.



* Hash Function :-

- A hash function is a mathematical function which maps a given key of the dictionary to its corresponding location in the storage table. [Known as hash table]
- The process of mapping the keys to their respective position in the hash table is called as hashing.
- The choice of the hash function plays a significant role in the performance of the hash table.

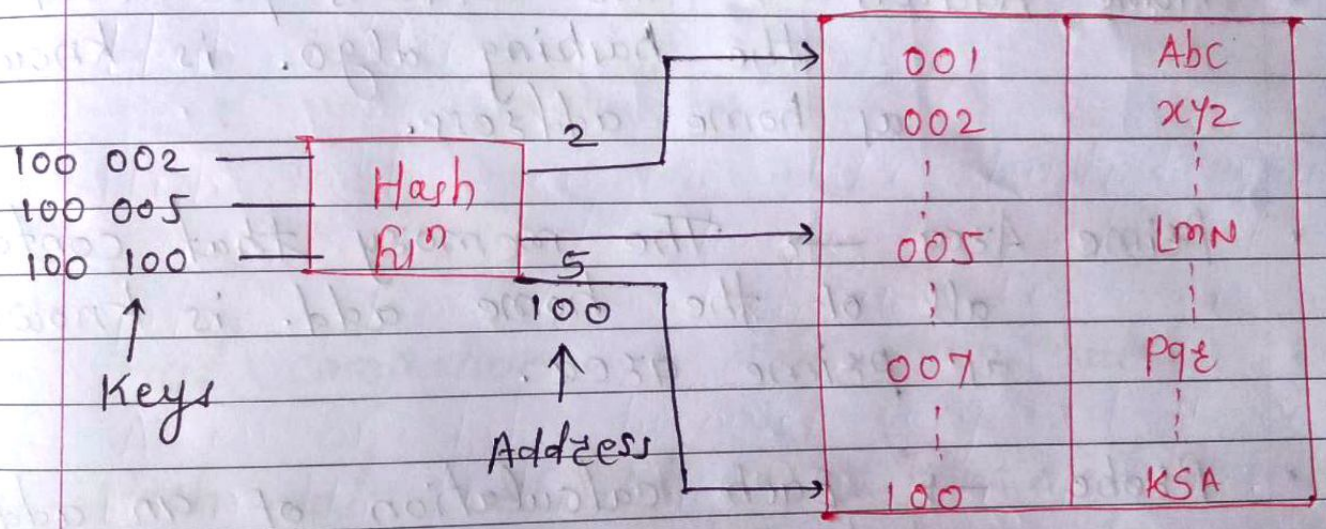


Fig (Hash Search)

* Characteristics of Hash fun :-

① Easy & quick to compute.

② Even distribution of keys across the hash table.

③ A has function must minimize collisions.

* Basic Definitions of Hashing :-

- Synonyms → The set of keys that hash to the same Location in our list is called as synonyms.

- Collision → Collision is the event that occurs when a hashing algo. produces an address for an insertion key & that address is already occupied.

- Home Address → The address produced by the hashing algo. is known as home address.

- Prime Area → The memory that contains all of the home add. is known as prime area.

- Probe → Each calculation of an add. & test for success is known as a probe.

- Bucket → A hash table uses a hash fun to compute an index into an array of buckets from which desired value can be found.

- Overflow → An overflow occurs when the home bucket for new pair is full.
- Open Hashing → In open hashing, keys are stored in linked list attached to cells of hash table.
- Closed Hashing → In closed hashing all keys are stored in the hash table itself without the use of linked list.
- Load Density → The loading density of hash table is $a = n/(sb)$

  s is the no. of slots.

  b is the no. of buckets.

* Properties of Good Hashing Function →

- Fast computation of the hash value O(1).
- Hash values should be distributed uniformly.
  - Every hash value has equal probability.
  - This should hold even if keys are non-uniformly distributed.
- The goal of a hash function is disperse the keys in an apparently random way.

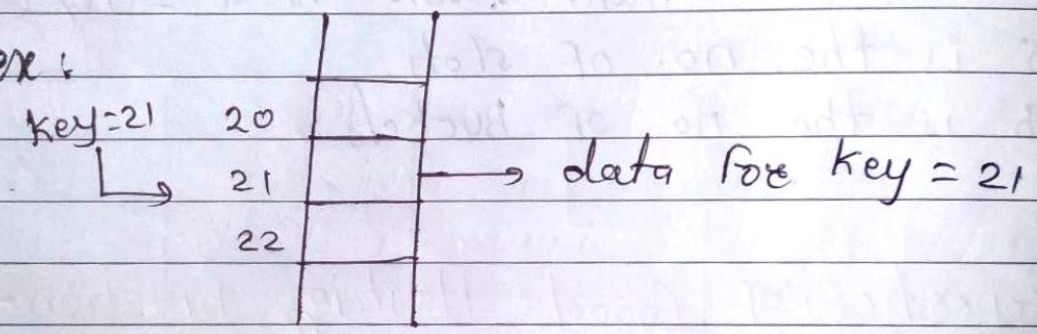
* Direct Address Tables :-

- Direct Address Table is a data structure that has the capability of mapping records to their corresponding keys using array.

- In direct address tables records are placed using their keys values directly as indexes.

- They faciliate fast searching, insertion & deletion operations.

- ex :



* Advantages :-

- Searching in O(1) Time -

Direct address tables use array which are random access data structure. So the key value can be easily used to search the records in O(1) Time.

- Insertion in O(1) Time -

We can easily insert an element in an

array in O(1) time.

- Deletion in O(1) Time –

Deletion of an element takes O(1) time in an array.

* Limitations →

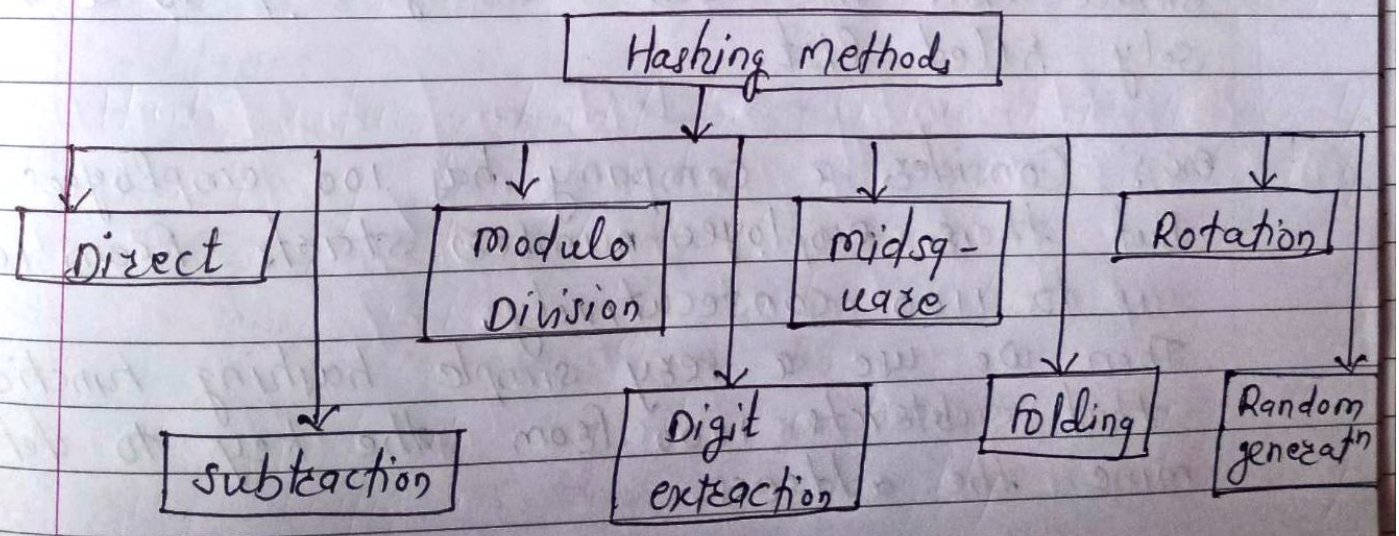
① Prior knowledge of maximum key value.

② Practically useful only if the maximum value is very less.

③ It causes wastage of memory space.

Hashing can overcome these limitations of direct address table.

* Hash Function →



① Direct Hashing :-

- In direct hashing, address for a key is generated without any algorithmic manipulation. Therefore the data structure must contain an address for every possible key.

ex. A small organization has 100 employees. Each employee is assigned an employee number betn 1 to 100. Hence we create an array of 100 employee records. the employee number can be directly used as the address of any individual record.

② Subtraction Method :-

- Sometimes we have keys that are consecutive but don't start from one.
- This method is simple & it guarantees no collisions.
- Limitation of this method can be used for small lists in which the keys map to a densely filled list.

ex: Consider a company has 100 employees, but their employee number starts from 1000 up to 1100 consecutively.

Then we use a very simple hashing function that subtracts 1000 from the key to determine the address.

③ Modulo-Division Method :-

This method divides the key by an array or bucket size & uses the remainder plus one for the address.

Address = (Key % list size) + 1

A list size that is a prime no. produces fewer collisions than other list size.

ex: Suppose we have 300 employee. The first prime no. greater than 300 is 307. We therefore choose 307 as our list size.

employee no. = 121267

(121267 % 307) + 1 = 2 + 1 = 3

④ Digit Extraction Method :

Using digit extraction, selected digits are extracted from the key & used as the address.

ex: 379452 → 394

121267 → 121

378845 → 388

⑤ Midsquare Method :

Key is a squared & the address is selected from the middle of the squared number.

ex: 9452 * 9452 = 89340304

Address is 3403.

(6) Folding Method :-

(a) Fold Shift → The key value is divided into parts whose size matches the size of the required address.

Then the left & right parts are shifted & added with middle part.

ex. suppose we have 3 digit address & key is 123456789

```
   123
 + 456
   789
 -----
 (1)368
```

Discard (1) so address is 368.

(b) Fold Boundry → Left and right numbers are folded on a fixed boundry betn them & the center no. This results in the two outside values beig reversed.

ex:

Suppose we have 3 digit address & key is 123456789

$$\begin{array}{r} 321 \\ +\ 456 \\ \hline 987 \\ \hline (1)764 \end{array}$$

→ Reversed digit of 123

→ Reversed digit of 789

Discard (1). So address is 764.

(7) Rotation Method :-

Rotation method is incorporated in combination with other hashing methods. It is most useful when keys are assigned serially, such as we often see in employee no. & part no.

Original key	Rotation	Rotated key
600101	60010	60010

(8) Multiplicative Hash function :-

ex $H(K) = \text{Floor}(P * \text{Fractional part of key} * A)$

ex :- P = constant Integer

A = constant real no.

K = 107, P = 50.

A = 0.6180339887

$H(K) = \text{Floor}(50 * (107 * 0.6180339887))$

$= \text{Floor}(3306.4818458045)$

$H(K) = 3306$

* Open Addressing →

- Open Addressing is a method for handling collisions.

- In open addressing all elements are stored in the hash table itself.

- The entire process of addressing is based upon probing.

- insert (K) – Keep probing until an empty slot is found. Once an empty slot is found, insert K.

- search (K) – Keep probing until the slot's key doesn't become equal to K or an empty slot is reached.

Delete (K) – Slot of deleted key are marked specially as "deleted".

① * Linear Probing →

- In linear probing, the hash table is searched sequentially that starts from the original location of the hash.

- In case the location that we get is already occupied, then we check for the next location.

- The function used for rehashing is as follow

rehash (key) = (n+1) % table-size

* Double Hashing →

- Double Hashing is a technique that reduces clustering in an optimized way.
- In this technique, the increments for the probing sequence are computed by using another hash function.

$h_1(K) = K \bmod$ table size

ex. $h_1(K) = K \bmod 7$ [K % 7]

$h_2(K) = 1 + (K \bmod 5)$

double Hashing → $[h_1(K) + i * (h_2(K))] \% 7$

$i = i^{th}$ rotation

ex: Keys - 3, 33, 42, 63, 89, 45, 93.

Hash function → key % 10.

	Empty	3 After	33 After	After 42	After 63	After 89	After 45	After 93
0	-	-	-	-	-	-	-	-
1	-	-	-	-	-	-	-	-
2	-	-	-	42	42	42	42	42
3	-	3	-	3	3	3	3	3
4	-	-	33	33	33	33	33	3
5	-	-	-	-	63	63	63	63
6	-	-	-	-	-	-	45	45
7	-	-	-	-	-	-	-	93
8	-	-	-	-	-	-	-	-
9	-	-	-	-	-	89	89	89

(b) Linear Probing with Chaining :- [without Replacement]

- Excessive collision can be dealt by means of chaning.
- All the records mapped to the same location are stored in a chain.

ex. 3, 33, 42, 63, 89, 45, 93

Hash Function → key % 10

Index	Key	Chain
0	-	-1
1	-	-1
2	-	-1
3	-	-1
4	-	-1
5	-	-1
6	-	-1
7	-	-1
8	-	-1
9	-	-1

← -1 shows there is no chaining yet.

0	-	-1
1	-	-1
2	42	-1
3	3	index → 4
4	33	index → 5
5	63	index → 6
6	45	-1
7	93	-1
8	-	-1
9	89	-1

Here 3, 33, 63 & 93 are supposed to be mapped at location 3. Hence all 3 are chained by index no. at chain coloumn.

(c) Linear Probing with Chaining (With Replacement) :-

In last ex. key 45 has misplaced its starting location is at index 5. So to overcome this issue we replace key at index 5 by 45 & shift 63 to another subsequent empty location.

ex. Keys - 3, 33, 42, 63, 89, 45, 93

Hash fun → key % 10.

Index	Key	chain	
0	-	-1	Now to insert 45, we have shift 63 to next index 6 & place 45 at index 5.
1	-	-1	
2	42	-1	
3	3	4	→ index of 33
4	33	5	→ index of 63
5	63	-1	
6	-	-1	
7	-	-1	
8	-	-1	
9	89	-1	

After arrival of 45 & 93

0	-	-1	
1	-	-1	
2	42	-1	
3	3	4	→ index of 33
4	33	6	→ New index of 63
5	45	-1	
6	63	7	→ index of 93
7	93	-1	
8	-	-1	
9	89	-1	

② Quadratic Probe :-

- In quadratic probe the increment is the collision probe no. squared.
- Thus for 1st probe we add 1^2; for 2nd probe we add 2^3 & for 3rd probe we add 3^2 so on until we find an empty element or we exhaust the possible elements.
- This probe does not ensure that all cells will be examined to find empty cell. Thus it may be possible that key won't be inserted even if there is an empty cell.

Probe No.	Collision location	$(Probe)^2$ & Increment	New Address
1	1	$(1)^2 = 1$	$1+1 = 2$
2	2	$(2)^2 = 4$	$2+4 = 6$
3	6	$(3)^2 = 9$	$6+9 = 15$
4	15	$(4)^2 = 16$	$15+16 = 31$
5	31	$(5)^2 = 25$	$31+25 = 56$

ex: Keys → 4371, 1323, 6173, 4199, 4344, 9679, 1989.

Hash fun. → key % 10

① 4371 % 10 = 1 (1st location is empty, put the element directly)

Index	Key
0	9679
1	4371
2	–
3	1323
4	6173
5	4344
6	–
7	–
8	1989
9	4199

② $1323 \% 10 = 3$ [put the element directly]

③ $6173 \% 10 = 3$
3rd location is not empty
$\therefore 6173 \% 10 = 3 + (1)^2 = 4$
put the element at 4th index.

④ $4199 \% 10 = 9$ [put the element directly]

⑤ $4344 \% 10 = 4$ [4th loc. is not empty]
$\therefore 4344 \% 10 = 4 + (1)^2 = 5$
put the element at 5th loc.

⑥ $9679 \% 10 = 9$ [9th loc. is not empty]
$\therefore 9679 \% 10 = 9 + (1)^2 = 10 \% 10 = 0$
put the element at 0th loc.

⑦ $1989 \% 10 = 9$ [9th loc. is not empty]
$\therefore 1989 \% 10 = 9 + (2)^2 = 9 + 4 = 13 \% 10 = 3$ but 3rd is not empty
$\therefore 1989 \% 10 = 9 + (3)^2 = 18 \% 10 = 8$
8th loc. is empty so put the element there.

③ Key Offset →

- Key offset is a double hashing method that produces diff. collision paths for diff. keys.
- Key offset calculates the new add. as a fun of the old add. and the key.

Offset = [key / list size]

Address = [(Offset + old address) % List size] + 1

ex: key → 166702
size → 307

1st hash fun = key % 10
166702 % 10 = 2

Offset = (166702/307) = 543

Address = [(543 + 002) % 307] + 1 = 239

If the 239 is not empty location then repeat the steps to find next address.

∴ offset = (166702/307) = 543)

Address = [(543 + 239) % 307] + 1 = 169

④ Dynamic Hashing / Linked list Resolution :-

- Linked list is an ordered collection of data in which each element contains the location of the next element.
- Linked list resolution uses a separate area to store collisions & chains all synonyms together in a linked list.
- It uses two storage areas, prime area and overflow area.

ex: key - 3, 33, 42, 63, 89, 45, 93

Hash fun → key % 10

Index	key
0	
1	
2	42
3	33
4	
5	45
6	
7	
8	
9	89

→ [→] 63 [→] 93

⑤ Bucket Hashing :-

- Bucket nodes that accommodate Multiple Data Occurences. Becoz a bucket can hold multiple pieces of data, collision are

postponed until the bucket is full.

ex: Keys → 3, 33, 42, 63, 89, 45

Hash fun = key % 10

0	
1	
2	42

3	3
	33
	63
4	
5	45

⋮

⑥ Rehashing :-

- Rehashing tells us what to do when the hash table gets full instead of waiting for the hash table to get completely full, it is more efficient to rehash when the table is about 70% or 80% full.

- The most common rehashing technique is to construct a new table of approximately double size of the original hash table.

ex: 13 15 6 24 23 into an initially empty hash table. Assume table size = 7 & use linear probing for collision resoultion.

since table size = 7

Hash (x) = x mod 7

After 23 is inserted table is 70% full,

Rehash : New table size = 7 * 2 = 14

14 is not prime number.

So, we select the prime no. closest to & greater than 14 i.e. 17 as the new table size. New hash fun is : hash(x) = x mod 17.

⑦ Perfect Hashing :-

- A perfect hash fun for a set S is a hash fun that maps distinct elements in S to a set of integers, with no collisions.

- Perfect hash function may be used to implement a lookup table with constant worst-case access time.

- Perfect hashing is a technique for storing records in a hash table in a way that guarentees no collisions.

* Real time appn of Hash table :-

① Hash is used for cache mapping for fast access to the data.

② Hash can be used for password verification.

③ Hash is used in cryptography as a message digest.