MODULE 1

MODULE 1 : INTRODUCTION TO DATA STRUCTURES

Data Structures :- Definition :

Data Structure is a logical or mathematical model of storing and organizing data in a particular way in a computer required for designing and implementing efficient algorithms and program development. (or)

* Data Structure is a way of organizing the data along with relationship among data.

The study of data structures includes :

1. Defining operations that can be performed on data.
2. Representing data in the memory.
3. Determining amount of memory required to store and amount of time needed to process the data.

* Need for Data Structures :

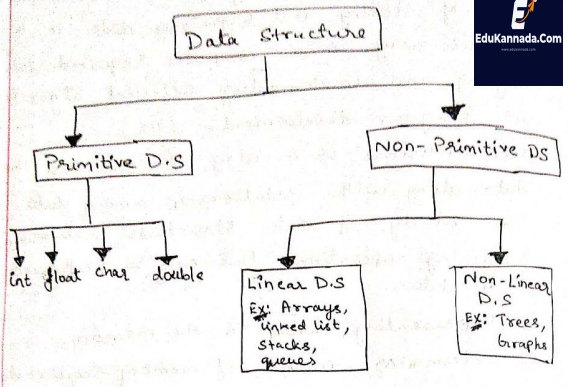
The computers are electronic data processing machines. In order to solve a particular problem, we need to know :

1. How to represent data in computer?
2. How to access them?
3. What are the steps to be performed to get the needed output.

These tasks can be achieved with the knowledge of data structures and Algorithms

Classification of data Structures :-

* D.S are classified into Primitive & Non-Primitive data structure.



1.) Primitive Data Structures :-

- These are the fundamental standard data types.
- These are used to represent single values.

Eg: int, float, char, double.

2.) Non-Primitive data Structures :-

- These are derived from primitive data types.
- are used to store group of values.

Eg: Arrays, stacks, queues, linked list, trees etc.

* Non-Primitive data structure are further classified into Linear and Non-Linear D.S.

a) Linear D.S : A data structure is said to be Linear, if its elements are stored in a sequential order. Ex: Arrays, stacks, queues, linked list.

b.) Non Linear D.S : A D.S. is said to be non-linear if the data is not arranged in a sequential order. Ex: Trees, Graphs

- Elements are stored on hierarchical relationship among the data. Ex: Trees.
- Graphs : used to represent data that has relationship between pair of elements. Ex: Graph, Routes, networks.

Data Structure Operations :-

* Data appearing in our data structures are processed by means of certain operations:

4 Major operations:

① Traversing : accessing each record (element) exactly once so that certain items may be processed.

② Searching : Finding the location of the record with a given key value or finding the location of all records which satisfy one/more conditions

③ Insertion : adding a new record into the data structure.

④ Deletion : removing a record from the structure

→ Sometimes 2 or more operations may be used in a given situation. Ex: we want to delete the element with given key ie; first search for location and delete.

Other operations like:

1.) Sorting : arranging the records in ascending/descending Order.

2.) Merging : combining two data structures.

Review Of Arrays :-

Definition: An array is collection of similar data items. The elements of an array are of same type & each element can be accessed using same name, but with different index values.

- Formally, an array is defined as a set of pairs <index, value>, where index is position and value is data stored in position index

Ex: Array of marks <index, value>

<0, 80>
<1, 90>
<2, 45>
<3, 98>
<4, 31>

Note: Array is implemented as consecutive set of memory locations

marks[0]	80
marks[1]	90
marks[2]	45
marks[3]	98
marks[4]	31

ADT (Abstract Data Type) for Array:

An ADT of array Provides details of array implementation and various operations that can be performed on an array.

P.T.O

ADT array is

Object: A set of Pairs <index, value> where value is data stored at index position in an array. for 1D. array index values are

[0, 1, 2 ..... n-1]

for 2D. array index values are:

[(0,0), (0,1), (0,2),
(1,0), (1,1), (1,2),
(2,0), (2,1), (2,2)]

Functions: A ∈ Array, i ∈ index, x ∈ item, n ∈ integer

return_type	Function_name	Operations Performed
Array	Create(n, a) ::=	Returns array a of size n.
item	Retreive(a, i) ::=	if i ∈ index then return item stored in array a at index position i.
Array	Store(a, i, x) ::=	if i ∈ index then return array A, by storing x at index position i in a.

end Array

Syntax of declaring an array:

data type name[size];

Example: int a[50];

Initialize an array:

int a[5] = {4, 6, 3, 1, 9};

Structures :-

- A structure is a user-defined datatype.
- It is defined as a group of dissimilar or heterogeneous data items, where items can be of a different data type

Structure declaration:

Syntax:

```
struct tagname
{
type1 var1;
type2 var2;
 :     :
 :     :
type n var n;
};
```

Ex:

```
Struct student
{
char name[30];
int age;
int rno;
};
```

where struct – is a keyword (for structures)

type1, type2 ... type n – standard data types like int, float, char, double

Var1, Var2 .... Var n – are members of the structures.

tagname — any identifier.

* To allocate the memory for the structure, we have to declare the variable as shown:

```
struct  student  cse,ise;
```

Keyword — Structure name — Structure Variables.

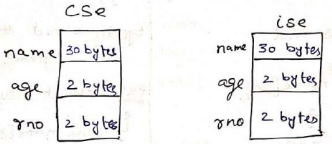
* Once structure variables are declared, compiler allocates memory for the structure variables.

→ Two ways to declare variables:

①
```
struct student
{
  ---
  ---
  ---
};
struct student cse, ise;
```

②
```
struct student
{
  ---
  ---
  ---
} cse, ise;
```

→ Memory Representation of structure : cse and ise is as follows:



∴ 34 bytes of memory is allocated each for the structure variable cse and ise.

→ Structure Initialization : assigning values to the structure variable.

Method 1:
```
struct student
{
  char name[30];
  int age;
  int rno;
} cse = {"Kumar", 19, 002};
```

Method 2:
```
struct student
{
  char name[30];
  int age;
  int rno;
};
struct student cse = {"Kumar", 19, 002};
```

→ Accessing Structure members : using dot operator (.)

Ex: ① cse.name = "Kumar"; ② pf("%d", cse.rno);

→ Type - defined structure:

* The structure associated with keyword typedef is called type - defined structure.
* Most powerful way of defining the structure.

→ There are 2 methods:

① typedef struct
{
type1 member1;
type2 member2;
:
} typedef_structure_name;

Ex: typedef struct
{
char name[30];
int age;
int rno;
} student;

create variables:
typedef_name variables;
Ex: student cse, ise;

② Ex: struct structurename
{
type1 member1;
type member2;
:
};

typedef struct structure_name typedef_name;

Ex: struct student
{
char name[30];
int age;
int rno;
};

typedef struct student St;
create variables:
St cse, ise;

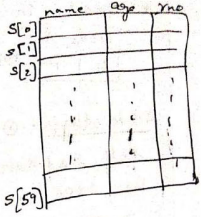
→ Array of structures :-

If one has to store a large number of objects then array of structures is used.

Ex: If there are 60 students in a class, then the record of each student is in structure. and 60 such records can be stored in array of structure.

```
struct student
{
    char name[30];
    int age;
    int rno;
} s[60];
```

Memory Representation ⇒



→ Nested Structures / structure within a structure:-

* Nested structure – embeds a structure within another structure.

Ex:

```
struct date
{
    int month;
    int day;
    int year;
};
struct person
{
    char name[30];
    int age;
    struct date dob;
} p1;
```

Note: Thus, a person born on 24, Sept, 2019, would have the values for the struct dob set as:

```
p1.dob.month = 09;
p1.dob.day = 24;
p1.dob.year = 2019;
```

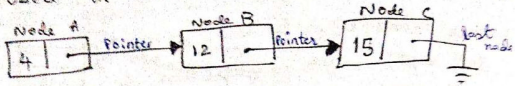
Self-Referential Structure:-

* A self Referential structure is a structure definition which includes atleast one member that is a pointer to the structure of its own kind. ie; Pointer stores the address of the structure of the same type.

Ex:

```
struct node
{
    int data;
    struct node *ptr;   // pointer to struct node.
};
```

* Used in linked list:



Advantages: ① Unlike static data structures, self referential structure can dynamically be expanded or contracted. Thus, require DMA functions (malloc, free) to explicitly obtain & release memory.

② Useful in applications that involve data structures like linked list and trees.

Unions :-

* Union is a user-defined datatype in C, which is used to store collection of dissimilar data items (just like a structure).
* Unions and structures are same, except allocating memory for their members.
* Structure allocates storage space for all its members separately, whereas unions allocates one common storage space for all its members.

Ex:
```
union student
{
    char name[20];
    int age;
    int rno;
};
```
→ Here, union allocates only 20 bytes of memory, since the name field requires the maximum space (1 byte x 20) compared to age and rno (only 2 bytes each). However, a structure would allocate 24 bytes total (includes the size of all its members).

Union syntax:

```
union tagname
{
    type1 member1;
    type2 member2;
    .     .
    .     .
    typen membern;
};
```

(OR)

```
typedef union
{
    ---
    ---
} tag_name;
```

Declaration of Union Variables:

union tagname Var1, Var2, ... Varn;

Ex: union student S1, S2, S3;

Pointers:-

* A pointer is a variable that stores the address of another variable.

4 steps to use pointer:

	Ex:
Step 1: Declare a variable	int a;
Step 2: Declare a pointer variable.	int *Pa;
Step 3: Initialize a pointer variable	Pa = &a;
Step 4: Access the pointer	Pf("value of a is %d", *Pa);

Pointer declaration: datatype *pointer_var;

Ex: int *Pa

Pointer Initialization

Pa = &a;

Here the Pointer Pa, holds the address of a.

* C allows us to perform arithmetic operations such as addition, subtraction etc, since a Pointer is a non-negative integer.

Dynamic Memory Allocation :-

* Memory management is important task in computer programming.
* There are two types of memory management
  1. Static memory mgmt.
  2. Dynamic Memory Mgmt.

1. **Static Memory Management** : The allocation & deallocation of memory is done during compilation time of the Program.

   Ex: int a[10];

2. **Dynamic Memory Mgmt** :- The memory allocation & deallocation is performed during run-time of the Program. Thus, when program is getting executed at that time memory is managed. This is the efficient method compared to static memory management.

Static	Dynamic
1. Memory allocation is performed at compile time.	Memory allocation is performed at run-time.
2. Prior to allocation, fixed size of memory has to be decided.	No need to know size of memory prior to allocation
3. wastage/shortage of memory occurs.	Memory is allocated as per requirement
Ex: Array	Ex: Linked List.

* There are four functions used in DMA:

1. **malloc()** : allocates a block of memory.
   * It is used to allocate memory space as per requirements.
   * Function allocates memory & return a pointer

of type void * to the start of that memory block.
* If function fails, it returns NULL. ∴ it is necessary to verify pointer returned is not NULL
* This function, does not initialize the memory allocated during execution. It carries garbage value.

Syntax: ptr = (datatype *) malloc (size);

ptr – pointer variable of type datatype.
datatype – any C datatype or user defined data type
Size – no. of bytes required.

Ex: int *ptr;
ptr = (int *) malloc (10);

② calloc() : allocate multiple blocks of memory.
* It is similar to malloc, but it initializes the allocated memory to zero.

Syntax: ptr = (datatype *) calloc (n, size)

n → no. of blocks to be allocated.

Ex: ptr = calloc (20, sizeof(int));

This fn computes memory required for 20 X 2 bytes (for int) = 40 bytes of memory block. is allocated

③ realloc() : used to modify the size of allocated block by malloc(), calloc() to new size.
* If allocated memory space is not sufficient, than additional memory can be taken using realloc().
* Can also be used to reduce the size of already allocated memory.

* Syntax:

ptr = (datatype *) realloc (ptr, size);
↓ new size

Ex: Char *str;

str = (char *) malloc (10);

str = (char *) realloc (str, 40);

(4) free(): deallocate the allocated memory which was done using malloc(), calloc or realloc().

Syntax: free(pointername)

Ex: free (str);

Dynamically Allocated Arrays :-

The array can be dynamically allocated using malloc(), calloc(), or realloc() fns.

Ill by allocated memory can be freed.

* Advantage: memory for array of any desired size can be allocated. No need of fixed sized array.

C program for Dynamic Arrays

```
#include <stdio.h>
int main()
{
    int n, i, sum=0;
    int *a;
    printf("Enter no. of elements\n");
    scanf("%d", &n);
    a = (int *) malloc (n * sizeof(int));
```

```
if (a == NULL)
    Printf(" NO memory allocated ");
Printf(" Enter array elements\n");
for(i=0; i<n; ++i)
{
    scanf("%d", a+i);
    sum+ = *(a+i);
}
Printf(" sum = %d\n", sum);
free(a);
return 0;
}
```

Linear Arrays :-

* It is a list of finite number n of homogeneous elements of same type such that:

a) elements of array are referenced by respective index set consisting of consecutive numbers.

b) elements of array are stored respectively in successive memory locations.

* No. of elements are called Length/size of array.

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

$\therefore$ Length $= 4 - 0 + 1 = 5$ $\quad UB = 4;\ LB = 0$

* The elements of array A can be denoted as $A1, A2, A3 \ldots An$ or $A[1], A[2] \ldots A[n] \ldots$

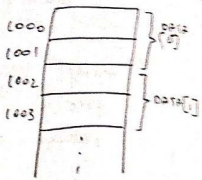
Ex: An automobile company wants to store sales data in array from 1930 to 1984.

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

$\therefore$ Length $= UB - LB + 1$
$= 1984 - 1930 + 1$
$= 55$ locations

Linear Arrays Representation in Memory :-

* Array elements are stored consecutively in memory and memory locations are continuous.
* Each location has an address,
* The computer keeps track on only first element address. ie called as base address (Base)



Now, Let us use the notation:

> LOC (DATA[K]) = address of element DATA[K] of array DATA

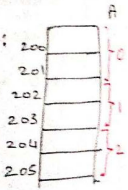
Ex: LOC (DATA[1]) = 1002

* Using base address, computer calculates other location addresses:

> LOC (DATA[K]) = Base (DATA) + W (K - lowBound)

where W → no. of words per memory cell for array DATA.

Example: Consider an array A:



Here: Base (A) = 200, W = 2

Then find location of index position 2:

LOC (A[2]) = Base (A) + w (K - lowerbound)
= 200 + 2 (2 - 0)
= 204 //

Array Operations :-

→ Traversing Linear Arrays:

* Let A be a collection of data elements stored in the memory of the computer.
* Suppose we want to print the contents of each element of A or suppose we want to count the no. of elements of A with a given property.
* This can be accomplished by traversing A; ie; by accessing & processing (visiting) each element of A exactly once.
* Algorithm: Let LA be a linear array with Lower bound LB and Upper bound UB. This algorithm traverses LA applying an operation PROCESS to each element of LA.

Step 1. [Initialize Counter] Set K := LB
Step 2: Repeat step 3 and 4 while K ≤ UB
Step 3. [visit Element] Apply PROCESS to LA[K]
Step 4. [Increase Counter] Set K := K+1
[End of Step 2 loop]
Step 5. Exit

→ Inserting and deleting in Array

* Inserting refers to the operation of adding another element to the array.
* Deletion refers to the operation of removing one element from the array.
* Inserting an element at the end of a linear array can be easily done provided the memory space allocated for the array is

is large enough to accommodate the additional element. But suppose we need to insert an element in the middle of the array, then half of the elements must be moved downward to new location & thus keeping the order of the other elements.

* Illy, deleting an element at the end of the array presents no difficulties, but deleting an array in the middle of the array requires each subsequent element be moved one location upward in order to fillup the array.

Algorithm for Inserting in K^{th} position in an array LA with N elements, and using J as a counter:

INSERT (LA, N, K, Item)

Step 1. [Initialize] Set J := N

Step 2. Repeat steps 3 and 4 while J ≥ K

Step 3. [Move J^{th} element downward] Set LA[J+1] := LA[J]

Step 4. [Decrease counter] Set J := J-1

[End of step 2 loop]

Step 5. [Insert element] Set LA[K] := Item

Step 6. [Reset N] Set N := N+1

Step 7. Exit

Algorithm for Deleting a K^{th} element from array LA

[Move J + 1st element upward] Set LA[J] := LA[J+1]

[End of Loop]

3. [Reset N in LA] Set N := N-1

4. Exit.

→ **Sorting:-** Sorting refers to the operation of rearranging the elements of a list, so that they are in increasing or decreasing order.

Eg: Suppose array A originally has the list: 8, 4, 12, 5

After sorting, A has the list: 4, 5, 8, 12

* There are many different sorting algorithms: like Bubble Sort, Selection sort, heap sort etc.

Bubble Sort: Suppose the list of Numbers A[1], A[2] .... A[N] is in Memory. The bubble sort algorithm works as follows:

Step 1: Compare A[1] and A[2] and arrange them in desired order so that A[1] < A[2]. Then compare A[2] and A[3] and also A[3] and A[4] and so on.

[Observe that Step1 involves n-1 comparisons and also the largest element sinks to the n^{th} position.]

Step 2: Repeat Step1 with one less comparison (i.e. the largest element is already sorted) ie, step2 requires n-2 comparison, the second largest element will occupy A[n-1] or $(n-1)^{th}$ position.

⋮

Step n-1: After n-1 steps, the list will be sorted in increasing order finally.

Note: Bubble sort requires n-1 Passes, where n is the no. of input elements.

Formal algorithm for Bubble sort

Bubble (DATA, N)

Here DATA is an array with N elements.

Step 1 : Repeat steps 2 and 3 for K=1 to N-1

Step 2 : Set pass := 1

Step 3 : Repeat while pass $\leq$ N-k

a) if DATA[pass] > DATA[pass+1]

b) set pass := pass + 1

Step 4 . Exit

Complexity of the Bubble Sort

* Traditionally, the time for a sorting algo is measured in terms of the the no. of comparisons. The number f(n) of comparisons in bubble sort is:

$$f(n) = (n-1) + (n-2) + \cdots 2 + 1 \quad \text{// No. of comparisons in each pass}$$

$$= \frac{n(n-1)}{2} = \frac{n^2}{2} + O(n)$$

$$= O(n^2)$$ //

Example for Bubble Sort: 32, 27, 13, 85

A	Initial	pass 1			pass 2		pass 3
0	32	27	27	27	13	13	13
1	27	32	13	13	27	27	27
2	13	13	32	32	32	32	32
3	85	85	85	85	85	85	85

Searching :-

* Searching refers to finding the locations of the record with a given key value. The search is said to be successful if the key value is found, otherwise is said to be unsuccessful.

i) **Linear Search :** is a method which searches for a given key value with each element one by one sequentially in the array.

Algorithm: LinearSearch (DATA, N, KEY, LOC)

The algo finds the location LOC of KEY in DATA, and sets LOC = 0 if the search is unsuccessful.

Step 1: Set LOC := 1

Step 2: [Search for Item]

Repeat while DATA[LOC] ≠ Item

Set LOC := LOC + 1

[End of Loop]

Step 3: [Successful ?] If LOC = N+1, then set LOC := 0

step 4: Exit.

Complexity of Linear Search: complexity is measured using 3 cases: ① Best case ② Average case ③ worst case.

* while searching an item in an array, if the item is present in 1st location, that is search is successful during 1st comparison - it is Best case.
* Search is successful in case where item is found in 2nd to (n-1) comparison - it is average case.
* If item is found in last comparison or if no item was found — is the worst case.
* For Linear Search: Best case = 1, Average case = n/2 and worst case = n.

2.) Binary Search: The data in array should be stored in the increasing numerical order or equivalently in alphabetical order in this Binary search method.

* This is one of the efficient Searching algorithm.
* Idea: Suppose we want to find location of some name in a telephone directory. One doesn't perform linear search, rather opens the directory in middle to determine which half contains the name being sought. Then open that half in the middle to determine to determine which quater of the directory contains name. and so on.
* Thus, we find the location of name, since we reduce the no. of locations to be searched.

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

Here DATA is stored in an array with LB & UB. ITEM is given item of information. BEGIN & END to denote beginning & ending locations. MID for denoting middle location of DATA. This algo finds location LOC of ITEM in DATA Or sets LOC = NULL.

Step1: [Initialize segment variables]
Set BEGIN = LB, END = UB & MID = INT (BEGIN+END)/2

Step 2: Repeat Steps 3 & 4
while BEGIN ≤ END & DATA[MID] ≠ ITEM

Step 3: If ITEM < DATA[MID] then set
END = MID - 1
Else
BEGIN = MID + 1
[End of If stmt]

Step 4: Set MID = INT ((BEGIN + END)/2)

[End of step2 loop]

Step 5: If DATA[MID] = ITEM then:

set LOC = MID

else

set LOC = NULL

[End of if stmt]

Step 6. Exit

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

Search ITEM = 40

1. Initially BEGIN = 1 and END = 13

MID = (1+13)/2 = 7 and so DATA[MID] = 55

2. Since 40 < 55, END is set to END = MID - 1 = 6

Hence MID = (1+6)/2 = 3 and so DATA[MID] = 30

3. Since 40 > 30, BEGIN is set to BEGIN = MID + 1 = 4

Hence MID = (4+6)/2 = 5 & so DATA[MID] = 40

4. we have found the ITEM 40 in Location

in LOC = MID = 5 //

Complexity of Binary Search: No of comparisons to locate the data:

- Best case: if DATA[MID] is exactly middle value, then

Best case = 1 //.

- Average case: Observe each comparison has reduced sample size in half.

∴ no. of comparisons $f(n) = \log_2 n$ //

- worst case: If item not found, then also

$f(n) = \log_2 n$ //

Ex: If 1,00,000 items in array.

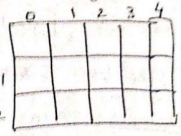
$f(n) = \log_2 (1,00,000) = \frac{\log (1,00,000)}{\log 2} = 20$ comparisons

Multidimensional Arrays :-

* An array that has more than one subscripts are called as Multidimensional arrays.
  Ex: 2-D array, 3D and so on.

Two dimensional array: An array that has two subscripts are called as 2D array.

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



* This defines a new array containing 3 elements. Each of these elements is itself an array containing five integers.
* The elements of this array can be accessed by specifying a row number and a column number.
  Eg: The element in row 1 and column 3 can be referenced as `a[1][3]`
* The no. of elements in the array is the product of the ranges of all its dimension ie; the above array a contains $3 \times 5 = 15$ elements.
* To implement a 2-D array in a linear fashion, there are 2 methods: row-major and column-major order. In row-major order, the elements are arranged row-by-row manner, whereas in column-major order, elements are arranged column-by-column.

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

Fig: Row-major order

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

Fig: Column-Major order

Representation of 2D array: Let us consider a row-major order. Let ar be an array, where base(ar) is the address of the first element of the array. ie; if ar is declared by: int ar[r][c];

where r → no. of rows and c → no. of columns.

esize → size of each element in the array.

* To calculate the address of an arbitrary element ar[i1][i2], apply the following formula:

ar[i1][i2] = base(ar) + (i1 * c + i2) * esize.

Example: To calculate ar[0][2]

Base address is 200; apply the above eqn:

200	a[0][0]
202	a[0][1]
204	a[0][2]
206	a[0][3]

ar[0][2] = 200 + (0 * 3 + 2) * 2

= 200 + (0 + 2) * 2

= 204 //

Polynomials :-

* A Polynomial is sum of terms where each term has a form:

$a\,x^{e}$ → exponent

a → coefficient, x → variable

* The largest exponent (Leading exponent) of a Polynomial is called its degree.

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

This polynomial is sum of three terms. Since 25 is the largest exponent, the degree of this polynomial is 25. The last term 35 can also be written $35x^{0}$.

* The Abstract Datatype: An ADT Polynomial is the one that shows various operations that can be performed on polynomials. These operations are implemented as functions subsequently in the program:

ADT Polynomial is :

Objects : $P(x) = a_1 x^{e_1} + \ldots\ldots + a_n x^{e_n}$ where a_i and e_i are coefficients & exponents respectively ; are integers $>= 0$.

Functions : Parameters used : poly , poly1 , poly2 ∈ Polynomial, coef ∈ coefficients, expon ∈ exponents

Boolean IsZero(poly) ::= if polynomial poly does not exists return true else return false.

Coefficient coef(poly, expon) ::= if expon ∈ poly, then return coefficient else return 0.

Exponent LeadExp(poly) ::= return the largest exponent of poly.

Polynomial zero() ::= return empty polynomial.

Polynomial Attach(poly, coeff, expon) ::= If expon ∈ poly then return error else insert term <coeff, expon> into poly & return poly.

Polynomial Remove(poly, expon) ::= If expon ∈ poly, delete the term <coeff, expon> & return poly. else return error.

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

Polynomial mult(poly1, poly2) ::= return poly1 * poly2

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

(i) LeadExp(a) = 6

(ii) Coef(a, LeadExp(a)) = 25

(iii) IsZero(a) = False

(iv) Attach(a, 15, 3) ⇒ $25x^6 + 10x^5 + \boxed{15x^3} + 6x^2 + 9$ ↑ insert

(v) Remove(a, 6) ⇒ $10x^5 + 6x^2 + 9$

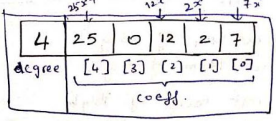
→ **Polynomial Representation:** ① Polynomials can be represented in C using a structure as shown below:

```
#define MAX_DEGREE 101
typedef struct
{
    int degree;
    float coeff[MAX_DEGREE];
} Polynomial;

Polynomial a, b;
```

* Using the variable a, the degree of the polynomial can be accessed using a.degree and the coefficients can be accessed using a.coeff[i] for i = 0, 1, 2 ...

Ex: The polynomial $a(x) = 25x^4 + 12x^2 + 2x + 7$ with degree 4 can be represented as shown:



Advantages: ① Simple

② If few terms with zero coefficients are present, it uses less space.

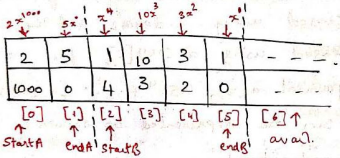
Disadv: If most of the terms are not present, then we store 0's in corresponding coeff's & occupy more space. This can be overcomed using array of structures.

② **Another method using Array of structures:-**

* can be used to store several polynomials.
* This method is used to save space.
* Each Term of a polynomial with 2 fields: coeff and expo can be stored in array location P[0], P[1], P[2] ...

```
#define MAX-DEG 100
typedef struct
{
    float coef;
    int expo;
} Polynomial;
Polynomial P[MAX-DEG];
```

Ex: $A(x) = 2x^{1000} + 5$ and $B(x) = x^4 + 10x^3 + 3x^2 + 1$ can be represented as:



* StartA – index of first term of poly a
  endA – index of last term of poly a
  StartB – index of first term of polyb
  endB – index of last term of polyb,
  avail – index of next free space, where a term of a polynomial can be stored.

Disadv: If more no. of non-zero coefficients are present, then it occupies twice the memory occupied by previous method.

→ Polynomial Addition :-

(1) Design an algorithm to add 2 polynomials using ADT Polynomial [c = a+b].

* The addition of 2 polynomials can be done by considering various cases as shown:

Case 0: Powers of two terms to be added are equal.

⇒ Assume that the 1st term of poly nomials 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 + 10

= 40 // Now add into poly C

* This can be done by calling the attach fn 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 terms ie;

a = Remove(a, LeadExp(a)); ie; $25x^6$

b = Remove(b, LeadExp(b)); ie; $15x^6$

∴ $a = 10x^5 + 6x^2 + 9$ and

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

⇒ The 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 1: Power of 1st term of poly a is greater than power of 1st term of poly b. ie;

$a = 10x^5 + 6x^2 + 9$ ⟶ 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 of poly a ie;

➡ Complete code :

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

default : If I^{st} term of poly B > I^{st} term of poly A

```
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 until one or both polynomials become empty. If one of the poly becomes empty, copy the remaining terms into C.
* The 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));
    }
}
// Insert any remaining items of a or b into c.
```

② To add Two polynomials using array of structures :-

```
void polyAdd(Polynomial *P, int startA, int endA, int startB,
             int endB, int *startC, int *endC)
{   *startC = avail; // get index of 1st term of resulting poly
    while(startA <= endA && startB <= endB)
    {
        switch(COMPARE(P[startA].expon, P[startB].expon))
        {
            case 0: coef = P[startA].coef + P[startB].coef;
                    if(coef != 0) attach(coef, P[startA].expon, P);
                    startA++; startB++; break;
            case 1: attach(P[startA].coef, P[startA].expon, P);
                    startA++; break;
            default: attach(P[startB].coef, P[startB].expo, P);
                     startB++;
        }
    }
while(startA <= endA) // Add remaining terms of poly A.
{ attach(P[startA].coef, P[startA].expo, P);
  startA++;
}
while(startB <= endB)  // Add remaining terms of poly B.
{ attach(P[startB].coef, P[startB].expo, P);
  startB++;
}
```

* endC = avail - 1;
}

Sparse Matrices :-

* A sparse matrix is a matrix which has more number of zero elements or a very few non-zero elements.
* It can be a 1D, 2D etc.

Ex: a

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

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

* The Abstract Data Type : An ADT sparse matrix is the one that shows the various operations that can be performed on sparse matrices. It consists of various objects (ie: variables) and functions as shown below:

ADT Array is

Objects : A set of triples <row, col, val> where val is the data stored in a[row][col].

Functions : parameters used : a, b ∈ Array, MaxRow, maxCol ∈ index, n ∈ integer.

SparseMatrix Create(maxRow, maxCol) ::= Creates a 2D array with row & cols.

SparseMatrix Transpose(a) ::= Returns the transpose of a given matrix. The row elements are changed to column elements & vice-versa

SparseMatrix Add(a,b) ::= If size of two matrices are same, the fn performs sum of a and b else return 0.

SparseMatrix Multiply(a,b) ::= Let mxn and pxq are the size of matrix A & B resp.

If n and p are same then multiplication is possible. Otherwise return 0.

The matrix C:

$$c[i][j] = \sum_{k=0}^{n-1} (a[i][k] * b[k][j])$$

for i = 0 to m-1 and
j = 0 to q-1

End Array.

* **Disadvantages of Sparse matrices:** A sparse matrix contains many 0's. If we are manipulating only non-zero values, then we are wasting the memory space by storing unecessary zero values.
→ This can be overcomed by storing only non-zero values.

* **Sparse matrix Representation:** A sparse matrix can be created using the array of triples as:

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

Example: Represent the given matrix using triples in a 1D array.

* The non-zero elements of the matrix along with row & col position, starting from a[1].

	row	col	val	
a[0]	5	4	8	
a[1]	0	0	10	row 0
a[2]	0	3	40	
a[3]	1	0	11	row 1
a[4]	1	2	22	
a[5]	3	0	20	row 3
a[6]	3	3	50	
a[7]	4	1	15	row 4
a[8]	4	3	25	

	0	1	2	3
0	10	0	0	40
1	11	0	22	0
2	0	0	0	0
3	20	0	0	50
4	0	15	0	25

* The various information can be accessed as shown:
  - → The size of matrix : a[0].row, a[0].col
  - → The no. of non-zero values : a[0].val
  - → The row index of a non-zero element : a[j].row
  - → The column index of a non-zero element : a[j].col
  - → The index of non-zero element : a[j].val

* Function to read the sparse matrix as a triple

```
void readspmatrix(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.

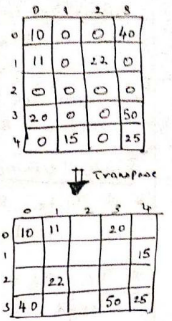
⇨ How to represent a sparse matrix into 1D array with triples as well as transform it into a transpose matrix.

Sparse matrix.

	rows	col	val
a[0]	5	4	8
a[1]	0	0	10
a[2]	0	3	40
a[3]	1	0	11
a[4]	1	2	22
a[5]	3	0	20
a[6]	3	3	50
a[7]	4	1	15
a[8]	4	3	25

Transpose Sparse matrix

	row	col	val
b[0]	4	5	8
b[1]	0	0	10
b[2]	0	1	11
b[3]	0	3	20
b[4]	1	4	15
b[5]	2	1	22
b[6]	3	0	40
b[7]	3	3	50
b[8]	3	4	25



* Function to find the transpose of 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++;
            }
}
```

Strings :-

* One of the primary applications of computers today is in the field of word processing. Such processing usually involves some type of pattern matching, as in checking to see if a particular word S appears in a given text T.
* Computer terminology usually uses the term "String" for a sequence of characters (rather than word).

Basic Terminology of strings :-

* A finite sequence of zero or more characters is called as a string.
* The no. of characters in a string is called its length. The string with zero characters is called the empty or NULL string.
  Ex: "The End" , "Welcome"
* Let S1 and S2 be strings. The string consisting of the characters of S1 followed by the characters of S2 is called concatenation of S1 and S2.
* A string Y is called a substring of a string S, if there exists strings X and Z such that,

$$S = XYZ$$

→ **Storing Strings :-** Strings are stored in 3 types of structures: Fixed-length, variable-length and Linked structures.

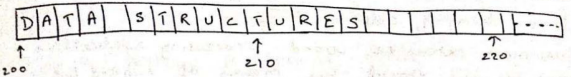
(i) **Record-Oriented, Fixed Length storage :**

* Here each line of print is viewed as a record, where all records have the same length ie; each record accomodates the same number of characters.

Ex: Since data are frequently input on terminals

appear in memory as pictured below:

String → "DATA STRUCTURES"



Advantages: ① Ease of accessing data from any given record.

② Ease of updating data in any given record (as long as it doesn't exceed the record length)

Disadvantages: ① Time is wasted reading an entire record if most of the storage is blank.

② Certain records may require more space than available.

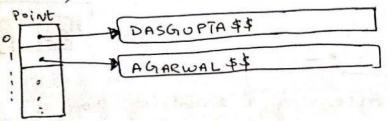
(ii) Variable-Length storage with fixed Maximum:

* can be done in 2 ways,

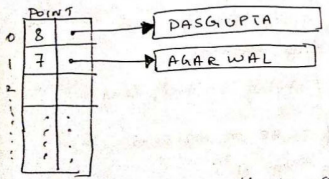
① use a marker, like two dollar signs ($$), to signal the end of the strings.

② List the length of the string – as an additional item in the pointer array.

Figure a) 1st method using a marker (Sentinels)



(b) 2nd Method, record whose lengths are listed

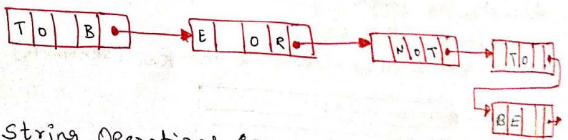


* However, these methods are usually insufficient when the strings and their lengths are frequently changing.

(iii) Linked Storage:-

* The fixed length memory cells do not easily lend themselves to operations in word processing like changing, deleting etc.
* For most extensive word processing applications, strings are stored by means of Linked List.
* Linked List - is a linearly ordered sequence of memory cells, called nodes, where each node contains an item called a link - which points to the next node in the list.
* Strings may be stored in Linked List as follows:
→ Each memory is assigned a fixed no. of characters and a link which gives the address of the cell containing the next node.

Figure: 4 characters per node representing the string "TO BE OR NOT TO BE".



String Operations:-

(i) Substring: Accessing a substring from a given string requires 3 pieces of info: name of the string, the position of 1st character of the substring in the given string and also the length of the substring.

Syntax: SUBSTRING(string, initial, length)

Example: SUBSTRING('TO BE OR NOT TO BE', 4, 7)
= 'BE OR N'

(ii) Indexing: (Pattern matching)

* refers to finding the position where a string pattern P first appears in a given string text T.

Syntax: INDEX (text, Pattern)

- If the pattern P doesn't appear in T, then INDEX is assigned the value 0.

Ex: T = 'HIS FATHER IS THE PROFESSOR'; then INDEX(T, 'PROF') = 19

(iii) Concatenation (S1 // S2)

* Let S1 and S2 be strings. The concatenation of string S1 and S2, is denoted by S1 // S2 - is the string consisting of the characters of S1 followed by S2.

Ex: S1 = 'MARK" S2 = 'TWAIN' ; then

S1 // S2 = 'MARKTWAIN'

(iv) Length: The no of characters in a string.

⇒ Length (strname)

Ex: LENGTH ('COMPUTER') = 8

Pattern Matching Algorithms:-

* Pattern matching is the problem of deciding whether or not a given string pattern P appears in a string text T.

Remarks: Assume that length of P does not exceed length of T.

* This section discusses 2 algorithms.
* characters are denoted as lowercase letters (a, b, c ...), and exponents used to denote repetition.

$a^2 b^3 a b^2$ ⇒ aabbbabb

$(cd)^3$ ⇒ cdcdcd

* empty string $\wedge$ (Lambda)

* Concatenated string = X.Y or XY

① First pattern matching Algorithm:

Compares a given pattern P with each of the substrings of T, left to right, until match is found.

② Second pattern matching algorithm:

uses a table which is derived from a particular pattern P, but it is independent of T.

① First Algorithm:

Let W_K = SUBSTRING(T, K, LENGTH(P))

W_K = substring of T having same length as P and beginning with K^{th} character of T.

→ First we compare P, char by char, with W_1.

→ If all characters match, then $P = W_1$ and so P appears in T and INDEX(T, P) = 1

→ If any one character in P is not same, then $P \neq W_1$ and we shift one character of W_1 to next obtaining W_2.

→ If $P \neq W_2$, Proceed to W_3 and so on until P is found in T.

→ The max value of K is: LENGTH(T) − LENGTH(P) + 1

Ex: Text T = 20 characters

Pattern P = 4 characters.

⇒ $W_1 = T_1\ T_2\ T_3\ T_4$

$P = P_1\ P_2\ P_3\ P_4$

} If all characters are same, then $P = W_1$ & INDEX(T,P)=1

else

$w_2 = T_2\ T_3\ T_4\ T_5$
$P = P_1\ P_2\ P_3\ P_4$
} If all characters are same $P = w_2$, INDEX(T,P) = 2

else

$w_3 = T_3\ T_4\ T_5\ T_6$.
$P = P_1\ P_2\ P_3\ P_4$
} If all characters are same $P = w_3$, INDEX(T,P) = 3

else ---. so on.....

till last $w_{17} = T_{17}\ T_{18}\ T_{19}\ T_{20}$
$P = P_1\ P_2\ P_3\ T_4$
} If $P = w_{17}$, INDEX(T,P) = 17
else INDEX(T,P) = 0 ie; Not found.

$\therefore\ K = 20 - 4 + 1 = 16 + 1 = 17$

Pseudocode / algorithm: (Pattern Matching) P and T are strings with length R and S respectively. This algo. finds INDEX of P in T.

Step 1. [Initialize] Set K = 1 and MAX = S - R + 1
2. Repeat Step 3 to 5 while K <= MAX
3. Repeat for L = 1 to R [test each char of P]
If P[L] != T[K + L - 1], then Goto 5.
[End of Inner loop]
4. [Success] Set INDEX = K and exit
5. Set K = K + 1
[End of step 2 outer loop]
6. [Failure] set INDEX = 0
7. Exit

② Second pattern matching Algorithm

* It uses a table which is derived from a particular pattern P, but it is independent of text T.

→ Consider, P = aaba

First, we give reason for table entries and how they are used. Suppose T = $T_1 T_2 T_3 T_4$ ... and first 2 characters of T make P ie; T = aa...

P = aa then T may be,

(i) T = aab... (ii) T = aaa.... (iii) T = aax

⇒ b or a or x (any other char)

→ Suppose T_3 = b, then read T_4 to see which character. If T_4 = a, then P = w_1

If T_4 = b or

If T_4 = x then P ≠ w_1

* we proceed, to T = $T_2 T_3 T_4 T_5$ and read the further characters for w_2.

III[ly] In w_2 = aa., two characters match, then advance to T_4 = a, T_4 = b or T_4 = x

then T = aaa, T = aab or T = aax

W.K.T., If T_5 = a then P = w_2 otherwise

T_5 = b or T_5 = x then P ≠ w_2

and so on, keep checking for sequence of characters till the LENGTH(T) − LENGTH(P) + 1

→ we write the table in the state diagram/graph as shown:

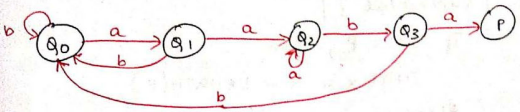
$f(q_i, t)$

	a	b	c
q_0	q_1	q_0	q_0
q_1	q_2	q_0	q_0
q_2	q_2	q_3	q_0
q_3	P	q_0	q_0

$q_0 = \Lambda$ (empty)

$q_1 = a$, $q_2 = a^2$

$q_3 = a^2b$, $q_4 = a^2ba = P$



Ex① Suppose T= aabcabq , P = aaba

Beginning with q_0, we use the characters of T and graph (or table) to obtain foll sequence of states.

$$q_0 \xrightarrow{a} q_1 \xrightarrow{a} q_2 \xrightarrow{b} q_3 \xrightarrow{c} q_0 \xrightarrow{a} q_1 \xrightarrow{b} q_0 \xrightarrow{a} q_1$$

∴ we did not reach P, ∴ P is not in T.

Ex② Suppose T = abcaabaca , P = aaba

$$q_0 \xrightarrow{a} q_1 \xrightarrow{b} q_0 \xrightarrow{c} q_0 \xrightarrow{a} q_1 \xrightarrow{a} q_2 \xrightarrow{b} q_3 \xrightarrow{a} P$$

Here we obtain pattern P at state s_8. Hence P does appear in T and its index is

$8 - (\text{LENGTH}(P)) = 8 - 4$

$= 4$,,

Pseudocode / Algorithm: The pattern matching table $f(q_i, T)$ of a pattern P is in memory and input is an N-character string $T = T_1 T_2 \ldots T_N$. This algorithm finds INDEX of P in T.

1. [Initialize] Set $K = 1$ and $S1 = S_0$
2. Repeat Step 3 to 5 while $S_K \neq P$ and $K >= N$
3. Read T_K
4. Set $S_{K+1} = F(S_K, T_K)$ [finds next state]
5. Set $K = K + 1$ [updates counter]
   [End of step 2 loop]
6. [Successful ?]
   If $SK = P$, then
   INDEX = K − LENGTH(P)
   Else :
   INDEX = 0
7. Exit