Data Structures

Module - 1

Syllabus:

Introduction: Data structures, classifications (Primitive and Non-primitive), Data structure operations (Traversing, Inserting, Deleting, Searching & Sorting). Review of Arrays
Structures: Array of structures, self-referential structures
Dynamic Memory allocation functions, Representation of linear arrays in memory, dynamically allocated arrays & Multidimensional arrays.
Demonstration of representation of polynomials and Sparse Matrices with arrays.

Introduction to Data Structures:-

- Data is a value or a set of values. Data as such may not convey any meaning.
  Eg:- 90, Bob
- When data is interpreted to convey a meaning we call it an information
  Eg:- Bob scored 90 Marks.
- A data item refers to a single unit of values. Data items that are divided into sub items are called group items
  Eg:- Name of an employee can be divided into 3 sub items i.e, first name, middle name & last name.
- Data items that are not divided into sub items are called elementary items.

Data structures:-

- A data structure is a particular method of storing and organizing data in a computer so that it can be used efficiently.
- It is also a way of collecting and organizing data in such a way that we can perform operations on these data in an effective way

Classification of data structures :-

Data structures are generally categorized into 2 classes

1) Primitive data structures
2) Non - primitive data structures

1) Primitive data structures :-

- Primitive data structures are the fundamental data types which are supported by a programming language
- These can be manipulated directly by the machine instructions.
- Some of the basic data types are the examples of primitive data structure.

Eg :- Integer, real, character, boolean, float etc

2) Non - primitive data structures :-

- Non - primitive data structures are those data structure which are created using primitive data structures.
- They cannot be manipulated directly by the machine instructions.

  Eg :- Linked lists, stacks, trees and graphs.
- Based on the structure and arrangement of data, non - primitive data structures are further classified into as follows.

  i) Linear data structures
  ii) Non - linear data structures

i) Linear data structure :-

- If the elements of a data structure are stored in a linear or sequential order, then it is a linear data structure

Eg :- Arrays, linked list, stacks & queues.

- Linear data structures can be represented in memory in 2 different ways.
  * One way is to have a linear relationship between elements by means of sequential memory locations
  * The other way is to have a linear relationship between elements by means of links.

ii) Non-linear data structures :-

- If the elements of a data structure are not stored in a sequential order, then it is a non-linear data structure.

Eg :- Trees and Graphs.

- The relationship of adjacency is not maintained between elements of a non-linear data structure.
- This structure is mainly used to represent data containing a hierarchical relationship between elements

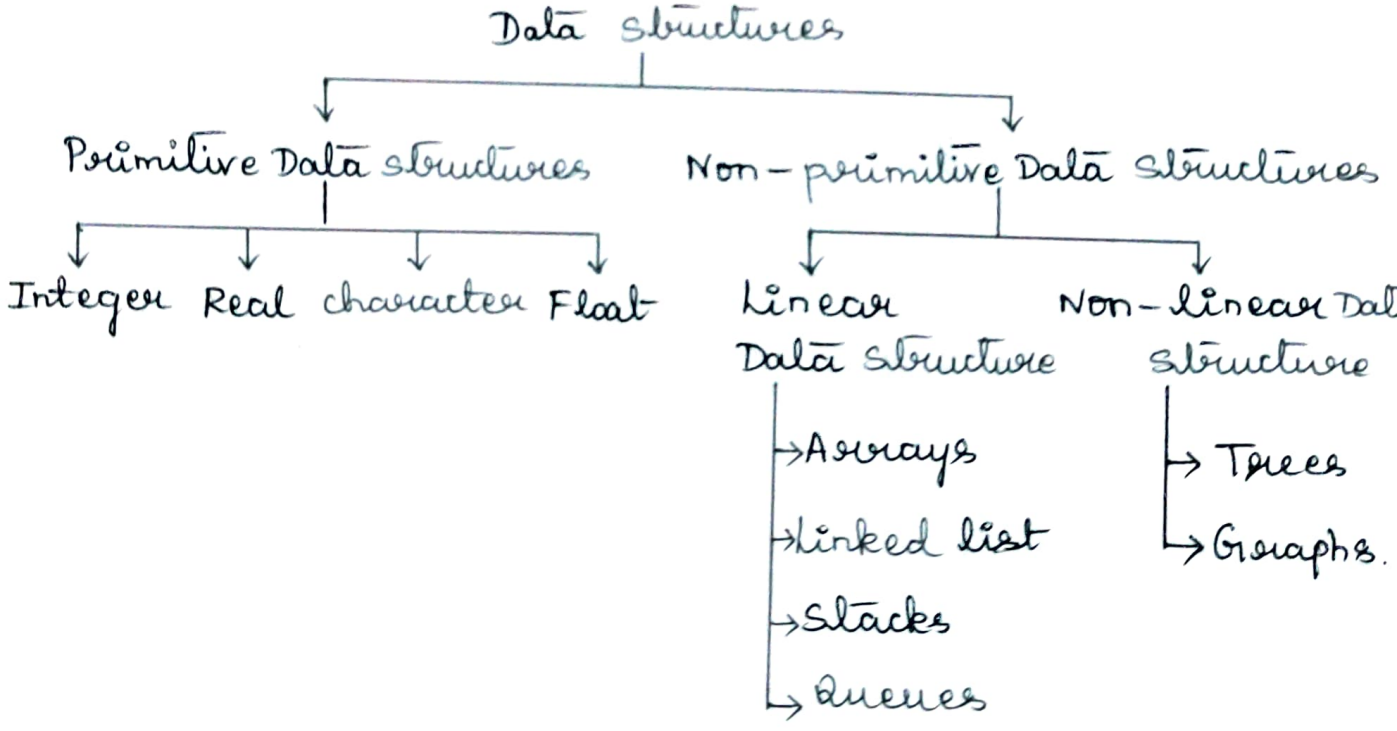


Figure - 1 :- Classification of Data Structures

Data Structure Operations :-

- Data are processed by means of certain operations which appearing in the data structure. This section introduces the reader to some of the most frequently used of these operations.

1) Traversing :-

- Accessing each record exactly once so that certain items in the record may be processed.
- This accessing and processing is sometimes called visiting the record.

2) Inserting :-

- Adding a new record to the structure

3) Deleting :-

- Removing the record from the structure

4) Searching :-

- Finding the location of the record with a given key value or finding the location of all the record which satisfy one or more conditions.

5) Sorting :-

- Arranging the data or record in some logical order either in ascending or descending order.

6) Merging :-

- Combining the record in two different sorted files into a single file.

Review of Arrays:-

Arrays:-

- Array is a collection of elements of the same data type
- All the elements of an array are stored in consecutive memory locations
- Each elements can be accessed using the same 'name' but different 'index' value.
- An array is a set of pairs, <index, value>, such that each index has a value associated with it. It can be called as corresponding or mapping.

Eg:- <index, value>

<0, 25> list[0] = 25
<1, 15> list[1] = 15
<2, 20> list[2] = 20
<3, 17> list[3] = 17

- Here, 'list' is the name of the array. By using list[0] to list[3] the elements in list can be accessed.

Declaration of arrays:-

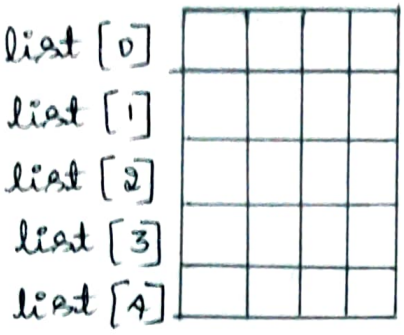
- A one dimensional array in C is declared by adding brackets to the name of a variable.

  Eg:- int list[5], *plist[5];

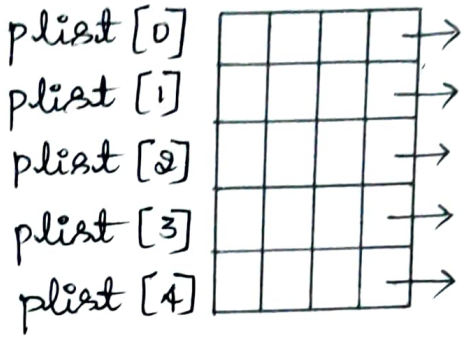
- Syntax:- date_type array_name[array_size];
- The array list[5], defines 5 integers and in C array start at index 0, so list[0], list[1], list[2], list[3] list[4] are the names of 5 array elements which contains an integer value.
- The array *plist[5], defines an array of 5 pointers to integers. where plist[0], plist[1], plist[2], plist[3]

plist [4] are the 5 array elements which contains a pointer to an integer.

int list [5]



int *plist [5]



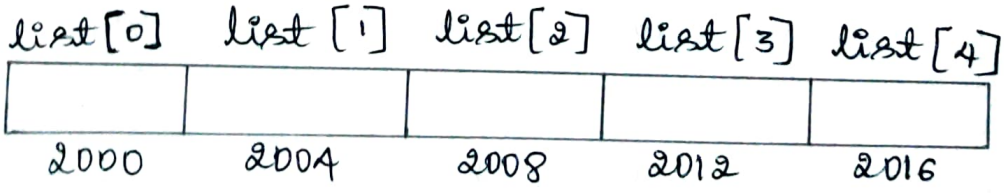
- When the compiler encounters an array declaration, list [5], it allocates 5 consecutive memory locations Each memory is large enough to hold a single intege
- Address of 1st element of an array is called 'Base address'

  Eg:- For list [5], the address of list [0] is called the base address.
- If the memory address of list [i] need to compute by the compiler, then the size of the 'int' would get by sizeof (int) i.e, 4 bytes, then memory address of list [i] is as follows:

$$list[i] = \alpha + i * sizeof(int)$$

where α = base address.

Eg :-



$$list[3] = \alpha + 3 * sizeof(int)$$

$$= 2000 + 3 * 4$$

$$list[3] = 2012$$

$\alpha \rightarrow$ address of 1st element i.e, list [0]

- Size of array defines the number of elements in an array
- First element is always numbered as '0' and so on. Array indexing always starts from 0 to n-1
  Eg:- int age[5];
  Here, the size of array is 5, so number of elements an array can store is 5.
- Arrays are generally used when we want to store large amount of similar type of data. But they have the following limitations:-
  i) Arrays are of fixed size
  ii) Data elements are stored in contiguous memory locatio which may not be always available.
  iii) Insertion & deletion of elements can be problematic because of shifting of elements from their positions

1) Program to find sum of 'n' numbers using array:-

```
#define MAX_SIZE 100
float sum (float [], int);
float input [MAX_SIZE], answer;
int i;
main()
{
  for (i=0; i< MAX_SIZE; i++)
  input [i] = i;
  answer = sum (input, MAX_SIZE);
  printf ("The sum is: %f\n", answer);
}
float sum (float list [], int n)
{ int i;
  float tempsum = 0;
  for (i=0; i<n; i++)
  tempsum += list [i];
  return tempsum;
}
```

Structures :-

- Structure is a user defined data type that can store related information that may be of same or different data types together.
- In C language, structures are used to group together different types of variables under the same name.
- To declare a structure, a keyword called 'struct' must be used.
- The struct keyword defines a new data type with more than one member
- Syntax for structure is as follows:

```
struct
{
  data_type member1;
  data_type member2;
  .
  .
  data_type membern;
} variable_name;
```

Eg :-

```
struct {
        char name[10];
        int age;
        float salary;
} Person;
```

→ members of a structure

The above example creates a structure and variable name is Person and that has 3 fields:

name → name that is a character array

age → an integer value representing the age of a person.

salary → a float value representing the salary of an individual.

- The major difference between a structure and an array is that, an array can store only information of same data type.
- A structure is therefore a collection of variables under a single name.
- The variables within a structure are of different types and each has a name that is used to select it from the structure.
- Structure tag may be include before concluding the definition or can be included in the main function and each data member is a normal variable definition.
- At the end of the structure's definition, a semicolon is mandatory.

Assign values to fields :-

- To assign the values to the fields, we use • (dot) as the structure operator.
- This operator is used to select a particular member of the structure.

Eg :-
```
strcpy (Person.name, "james");
Person.age = 10;
Person.salary = 35000;
```

Structure initialization :-

Method-1 :-

- Specify the initializers within the braces and separate by commas when the variables are declared as shown below

```
struct employee
{
  char name[20];
  int salary;
```

```
    int id;
} a = { "MONALIKA", 10950, 2001};
```

Initializers

Method 2:-

- Specify the initializers within the braces and separated by commas when the variables are declared as shown below

```
/* Structure definition */
Struct employee
{
  char name[20];
  int salary;
  int id;
};

/* structure declaration & initialization */
Struct employee a = { "MONALIKA", 10950, 2001};
```

- The members of the structure cannot be initialized in the definition. For example:

```
Struct employee
 {
   char name[20] = "RAMA";
   int salary = 20000;
   int id = 25;
 }
```

- During initialization, the number of initializers should not exceed the number of members. It leads to syntax errors. For example:

```
Struct employee a = { "Rama", 10950, 2001, 10.2};
```

For the above statement, compiler issues a syntax error saying 'too many initializers'.

- During partial initialization the values are assigned to members in the order specified & the remaining members are initialized with default values.

```
struct employee a = { "RAMA"};
```

In the above example, string "RAMA" will be assigned to structure member 'name'. The other members of the structure namely 'salary' & 'id' are initialized to zero by default.

- During initialization, there is no way to initialize members in the middle of a structure without initializing the previous members.

```
struct employee a = {10950, 2001};
```

Above example is invalid, because without initializing the first member we cannot initialize the last two members. In this case, the number 10950 will be copied to 1st member & the number 2001 will be copied to 2nd member.

Accessing structures :-

- The members of a structure can be accessed by specifying the variable followed by dot operator followed by the name of the member.

Eg :-
```
struct employee
{
  char name[20];
  int salary;
  int id;
} a = { "MITHIL", 10950, 2001};
```

- By specifying a.name we can access the name 'MITHIL'
- By specifying a.salary we can access the value 10950
- By specifying a.id we can access the value 2001.

- Programming statements to access the structures is as shown below:

```
printf("%.s\n", a.name);      output → MITHIL
printf("%d\n", a.salary);     output → 10950
printf("%d\n", a.id);         output → 2001
```

Type-Defined Structure :-

- The structure definition associated with keyword 'typedef' is called type-defined structure.

Syntax 1:-
```
typedef struct
{
  data_type member1;
  data_type member2;
  :
  data_type membern;
} Type_name;
```

where,

- typedef is the keyword added at the beginning of the definition and by using typedef user defined data type can be obtained.
- struct is the keyword which tells structure is defined to the compiler
- Members are declared with their datatype within the curly braces. They are also called as fields of structure
- Type_name is not a variable, it is user defined data type.

Eg :-
```
typedef struct student  → old type
{
  char name[10];
  int roll_number;
  float average_marks;
} STUDENT;  → new type
```

• In the above example, STUDENT is the type created by the user, it can be called as user-defined data type So, we can use STUDENT as data type & declare the variables. For example, consider the following declaration

```
STUDENT cse, ise;
```

STUDENT ← user defined data type
cse, ise ← structure variables

• This statement declares that the variables cse & ise are variables of type STUDENT.

Syntax 2:-

```
struct struct_name
{
    data_type member 1;
    data_type member 2;
    :
    data_type member n;
};
typedef struct struct_name Type_name;
```

• Here we use 'tag' for the structure & then we obtain the user-defined data type using the keyword typedef

Eg :-

```
struct student          ← Tag name
{
    char name[10];
    int roll_number;
    float average_marks;
};
typedef struct student STUDENT;
```

student → Tag name (old name)
STUDENT → user defined data type (new name)

• Consider the following declaration:

```
STUDENT cse, ise;
```

Above statement shows variables cse & ise are the variables of type STUDENT.

Structure Operations :-

- Various operations can be performed on structures and structure members.
- Any operation that can be performed on ordinary variable can also be performed on structure members. But some operations cannot be performed on structure variables.

1) Structure Equality check :- (or)

Comparison of 2 structure variables or members :-

- Equality or inequality check / comparison of 2 structure variable of same type or dissimilar type is not allowed

Eg :-
```
typedef struct {
        char name[10];
        int age;
        float salary;
    } humanBeing;
humanBeing person1, person2;
if (person1 == person2) is invalid.
```

- Members of 2 structure variables of same type can be compared using relational operators.

Eg :-
```
a.member1 == b.member2
a.member1 != b.member2
```
} Valid if both members have the same data type

- Arithmetic, relational, logical & other various operations can be performed on individual members of structures but not on structure variables.
- The 2 structures can be compared using the following function by assuming TRUE & FALSE as symbolic constants as shown:

```
int humansEqual(humanBeing person1, humanBeing person2)
{
 if (strcmp(person1.name, person2.name) != 0)
  return FALSE;
 if (person1.age != person2.age)
  return FALSE;
 if (person1.salary != person2.salary)
  return FALSE;
 return TRUE;
}
```

- Above function returns true if both employees have same information.

2) Assignment operation on structure variables :-

```
person1 = person2
```

- The above statement means that the value of every field of the structure of person2 is assigned as the value of the corresponding field of person1, but this is invalid statement.
- Valid statement is given below :

```
strcpy(person1.name, person2.name);
person1.age = person2.age;
person1.salary = person2.salary;
```

- Assigment operation can be performed only when both the structure's variables are of same data type.
- Copying of structure variables from one structure to another structure can also be performed only when variables are of same data type.

Nested Structures (or)

Structure within a structure :-

- There is a possibility to embed a structure within a structure. There are 2 ways to embed structure.

1) The structures are defined separately and a variable of structure type is declared inside the definition of another structure. as a member

- The accessing of the variable of a structure type that are nested inside another structure in the same way as accessing other member of that structure.

Example :-

The following example shows two structures, where both the structure are defined separately.

```
typedef struct {
            int month;
            int day;
            int year;
        } date;
typedef struct {
            char name[10];
            int age;
            float salary;
            date dob;
        } humanBeing;
humanBeing person1;
```

- A person born on Feb 11, 1990, would have the values for the date struct set as :

```
person1.dob.month = 2;
person1.dob.day = 11;    person1.dob.year = 1990;
```

2) The complete definition of a structure is placed inside the definition of another structure.

Example :-

```
typedef struct {
                char name[10];
                int age;
                float salary;
        struct {
                int month;
                int day;
                int year;
        } date;
} humanBeing;
```

Array of structures :-

- An array of structures in C can be defined as the collection of multiple structure variables where each variable contains information about different entities.
- The array of structures in C are used to store information about multiple entities of different data types.
- Array of structures is also known as the collection of structures.

Example :-

```
struct employee
{
  int id;
  char name[5];
  float salary;
};
struct employee emp[2];
```

Given example stores the information of 2 employees in one single structure, using arrays concept

- To declare an array of structure, first the structure must be defined and then an array variable of that type should be defined.

Eg:- struct book b[10]; // 10 elements in an array of structures of type 'book'.

- Example of an array of structures that stores information of 5 students and prints it.

```
#include <stdio.h>
#include <string.h>
struct student
{
   int rollno;
   char name[10];
};
int main()
{
 int i;
 struct student st[5];
 printf("Enter records of 5 students");
 for(i=0; i<5; i++)
{
 printf("Enter rollno\n");
 scanf("%d", &st[i].rollno);
 printf("Enter name\n");
 scanf("%s", &st[i].name);
}
 printf("\n student information list\n");
 for(i=0; i<5; i++)
{ printf("\n Rollno: %d, Name: %s", st[i].rollno,
                                    st[i].name);
} return 0;
```

Self-Referential Structures :-

- Self referential structures are those structures that have one or more pointers which point to the same type of structure as their member.
- In other words, structures pointing to the same type of structures are self-referential in nature.
- Self referential structures are widely used in dynamic data structures such as trees, linked lists etc.
- Pointer to a structure is similar to a pointer to any other variable.

Eg :-
```
struct node
  {
    int data1;
    char data2;
    struct node *link;
  };
int main()
{
  struct node ob;
  return 0;
}
```

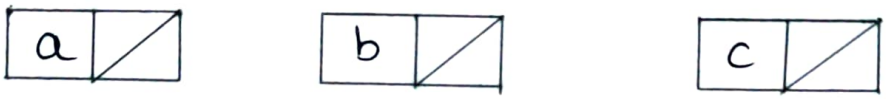
- In the above example, 'link' is a pointer to a structure of type 'node'. Hence, the structure 'node' is a self-referential structure with 'link' as the referencing pointer.

Eg :-
```
typedef struct
  {
    char data;
    struct list *link;
  } list;
```

- Each instance of the structure 'list' will have two components 'data' and 'link'.

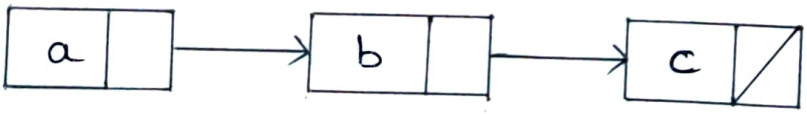
- Data → is a single character
- Link → is a pointer to a list structure. The value of link is either the address in memory of an instance of list or the null pointer.
- Consider these statements, which create 3 structures and assign values to their respective fields:

```
list item1, item2, item3;
item1.data = 'a';
item2.data = 'b';
item3.data = 'c';
item1.link = item2.link = item3.link = NULL;
```



- Structures item1, item2 & item3 each contain the data item a, b & c respectively and the null pointer.
- These structures can be attached together by replacing the 'null link' field in item2 with one that points to item3 and by replacing the null link field in item1 with one that points to item2.

```
item1.link = &item2;
item2.link = &item3;
```



Syntax :-

```
struct structure_name
{
  data type datatype_name;
  structure_name *pointer_name;
};
```

Dynamic Memory Allocation Functions :-

- This is a process of allocating memory space during execution time or run time.
- This is used if there is an unpredictable storage requirement.
- It uses predefined functions to allocate and release memory for data while the program is running.
- To use dynamic memory allocation the programmer must use either standard data types or must declare derived data types.
- In C dynamic memory allocation functions are defined in "stdlib.h".
- Dynamic memory allocation functions include 4 types of functions such as:

1) Malloc (Memory Allocate)
2) Calloc (Contiguous Memory Allocate)
3) Realloc (Resize Memory)
4) Free (De-allocate Memory)

1) Malloc :-

- Malloc function is used to allocate required amount of memory space during run-time
- When a malloc function is invoked requesting for memory, it allocates a block of memory that contains the number of bytes specified in its parameter and returns a pointer to the start of the allocated memory
- When the requested memory is not available the pointer NULL is returned.
- malloc does not initialize the allocated memory.
- malloc takes a single argument i.e, amount of memory to allocate in bytes.

Syntax:-

```
data_type *x;
x = (data_type *)malloc(size);
```

where,

x is a pointer variable of data_type
size is the number of bytes to be allocated.

Example:-

```
int *ptr;
ptr = (int *) malloc(100 * sizeof(int));
```

Functions to allocate memory for integers and floats:-

a)
```
int *MALLOC_INT(int n)
{
 int *x;
 x = (int*) malloc(n*sizeof(int))
 if (x == NULL)
 {
  Printf("Insufficient memory\n");
  exit(0);
 }
  return x;
}
```

b)
```
float *MALLOC_FLOAT(int n)
{
  float *x;
  x = (float *)malloc(n * sizeof(float));
  if (x == NULL)
  {
   printf("Insufficient memory\n");
   exit(0);
  }
  return x;
```

2) Calloc :-

- Calloc function is used to allocate contiguous block of memory.
- It is primarily used to allocate memory for arrays.
- This function allocates a user-specified amount of memory and initializes the allocated memory to zero.
- Pointer to the start of the allocated memory is returned
- If there is insufficient memory to make the allocation, the returned value is NULL.
- The only difference between malloc() and calloc() is that malloc() allocates single block of memory whereas calloc() allocates multiple blocks of memory each of same size and sets all bytes to zero.

Syntax :-

```
data_type *x;
x = (data_type *) calloc (n, size);
```

where,

x is a pointer variable of type int
n is the number of blocks to be allocated
size is the number of bytes in each block

Example :-

```
int *x;
x = calloc (10, sizeof (int));
```

- The above example is used to define a one-dimensional array of integers.
- The capacity of this array is n=10 and x[0:n-1] (x[0,9]) are initially zero.

Functions to allocate memory for integers and characters

a)
```
int *CALLOC_INT(int n)
{
 int *x;
 x = (int*)calloc(n, size);
 if(x==NULL)
 {
  printf("Insufficient memory\n");
  exit(0);
 } return x; }
```

b)
```
char *CALLOC_CHAR(int n)
{
  char *x;
  x = (char*)calloc(n, size);
  if(x==NULL)
 {
  printf("Insufficient memory\n");
  exit(0);
 }
  return x;
}
```

3) Realloc :-

- Before using the realloc() function, the memory should have been allocated using malloc() or calloc() functions.
- The function realloc() resizes memory previously allocated by either malloc() or calloc(), which means, the size of the memory changes by extending or deleting the allocated memory.
- If the existing allocated memory need to extend, the pointer value will not change.

- If the existing allocated memory cannot be extended the function allocates a new block and copies the contents of existing memory block into new memory block and then deletes the old memory block.
- When realloc is able to do the resizing, it returns a pointer to the start of the new block and when it is unable to do the resizing, the old block is unchanged and the function returns the value NULL.

Syntax :-

```
data_type *x;
x = (data_type *) realloc (ptr, size);
```

where,

ptr is a pointer to a block of previously allocated memory either using malloc() or calloc().
size is the new size of the block.

Example :-

```
#include <stdio.h>
#include <stdlib.h>
# include <string.h>
Void main()
{
  char *str;
  str = (char *) malloc(10); // allocates memory for the string
  strcpy (str, "computer"); // copy the string "computer"
  printf ("String = %s\n", str); output : computer
  str = (char *) realloc (str, 40); //reallocates the memory for
                                    larger string
  strcpy (str, "computer Science & Engineering"); copy the string
  printf (" string = %s\n", str);  into new memory space
}
```

4) Free :-

- This function is used to de-allocate (free) the allocated block of memory which is allocated by using the function malloc() or calloc().
- It is the responsibility of the programmer to de-allocate memory whenever it is not required by the program/application and initialize pointer variable to NULL.

Example :-

```
#include <stdio.h>
Void main()
{
  int i, *pi;
  pi = (int *) malloc(sizeof(int));
  *pi = 1024;
  printf("An integer = %d", pi);
  free(pi);
}
```

Representation of Linear Arrays in Memory :-

Linear Array :-

- Linear array is a list of a finite number 'n' of homogeneous data element such that

a) The elements of the array are referred respectively by an index set consisting of n consecutive numbers.

b) Elements of the array are stored respectively ~~by an index set~~ in successive memory locations.

- The number of 'n' of elements is called the length or size of the array.
- The length or the numbers of elements of the array can be obtained from the index set by the formula.

When LB = 0

$$\text{Length} = UB - LB + 1$$

When LB = 1

$$\text{Length} = UB$$

where,

UB → largest index called the upper bound

LB → smallest index called the lower bound

Eg :-

An automobile company wants to store sales data in an array from 1930 to 1984.

Auto [1930]	50
Auto [1931]	65
	⋮
	⋮
Auto [1984]	99

$\therefore$ Length = UB − LB + 1

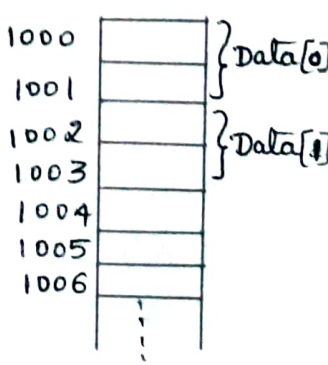
= 1984 − 1930 + 1

= 55 locations

Representation in Memory :-

- Array elements are stored consecutively in memory and memory locations are continuous.
- Each location has an address and the computer keeps track of the address of 1st element i.e, called as base address of linear array denoted by

Base (LA)



- Let us use the notation;

LOC (LA[K]) = address of the element LA[K] of the array LA

Eg :-

Loc(Data[K]) = address of element Data[K] of array data

∴ Loc(Data[1]) = 1002

- Using the base address, Base(LA) the computer calculates the address of any element of LA by the following formula

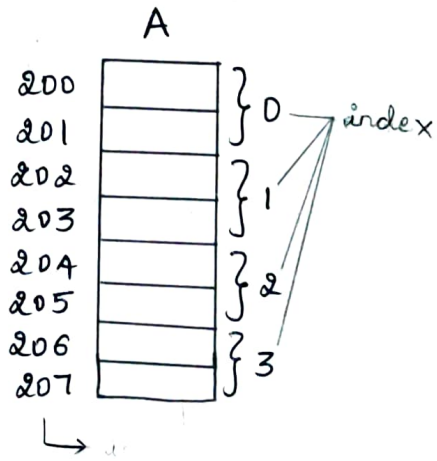
$$Loc(LA[K]) = Base(LA) + w(K - lowerbound)$$

where, w → no of words per memory cell for the array LA

Eg :- consider an array 'A'

where Base(A) = 200 & w = 2

w = 2 → because 2 words are occupied for each memory cell from 0 to 3



$$Loc(A[2]) = Base(A) + w(K - lowerbound)$$
$$= 200 + 2(2-0)$$
$$= 200 + 4 \quad \hookrightarrow A(0)$$
$$= 204$$

∴ Location for the element A[2] in memory is 204

Dynamically Allocated Arrays :-

One dimensional array :-

- When we cannot determine the exact size of the array, space of the array can be allocated at runtime.

Consider the following example :-

```
int i, n, *list;
printf("enter the number of numbers to generate");
scanf("%d", &n);
if (n<1)
```

```
{
  printf ("Improper value");
  exit (0);
}
MALLOC ( list, n*sizeof (int));
```

- Above code would allocate an array of exactly the required size and hence would not result in any wastage.

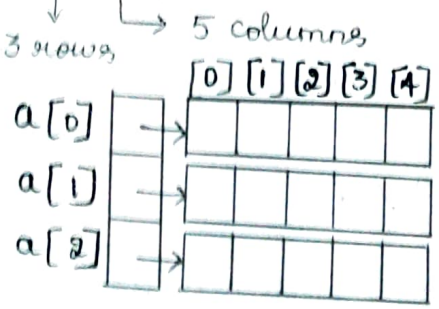
Two dimensional Arrays :-

- C uses array-of-arrays representation to represent a multi dimensional array.
- Two dimensional array is represented as a one-dimensional array of pointers ~~which~~ where each pointers contain the address of one-dimensional array.

(or)

- Two dimensional array is represented as one-dimensional array in which each element is itself a one-dimensional array.
- To represent the two-dimensional array, consider

```
int a[3][5];
```



It is a single dimensional array which can hold 3 pointers

Each of the 3 pointer points to a one-dimensional array consisting of 5 elements

Arrays of Arrays representation

- Above diagram represents a one-dimensional array 'x' whose length is '3', each element of 'x' is a one-dimensional array whose length is '5'.

Allocating memory for 2-Dimensional array dynamically :-

```
int ** make2DArray(int rows, int cols)
{
  int **X;        // 2 stars for storing address of 2D array
  int i;
  MALLOC (X, rows, * sizeof(*X));  // Allocate memory for row pointers
  for (i=0; i<rows; i++)
  {
    MALLOC(X[i], cols * sizeof(**X));  // Allocate memory for specified no of cols
  }
  return X;
}
```

Multidimensional Arrays :-

- C programming language allows multidimensional arrays
- An array that has more than one subscripts are called as multidimensional arrays.
- General form of multidimensional array declaration

  Datatype name [size1] [size2] ------ [sizeN];
- Consider the following example to create a 3 dimensional integer array.

  int three [5] [10] [4];
- Simplest form of multidimensional array is the two dimensional array.
- To declare a 2dimensional integer array of size [X] [Y]

  Datatype ArrayName [X] [Y];
- Multidimensional arrays may be initialized by specifying bracketed values for each row.

- Following is an array with 3 rows and each row has 4 columns.

```
int a[3][4] = { {0,1, 2, 3},
                {4, 5, 6, 7},
                {8, 9, 10, 11}
              };
```

- A 2 dimensional array which contains 3 rows & 4 columns can be shown as follows.

	col0	col1	col2	col3
Row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

- Every element in the array 'a' is identified by an element name of the form a[i][j]

  where 'a' – name of the array
  'i' – subscript of row
  'j' – subscript of column.

- Programming language stores the array either
  i) column by column called column major order
  ii) Row by row called row major order

a[0][0]
a[0][1]
a[0][2]
a[1][0]
a[1][1]
a[1][2]

a[0][0]
a[1][0]
a[2][0]
a[0][1]
a[1][1]
a[2][1]

Fig:- Row Major Order Column Major Order

Array Operations :-

Different operations that can be performed on arrays are

1) Traversing
2) Inserting
3) Deleting
4) Searching
5) Sorting

1) Traversing :-

- In traversing operation of an array, each element of an array is accessed exactly once for processing. This is also called visiting of an array.
- Each element is accessed in linear order either from left to right or from right to left.
- Traversing is a accessing and processing each element in the array exactly once.

Algorithm :-

- LA is a linear array with lower bound LB and Upper bound UB.
- This algorithm traverses LA by applying an operation PROCESS to each element of LA using 'for loop'/ 'while loop'

Steps :-

1. [Initialize Counter]	set K : = LB
2. Repeat step 3 & 4	while K ≤ UB
3. [Visit element]	Apply PROCESS to LA[K]
4. [Increase Counter]	set K : = K+1
5. [End of step-2 loop]	
5. Exit	

Example :-

```
int a[50],i,size;
printf (" Enter the size of array:");
scanf (" %d", &size);
printf(" Enter elements of array:");
for (i=0; i<size; i++);   // Read the data from the keyboard
{
 scanf(" %d", &a[i]);   // &a[i] → data will be stored in the address of index value
} Printf (" Elements in array are:");
for(i=0; i<size; i++)
{                          // To display the data stored in an array after traversing one by one
 printf (" %d", a[i]);   // a[i] → prints the data stored in each index from 0 to size-1
}
```

2) Inserting :-

- Let A be a collection of data elements stored in the memory of the computer.
- Inserting refers to the operation of adding another element to the collection A.
- Inserting an element at the "end" of the linear array can be easily done provided the memory space allocated for the array is large enough to accomodate the additional element.
- Inserting an element in the middle of the array, then on average half of the elements must be moved downwards to new locations to accomodate the new element and keep the order of the other elements.
- Inserting new element does not mean increasing its size, it is required to insert an item at the specified position

Algorithm :-

INSERT (LA, N, K, ITEM)

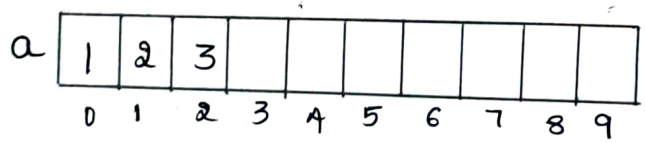
- Here LA is a linear array with N elements and K is a positive integer such that $K \leq N$
- This algorithm inserts an element ITEM into the K^{th} positio in LA.

Steps :-

1. [Initialize counter] — set J: = N
2. Repeat step 3 & 4 — while $J \geq K$
3. [Move J^{th} element downward] — Set LA[J+1]: = LA[J]
4. [Decrease counter] — Set J: = J-1
5. [End of step 2 loop]
6. [Insert element] — Set LA[K]:= ITEM
7. [Reset N] — Set N: = N+1
8. Exit

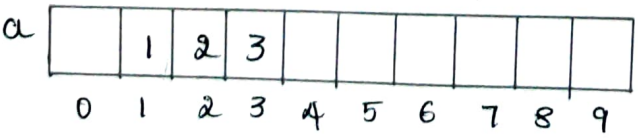
Example :-

- Consider an array a[10] having three items in it initially a[0] = 1, a[1] = 2 and a[2] = 3

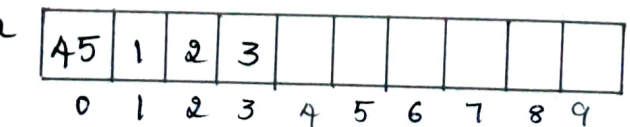


Now, we need to insert a no 45 at position "1" i.e, at a[0] = 45.

- We have to move elements one step towards right, such that a[1] = 1, a[2] = 2, a[3] = 3 and a[0] = vacant



- Now we insert a[0] = 45



Function to insert an item at specified position in an array :-

```
int insert(int item, int a[], int n, int pos)
{
  int i;
  if (pos > n || pos < 0)
  {
    printf("Invalid position");
    return n;
  }
  for (i = n-1; i>pos-1 ; i--);
    {
      a[i+1] = a[i]
    }
  a[pos-1] = item;
  return n+1;
}
```

3) Deleting :-

- It is a process of deleting an item from an array based on position.
- Deleting an element at the 'end' of the linear array can be easily done with, the difficulties.
  out
- If element at the middle of the array needs to be deleted then each subsequent elements be moved one location upward to fill up the array.

Algorithm :-

DELETE (LA, N, K, ITEM)

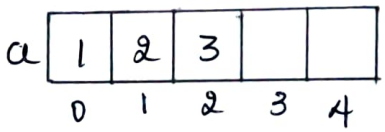
- Here LA is a linear array with N elements and K is a positive integer such that $K \leq N$, this algorithm deletes the K^{th} element from LA

Steps :-

1. Set ITEM := LA[K]
2. Repeat for J = K to N-1
   [Move J+1 element upward] set LA[J] := LA[J+1]
   [End of loop]
3. [Reset the number N of elements in LA] set N := N-1
4. Exit

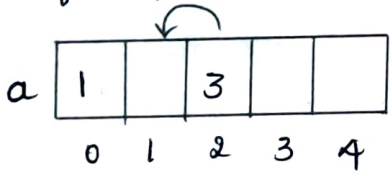
Example :-

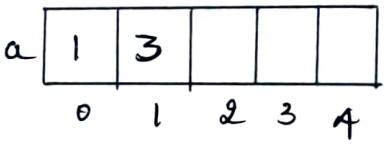
- Consider the following example of array a[5] having 3 items in it initially a[0]=1, a[1]=2 and a[2]=3



Now, we need to delete an item '2' at position a[1]=2

- We have to move all items from a[1] = 2 onwards towards left by one position





Function to delete an item from specified position in an array

```
int delete (int a[], int n, int pos)
{
  int i;
  if (pos>n || pos < 0)
  {
    printf ("invalid position");
    return n;
  }
printf ("item deleted = %d\n", a[pos-1]);
for (i= pos-1 ; i<n-1; i++)
 { a[i] = a[i+1] } return n-1; }
```

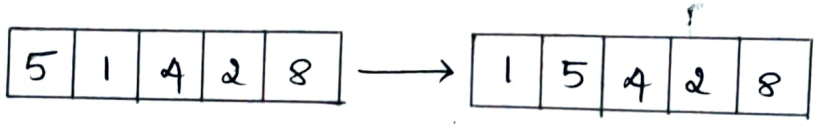
4) Sorting :-

- Sorting refers to the operation of rearranging the elements of a list, so that they are in increasing or decreasing order.
- There are different sorting techniques such as bubble sort, selection sort, merge sort, quick sort etc

Bubble sort :-

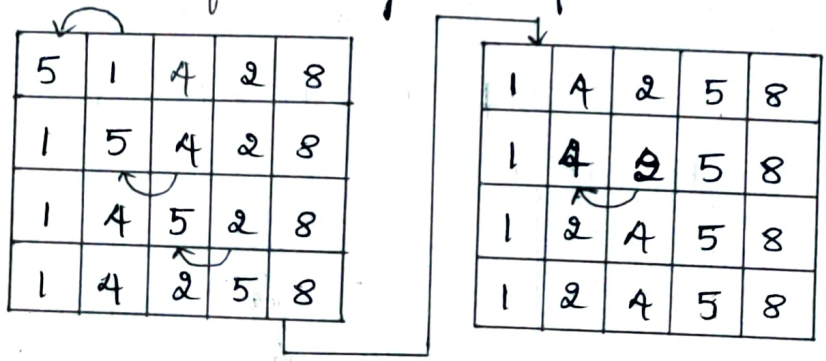
- Bubble sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements, if they are in wrong order.

Eg :-



- Here, algorithm compares the 1st two elements and swap, since 5>1

consider the following example :



Algorithm :-

BUBBLE (DATA, N)

- Here DATA is an array with N elements. This algorithm sorts the elements in DATA

Steps :-

1. Repeat steps 2 and 3 for K = 1 to N-1
2. Set pass := 1
3. Repeat while pass ≤ N-K
   a) if DATA[pass] > DATA[pass+1]
   b) Set Pass := Pass+1
4. Exit

Function to implement bubble sort :

```
Void bubble_sort (int a[], int n)
{
  int i, j, temp;
for (j=1; j<n; j++)
  {
    for (i=0; i<n-1; i++)
    {
      if (a[i] >= a[i+1])
      {
        temp = a[i];
        a[i] = a[i+1];
        a[i+1] = temp;
      }
    }
  }
}
```

Example :- 32, 27, 13, 85

A	Initial	Pass 1			Pass 2			Pass 3	
0	32 ↓	27	27	27	27 ↓	13	13	13 ↓	13
1	27 ↓	32 ↓	13	13	13 ↓	27 ↓	27	27 ↓	27
2	13	13 ↓	32 ↓	32	32	32 ↓	32	32	32
3	85	85	85 ↓	85	85	85	85	85	85

5) Searching :-

- Searching refers to the operation of finding the location of an element in data or printing some message that an element does not appear there.
- Search is said to be successful if an element does appear in data and unsuccessful otherwise

- Searching is used to find whether a given number is present in an array and if it is present then at what location it occurs.
- Two important searching techniques are
  i) Linear search
  ii) Binary search

i) Linear search:-

- It is a method which searches for a given key value with each element one by one sequentially in the array
- Linear search is also called as sequential search
- Start from the leftmost element of array and one by one compare 'X' with each element of array.
- If 'X' matches with an element, return the index otherwise return -1

Example:-

arr [ ] =

10	20	30	40	50	60	70	80	90	110	130
0	1	2	3	4	5	6	7	8	9	10

X = 110

Output = 9 i.e, Element 'X' is present at index '9'

Algorithm:-

LINEAR (DATA, N, ITEM, LOC)

- Here DATA is a linear array with N elements and ITEM is a given item of information
- This algorithm finds the location LOC of ITEM in DATA or sets LOC := 0 if the search is unsuccessful.

Steps:-

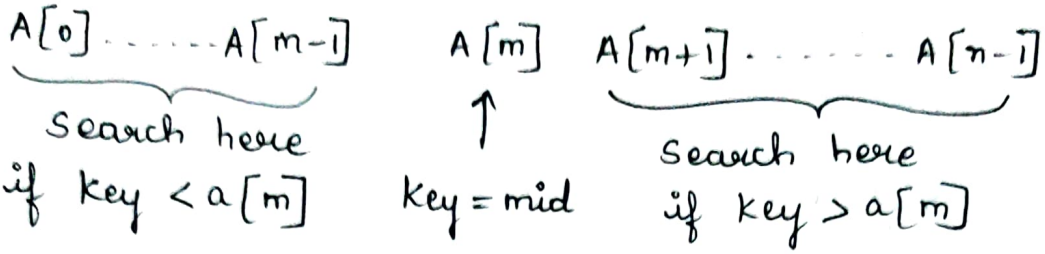
1. Set LOC := 1
2. [Search for item]
   Repeat while DATA[LOC] ≠ ITEM
   Set LOC := LOC + 1
   [End of loop]
3. [Successful?] If LOC = N+1, then set LOC := 0
4. Exit

Function to implement linear search is :-

```
int linear (int key, int a[], int n)
{
  int i;
  for (i=0; i<n; i++)
  {
    if (key == a[i])
    return i;
  }
  return -1;
}
```

ii) Binary Search :-

- The data in an array should be stored in the increasing numerical order or equivalently in alphabetical order
- Key element is searched with the middle element of the array
- If they are equal, the position of middle element is returned, else
- If the key is found to be greater than middle element, then the searching is applied on second half of array, otherwise on the first half of array.



Algorithm :-

BINARY (DATA, LB, UB, ITEM, LOC)

- Here DATA is a sorted array with LB and UB, ITEM is a given item of information.
- Variables BEG, MID and END denote beginning, middle and end locations respectively in DATA.
- This algorithm finds the location LOC of ITEM in DATA or sets LOC = NULL.

Steps :-

1. [Initialize segment variables]
   set BEG := LB, END := UB and MID = INT[(BEG + END)/2]
2. Repeat steps 3 and 4 while BEG ≤ END and DATA[MID] ≠ ITEM
3. If ITEM < DATA[MID], then :
   set END := MID - 1
   Else
   set BEG := MID + 1
   [End of If structure]
4. Set MID := INT[(BEG + END)/2]
   [End of step 2 loop]
5. If DATA[MID] = ITEM, then
   set LOC := MID
   Else
   Set LOC := NULL
   [End of If structure]
6. Exit.

Function to implement binary search:-

```
int bin_search(int key, int a[], int n)
{
  int low, high, mid;
  low = 0;
  high = n-1;
 while (low ≤ high)
 {
   mid = (low+high)/2;
  if (key = a[mid]) return mid;
  if (key < a[mid]) high = mid-1;
  if (key > a[mid]) low = mid+1;
 }
  return -1;
}
```

Example:-

DATA : 11, 22, 30, 33, 40, 44, 55, 60, 66, 77, 80, 88, 99

Search ITEM = 40

low, high, mid } Index Values

- Initially low = 0 and high = 12

  mid = (low + high)/2 = (0+12)/2 = 6

  So, DATA[mid] = DATA[6] = 55

- Since, 40 < 55, i.e, ITEM < DATA[mid]

  high is set to, high = mid − 1

  high = 6 − 1 = 5

  Hence, mid = (0 + 5)/2 = 2.5 ≃ 2 and

  DATA[mid] = DATA[2] = 30

- Since, 40 > 30 i.e, ITEM > DATA[mid]

  low is set to, low = mid + 1

  low = 2 + 1 = 3

Hence, mid = (3+5)/2 = 4 and

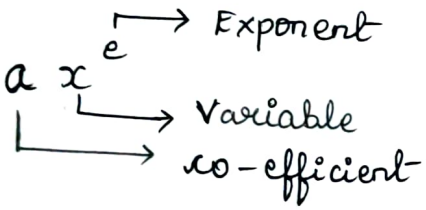
DATA[mid] = DATA[4] = 40

- We have found the ITEM '40' in location of DATA where LOC = 4 i.e, an index value

Representation of Polynomials with Arrays:-

Polynomial :-

- A polynomial is sum of terms where each term has a form



- The largest exponent of a polynomial is called its degree.

Eg :- $6x^{25} + 5x^{10} + 35$

This polynomial is a sum of 3 terms. Since 25 is the largest exponent, the degree of this polynomial is 25.

- Operation on polynomials includes returning the degree, extracting the co-efficient, addition, multiplication etc
- The different types of polynomial operations are
  - i) Representation
  - ii) Addition
  - iii) Multiplication
- A polynomial is an expression that contains more than two terms. A term is made up of co-efficient and exponent

Eg :- $p(x) = 4x^3 + 6x^2 + 7x + 9$

Abstract Data Type :-

- An ADT polynomial is the one that shows various operation that can be performed on polynomials. These operations are implemented as functions subsequently in the program.

ADT polynomial is :

Objects : $p(x) = a_1 x^{e_1} + \cdots + a_n x^{e_n}$; a set of ordered pairs of $\langle e_i, a_i \rangle$ where a_i and e_i are co-efficients and exponents respectively and are integers $>= 0$

Functions : Parameters used : poly, poly1, poly2 $\in$ polynomial
coef $\in$ co-efficients
expon $\in$ Exponents

Boolean IsZero(poly) ::= If polynomial 'poly' does not exist return true else return false

Co-efficient Coef(poly, expon) ::= if expon $\in$ poly, then return its coefficient else return 0

Exponent LeadExpo(poly) ::= return the largest exponent of poly

Polynomial zero() ::= return empty polynomial i.e, $p(x) = 0$

Polynomial Attach (poly, coef, expon) ::= If expon $\in$ poly return error else insert term $\langle$coef, expon$\rangle$ into poly & return polynomial poly.

Polynomial Remove (poly, expon) ::= If expon $\in$ poly, delete the term $\langle$coef, expon$\rangle$ and return poly else return error.

Polynomial Add (poly1, poly2) ::= return the polynomial
poly1 + poly2

Polynomial Multi (poly1, poly2) ::= return the polynomial
poly1 * poly2

Polynomial SingleMulti (poly, coef, expon) ::= return polynomial
poly . coef . x^{expon}

Example :- Let $a(x) = 25x^6 + 10x^5 + 6x^2 + 9$

calculate the following

1) LeadExpo (a) = 6
2) Coef (a, LeadExpo(a)) = 25
3) IsZero(a) = False
4) Attach (a, 15, 3) = $25x^6 + 10x^5 + \boxed{15x^3} + 6x^2 + 9$ ↑ insert
5) Remove (a, 6) = $10x^5 + 6x^2 + 9$

Polynomial Representation :-

- A polynomial can be represented using
  - i) Array
  - ii) Linked list

- Array representation assumes that the exponents of the given expression are arranged from 0 to highest value of degree which is represented by the index of array beginning with '0'
- The co-efficients of the respective exponent are placed at the appropriate index in the array.
- Consider the following example of polynomial using array representation.

$P(x) = 4x^3 + 6x^2 + 7x + 9$ { * Store the co-efficients in decreasing order of exponents

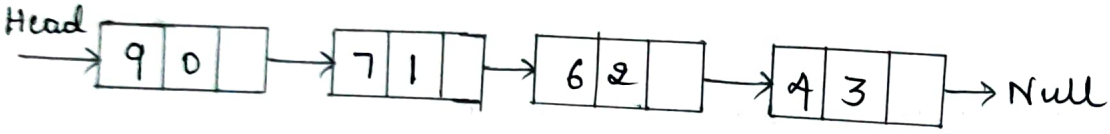
4	6	7	9	— Co-efficients
3	2	1	0	— Exponents

- A polynomial may also be represented using a linked list
- A structure may be defined such that it contains 2 parts one is the co-efficient and second is corresponding exponent.

```
struct polynomial
{
  int co-efficient;
  int exponent;
  struct polynomial *next;
};
```

- Consider the example of polynomial representation using linked list.

$$P(x) = 4x^3 + 6x^2 + 7x + 9$$



- Array of structures method can be used to store several polynomials.

```
#define MAX_DEG 100
typedef struct
{
  float coef;
  int expo;
}polynomial;
Polynomial P[MAX_DEG];
```

Eg :- $A(x) = 2x^{1000} + 5$

$B(x) = x^4 + 10x^3 + 3x^2 + 1$ can be represented as

	$2x^{1000}$	$5x^0$	x^4	$10x^3$	$3x^2$	x^0	
Coef	2	5	1	10	3	1	- - - - - -
Expo	1000	0	4	3	2	0	- - - - - -
	[0]	[1]	[2]	[3]	[4]	[5]	[6]
	Start A	End A	Start B			End B	Avail

- Start A – index of 1st term of poly A
  End A – index of last term of poly A

Start B – index of 1st term of poly B
End B – index of last term of poly B
Avail – index of next free space, where a term of a polynomial can be stored.

Polynomial Addition :-

Design an algorithm to add 2 polynomial using ADT Polynomial [C = a + b]

- Addition of 2 polynomials can be done by considering various cases as shown

Case – 1 :-

Powers of 2 terms to be added are equal.

- Assume that the 1st term of following polynomials are to be added.

$a = 25x^6 + 10x^5 + 6x^2 + 9$ ⇒ LeadExp(a) = 6
$b = 15x^6 + 5x^4 + 4x^3$ ⇒ LeadExp(b) = 6

$C = 40x^6$

- LeadExp(a) and LeadExp(b) are same, they can be added

sum = Coef (a, LeadExp(a)) + Coef (b, LeadExp(b))
= 25 + 15
= 40 → Now add 40 into poly 'C'

- This can be done by calling the attach function as:

```
if (sum != 0)
Attach (c, sum, LeadExp(a));
```

- Now, move to the next term of poly 'a' and poly 'b' by removing the added term i.e,

a = Remove (a, LeadExp(a)); → i.e, $25x^6$

b = Remove (b, LeadExp(b)); → i.e, $15x^6$

∴ a = $10x^5 + 6x^2 + 9$ and

b = $5x^4 + 4x^3$

- Complete code can be written as:

```
if (LeadExp(a) == LeadExp(b))
{
 sum = Coef (a, LeadExp(a)) + Coef (b, LeadExp(b));
 if (sum != 0) Attach (c, sum, LeadExp(a));
 a = Remove (a, LeadExp(a));
 b = Remove (b, LeadExp(b));
}
```

Case - 2 :-

Power of 1st term of poly 'a' is greater than power of 1st term of poly 'b'

a = $10x^5 + 6x^2 + 9$ ⇒ LeadExp(a) = 5

b = $5x^4 + 4x^3$ ⇒ LeadExp(b) = 4

---

c = $10x^5$

- $10x^5 > 5x^4$, we attach $10x^5$ into c and then we remove it before moving to the next term of poly 'a'
- Complete code can be written as:

```
if (LeadExp(a) > LeadExp(b))
{
  Attach (c, Coef (a, LeadExp(a)), LeadExp(a));
  a = Remove (a, LeadExp(a));
}
```

∴ a = $6x^2 + 9$

b = $5x^4 + 4x^3$

Default:

If 1st term of poly 'b' is greater than 1st term of poly 'a'

(or)

If 1st term of poly 'a' is lesser than 1st term of poly 'b'

$a = 6x^2 + 9 \Rightarrow LeadExp(a) = 2$

$b = 5x^4 + 4x^3 \Rightarrow LeadExp(b) = 4$

$C = 5x4$

- $6x^2 < 5x^4$ (or) $5x^4 > 6x^2$, we attach $5x^4$ into C and then remove it before moving to the next term of poly 'b'
- Complete code can be written as:

```
if (LeadExp(b) > LeadExp(a))
{
  Attach (c, coef (b, LeadExp(b)), LeadExp(b));
  b = Remove (b, LeadExp(b));
}
```

- All the above cases have to be repeated untill one or both polynomials become empty.
- If one of the poly becomes empty, copy the remaining terms into poly 'c'.
- Above logic can be written using a COMPARE() macro, i.e, compares whether the exponents of 'a' and 'b' are same greater than or lesser than as shown below:

```
C = zero();
while (! IsZero(a) && ! IsZero(b))
{
 switch (COMPARE (LeadExp(a), LeadExp(b)))
{
  case 0: sum = Coef (a, LeadExp(a)) + Coef (b, LeadExp(b));
          if (sum! =0) Attach (c, sum, LeadExp(a));
```

```
        a = Remove (a, LeadExp(a));
        b = Remove (b, LeadExp (b));
        break;
Case 1: Attach (c, Coef (a, LeadExp(a)), LeadExp(a));
        a = Remove (a, LeadExp (a));
        break;
Default: Attach (c, Coef (b, LeadExp(b)), LeadExp(b));
        b = Remove (b, LeadExp (b));
    }
}
```

Sparse Matrices :-

- A matrix which contains many zero entries or very few non-zero entries is called as sparse matrix.
- A sparse matrix can be represented in 1D, 2D and 3D arra when a sparse matrix is represented as a 2D array, more space is wasted.

Eg :-

a

0	1	2	3	4	5
0	10	0	0	12	0

and

	0	1	2	3
0	10	0	0	40
1	11	0	20	0
2	0	0	0	0
3	20	0	0	50

Sparse Matrix Representation :-

- An element within a matrix can characterize by using the triple <row, col, value>. This means that an array of triple is used to represent a sparse matrix.
- Organize the triples so that the row indices are in ascending order.

- Sparse matrix can be created using the array of triples as shown,

```
#define MAX 100
typedef struct
{
  int row;
  int col;
  int val;
} TERM;
TERM a[MAX_TERMS];
```

- The below figure shows the representation of matrix in the array "a" a[0].row contains the number of rows, a[0].col contains the number of columns and a[0].value contains the total number of non-zero entries.

- Row → contains the index of row where non-zero elements are located.

  col → contains the index of column where non-zero element are located.

  Value → Value of non-zero elements located at index of (row, col).

Eq :-

a =

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

4 X 5

	Rows	Col	Value
a[0]	4	5	7
a[1]	0	2	4
a[2]	0	4	5
a[3]	1	3	3
a[4]	1	4	6
a[5]	2	3	2
a[6]	3	1	2
a[7]	3	2	3

Fig :- Sparse Matrix stored as triple

Function to read the sparse matrix as a triple :

```
Void sparsematrix ( TERM a[], int m, int n)
{
  int i, j, k, item;
 a[0]. row = m,  a[0]. col = n,  k = 1;
 for (i = 0 ; i < m ; i++)
 {
   for (j = 0; j < n; j++)
   {
     scanf ("%. d", &item);
   if ( item = = 0) continue;
   a[k]. row = i ;
   a[k]. col = j ;
   a[k]. val = item;
    k++;
 } }
     a[0]. val = K - 1;
}
```

Transpose of a Matrix :-

- A matrix which is obtained by changing row elements into column elements and vice-versa is called transpose of a matrix

Eg :- Represent the given sparse matrix into 1D array with triples as well as transform it into a transpose matrix

$$a = \begin{array}{c} \\ 0 \\ 1 \\ 2 \end{array} \begin{array}{c} \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \begin{bmatrix} 0 & 0 & 1 & 2 \\ 1 & 0 & 0 & 3 \\ 1 & 0 & 0 & 4 \end{bmatrix} \end{array}$$

3 × 4

	Rows	Cols	Value
a[0]	3	4	6
a[1]	0	1	1
a[2]	0	3	2
a[3]	1	0	1
a[4]	1	3	3
a[5]	2	0	1
a[6]	2	3	4

Fig :- Sparse Matrix stored as triple.

Transpose Matrix

$$b = \begin{bmatrix} 0 & 1 & 1 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \\ 2 & 3 & 4 \end{bmatrix}_{4 \times 3}$$

	Rows	Cols	Value
b[0]	4	3	6
b[1]	0	1	1
b[2]	0	2	1
b[3]	2	0	1
b[4]	3	0	2
b[5]	3	1	3
b[6]	3	2	4

Fig:- Transpose sparse matrix

Function to find the transpose of a given sparse matrix :-

```
Void transpose (TERM a[], TERM b[])
{
  int i, j, k;
  b[0].row = a[0].col;
  b[0].col = a[0].row;
  b[0].val = a[0].val;
  k = 1;          // position of 1st non-zero element
  for (i=0; i<a[0].col; i++)
  for (j=1; j<a[0].val; j++)
    if (a[j].col == i)
    {
      b[k].row = a[j].col;
      b[k].col = a[j].row;
      b[k].val = a[j].val;
      k++;
    }
}
```