UNIT-I

Chapter-1: Introduction to Data Structures

Definition

Data Structure is a particular way of storing and organizing data in the computer memory so that the data can easily be retrieved and efficiently utilized in the future when required. The data can be managed in various ways, like the logical or mathematical model for a specific organization of data is known as a data structure.

Data :- Data is termed as a collection of raw facts which can be stored on computer media. Data must have an implicit meaning.Data can be considered as either singular or plural. Data may be a collection of numbers, text or char,audio,image and video.

Information :- processed data or meaningful data is called information. Representation of information can increase the knowledge of a person who uses it. Information can be displayed in the form of tables,graphs or pictorial representation.

Ex:-

Data	Information
25	Roll number of the student
Rama	Name of the student
ADT	Image of an organization
Santa	Signature of the person
{10,12,14,16,18}	Set of Student Roll numbers

The terms data and information can be used interchangeably. But it is very essential to distinguish the data and information.

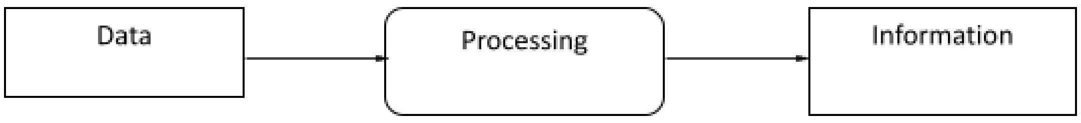


Fig : Data and Information.

Concept of Data Structures

Data Structures are the building blocks of any software or program. Selecting the suitable data structure for a program is an extremely challenging task for a programmer.

The following are some **fundamental terminologies** used whenever the data structures are involved:

1. **Data:** We can define data as an elementary value or a collection of values. For example, the Employee's name and ID are the data related to the Employee.
2. **Data Items:** A Single unit of value is known as Data Item.

3. **Group Items:** Data Items that have subordinate data items are known as Group Items. For example, an employee's name can have a first, middle, and last name.
4. **Elementary Items:** Data Items that are unable to divide into sub-items are known as Elementary Items. For example, the ID of an Employee.
5. **Entity and Attribute:** A class of certain objects is represented by an Entity. It consists of different Attributes. Each Attribute symbolizes the specific property of that Entity

Need for Data Structures :-

As applications are becoming more complex and the amount of data is increasing every day, which may lead to problems with data searching, processing speed, multiple requests handling, and many more. Data Structures support different methods to organize, manage, and store data efficiently. With the help of data Structures, we can easily traverse the data items. Data Structures provide Efficiency, Reusability, and Abstraction. We have to learn about data Structures for the following reasons:

- Data Structures and Algorithms are two of the key aspects of Computer Science.
- Data Structures allow us to organize and store data, whereas Algorithms allow us to process that data meaningfully.
- Learning Data Structures and Algorithms will help us become better Programmers.
- We will be able to write code that is more effective and reliable.
- We will also be able to solve problems more quickly and efficiently.

Characteristics of a Data Structure :-

- **Correctness** − Data structure implementation should implement its interface correctly.
- **Time Complexity** − Running time or the execution time of operations of data structure must be as small as possible.
- **Space Complexity** − Memory usage of a data structure operation should be as little as possible.

Features of Data Structures:-

Some of the significant features of data Structures are :

- **Robustness:** Generally, all computer programmers aim to produce software that yields correct output for every possible input, along with efficient execution on all hardware platforms. This type of robust software must manage both valid and invalid inputs.
- **Adaptability:** Building software applications like Web Browsers, Word Processors, and Internet Search Engine include huge software systems that require correct and efficient working or execution for many years. Moreover, software evolves due to emerging technologies or ever-changing market conditions.
- **Reusability:** The features like Reusability and Adaptability go hand in hand. It is known that the programmer needs many resources to build any software, making it a costly enterprise. However, if the software is developed in a reusable and adaptable way, then it can be applied in most future applications. Thus, by executing quality data structures, it is possible to build reusable software, which appears to be cost-effective and timesaving.

Over view of Data Structures (or) Classification of Data Structures

We can classify Data Structures into two categories:

1. Primitive data structures
2. Non-Primitive data structures

The following figure shows the different classifications of data structures.

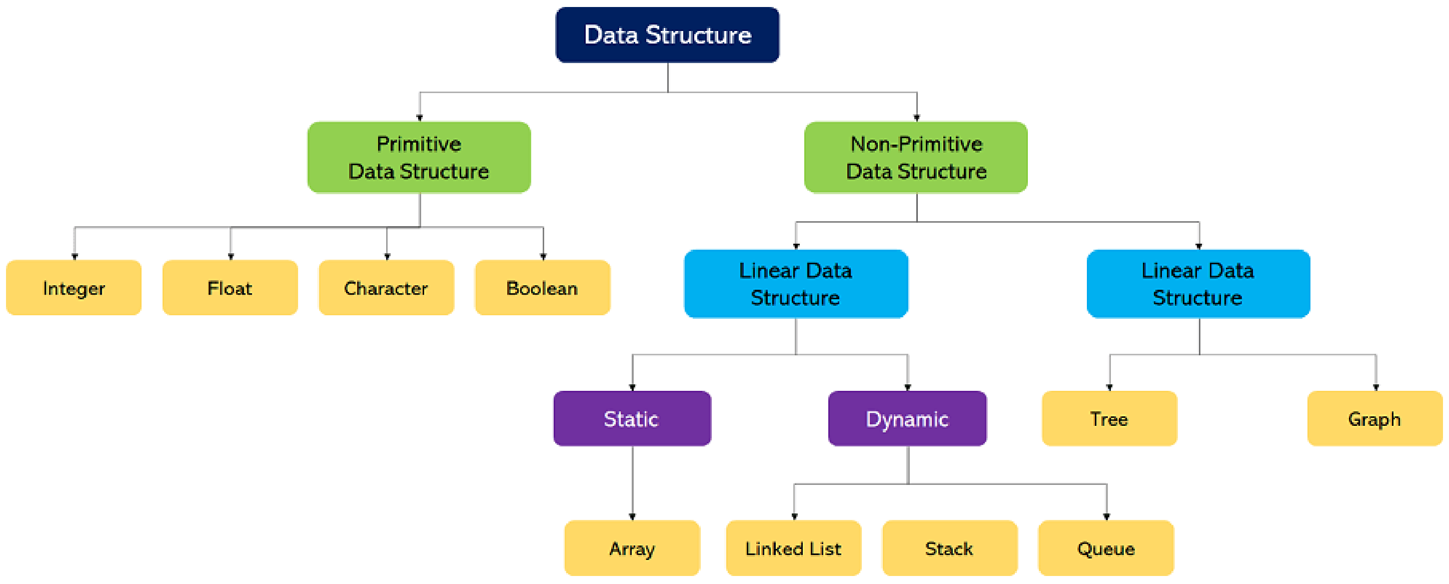


Fig: Classification of Data Structures

Primitive Data Structures:-

- **Primitive Data Structures** are the data structures consisting of the numbers and the characters that are **in-built** into programs.
- These data structures can be manipulated or operated directly by machine-level instructions.
- Basic data types like **Integer, Float, Character**, and **Boolean** come under the Primitive Data Structures.
- These data types are also called **Simple data types**, as they contain characters that can't be divided further

Non-Primitive Data Structures :-

- **Non-Primitive Data Structures** are those data structures derived from Primitive Data Structures.
- These data structures can't be manipulated or operated directly by machine-level instructions.
- The focus of these data structures is on forming a set of data elements that is either **homogeneous** (same data type) or **heterogeneous** (different data types).

- Based on the structure and arrangement of data, we can divide these data structures into two sub-categories :

  1. Linear Data Structures
  2. Non-Linear Data Structures

Linear Data Structures:-

A data structure that preserves a linear ordering or sequence among its data elements is known as a Linear Data Structure. Based on memory allocation, the Linear Data Structures are further classified into two types:

1. **Static Data Structures:** The data structures having a fixed size are known as Static Data Structures. The memory for these data structures is allocated at the compiler time, and their size cannot be changed by the user after being compiled; however, the data stored in them can be altered.
   The **Array** is the best example of the Static Data Structure as they have a fixed size, and its data can be modified later.
2. **Dynamic Data Structures:** The data structures having a dynamic size are known as Dynamic Data Structures. The memory of these data structures is allocated at the run time, and their size varies during the run time of a program. Moreover, the user can change the size as well as the data elements stored in these data structures at the run time of the program.
   Linked Lists, Stacks, and **Queues** are common examples of dynamic data structures.

Non Linear Data Structures :-

This data structure does not form a sequence or linear order among the elements is called a Non-Linear data structure. Each item or element is connected with two or more other items in a non-linear arrangement. The data elements are not arranged in sequential order. The different types of non-linear data structures are: Trees and Graphs.

Differences between Linear and Non-Linear Data Structures :-

S.NO	Linear Data Structure	Non-linear Data Structure
1.	In a linear data structure, data elements are arranged in a linear order where each and every element is attached to its previous and next adjacent.	In a non-linear data structure, data elements are attached in hierarchical manner.
2.	In linear data structure, elements are involved in single level.	Whereas in Non-linear data structure, elements are involved in multiple levels.
3.	Its implementation is easy in comparison with non-linear data structure.	While its implementation is complex in comparison with linear data structure.
4.	In linear data structure, data elements can be traversed in a single run only.	While in non-linear data structure, data elements can't be traversed in a single run.
5.	In a linear data structure, memory is not utilized in an efficient way.	While in a non-linear data structure, memory is utilized in an efficient way.
6.	Its examples are: array, stack, queue, linked list, etc.	While its examples are: trees and graphs.
7.	Applications of linear data structures are application software development.	Applications of non-linear data structures are Artificial Intelligence and image processing.

Implementation of Data Structures

Implementation of data structures can be done in two phases:

Phase 1: Storage Representation:

Storage Representation involves how a data structure can be stored in a computer memory. This storage representation in general is based on the use of other data structures.

Phase 2: Algorithmic notation of various operations of the data structure:

A function for manipulating a data structure is expressed in terms of algorithms so that the details of the operation can be understood easily and the reader can implement them with the help of any programming language.

In order to express the algorithm for a given operation, we will assume different control structures and notations as shown below:

Algorithm <Name of the Operation>

Input: <Specification of input data for the operation>

Output: <Specification of output after the successful performance of the operation>

Remark: <If the operation assumes other data structure for its implementation or something important>

Steps:

```
1.  . . . . . . . . . . . . . . . . . . . . . . . .
2.  If <condition> then                  // Comment on this step, if any applicable
3.      . . . . . . . . . . . . . . . . . . . . . . . .
4.      . . . . . . . . . . . . . . . . . . . . . . . .
5.  Else
6.      . . . . . . . . . . . . . . . . . . . . . . . .
7.      . . . . . . . . . . . . . . . . . . . . . . . .
8.  End if
9.  . . . . . . . . . . . . . . . . . . . . . . . .
10. . . . . . . . . . . . . . . . . . . . . . . . .
11. While <condition> do
12.     . . . . . . . . . . . . . . . . . . . . . . . .
13.     . . . . . . . . . . . . . . . . . . . . . . . .
14. EndWhile
15. . . . . . . . . . . . . . . . . . . . . . . . .
16. . . . . . . . . . . . . . . . . . . . . . . . .
17. For <loop condition> do
18.     . . . . . . . . . . . . . . . . . . . . . . . .
19.     . . . . . . . . . . . . . . . . . . . . . . . .
20. EndFor
21.     . . . . . . . . . . . . . . . . . . . . . . . .
22.     . . . . . . . . . . . . . . . . . . . . . . . .
23. Stop
```

Chapter-2: Arrays

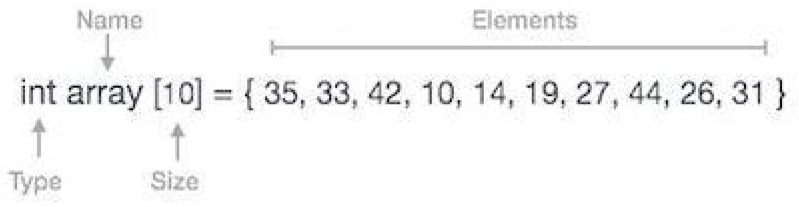
Array Definition and Terminology

Arrays are defined as the collection of similar types of data items stored at contiguous memory locations. It is one of the simplest data structures where each data element can be randomly accessed by using its index number.

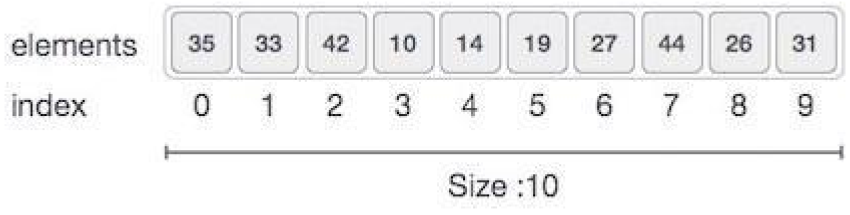
Array is a container which can hold a fix number of items and these items should be of the same type. Most of the data structures make use of arrays to implement their algorithms. Following is the important **Terminology** to understand the concept of Array.

- **Element:-** Each item stored in an array is called an element.
- **Index or Subscript :-** Each location of an element in an array has a numerical index, which is used to identify the element. All the elements in an array can be referenced by a subscript like A_i or A[i]: this subscript is known as index.
- **Size or Length or Dimension:-** Size refers to the number of elements that the array will contain.
- **Data Type:**- which refers to the kind of data items that are to be stored in to the array.
- **Base:**- The base of an array is the address of the memory location where the first element of the array is located.
- **Range of Indices:**- Indices of an array elements may change from a lower bound (L) to an upper bound (U), and these bounds are called the boundaries of an array.

Array Representation:-
Arrays can be declared in various ways in different languages. For illustration, let's take C array declaration.



Arrays can be declared in various ways in different languages. For illustration, let's take C array declaration.



As per the above illustration, following are the important points to be considered.

- Index starts with 0.
- Array length is 10 which means it can store 10 elements.
- Each element can be accessed via its index. For example, we can fetch an element at index 5 as 19.

One Dimensional Array

If only one subscript/index is required to reference all the elements in an array, then the array is termed as One-Dimensional array or simply an array. For example we have to declare an array like : int A[5];

In the above array all the elements are referred as A[1],A[2],A[3],A[4] and A[5] with only one dimension.

Operations on Arrays:- The following are the different operations performed on Arrays:

- **Traversal** :- This operation is used to print the elements of the array.
- **Insertion** :- It is used to add an element at a particular index.
- **Deletion** :- It is used to delete an element from a particular index.

- **Search** :- It is used to search an element using the given index or by the value.
- **Update** :- It updates an element at a particular index.

Traversal operation:-This operation is performed to traverse through the array elements. It prints all array elements one after another.

```
#include <stdio.h>
void main()
 {
    int Arr[5] = {18, 30, 15, 70, 12};
    int i;
    printf("Elements of the array are:\n");
    for(i = 0; i<5; i++)
    {
      printf("Arr[%d] = %d,  ", i, Arr[i]);
    }
  }
```

Insertion operation:- This operation is performed to insert one or more elements into the array. As per the requirements, an element can be added at the beginning, end, or at any index of the array. For example consider the following example of inserting an element into the array.

```
#include <stdio.h>
void main()
{
 int arr[20] = { 18, 30, 15, 70, 12 };
 int i, x, pos, n = 5;
 printf("Array elements before insertion\n");

 for (i = 0; i < n; i++)
    printf("%d ", arr[i]);
 printf("\n");
 x = 50; // element to be inserted
 pos = 4;
 n++;
 for (i = n-1; i >= pos; i--)
   arr[i] = arr[i - 1];
 arr[pos - 1] = x;
 printf("Array elements after insertion\n");
 for (i = 0; i < n; i++)
   printf("%d ", arr[i]);
 printf("\n");
}
```

Deletion operation:-This operation removes an element from the array and then reorganizes all of the array elements. Consider the following example:

```
#include <stdio.h>
void main()
{
  int arr[] = {18, 30, 15, 70, 12};
  int k = 30, n = 5;
```

```
   int i, j;
   printf("Given array elements are :\n");
   for(i = 0; i<n; i++)
   {
     printf("arr[%d] = %d,  ", i, arr[i]);
   }
   j = k;
  while( j < n)
  {
    arr[j-1] = arr[j];
    j = j + 1;
  }
 n = n -1;
 printf("\nElements of array after deletion:\n");
 for(i = 0; i<n; i++)
 {
   printf("arr[%d] = %d,  ", i, arr[i]);
 }
}
```

Search operation :-This operation is performed to search an element in the array based on the value or index.

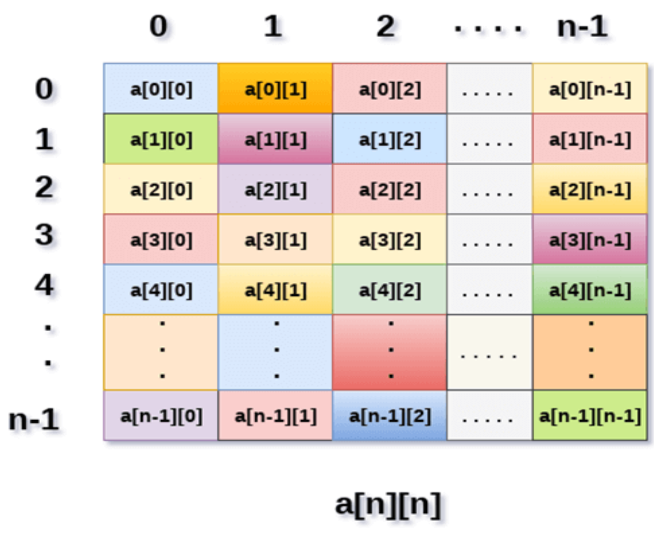
```
#include <stdio.h>
void main()
{
  int arr[5] = {18, 30, 15, 70, 12};
  int item = 70, i, j=0 ;
  printf("Given array elements are :\n");
  for(i = 0; i<5; i++)
   {
     printf("arr[%d] = %d,  ", i, arr[i]);
   }
   printf("\nElement to be searched = %d", item);
  while( j < 5)
  {
    if( arr[j] == item )
     {
     break;
     }
   j = j + 1;
  }
 printf("\nElement %d is found at %d position", item, j+1);
}
```

Update operation:-This operation is performed to update an existing array element located at the given index.

```
#include <stdio.h>
void main()
{
  int arr[5] = {18, 30, 15, 70, 12};
  int item = 50, i, pos = 3;
  printf("Given array elements are :\n");
  for(i = 0; i<5; i++)
  {
   printf("arr[%d] = %d,  ", i, arr[i]);
  }
  arr[pos-1] = item;
  printf("\nArray elements after updation :\n");
  for(i = 0; i<5; i++)
  {
    printf("arr[%d] = %d,  ", i, arr[i]);
  }
}
```

Two-Dimensional(2D) Arrays

Two-dimensional (2D) array can be defined as an array of arrays. The 2D array is organized as matrices which can be represented as the collection of rows and columns. The syntax for declaring a 2D array is :
data type arrayname[rowsize][columnsize];



Above image shows the two dimensional array, the elements are organized in the form of rows and columns. First element of the first row is represented by a[0][0] where the number shown in the first index is the number of that row while the number shown in the second index is the number of the column.

The syntax to declare and initialize the 2D array is given as follows:

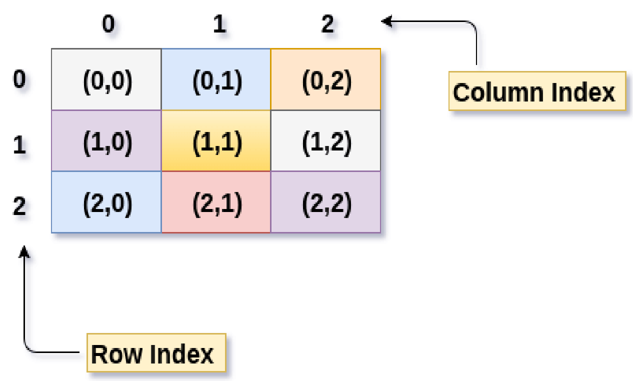
int arr[2][2] = {0,1,2,3};
The number of elements that can be present in a 2D array will always be equal to (**number of rows * number of columns**).

Example:-

```
#include <stdio.h>

void main ()

{

   int arr[3][3],i,j;

   for (i=0;i<3;i++)
```

```
    {
        for (j=0;j<3;j++)
        {
            printf("Enter a[%d][%d]: ",i,j);
            scanf("%d",&arr[i][j]);
        }
    }
    printf("\n printing the elements ....\n");
    for(i=0;i<3;i++)
    {
        printf("\n");
        for (j=0;j<3;j++)
        {
            printf("%d\t",arr[i][j]);
        }
    }
}
```

In computer memory, the storage technique for 2D array is similar to that of one dimensional array. The size of a two dimensional array is equal to the multiplication of number of rows and the number of columns present in the array. A 3 X 3 two dimensional array is shown in the following image. However, this array needs to be mapped to a one dimensional array in order to store it into the memory.



There are two main techniques of storing 2D array elements into memory. They are :

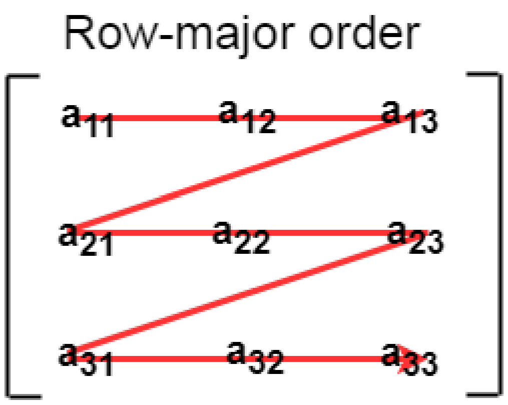
1. Row Major ordering
2. Column Major ordering

1. Row Major ordering :-

In row major ordering, all the rows of the 2D array are stored into the memory contiguously. Considering the array shown in the above image, its memory allocation according to row major order is shown as follows.

(0,0)	(0,1)	(0,2)	(1,0)	(1,1)	(1,2)	(2,0)	(2,1)	(2,2)

First, the 1st row of the array is stored into the memory completely, then the 2nd row of the array is stored into the memory completely and so on till the last row.

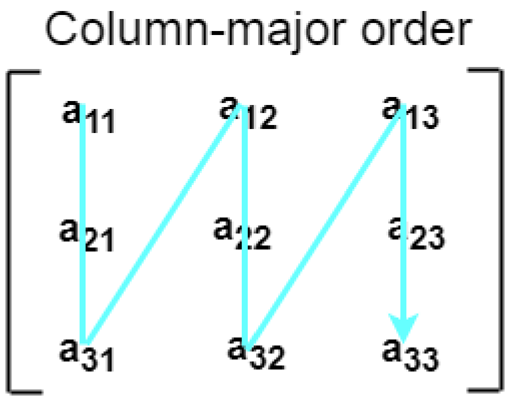

Row-major order
a_{11} a_{12} a_{13}
a_{21} a_{22} a_{23}
a_{31} a_{32} a_{33}

2. Column Major ordering :-

According to the column major ordering, all the columns of the 2D array are stored into the memory contiguously. The memory allocation of the array which is shown in the above image is given as follows.

(0,0)	(1,0)	(2,0)	(0,1)	(1,1)	(2,1)	(0,2)	(1,2)	(2,2)

First, the 1st column of the array is stored into the memory completely, then the 2nd row of the array is stored into the memory completely and so on till the last column of the array.



Pointer Arrays

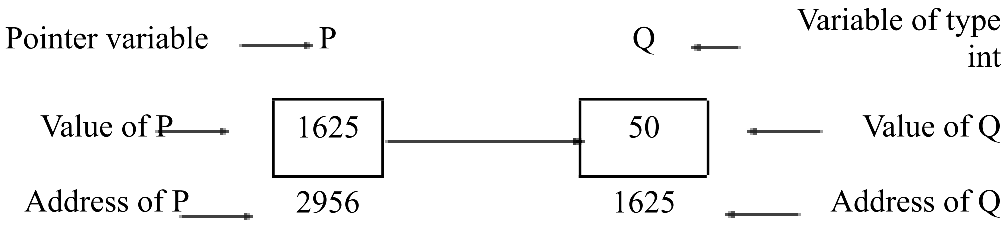
Using pointer concept, we can handle the memory conveniently. The pointer concept helps us to create and manage the memory in a convenient manner and also increase the execution speed of the program.

A **pointer** is a variable that holds the memory address of a data item such as a variable, an array element, etc. The organization of memory is designed in such a way that it contains continuous locations called memory cells to store data. The size of memory location that is assigned to each data value depending on the type of data declared. A unique numeric address is allotted to each of these memory locations for easy accessing of data. It is possible for the programmer to access this memory address using a special variable called ***pointer***.

Uses or Advantages of Pointers:-

- Pointers provide the flexibility of creation of dynamic variables, accessing and manipulating the contents of memory locations and releasing the memory occupied by dynamic variables which are no longer needed.
- Array elements can be easily accessed.
- Arguments passed to a function can be modified. Thus making a function to return more than one value. Simply it provides call-by-reference mechanism.
- Dynamic data structures like stacks,queues,linked lists,graphs,etc can be created.
- Memory can be efficiently used through dynamic memory allocation (DMA).
- By using pointers execution of program will be faster.

For instance, let P is a pointer variable that contains the address of another variable Q. then P is said to be pointing to Q. If the address of Q is 1625 then the value of P would be 1625. Using this address we can manipulate the value of Q. It can be as shown below:



Declaring a Pointer Variable:-

A pointer declaration contains a base type and asterisk (*) followed by a variable. A pointer variable is an unsigned integer and hence it occupies 2 bytes of memory. The general format for declaring a pointer variable is as follows:

Syntax:- datatype *variable;

Here, the data type is the base type of the pointer. The base type defines the type of variable to which the pointer will point. The base type must match to the type of object to which a pointer points.

Example:

```
int *p;
float *m;
```

The above declaration indicates that p is a pointer variable to hold the address of an integer variable. The variable 'm' holds the address of a float variable.

Pointer Operators:-

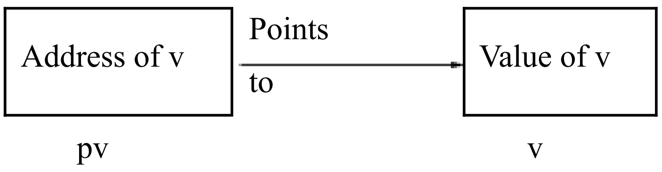
'C' language provides two special operators to manipulate the data items directly from memory location using pointer variables.

1. Address Operator (&):

The address operator is used to get and assign the address of a variable to a pointer variable.

Ex: pv = &v;

Where 'pv' is a pointer variable which holds the address of the variable 'v'. So 'pv' is a pointer to 'v' that points to the location of 'v' as shown below:



The above diagram shows the relationship between 'pv' and 'v'. Hence pv=&v and v=*pv.

2. Indirection Operator (*):

The asterisk (*) character is used as indirection operator. It is a unary operator that returns the value located at the address.

For example, v=*pv;

This statement assigns the value stored at address 'pv' into the variable 'v'. Hence *pv can also be assumed as *(&v) is equal to 'v'.

Example:

```
main( )
{
 int a=50;
 int *p;
 p = &a;
 printf("Value of a = %d ",a);
 printf("Address of a = %u ",&a);
 printf("Value of p = %u",p);
```

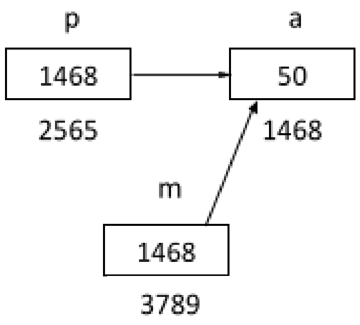
```
  printf("Address of p = %u",&p);
  printf("Address of a = Value of p = %u",p);
  printf("Value of a = *p = %d",*p);
}
```

Pointer Assignment:-

Like simple variables, a pointer variable can also be assigned to another pointer variable.

Example:

```
Main( )
{ int a=50;
    int *p, *m;
  p=&a;
  m = p;
  printf("Value of p = %u",p);
  printf("Value of m = %u",m);
}
```



Pointer Expressions and Pointer Arithmetic:-

Like other variables, pointer variables can be used in expression and also construct the pointer arithmetic. For example: if **P1** and **P1** are declared and initialized as pointer variables. Then we can construct the arithmetic expression like the following.

```
X = *p1 * *p2;
Sum = *p1 + *p2;
*p2 = *p1 – 35;
```

In the above example, the values of pointer variables P1 and P2 are evaluated and assigned the result to the variables.

Example:

```
main()
{
        int  *p1, *p2,x,sum,a,b;
        clrscr();
        a=20;
        b=30;
        p1=&a;
        p2=&b;
        x= *p1 * *p2;
        sum = *p1 + *p2;
        printf(" Multiplication = %d\n",x);
        printf("Addition =%d\n",sum);
        getch();
```

```
                    }
```

Pointer Array:-

There is a close relationship between pointers and arrays. The elements of an array can be efficiently accessed by using pointers. 'C' Language provides two methods of accessing array elements. They are pointer arithmetic method and array indexing method. However, pointer arithmetic will be faster.

To access array elements, the address of first element (base address) of an array can be assigned to the pointer variable and using this address we can access the remaining elements of that array.

For example,

```
        int a[10], *p;

        p = &a;
```

Assigns the base address of the array variable 'a' to the pointer variable 'p'. Now to access 5th element of 'a', we can write either a[4] or *(p+4).

Example:

```
main( )

{

 int a[20],i , n, *p;

 clrscr( );

 printf("How many values ");

 scanf("%d",&n);

 p = a;  /* assigns beginning address of 'a' to 'p' */

 for(i=0;i<=n-1;i++)

    scanf("%d",(p+i));      /* inputs value to array through pointer    */

 for(i=0;i<=n-1;i++)

    printf("%3d",*(p+i));      /* displays value of array through pointer    */

}
```

Array of Pointers:-

A pointer can also be declared as an array. As a pointer variable always contains an address, an array of pointers contains a collection of addresses. Like an array, the elements of an array of pointers are also stored in the memory in contiguous memory locations.

The general form to declare an array of pointers is as follows:

Syntax:- **datatype   *arrayname[size];**

Here 'data type' is the base type of pointer array and the size indicates the number of elements in the pointer array.

Example:-

```
main( )
```

```
{
    int a[5]={10,20,30,40,50}, i;
    int *p[5];
    for(i=0;i<5;i++)
        p[i] = &a[i];
    for(i=0;i<5;i++)
        printf("%u ",p[i]);
}
```

In the above example, the pointer variable ‘p’ holds the address of the elements of array variable ‘a’. This can be as shown below:

a[0]	a[1]	a[2]	a[3]	a[4]
10	20	30	40	50
1001	1003	1005	1007	1009

1001	1003	1005	1007	1009
p[0]	p[1]	p[2]	p[3]	p[4]

Searching

Searching is the process of finding some particular element in the list. If the element is present in the list, then the process is called successful, and the process returns the location of that element; otherwise, the search is called unsuccessful. There following are two popular search methods:

- **Linear Search**
- **Binary Search**

Linear Search

Linear search is also called as **Sequential Search.** It is the simplest searching algorithm. In Linear search, we simply traverse the list(array) completely and match each element of the list with the item(searching element) whose location is to be found. If the match is found, then the location of the item is returned; otherwise, the algorithm returns NULL.

The steps used in the implementation of Linear Search are listed as follows:

- First, we have to traverse the array elements using a **for** loop.

 In each iteration of **for loop,** compare the **search(key)** element with the current array element, and :

- If the element matches, then return the index of the corresponding array element.
- If the element does not match, then move to the next element.

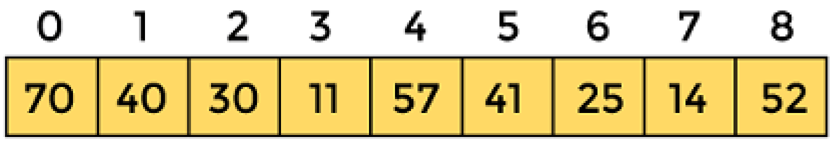
 If there is no match or the search element is not present in the given array, return **-1.**

Algorithm:-

```
Linear_Search(a, n, key) // 'a' is the given array, 'n' is the size of given array, 'key' is the element to search
Step 1: set pos = -1
Step 2: set i = 1
Step 3: repeat step 4 while i <= n
Step 4: if a[i] == key
    set pos = i
    print pos
    go to step 6
    end of if
    set i = i + 1
end of loop
Step 5: if pos = -1
            print " key is not present in the array "
        end of if
Step 6: exit
```

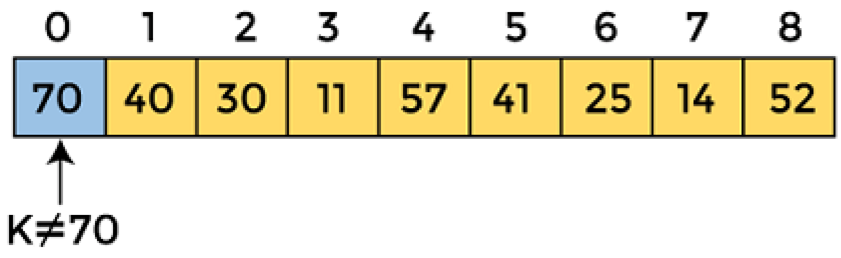
To understand the working of linear search algorithm, consider the following unsorted array.

Let the elements of array are :

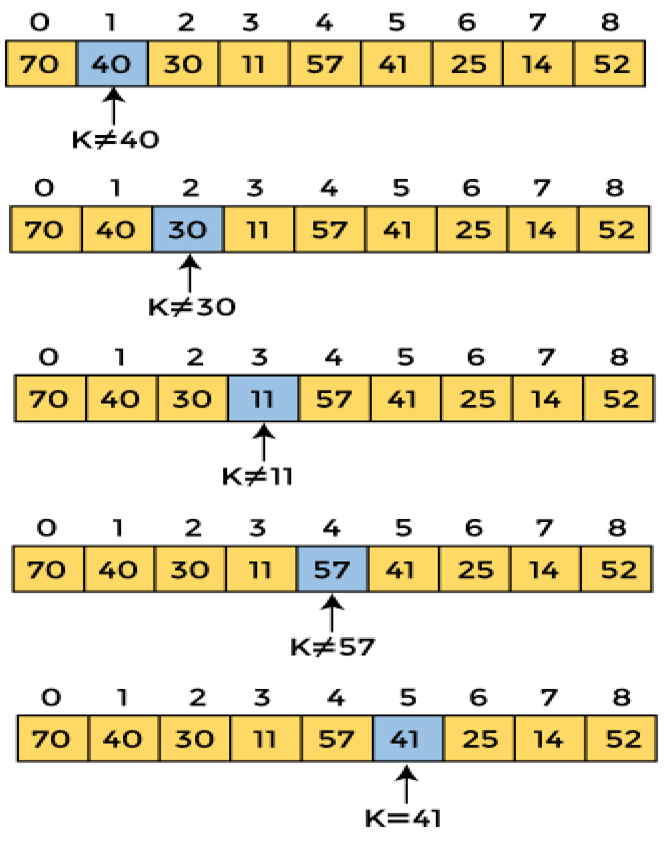


Let the element to be searched is **K = 41**

Now, start from the first element and compare **K** with each element of the array.



The value of **K,** i.e., **41,** is not matched with the first element of the array. So, move to the next element. And follow the same process until the respective element is found.



Now, the element to be searched is found. So algorithm will return the **index** of the matched element as ‘5’.

Binary search

Binary search is the search technique that works efficiently on sorted list (array). Hence, to search an element into some list using the binary search technique, we must ensure that the list is sorted intially.

Binary search follows the divide and conquer approach in which the list is divided into two halves, and the item is compared with the middle element of the list. If the match is found then, the location of the middle element is returned. Otherwise, we search into either of the halves depending upon the result produced through the match.

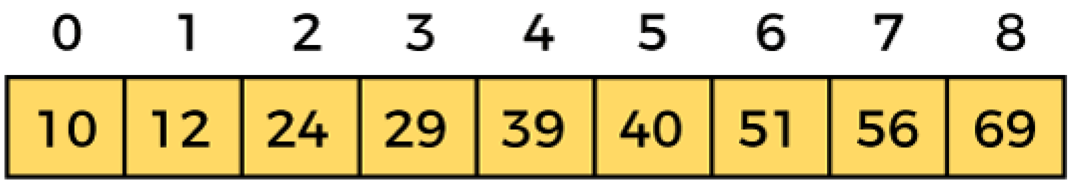
<u>**Algorithm**</u>:-

```
Binary_Search(a, lower_bound, upper_bound, key)
Step 1: set beg = lower_bound, end = upper_bound, pos = - 1
Step 2: repeat steps 3 and 4 while beg <=end
Step 3: set mid = (beg + end)/2
Step 4: if a[mid] = key
                set pos = mid
                print pos
                go to step 6
        else if a[mid] > val
                set end = mid - 1
        else
                set beg = mid + 1
        end of if
end of loop
Step 5: if pos = -1
                print "key is not present in the array"
end of if
```

Step 6: exit

To understand the working of the Binary search algorithm, consider a sorted array.

Let the elements of array are :



Let the element to search is, **K = 56**

We have to use the below formula to calculate the **mid** of the array :

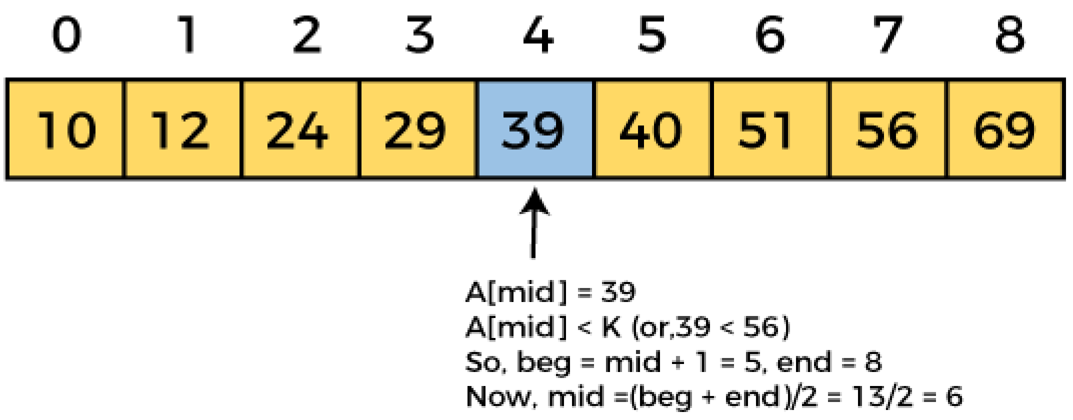
mid = (beg + end)/2

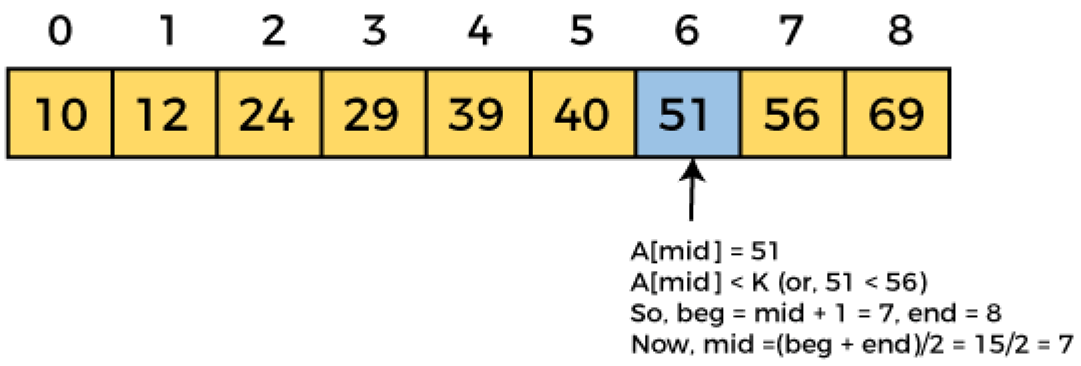
So, in the given array -

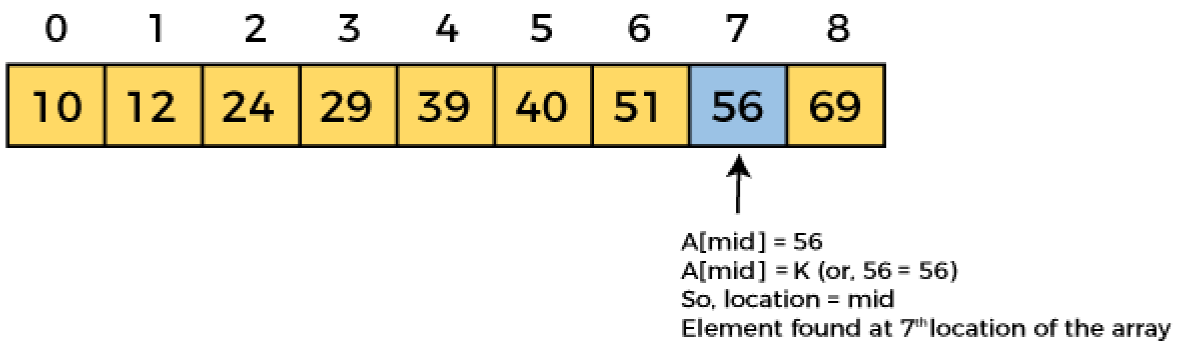
beg = 0

end = 8

mid = (0 + 8)/2 = 4. So, 4 is the mid of the array.







Now, the element to search is found. So algorithm will return the index of the matched element as '7'.

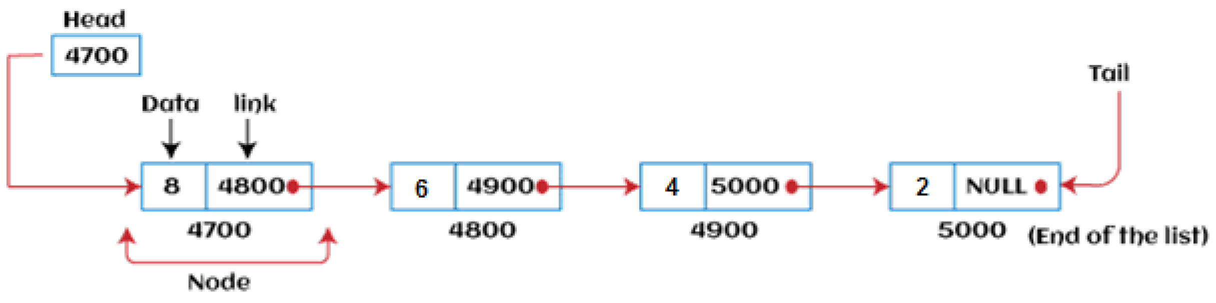
<u>UNIT-II : Linked Lists</u>

<u>Definition</u>

Linked list is a linear data structure that includes a series of connected nodes. Linked list can be defined as the collection nodes that are randomly stored in the memory. A node in the linked list contains two parts, i.e., first is the data part and second is the address part. The last node of the list contains a pointer to the null. After array, linked list is the second most used data structure. In a linked list, every link contains a connection to another link.

<u>Representation of a Linked list in Memory</u>:-

Linked list can be represented as the connection of nodes in which each node points to the next node of the list. The representation of the linked list is shown below :



Linked list is a data structure that overcomes the limitations of arrays. The following are some of the limitations of arrays :

- The size of the array must be known in advance before using it in the program.
- Increasing the size of the array is a time taking process. It is almost impossible to expand the size of the array at run time.
- All the elements in the array need to be contiguously stored in the memory. Inserting an element in the array needs shifting of all its predecessors.

Linked list is useful because :

- It allocates the memory dynamically. All the nodes of the linked list are non-contiguously stored in the memory and linked together with the help of pointers.
- In linked list, size is no longer a problem since we do not need to define its size at the time of declaration. List grows as per the program's demand and limited to the available memory space.

The declaration of linked list is given as follows :

```
struct node
{
  int data;
  struct node *next;
}
```

In the above declaration, a structure named as **node** that contains two variables, one is **data** that is of integer type, and another one is **next** that is a pointer which contains the address of next node.

Types of Linked lists

Linked list is classified into the following types :

- **Single Linked List or Singly linked list:-** Single linked list can be defined as the collection of an ordered set of elements. A node in the singly linked list consists of two parts: data part and link part. Data part of the node stores actual information that is to be represented by the node, while the link part of the node stores the address of its immediate successor.
- **Double linked list or Doubly Linked List :-** Double linked list is a complex type of linked list in which a node contains a pointer to the previous as well as the next node in the sequence. Therefore, in a double-linked list, a node consists of three parts: node data, pointer to the next node in sequence (next pointer or successor), and pointer to the previous node (previous pointer or predecessor).

- **Circular Single linked list or Circular Singly Linked List :-** In a circular single linked list, the last node of the list contains a pointer to the first node of the list. We can have circular single linked list as well as circular double linked list.
- **Circular Double Linked list or Circular Doubly Linked List :-** Circular double linked list is a more complex type of data structure in which a node contains pointers to its previous node as well as the next node. Circular double linked list doesn't contain NULL in any of the nodes. The last node of the list contains the address of the first node of the list. The first node of the list also contains the address of the last node in its previous pointer.

Advantages of Linked list

- **Dynamic data structure :-** The size of the linked list may vary according to the requirements. Linked list does not have a fixed size.
- **Insertion and deletion :-** Unlike arrays, insertion, and deletion in linked list is easier. Array elements are stored in the consecutive location, whereas the elements in the linked list are stored at a random location. To insert or delete an element in an array, we have to shift the elements for creating the space. Whereas, in linked list, instead of shifting, we just have to update the address of the pointer of the node.
- **Memory efficient :-** The size of a linked list can grow or shrink according to the requirements, so memory consumption in linked list is efficient.
- **Implementation :-** We can implement both stacks and queues using linked list.

Disadvantages of Linked list

The limitations of using the Linked list are given as follows :

- **Memory usage :-** In linked list, node occupies more memory than array. Each node of the linked list occupies two types of variables, i.e., one is a simple variable, and another one is the pointer variable.
- **Traversal :-** Traversal is not easy in the linked list. If we have to access an element in the linked list, we cannot access it randomly, while in case of array we can randomly access it by index. For example, if we want to access the 3rd node, then we need to traverse all the nodes before it. So, the time required to access a particular node is large.
- **Reverse traversing :-** Backtracking or reverse traversing is difficult in a linked list. In a doubly-linked list, it is easier but requires more memory to store the back pointer.

Applications of Linked lists

The applications of the Linked list are given below:

- With the help of a linked list, the polynomials can be represented as well as we can perform the operations on the polynomial.
- A linked list can be used to represent the sparse matrix.
- The various operations like student's details, employee's details, or product details can be implemented using the linked list as the linked list uses the structured data type that can hold different data types.
- Using linked list, we can implement stack, queue, tree, and other various data structures.

- The graph is a collection of edges and vertices, and the graph can be represented as an adjacency matrix and adjacency list. If we want to represent the graph as an adjacency matrix, then it can be implemented as an array. If we want to represent the graph as an adjacency list, then it can be implemented as a linked list.
- A linked list can be used to implement dynamic memory allocation. The dynamic memory allocation is the memory allocation done at the run-time.

Operations on Linked list

The basic operations performed on Linked list are given below:

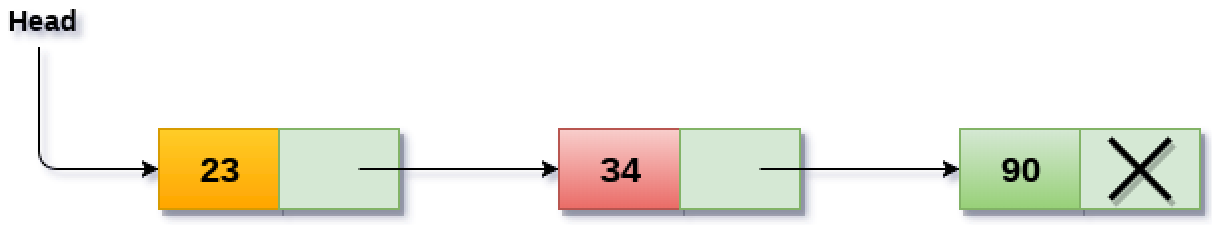
- **Insertion :-** This operation is performed to add an element into the list.
- **Deletion :-** It is performed to delete an operation from the list.
- **Display :-** It is performed to display the elements of the list.
- **Search :-** It is performed to search an element from the list using the given key.

Singly linked list and its operations

Single linked list can be defined as the collection of ordered set of elements. The number of elements may vary according to need of the program. A node in the single linked list consist of two parts: **data part** and **link part**. Data part of the node stores actual information that is to be represented by the node while the link part of the node stores the address of its immediate successor.

Single linked list can be traversed only in one direction so it is also called as One-Way-List or Chain. In other words, we can say that each node contains only next pointer, therefore we cannot traverse the list in the reverse direction.

Consider an example where the marks obtained by the student in three subjects are stored in a linked list as shown in the figure.



In the above figure, the arrow represents the links. The data part of every node contains the marks obtained by the student in the different subjects. The last node in the list is identified by the null pointer which is present in the address part of the last node.

Node Creation of singly linked list is as follows:

```
struct node
{
  int data;
  struct node *next;
```

```
        };
        struct node *head, *ptr;
        ptr = (struct node *)malloc(sizeof(struct node *));
```

Operations on Singly Linked List:-

There are various operations which can be performed on singly linked list. A list of all such operations is given below:

Insertion:-

The insertion into a single linked list can be performed at different positions. Based on the position of the new node being inserted, the insertion is categorized into the following categories.

SNO	Operation	Description
1	Insertion at beginning	It involves inserting any element at the front of the list.
2	Insertion at end of the list	It involves insertion at the last of the linked list. The new node can be inserted as the only node in the list or it can be inserted as the last one.
3	Insertion after specified node	It involves insertion after the specified node of the linked list. We need to skip the desired number of nodes in order to reach the node after which the new node will be inserted. .

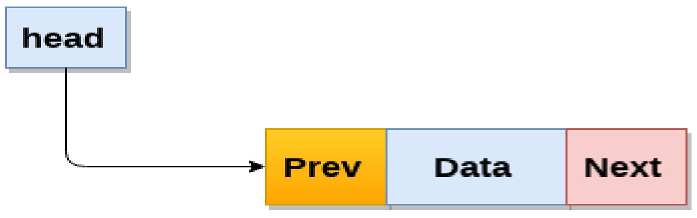
Deletion, Traversing and searching:-

The Deletion of a node from a single linked list can be performed at different positions. Based on the position of the node being deleted, the operation is categorized into the following categories.

SNO	Operation	Description
1	Deletion at beginning	It involves deletion of a node from the beginning of the list. This is the simplest operation among all.
3	Deletion after specified node	It involves deleting the node after the specified node in the list. we need to skip the desired number of nodes to reach the node after which the node will be deleted. This requires traversing through the list.
4	Traversing	In traversing, we simply visit each node of the list at least once in order to perform some specific operation on it, for example, printing data part of each node present in the list.
5	Searching	In searching, we match each element of the list with the given element. If the element is found on any of the location then location of that element is returned otherwise null is returned. .

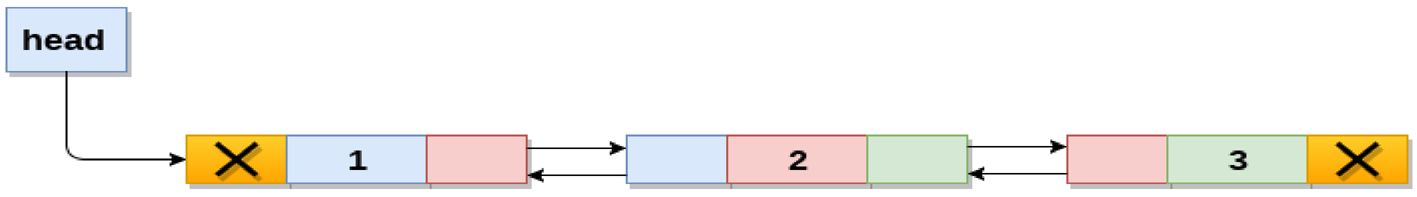
<u>Double linked list and its operations</u>

Double linked list is a complex type of linked list in which a node contains a pointer to the previous as well as the next node in the sequence. Therefore, in a doubly linked list, a node consists of three parts: node data, pointer to the next node in sequence (next pointer) , pointer to the previous node (previous pointer). A sample node in a doubly linked list is shown in the figure.



Node

A double linked list containing three nodes having numbers from 1 to 3 in their data part, is shown in the following image.



Doubly Linked List

In C, structure of a node in doubly linked list can be given as :

```
struct node
{
   struct node *prev;
   int data;
   struct node *next;
}
```

The **prev** part of the first node and the **next** part of the last node will always contain null indicating end in each direction.

In a single linked list, we could traverse only in one direction, because each node contains address of the next node and it doesn't have any record of its previous nodes. However, double linked list overcome this limitation of single linked list. Due to the fact that, each node of the list contains the address of its previous node, we can find all the details about the previous node as well by using the previous address stored inside the previous part of each node.

Operations on double linked list:-

All the operations regarding double linked list are described in the following table.

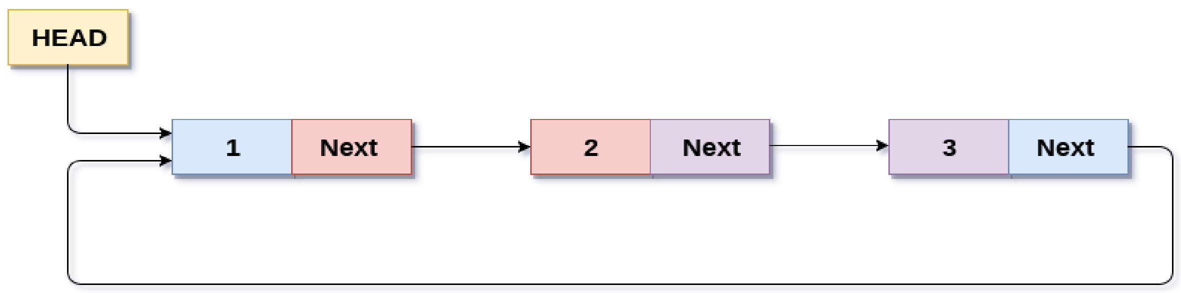
SNO	Operation	Description
1	Insertion at beginning	Adding the node into the linked list at beginning.
2	Insertion at end	Adding the node into the linked list to the end.
3	Insertion after specified node	Adding the node into the linked list after the specified node.
4	Deletion at beginning	Removing the node from beginning of the list
5	Deletion at the end	Removing the node from end of the list.
6	Deletion of the node having given data	Removing the node which is present just after the node containing the given data.
7	Searching	Comparing each node data with the item to be searched and return the location of the item in the list if the item found else return null.
8	Traversing	Visiting each node of the list at least once in order to perform some specific operation like searching, sorting, display, etc.

Circular Single Linked List and its operations

In a Circular Single linked list, the last node of the list contains a pointer to the first node of the list. We can have circular single linked list as well as circular double linked list.

We traverse a circular single linked list until we reach the same node where we started. The circular single liked list has no beginning and no ending. There is no null value present in the next part of any of the nodes.

The following image shows a circular single linked list.



Circular Singly Linked List

Circular linked list are mostly used in task maintenance in operating systems. There are many examples where circular linked list are being used in computer science including browser surfing where a record of pages visited in the past by the user, is maintained in the form of circular linked lists and can be accessed again on clicking the previous button.

Operations on Circular Single linked list:-

Insertion:-

SNO	Operation	Description
1	Insertion at beginning	Adding a node into circular single linked list at the beginning.
2	Insertion at the end	Adding a node into circular single linked list at the end.

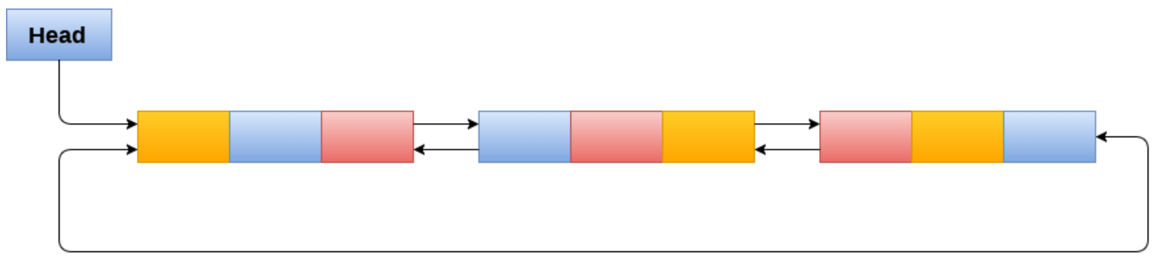
Deletion, Searching and Traversing:-

SNO	Operation	Description
1	Deletion at beginning	Removing the node from circular single linked list at the beginning.
2	Deletion at the end	Removing the node from circular single linked list at the end.
3	Searching	Compare each element of the node with the given item and return the location at which the item is present in the list otherwise return null.
4	Traversing	Visiting each element of the list at least once in order to perform some specific operation.

Circular Doubly Linked List and its operations

Circular double linked list is a more complex type of data structure in which a node contains pointers to its previous node as well as the next node. Circular double linked list doesn't contain NULL in any of the node. The last node of the list contains the address of the first node of the list. The first node of the list also contains address of the last node in its previous pointer.

A circular double linked list is shown in the following figure.



Circular Doubly Linked List

Due to the fact that a circular double linked list contains three parts in its structure therefore, it demands more space per node and more expensive basic operations. However, a circular double linked list provides easy manipulation of the pointers and the searching becomes twice as efficient.

Operations on circular double linked list :-

There are various operations which can be performed on circular double linked list. The node structure of a circular double linked list is similar to doubly linked list. However, the operations on circular double linked list are described in the following table.

SNO	Operation	Description
1	Insertion at beginning	Adding a node in circular double linked list at the beginning.
2	Insertion at end	Adding a node in circular double linked list at the end.
3	Deletion at beginning	Removing a node in circular double linked list from beginning.
4	Deletion at end	Removing a node in circular double linked list at the end.