NOTES

Subject Name: Data Structures & Analysis of Algorithms **Subject Code**: KCA-205 **Semester**: II

UNIT-1

Introduction to data structure: Data, Entity, Information, Difference between Data and Information, Data type, Build in data type, Abstract data type, Definition of data structures, Types of Data Structures: Linear and Non-Linear Data Structure, Introduction to Algorithms: Definition of Algorithms, Difference between algorithm and programs, properties of algorithm, Algorithm Design Techniques, Performance Analysis of Algorithms, Complexity of various code structures, Order of Growth, Asymptotic Notations.
Arrays: Definition, Single and Multidimensional Arrays, Representation of Arrays: Row Major Order, and Column Major Order, Derivation of Index Formulae for 1-D,2-D Array Application of arrays, Sparse Matrices and their representations.
Linked lists: Array Implementation and Pointer Implementation of Singly Linked Lists, Doubly Linked List, Circularly Linked List, Operations on a Linked List. Insertion, Deletion, Traversal, Polynomial Representation and Addition Subtraction & Multiplications of Single variable.

Data: Data is unorganized raw facts that need processing without which it is seemingly random and useless to humans. Data is the complete list of facts and details like text, observations, figures, symbols and description of things. It is the raw list of facts that are processed to gain information.

Data can be of two types:

- Qualitative data: It is the non-numerical data. For eg., texture of the skin, colour of eyes, etc.
- Quantitative data: Quantitative data is given in numbers. Data in the form of questions such as "how much", "how many", gives the quantitative data.

Entity: An **entity** can be a real-world object, either animate or inanimate, that can be easily identifiable. For example, in a school database, students, teachers, classes, and courses offered can be considered as entities. All these entities have some attributes or properties that give them their identity.

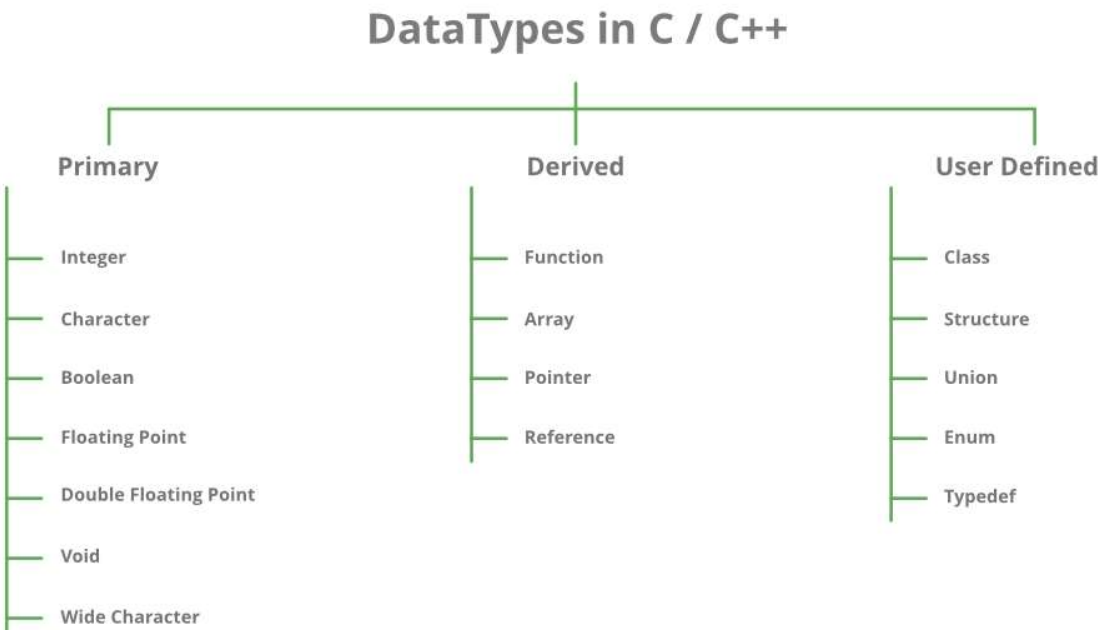
Information: Information is the processed, organized and structured data. It provides context for data. However, both the terms are used together; information can be easily understood than data.

Difference between Data and Information:

Data	Information
Data is unorganised raw facts that need processing without which it is seemingly random and useless to humans	Information is a processed, organised data presented in a given context and is useful to humans.
Data is an individual unit that contains raw material which does	Information is a group of data that collectively carry a logical meaning.

not carry any specific meaning.	
Data doesn't depend on information.	Information depends on data.
It is measured in bits and bytes.	Information is measured in meaningful units like time, quantity, etc.
Data is never suited to the specific needs of a designer.	Information is specific to the expectations and requirements because all the irrelevant facts and figures are removed, during the transformation process.
An example of data is a student's test score	The average score of a class is the information derived from the given data.

Data type: In computer science and computer programming, a **data type** or simply **type** is an attribute of **data** which tells the compiler or interpreter how the programmer intends to use the **data**. This **data type** defines the operations that can be done on the **data**, the meaning of the **data**, and the way values of that **type** can be stored.



Built-in Data types: The term **built**-in means that they are pre-defined in C++ and can be used directly in a program. char, int, float and double are the most common **built-in data types**. Apart from these, we also have void and Boolean **data types**.

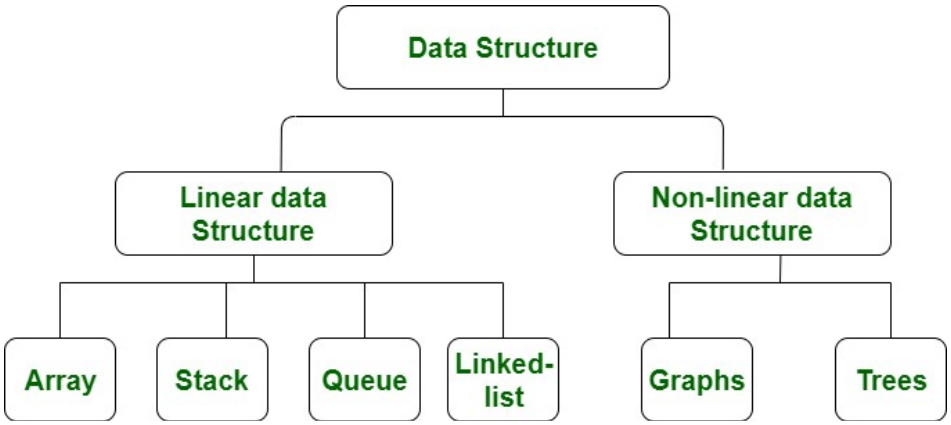
Abstract data types: An ADT may be defined as a "class of objects whose logical behavior is defined by a set of values and a set of operations". The abstract datatype is special kind of datatype, whose behavior is defined by a set of values and set of operations. The keyword "Abstract" is used as we can use these datatypes, we can perform

different operations. But how those operations are working that is totally hidden from the user. The ADT is made of with primitive datatypes, but operation logics are hidden.

Some examples of ADT are Stack, Queue, List etc

Definition of data structures, Types of Data Structures: Linear and Non-Linear Data Structure:

Data structure is a **data** organization, management, and storage format that enables efficient access and modification. More precisely, a **data structure** is a collection of **data** values, the relationships among them, and the functions or operations that can be applied to the **data**.



S.NO	Linear Data Structure	Non-linear Data Structure
1.	Data elements are arranged in a linear order where each and every elements are attached to its previous and next adjacent.	In a non-linear data structure, data elements are attached in hierarchically manner.
2.	In linear data structure, single level is involved.	Whereas in non-linear data structure, multiple levels are involved.
3.	Its implementation is easy in comparison to non-linear data structure.	While its implementation is complex in comparison to 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 only.
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 mainly in application software development.	Applications of non-linear data structures are in Artificial Intelligence and image processing.

Algorithms: An algorithm is a finite sequence of well-defined, computer-implementable instructions, typically to solve a class of problems or to perform a computation.

The following are the features of a good algorithm;

- Precision: a good algorithm must have a certain outlined steps. The steps should be exact enough, and not varying.
- Uniqueness: each step taken in the algorithm should give a definite result as stated by the writer of the algorithm. The results should not fluctuate by any means.
- Feasibility: the algorithm should be possible and practicable in real life. It should not be abstract or imaginary.
- Input: a good algorithm must be able to accept a set of defined input.
- Output: a good algorithm should be able to produce results as output, preferably solutions.
- Finiteness: the algorithm should have a stop after a certain number of instructions.
- Generality: the algorithm must apply to a set of defined inputs.

Difference between algorithm and program:

An algorithm is just a collection of instructions to do something. But that's pretty abstract, it's not tied to the specific way it's done. However if you take that algorithm and you write it up in a programming language, to perform that algorithm on a piece of hardware, on a system that can execute that code, then you have a program.

The algorithm is the logic; the program is how it's implemented.

An algorithm must have five properties:

- Input specified.
- Output specified.
- Definiteness.
- Effectiveness.
- Finiteness.

Algorithm Design Techniques and Performance Analysis of Algorithms:

In the field of computing, an algorithm is a set of instructions applied to solve a particular problem. Since algorithm design techniques are growing at a fast pace, it has become important for all of us to upgrade their knowledge in order to meet growing industry demand.

No matter which programming language you use, it is important to learn algorithm design techniques in data structures in order to be able to build scalable systems.

Selecting a proper design technique for algorithms is a complex but important task. Following are some of the main algorithm design techniques:

1. Brute-force or exhaustive search
2. Divide and Conquer
3. Greedy Algorithms
4. Dynamic Programming
5. Branch and Bound Algorithm
6. Randomized Algorithm

7. Backtracking

A given problem can be solved in various different approaches and some approaches deliver much more efficient results than others. Algorithm analysis is a technique used to measure the effectiveness and **performance** of the algorithms. It helps to determine the quality of an algorithm based on several parameters such as user-friendliness, maintainability, security, space usage and usage of other resources.

Complexity of various code structures, Order of Growth, and Asymptotic Notations.

The complexity of an algorithm is a measure of the amount of time and/or space required by an algorithm for an input of a given size (n). Though the complexity of the algorithm does depends upon the specific factors such as: The architecture of the computer i.e. the hardware platform representation of the Abstract Data Type(ADT) compiler efficiency the complexity of the underlying algorithm size of the input. Though you will see most significant factors are a complexity of underlying algorithm and size of the input.

(a) Time complexity: The time complexity of a program or algorithm is the amount of computational time it needs to completion. The time complexity is measured by counting the number of key operations in sorting and searching algorithms. The time complexity can be expressed as a function of number of key operations performed.
(b) Space complexity: The space complexity of program is the amount of memory that it needs to run to completion. The space is measured by counting the maximum memory needed by the algorithm in terms of variables used by algorithm in terms of variables used by algorithm.

Asymptotic Analysis

Asymptotic analysis refers to the computing of the running time of any piece of code or the operation in a mathematical unit of a computation. Its operation is computed in terms of a function like f(n). In mathematical analysis, asymptotic analysis, also known as asymptotics, is a method of describing limiting behavior.
The time required by the algorithm falls under the three types: Worst case - Maximum time required by an algorithm and it is mostly used or done while analyzing the algorithm. Best case - Minimum time required for the algorithm or piece of code and it is not normally calculated while analyzing the algorithm. Average case - Average time required for an algorithm or portion of code and it is sometimes done while analyzing the algorithm.

Asymptotic notation

The commonly used notation for calculating the running time complexity of the algorithm is as follows:

- Big O notation
- Big θ notation
- Big Ω notation

Big Oh Notation, O

Big O is used to measure the performance or complexity of an algorithm. In more mathematical term, it is the upper bound of the growth rate of a function, or that if a function g(x) grows no faster than a function f(x), then g is said to be a member of O(f).In general, it is used to express the upper bound of an algorithm and which gives the measure for the worst time complexity or the longest time an algorithm possibly take to complete.

Big Omega Notation, Ω

The notation Ω(n) is the formal way to express the lower bound of an algorithm's running time. It measures the best case time complexity or the best amount of time an algorithm can possibly take to complete.

Big Theta Notation, θ

The notation θ(n) is the formal way to express both the lower bound and the upper bound of an algorithm's running time.

Order of growth

An order of growth is a set of functions whose asymptotic growth behavior is considered equivalent. For example, 2n, 100n and n+1 belong to the same order of growth, which is written O(n) in Big-Oh notation and often called linear because every function in the set grows linearly with n.

All functions with the leading term n2 belong to O(n2); they are quadratic, which is a fancy word for functions with the leading term n2.

The following table shows some of the orders of growth that appear most commonly in algorithmic analysis, in increasing order of badness.

Order of Growth	Name
O(1)	Constant
O(logb n)	logarithmic (for any b)
O(n)	Linear
O(n logb n)	"en log en"
O(n2)	Quadratic
O(n3)	Cubic
O(cn)	exponential (for any c)

For the logarithmic terms, the base of the logarithm doesn't matter; changing bases is the equivalent of multiplying by a constant, which doesn't change the order of growth. Similarly, all exponential functions belong to the same order of growth regardless of the base of the exponent. Exponential functions grow very quickly, so exponential algorithms are only useful for small problems.

Arrays:

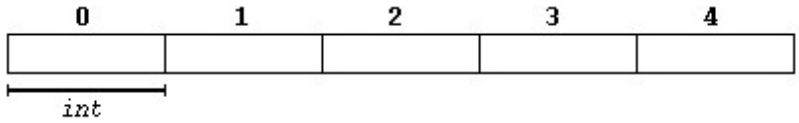
An array is defined as an ordered set of similar data items. All the data items of an array are stored in consecutive memory locations in RAM. The elements of an array are of same data type and each item can be accessed using the same name.

Declaration of an array:- We know that all the variables are declared before they are used in the program. Similarly, an array must be declared before it is used. During declaration, the size of the array has to be specified. The size used during declaration of the array informs the compiler to allocate and reserve the specified memory locations.

Syntax:- data_type array_name[n]; where, n is the number of data items (or) index(or) dimension. 0 to (n-1) is the range of array. Ex: int a[5]; float x[10];

An array is a collection of elements of the same type placed in contiguous memory locations that can be individually referenced by using an index to a unique identifier. Five values of type int can be declared as an array without having to declare five different variables (each with its own identifier).

For example, a five element integer array arr_name may be logically represented as;



where each blank panel represents an element of the array. In this case, these are values of type int. These elements are numbered from 0 to 4, with 0 being the first while 4 being the last; In C, the index of the first array element is always zero. As expected, an n array must be declared prior its use. A typical declaration for an array in C is:
Note: When an array is declared it contains garbage values.
Data_Type array_name[no. of elements in integer]= {value1, value2, value3,….};

Int arr_name[4]={1,2,3,4,5};

TYPES OF ARRAYS

- **One dimensional array**
- **Multi dimensional array**
  - **Two dimensional array**
  - **Three dimensional array**
  - **four dimensional array etc…**

One dimensional array:
A one-dimensional array (or single dimension array) is a type of linear array. Accessing its elements involves a single subscript which can either represent a row or column index. Conceptually you can think of a one-dimensional array as a row, where elements are stored one after another
float temp[20]; // temp is an array of type float, which can only store 20 elements of type float.

For example: The following program uses for loop to take input and print elements of a 1-D array.

```
#include<stdio.h>

int main()
{
   int arr[5], i;

   for(i = 0; i < 5; i++)
   {
      printf("Enter a[%d]: \n", i);
      scanf("%d", &arr[i]);
   }

   printf("\nPrinting elements of the array: \n\n");

   for(i = 0; i < 5; i++)
   {
      printf("%d ", arr[i]);
   }

   // signal to operating system program ran fine
   return 0;
}
```

Two dimensional array: An array consisting of two subscripts is known as two-dimensional array. These are often known as array of the array. In two dimensional arrays the array is divided into rows and columns. These are well suited to handle a table of data.

In 2-D array we can declare an array as :

Declaration:- Syntax: data_type array_name[row_size][column_size];

Ex:- int arr[3][3];

where first index value shows the number of the rows and second index value shows the number of the columns in the array.

Initializing two-dimensional arrays: Like the one-dimensional arrays, two-dimensional arrays may be initialized by following their declaration with a list of initial values enclosed in braces.

Ex: int a[2][3]={0,0,0,1,1,1};

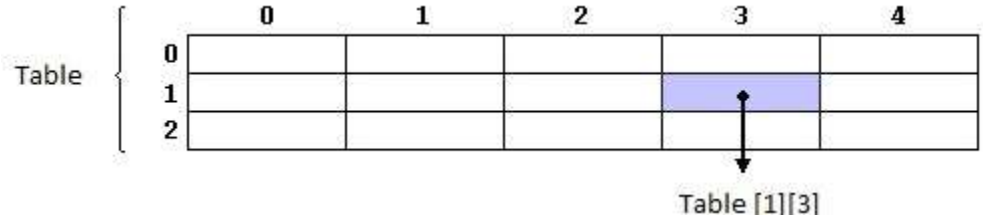
initializes the elements of the first row to zero and the second row to one. The initialization is done row by row. The above statement can also be written as

int a[2][3] = {{ 0,0,0},{1,1,1}};

by surrounding the elements of each row by braces. We can also initialize a two-dimensional array in the form of a matrix as shown below

```
int a[2][3]={
                {0,0,0},
                {1,1,1}
            };
```

MEMORY REPRESENTATION OF 2-D ARRAY FOR REFRENCE ONLY:



(remember that array indices always begin with zero).

Multidimensional arrays are not limited to two indices (i.e., two dimensions). They can contain as many indices as needed. Although be careful: the amount of memory needed for an array increases exponentially with each dimension. For example: char century [100][365][24][60][60];

These are stored in the memory as given below.

- Row-Major order Implementation
- Column-Major order Implementation

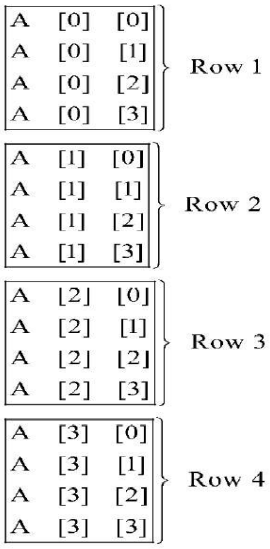


Fig. 2.4(a) Row major order

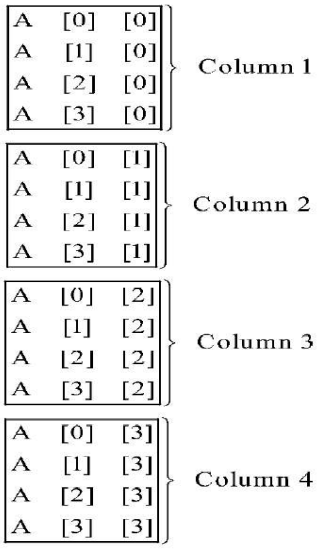
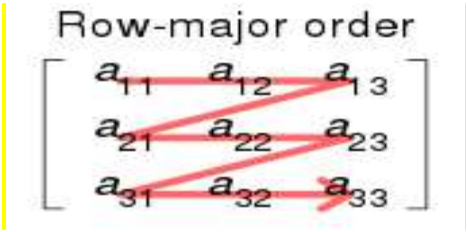


Fig. 2.4(b) Column major order

• In Row-Major Implementation of the arrays, the arrays are stored in the memory in terms of the row design, i.e. first the first row of the array is stored in the memory then second and so on.
Suppose we have an array named arr having 3 rows and 3 columns then it can be stored in the memory in the following manner :
int arr[3][3];

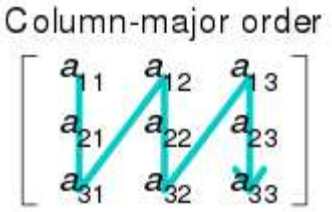
arr[0][0]	arr[0][1]	arr[0][2]
arr[1][0]	arr[1][1]	arr[1][2]
arr[2][0]	arr[2][1]	arr[2][2]

arr[0][0]
arr[0][1]
arr[0][2]
arr[1][0]
arr[1][1]
arr[1][2]
arr[2][0]
arr[2][1]
arr[2][2]
Thus an array of 3*3 can be declared as follows :
arr[3][3] = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
and it will be represented in the memory with row major implementation as follows :



1	2	3	4	5	6	7	8	9

In Column-Major Implementation of the arrays, the arrays are stored in the memory in the term of the column design, i.e. the first column of the array is stored in the memory then the second and so on. By taking above eg. we can show it as follows :
arr[3][3] = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
and it will be represented in the memory with column major implementation as follows :



1	4	7	2	5	8	3	6	9

Calculating the Address of the random element of an array:

SINGLE DIMENSIONAL ARRAY

In single dimensional array, all the elements are required to reference by an index of length n and the elements of the array are stored in successive memory locations. For example, int A [20], where A is the name of the array and int is the data type and 20 represents the length or size of array.
The number of data elements of the array can be obtained from the index set by the formula:

Number of elements = Upper bound – Lower bound + 1

and the lower bound is 0 because array index starts from zero, for example, A [0], so
number of elements = upper bound + 1.

Representation of Single Dimensional Array in Memory

Let A be a linear or one dimensional array with n elements. As we know array is stored in consecutive memory location so we have no need to keep the location of every element of A but we need to keep only the location of the first element. The location or address of the first element is also called the base address and denoted as Base [A]. Using the base address, we can find out the address of the kth element with the help of the following formula:

loc (A[k]) = Base (A) + w * k

where w is the number of words per memory cell.

Example 1. Consider an array A is declared as array of integers with size 50 and its first element is stored at memory address 101. So find out the address of 5th element.
Given w = 4 words per memory cell.

Solution:
Base (A) = 101
w = 4 words/ memory cell
By using the formula,
loc (A [k]) = Base (A) + w * k
loc (A[5]) = 101 + 4 * 5
= 101 + 20
= 121 **Answer.**

Example 2. Consider an array A, which records the number of T.V sold each year from 1975 to 2000. Suppose the Base (A) = 101 and w = 4 words per memory cell then find out the location of loc (A [1990]).

Solution:

Base (A) = 101
w = 4 words/memory cell
The location loc A[1975] = 101 i.e., A [0] = 101
By using the formula
loc (A[k]) = Base (A) + w * k
So we have to find out the address of the index 1990 – 1975 = 15
So loc (A[15]) = 101 + 4 * 15

= 161 **Answer.**

2-D Array formula

Due to the fact that, there are two different techniques of storing the two dimensional array into the memory, there are two different formulas to calculate the address of a random element of the 2D array.

By Row Major Order

The computer does not keep track of all the elements of the array, it keeps only the base address i.e., address of the first element and we can calculate the address of any element when required. The address of the element A[i] [j] can be calculated in row major-order by using the following formula:

LOC A[i] [j] = Base(A) + w [n (i – L1) + (j – L2)]

where w denotes the number of words per memory cell, n is the number of columns. L1 is

the lower bound of row and L2 is the lower bound of column.

By Column Major Order

The address of the element A[i] [j] can be calculated in column-major order by using the following formula:

LOC A [i] [j] = Base (A) + m [w (j – L2) + (i – L1)]

where m is the number of rows

Example 1. A two-dimensional array is defined as A[3 : 7, – 1 : 4] requires 4 words per memory cell. Find the location of A [5,2] if the array is implemented in row major order. The base address is given as 200.

Solution:

Base (A) = 200
w = 4 words per memory cell
i = 5 and j = 2
Lower bound of row L1 = 3
Lower bound of column L2 = – 1
Upper bound of row U1 = 7
Upper bound of column U2 = 4
Number of columns (n) = U2 – L2 + 1
= 4 – (–1) + 1
= 4 + 1 + 1 = 6
For row-major order, the formula for calculating the location of the elements is given as
LOC A[i] [j] = Base (A) + w [n (i – L1) + (j – L2)]
LOC A [5][2] = 200 + 4 [6(5 –3) + (2 – (–1)]
= 200 + 4 [6 × 2 + 3]
= 200 + 4 [15]

= 260 **Answer.**

Example 2. Consider a two-dimensional array A [20][5] of base address 200 and w = 4 words per memory cell. Then find out the address of A[12, 3] using column major order in C language.
Solution:
Base (A) = 200
w = 4 words/ memory cell
i = 12, j = 3
Number of rows (m) = 20
Number of columns(n) = 5

For column-major order, the formula for calculating the location of A[i] [j] is given as
LOC A[i] [j] = Base(A) + w [m (j – L2) + (i – L1)
Here L1 and L2 are the lower bound of row and column respectively and in C language as index starts from 0 so L1 = L2 = 0
LOC A[12] [3] = 200 + 4 [20 (3 – 0) + (12 – 0)
= 200 + 4 [60+12]
= 200 + 4 [72]
= 200 + 288

= 488 **Answer.**

Example 3. Each element of an array Data [20] [50] requires 4 bytes of storage. Base address of Data is 2000. Determine the location of Data [10] [10] when the array is stored as
(i) Row-major
(ii) Column-major
Solution:
Base (Data) = 2000
w = 4 bytes L1 = L2 = 0
i = 10, j = 10
Number of rows (m) = 20
Number of columns (n) = 50
(i) For row-major order, the formula for calculating the location of A[i][j] is given as
LOC A[i] [j] = Base (A) + w[(i – L1) + (j – L2)]
So LOC Data [10] [10] = 2000 + 4[50(10 – 0) + 10 – 0]
= 2000 + 4[50 × 10 + 10]
= 2000 + 4[510]
= 2000 + 2040
= 4040 Answer.
(ii) For column-major order, the formula for calculating the location of A[i][j] is given as
LOC A[i] [j] = Base (A) + w[m(i – L1) + (j – L2)]
So LOC Data [10] [10] = 2000 + 4[20(10 – 0) + 10 – 0]
= 2000 + 4[20 × 10 + 10]
= 2000 + 4 [210]
= 2000 + 840

= 2840 **Answer.**

APPLICATIONS OF ARRAY

The general applications of array are given as follows:
1. Array addition
2. Array subtraction
3. Array multiplication
4. Transpose of an array
5. Addition of rows and columns
6. To check whether the given matrix is sparse or not
7. To maintain polynomials in memory

Sparse Matrices

A matrix A is said to be sparse if more number of elements has value zero and similarly, the dense matrix has more number of non-zero elements.

Example:

$$A = \begin{bmatrix} 1 & 2 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 2 & 0 & 0 & 0 \\ 0 & 0 & 1 & 3 \end{bmatrix} \text{ or } A = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 2 & 0 & 0 \\ 1 & 2 & 3 & 0 \\ 1 & 2 & 3 & 4 \end{bmatrix} \text{ or } A = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 3 \end{bmatrix}$$

Representation of sparse matrices in memory

If we use the two-dimensional array representation method for representing the sparse matrices in memory then it is wastage of memory i.e., one may save space by storing only those entries which may be non-zero.

There are two ways for representing the sparse matrices in memory and they are:

(1) 3-Tuple Representation

The non-zero elements of the sparse matrix can be represented as a list of 3-tuples of the form (*i*, *j*, value) representing either row-major order or column-major order, where *i* represents rows and *j* represents columns.

Example: Consider a sparse matrix as

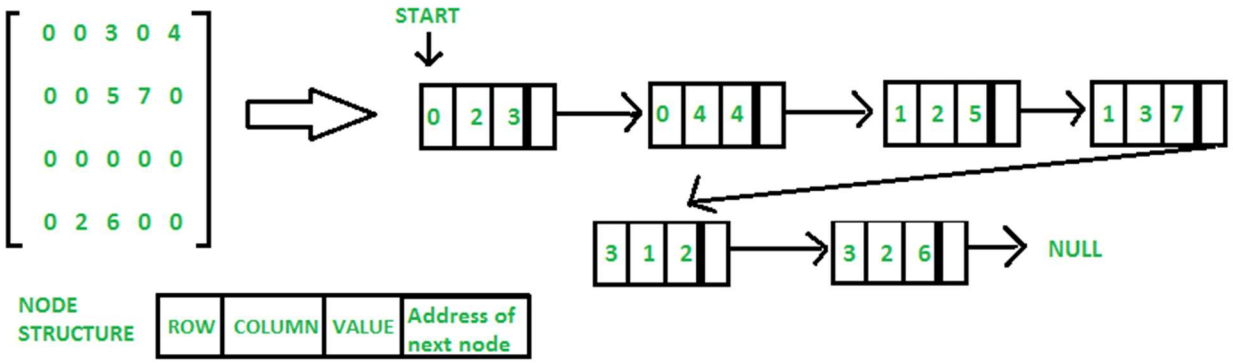
$$A = \begin{bmatrix} 10 & 0 & 0 & 0 & 5 \\ 0 & 6 & 0 & 0 & 0 \\ 0 & 0 & 7 & 0 & 2 \\ 0 & 0 & 0 & 9 & 8 \\ 0 & 0 & 2 & 0 & 0 \end{bmatrix}_{5 \times 5}$$

This matrix may be stored in memory in row-major order as shown:

i	*j*	*value*
0	0	10
0	4	5
1	1	6
2	2	7
2	4	2
3	3	9
3	4	8
4	2	2

(2) As Linked list:

Suppose a triangular array is placed in memory and is given as



Linked list:
A linked list is a linear collection of data elements called *nodes*, where the linear order is given by means of pointers. The node in linear linked list is divided into two parts— The first part contains the information and second part contains the address of the next node in the list. In linked list, data can be inserted or deleted from any position.
Example: The following figure shows a linear linked list with 5 nodes:

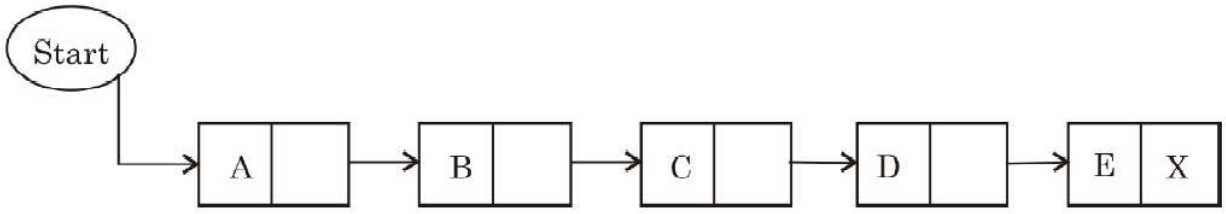


Fig. 1.3 Linear linked list

Basic Operations on Linked List

- Traversal: To traverse all the nodes one after another.
- Insertion: To add a node at the given position.
- Deletion: To delete a node.
- Searching: To search an element(s) by value.
- Updating: To update a node.
- Sorting: To arrange nodes in a linked list in a specific order.
- Merging: To merge two linked lists into one.

Implementation of Singly Linked Lists

```
:
//*************************************************************************//
//*********************  SINGLE LINEAR LINKED LIST  *********************//
//*************************************************************************//

#include<stdio.h>
#include<alloc.h>
#include<conio.h>
#include<process.h>

//*********************  DECLARATION OF STRUCTURE  *********************//
//*************************************************************************//

typedef struct simplelink
{
int data;
struct simplelink *next;
}node;

//*********************  FUN OF CREATE FIRST NODE  *********************//
//*************************************************************************//

node *create(node *p)
{
p=(node *)malloc(sizeof(node));
printf("Enter the data:");
scanf("%d",&p->data);
p->next=(node *)NULL;
return(p);
}

//*********************  FUN OF TRAVERSE IN ORDER  *********************//
//*************************************************************************//

void display(node *p)
{
printf("\n\n");
while(p!=NULL)
{
printf("[%d,%u]-->",p->data,p->next);
p=p->next;
}
printf("NULL");
printf("\n\n\npress any key to continue.....");
getch();
```

```
}

//***************  FUN OF TRAVERSE IN  REVERSE ORDER  ******************//
//*********************************************************************//

void revdisplay(node *p)
{
int a;
printf("\n\n");
node *brr[25];
int arr[25];
a=0;
while(p!=NULL)
{
//printf("\n\n[%d,%u]-->",p->data,p->next);
arr[a]=p->data;
brr[a]=p->next;
p=p->next;
a++;
}
//printf("NULL");
for(int i=(a-1);i>=0;i--)
printf("[%d,%u]<--",arr[i],brr[i]);
printf("HEAD");
printf("\n\n\npress any key to continue.....");
getch();
}

//**********************  FUN OF INSERT AT BEGINING  **********************//
//*********************************************************************//

node *insert_begin(node *p)
{
node *temp;
temp=(node *)malloc(sizeof(node));
printf("\nEnter the inserted data:");
scanf("%d",&temp->data);
temp->next=p;
p=temp;
return(p);
}

//************************  FUN OF INSERT AT END  ************************//
//*********************************************************************//

node *insert_end(node *p)
{
node *temp,*q;
q=p;
temp=(node *)malloc(sizeof(node));
```

```
printf("\nEnter the inserted data:");
scanf("%d",&temp->data);
while(p->next!=NULL)
{
p=p->next;
}
p->next=temp;
temp->next=(node *)NULL;
return(q);
}

//********************  FUN OF INSERT AFTER ELEMENT  **********************//
//*************************************************************************//

node *insert_after(node *p)
{
node *temp,*q;
int x;
q=p;
printf("\nEnter the data(after which you enter data):");
scanf("%d",&x);
while(p->data!=x)
{
p=p->next;
}
temp=(node *)malloc(sizeof(node));
printf("\nEnter the inserted data:");
scanf("%d",&temp->data);
temp->next=p->next;
p->next=temp;
return(q);
}

//********************  FUN OF COUNT THE NO. OF NODES  ********************//
//*************************************************************************//

int count(node *p)
{
int i=0;
while(p!=NULL)
{
p=p->next;
i++;
}
return(i);
}

//******************  FUN OF INSERT AT SPECIFIC POSITION  ******************//
//*************************************************************************//
```

```
node *insert_at_spe_pos(node *p)
{
node *temp,*q,*r;
int x,a,i=1;
a=count(p);
q=p;
printf("\nEnter the position(at which you want to enter data):");
scanf("%d",&x);
if(x>(a+1))
{
printf("Not able to insert node at such position :");
getch();
}
else
{
if(x==1)
{
temp=(node *)malloc(sizeof(node));
printf("\nEnter the inserted data:");
scanf("%d",&temp->data);
temp->next=p;
q=temp;
}
else
{
while(i!=x)
{
r=p;
p=p->next;
i++;
}
temp=(node *)malloc(sizeof(node));
printf("\nEnter the inserted data:");
scanf("%d",&temp->data);
temp->next=p;
r->next=temp;
}
}
return(q);
}

//*********************  FUN OF DELETE LAST NODE  **********************//
//*************************************************************************//

node *delend(node *p)
{
node *q,*r;
r=p;
q=p;
if(p->next==NULL)
```

```
{
r=(node *)NULL;
}
else
{
while(p->next!=NULL)
{
q=p;
p=p->next;
}
q->next=(node *)NULL;
}
free(p);
return(r);
}

//******************  FUN OF DELETE SPECIFIC ELEMENT  ******************//
//*************************************************************************//

node *del_speci_ele(node *p)
{
node *q,*r;
int x;
q=p;
r=q;
printf("\nEnter the data to remove:");
scanf("%d",&x);
while(q->data!=x)
{
r=q;
q=q->next;
}
if(q==r)
p=p->next;
else
r->next=q->next;
free(q);
return(p);
}

//**********************  FUN OF DELETE FIRST NODE  **********************//
//*************************************************************************//

node *delbegin(node *p)
{
node *q;
q=p;
p=p->next;
free(q);
return(p);
```

```
}

//*****************  FUN OF DELETE NODE AFTER ELEMENT  *****************//
//*************************************************************************//

node *delete_after(node *p)
{
node *temp,*q;
int x;
q=p;
printf("\nEnter the data(after which you want to delete):");
scanf("%d",&x);
while(p->data!=x)
{
p=p->next;
}
temp=p->next;
p->next=temp->next;
free(temp);
return(q);
}

//**************  FUN OF DELETE NODE AT SPECIFIC POSITION  ****************//
//*************************************************************************//

node *delete_at_spe_pos(node *p)
{
node *temp,*q,*r;
int x,a,i=1;
a=count(p);
q=p;
printf("\nEnter the position(at which you want to remove data):");
scanf("%d",&x);
if(x>(a))
{
printf("Not able to remove node at such position :");
getch();
}
else
{
if(x==1)
{
q=q->next;
free(p);
}
else
{
while(i!=x)
{
r=p;
```

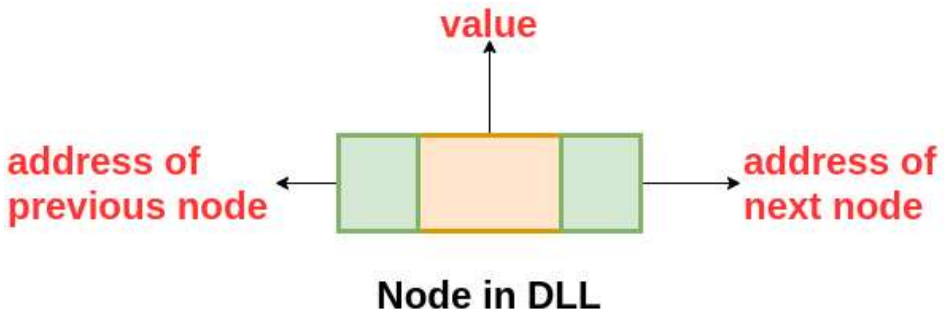
```
p=p->next;
i++;
}
r->next=p->next;
free(p);
}
}
return(q);
}

//*********************  FUN OF REVERSE THE LIST  **********************//
//*************************************************************************//

node *reverse(node *p)
{
node *q,*r;
q=(node *)NULL;
while(p!=NULL)
{
r=q;
q=p;
p=p->next;
q->next=r;
}
return(q);

}

//*************************  FUN OF MAIN SCREEN  *************************//
//*************************************************************************//

void screen(node *head)
{
clrscr();
int ch,a;
printf("\t\t\t     SINGLE LINEAR LINKED LIST");
printf("\n\t\t\t**********************************");
printf("\n\n  OPTIONS ARE--:");
printf("\n~~~~~~~~~~~~~~~~~~");
printf("\n\n 0-Exit");
printf("\n\n 1-Create first node");
printf("\n\n 2-Insert at begining");
printf("\n 3-Insert at end");
printf("\n 4-Insert after element");
printf("\n 5-Insert at specific position");
printf("\n\n 6-Delete at end");
printf("\n 7-Delete at begining");
printf("\n 8-Delete after element");
printf("\n 9-Delete specific element");
printf("\n10-Delete element from specific position");
```

```
printf("\n\n11-Traverse in order(Display)");
printf("\n12-Traverse in reverse order(Display)");
printf("\n\n13-Count no. of node");
printf("\n\n14-Reversed linked list");
printf("\n--------------------------------------------------------------------------------");
printf("\n\nEnter the choice:");
//printf("\n");
scanf("%d",&ch);
printf("\n--------------------------------------------------------------------------------");
switch(ch)
{
case 0:
	exit(0);
case 1:
	head=create(head);
	//	display(head);
	break;
case 2:
	head=insert_begin(head);
	break;
case 3:
	head=insert_end(head);
	break;
case 4:
	head=insert_after(head);
	break;
case 5:
	head=insert_at_spe_pos(head);
	break;
case 6:
	head=delend(head);
	break;
case 7:
	head=delbegin(head);
	break;
case 8:
	head=delete_after(head);
	break;
case 9:
	head=del_speci_ele(head);
	break;
case 10:
	head=delete_at_spe_pos(head);
	break;
case 11:
	display(head);
	break;
case 12:
	revdisplay(head);
	break;
```

```
case 13:
        a=count(head);
        printf("The no. of node in list-: %d",a);
        printf("\n\n\npress any key to continue.....");
        getch();
        break;
case 14:
        head=reverse(head);
        break;
default:
        printf("\n\n\nPlease enter the right choice.....");
        getch();
        break;
}
screen(head);
}

//*************************  DECLARATION OF MAIN  ***************************//
//*************************************************************************//

void main()
{
clrscr();
node *head;
//int a,ch;
head=(node *)NULL;
screen(head);
getche();
}
```

Doubly Linked List

A Doubly Linked List contains an extra memory to store the address of the previous node, together with the address of the next node and data which are there in the singly linked list. So, here we are storing the address of the next as well as the previous nodes.



Node in DLL

Advantages over Singly Linked List-

- It can be traversed both forward and backward direction.
- The delete operation is more efficient if the node to be deleted is given. (Think! you will get the answer in the second half of this blog)
- The insert operation is more efficient if the node is given before which insertion should take place. (Think!)

Disadvantages over Singly Linked List-

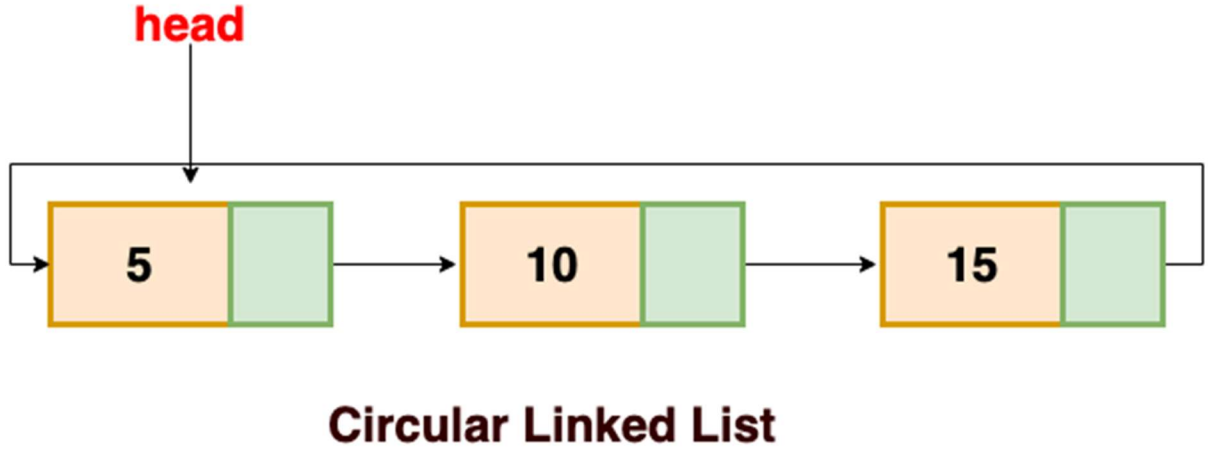
- It will require more space as each node has an extra memory to store the address of the previous node.
- The number of modification increase while doing various operations like insertion, deletion, etc.

```
/* CREATING A SIMPLE DOUBLY LINKED LIST */
/* DBLINK.C */
```

```
# include <stdio.h>
# include <malloc.h>

struct Double
{
        int info;
        struct Double *next;
        struct Double *previous;
};

int num ;
struct Double start;
void Doubly_link_list (struct Double *);
void display (struct Double *);

/* Function creates a simple doubly linked list */

void Doubly_link_list(struct Double *node)
{
        char ch;
        start.next = NULL;  /* Empty list */
        start.previous = NULL;
        node = &start;      /* Point to the start of the list */
        num = 0;
        printf("\n Input choice n for break: ");
        ch = getchar();

        while( ch != 'n')
        {
                node->next = (struct Double *) malloc(sizeof(struct Double));
                node->next->previous = node;
                node = node->next;
                printf("\n Input the values of the node : %d: ", (num+1));
                scanf("%d", &node->info);
                node->next = NULL;
                fflush(stdin);
                printf("\n Input choice n for break: ");
                ch = getchar();
                num ++;
        }
        printf("\n Total nodes = %d", num);
}

/* Display the list */

void display (struct Double *node)
{
        node = start.next;
```

```
        do {
                printf("\n 0x%x", node);
                printf(" %d", node->info);
                node = node->next;
        } while (node->next);  /* Show value of last node only one time */

        do {
                printf("\n 0x%x", node );
                printf(" %d", node->info);
                node = node->previous;
        } while (node->previous);
}

void main()
{
        struct Double *node = (struct Double *) malloc(sizeof(struct Double));
        Doubly_link_list(node);
        printf("\n Created doubly linked list is as follows\n");
        display(node);
}
```

Circular list: A circular linked list is either a singly or doubly linked list in which there are no NULL values. Here, we can implement the Circular Linked List by making the use of Singly or Doubly Linked List. In the case of a singly linked list, the next of the last node contains the address of the first node and in case of a doubly-linked list, the next of last node contains the address of the first node and prev of the first node contains the address of the last node



Circular Linked List

Advantages of a Circular linked list

- The list can be traversed from any node.
- Circular lists are the required data structure when we want a list to be accessed in a circle or loop.
- We can easily traverse to its previous node in a circular linked list, which is not possible in a singly linked list. (**Think!)**

Disadvantages of Circular linked list

- If not traversed carefully, then we could end up in an infinite loop because here we don't have any ***NULL*** value to stop the traversal.
- Operations in a circular linked list are complex as compared to a singly linked list and doubly linked list like reversing a circular linked list, etc.

.

/*Implementation of Circular list*/

```
#include<stdio.h>
#include<conio.h>
#include<alloc.h>
```

```
typedef struct n
{
int data;
struct n *next;
}node;
node *head=NULL;

void insert_cir_end(node *h,int d)
{
node *temp;
temp=(node*)malloc(sizeof(node)) ;
temp->data=d;
if(head==NULL)
{
head=temp;
temp->next=head;
return;
}
while(h->next!=head)
 h=h->next;
temp->next=h->next;
h->next=temp;
}
//-----------------------------------------------------------------
void insert_cir_beg(node *h,int d)
{
node *temp;
temp=(node*)malloc(sizeof(node)) ;
temp->data=d;
if(head==NULL)
{
head=temp;
temp->next=head;
return;
}
while(h->next!=head)
 h=h->next;
temp->next=head;
head=temp;
h->next=head;
}
//-----------------------------------------------------------
node* find(node *h,int aft)
{
while(h->next!=head && h->data!=aft)
            h=h->next;
if(h->next==head && h->data!=aft)
 return (node*)NULL;
else
```

```
  return h;
}
//-------------------------------------------------------------
void insert_cir_after(node *h,int aft,int d)
{
node *temp,*loc;
loc=find(h,aft);
if(loc!=(node*)NULL)
{
temp=(node*)malloc(sizeof(node));
temp->data=d;
temp->next=loc->next;
loc->next=temp;
}
else
printf("\nElement Not found.");
getch();
}
//-----------------------------------------------------------------------
void insert_cir_sp_pos(node *h,int pos,int d)
{
node *temp,*loc;
int p=0;
while(h->next!=head && p<pos-1)
        {
        loc=h;
        p++;
        h=h->next;
        }
if((pos>p+1 && h->next==head) || pos< 0)
        {
        printf("\nPosition not exists.");
        getch();
        return;
        }
if((p+2)==pos)
{
loc=h;
}
temp=(node*)malloc(sizeof(node));
temp->data=d;
temp->next=loc->next;
if(pos==1)
        { h=head;
        while(h->next!=head)
                h=h->next;
        h->next=temp;
        head=temp;
        }
else
```

```
            loc->next=temp;

}
//-------------------------------------------------------------------
void display(node *h)
{
while(h->next!=head)
{
printf("%d ",h->data);
h=h->next;
}
printf("%d ",h->data);
}
//-------------------------------------------------------------------
void delete_cir_beg(node *h)
{
node *temp;
if(head==NULL)
{
 printf("\nList is empty");
 getch();
 return;
}
if(h->next==head)
  {
  printf("\nNode deleted.List is empty");
  getch();
  head=NULL;
  free(h);
  return;
  }
temp=head;
while(h->next!=head)
 h=h->next;
head=head->next;
h->next=head;
free(temp);
}
//-------------------------------------------------------------------
void delete_cir_end(node *h)
{
node *temp;
if(head==NULL)
{
 printf("\nList is empty");
 getch();
 return;
}
if(h->next==head)
  {
```

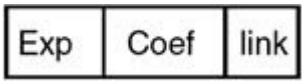
```
 printf("\nNode deleted.List is empty");
 getch();
 head=NULL;
 free(h);
 return;
 }
while(h->next!=head)
 {
 temp=h;
 h=h->next;
 }
temp->next=h->next;
free(h);
}
//-----------------------------------------------------------------
void delete_cir_after(node *h,int aft)
{
node *temp;
if(head==NULL)
{
 printf("\nList is empty");
 getch();
 return;
}
if(h->next==head && h->data==aft)
 {
 printf("\nNode deleted.List is empty");
 getch();
 head=NULL;
 free(h);
 return;
 }
temp=find(head,aft);
if(temp!=(node*)NULL)
{
 if(temp->next==temp)
 {
 printf("\nSingle Node.Deletion not possible");
 getch();
 return;
 }
 if(temp->next==head)
 {
  head=head->next;
  temp->next=head;
  free(h);
 }
 else
 {
 h=temp->next;
```

```
 temp->next=h->next;
 free(h);
 }
}   //end of if(!temp)
}
//-----------------------------------------------------------------
void free_list(node *list)
{
node *t;
while(list!=NULL)
 {
 t=list;
 list=list->next;
 free(t);
 }
}

//-----------------------------------------------------------------
void main()
{
int opt=0,val,pos,aft;
while(1)
{
clrscr();
printf("\nCircular List Operations.");
printf("\n\n*****************************************");
printf("\n1.Insert at begining.");
printf("\n2.Insert at end.");
printf("\n3.Insert after specific element.");
printf("\n4.Insert at specific position.");
printf("\n5.Delete from begining.");
printf("\n6.Delete from end.");
printf("\n7.Delete after specific element.");
printf("\n8.View the List.");
printf("\n9.Exit.");
printf("\n\nEnter the option.");
fflush(stdin);
scanf("%d",&opt);
switch(opt)
{
case 1: printf("\nEnter the data");
            fflush(stdin);
            scanf("%d",&val);
            insert_cir_beg(head,val);
            break;
case 2: printf("\nEnter the data");
            fflush(stdin);
            scanf("%d",&val);
            insert_cir_end(head,val);
            break;
```

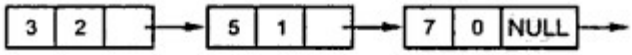
```
case 3: printf("\nEnter the data to be inserted.");
            fflush(stdin);
            scanf("%d",&val);
            printf("\nEnter the data after which to be inserted.");
            fflush(stdin);
            scanf("%d",&aft);
            insert_cir_after(head,aft,val);
            break;
case 4: printf("\nEnter the data");
            fflush(stdin);
            scanf("%d",&val);
            printf("\nEnter the position at for insertion.");
            fflush(stdin);
            scanf("%d",&pos);
            insert_cir_sp_pos(head,pos,val);
            break;
case 5: delete_cir_beg(head);
            break;
case 6: delete_cir_end(head);
            break;
case 7: printf("\nEnter the element after which is to be deleted.");
            fflush(stdin);
            scanf("%d",&aft);
            delete_cir_after(head,aft);
            break;
case 8: printf("\nElements in the list");
            display(head);
            getch();
            break;
case 9: free_list(head);
            exit(0);

}           //end of switch
}               //end of while
}               //end of main()
```

Polynomial Representation using Linked List: In each node the exponent field will store the corresponding exponent and the coefficient field will store the corresponding coefficient. Link field points to the next item in the polynomial.

Node of a Polynomial:



For example 3x^2 + 5x + 7 will represent as follows.

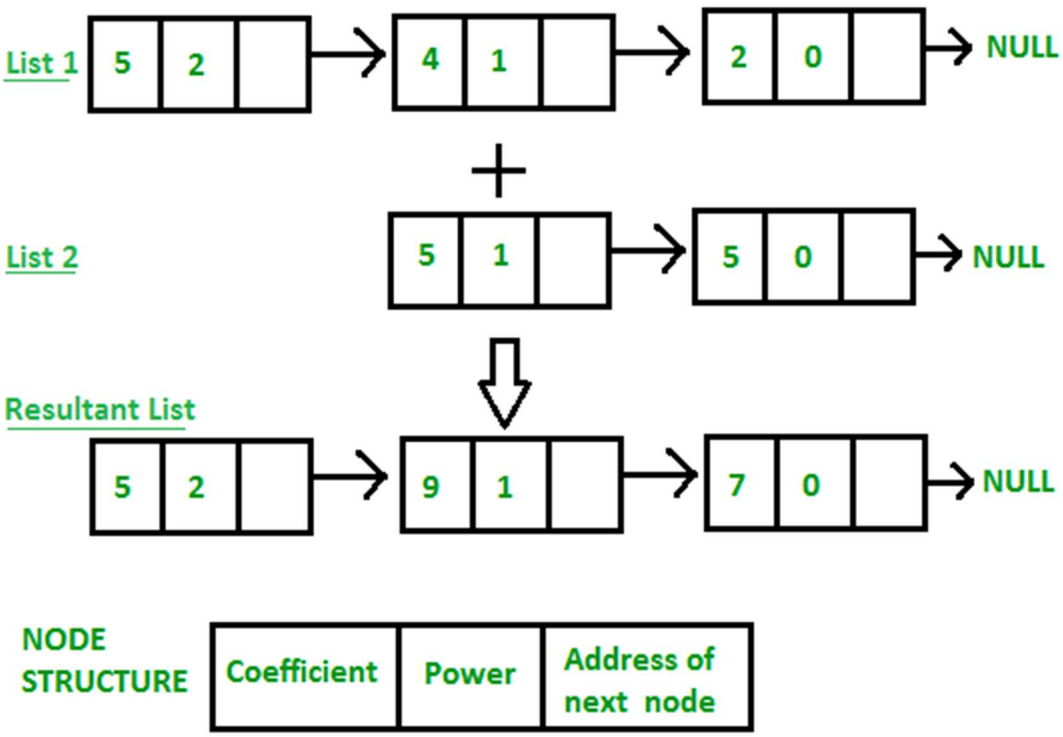


Structure will be:

```
struct Node {
    int coeff;
    int pow;
    struct Node* next;
};
```

For example:

```
Input:
     1st number = 5x² + 4x¹ + 2x⁰
     2nd number = -5x¹ - 5x⁰
Output:
        5x²-1x¹-3x⁰
Input:
     1st number = 5x³ + 4x² + 2x⁰
     2nd number = 5x^1 - 5x^0
Output:
        5x³ + 4x² + 5x¹ - 3x⁰
```



Code:

```
program for addition of two polynomials
// using Linked Lists
#include <bits/stdc++.h>
using namespace std;

// Node structure containing power and coefficient of
// variable
struct Node {
    int coeff;
    int pow;
    struct Node* next;
};

// Function to create new node
void create_node(int x, int y, struct Node** temp)
{
    struct Node *r, *z;
    z = *temp;
    if (z == NULL) {
        r = (struct Node*)malloc(sizeof(struct Node));
        r->coeff = x;
        r->pow = y;
        *temp = r;
        r->next = (struct Node*)malloc(sizeof(struct Node));
        r = r->next;
```

```
        r->next = NULL;
    }
    else {
        r->coeff = x;
        r->pow = y;
        r->next = (struct Node*)malloc(sizeof(struct Node));
        r = r->next;
        r->next = NULL;
    }
}

// Function Adding two polynomial numbers
void polyadd(struct Node* poly1, struct Node* poly2,
             struct Node* poly)
{
    while (poly1->next && poly2->next) {
        // If power of 1st polynomial is greater then 2nd,
        // then store 1st as it is and move its pointer
        if (poly1->pow > poly2->pow) {
            poly->pow = poly1->pow;
            poly->coeff = poly1->coeff;
            poly1 = poly1->next;
        }

        // If power of 2nd polynomial is greater then 1st,
        // then store 2nd as it is and move its pointer
        else if (poly1->pow < poly2->pow) {
            poly->pow = poly2->pow;
            poly->coeff = poly2->coeff;
            poly2 = poly2->next;
        }

        // If power of both polynomial numbers is same then
        // add their coefficients
        else {
            poly->pow = poly1->pow;
            poly->coeff = poly1->coeff + poly2->coeff;
            poly1 = poly1->next;
            poly2 = poly2->next;
        }

        // Dynamically create new node
        poly->next
            = (struct Node*)malloc(sizeof(struct Node));
        poly = poly->next;
        poly->next = NULL;
    }
    while (poly1->next || poly2->next) {
        if (poly1->next) {
            poly->pow = poly1->pow;
            poly->coeff = poly1->coeff;
            poly1 = poly1->next;
```

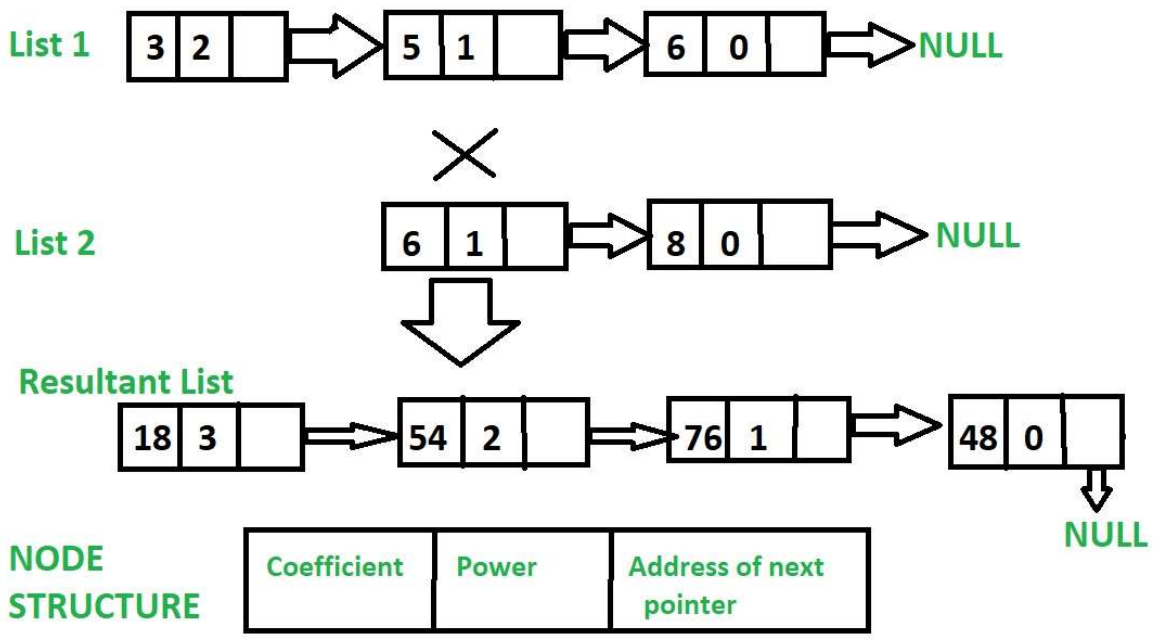
```
        }
        if (poly2->next) {
            poly->pow = poly2->pow;
            poly->coeff = poly2->coeff;
            poly2 = poly2->next;
        }
        poly->next
            = (struct Node*)malloc(sizeof(struct Node));
        poly = poly->next;
        poly->next = NULL;
    }
}

// Display Linked list
void show(struct Node* node)
{
    while (node->next != NULL) {
        printf("%dx^%d", node->coeff, node->pow);
        node = node->next;
        if (node->coeff >= 0) {
            if (node->next != NULL)
                printf("+");
        }
    }
}

// Driver code
int main()
{
    struct Node *poly1 = NULL, *poly2 = NULL, *poly = NULL;

    // Create first list of 5x^2 + 4x^1 + 2x^0
    create_node(5, 2, &poly1);
    create_node(4, 1, &poly1);
    create_node(2, 0, &poly1);

    // Create second list of -5x^1 - 5x^0
    create_node(-5, 1, &poly2);
    create_node(-5, 0, &poly2);

    printf("1st Number: ");
    show(poly1);

    printf("\n2nd Number: ");
    show(poly2);

    poly = (struct Node*)malloc(sizeof(struct Node));

    // Function add two polynomial numbers
    polyadd(poly1, poly2, poly);
```

```
    // Display resultant List
    printf("\nAdded polynomial: ");
    show(poly);

    return 0;
}
```

Multiplication of two polynomials using Linked list:

Input: Poly1: 3x^2 + 5x^1 + 6, Poly2: 6x^1 + 8
Output: 18x^3 + 54x^2 + 76x^1 + 48
On multiplying each element of 1st polynomial with
elements of 2nd polynomial, we get
18x^3 + 24x^2 + 30x^2 + 40x^1 + 36x^1 + 48
On adding values with same power of x,
18x^3 + 54x^2 + 76x^1 + 48

Input: Poly1: 3x^3 + 6x^1 - 9, Poly2: 9x^3 - 8x^2 + 7x^1 + 2
Output: 27x^6 - 24x^5 + 75x^4 - 123x^3 + 114x^2 - 51x^1 - 18



Approach:

1. In this approach we will multiply the 2nd polynomial with each term of 1st polynomial.

2. Store the multiplied value in a new linked list.
3. Then we will add the coefficients of elements having the same power in resultant polynomial.

Below is the implementation of the above approach:

Code:

```
/ C++ implementation of the above approach
#include <bits/stdc++.h>
using namespace std;

// Node structure containing powerer
// and coefficient of variable
struct Node {
    int coeff, power;
    Node* next;
};

// Function add a new node at the end of list
Node* addnode(Node* start, int coeff, int power)
{
    // Create a new node
    Node* newnode = new Node;
    newnode->coeff = coeff;
    newnode->power = power;
    newnode->next = NULL;

    // If linked list is empty
    if (start == NULL)
        return newnode;

    // If linked list has nodes
    Node* ptr = start;
    while (ptr->next != NULL)
        ptr = ptr->next;
    ptr->next = newnode;

    return start;
}

// Functionn To Display The Linked list
void printList(struct Node* ptr)
{
    while (ptr->next != NULL) {
        cout << ptr->coeff << "x^" << ptr->power ;
       if( ptr->next!=NULL && ptr->next->coeff >=0)
          cout << "+";

        ptr = ptr->next;
```

```
    }
    cout << ptr->coeff << "\n";
}

// Function to add coefficients of
// two elements having same powerer
void removeDuplicates(Node* start)
{
    Node *ptr1, *ptr2, *dup;
    ptr1 = start;

    /* Pick elements one by one */
    while (ptr1 != NULL && ptr1->next != NULL) {
        ptr2 = ptr1;

        // Compare the picked element
        // with rest of the elements
        while (ptr2->next != NULL) {

            // If powerer of two elements are same
            if (ptr1->power == ptr2->next->power) {

                // Add their coefficients and put it in 1st element
                ptr1->coeff = ptr1->coeff + ptr2->next->coeff;
                dup = ptr2->next;
                ptr2->next = ptr2->next->next;

                // remove the 2nd element
                delete (dup);
            }
            else
                ptr2 = ptr2->next;
        }
        ptr1 = ptr1->next;
    }
}

// Function two Multiply two polynomial Numbers
Node* multiply(Node* poly1, Node* poly2,
               Node* poly3)
{

    // Create two pointer and store the
    // address of 1st and 2nd polynomials
    Node *ptr1, *ptr2;
    ptr1 = poly1;
    ptr2 = poly2;
    while (ptr1 != NULL) {
        while (ptr2 != NULL) {
            int coeff, power;
```

```
            // Multiply the coefficient of both
            // polynomials and store it in coeff
            coeff = ptr1->coeff * ptr2->coeff;

            // Add the powerer of both polynomials
            // and store it in power
            power = ptr1->power + ptr2->power;

            // Invoke addnode function to create
            // a newnode by passing three parameters
            poly3 = addnode(poly3, coeff, power);

            // move the pointer of 2nd polynomial
            // two get its next term
            ptr2 = ptr2->next;
        }

        // Move the 2nd pointer to the
        // starting point of 2nd polynomial
        ptr2 = poly2;

        // move the pointer of 1st polynomial
        ptr1 = ptr1->next;
    }

    // this function will be invoke to add
    // the coefficient of the elements
    // having same powerer from the resultant linked list
    removeDuplicates(poly3);
    return poly3;
}

// Driver Code
int main()
{

    Node *poly1 = NULL, *poly2 = NULL, *poly3 = NULL;

    // Creation of 1st Polynomial: 3x^2 + 5x^1 + 6
    poly1 = addnode(poly1, 3, 3);
    poly1 = addnode(poly1, 6, 1);
    poly1 = addnode(poly1, -9, 0);

    // Creation of 2nd polynomial: 6x^1 + 8
    poly2 = addnode(poly2, 9, 3);
    poly2 = addnode(poly2, -8, 2);
    poly2 = addnode(poly2, 7, 1);
    poly2 = addnode(poly2, 2, 0);

    // Displaying 1st polynomial
```

```
    cout << "1st Polynomial:- ";
    printList(poly1);

    // Displaying 2nd polynomial
    cout << "2nd Polynomial:- ";
    printList(poly2);

    // calling multiply function
    poly3 = multiply(poly1, poly2, poly3);

    // Displaying Resultant Polynomial
    cout << "Resultant Polynomial:- ";
    printList(poly3);

    return 0;
}
```