DATA STRUCTURES AND APPLICATIONS [BCS 304]

MODULE-1

Introduction:

- Data Structures
- Classifications (Primitive & Non-Primitive)
- Data structure operations
  - Traversing
  - Inserting
  - Deleting
  - Searching
  - Sorting
- Review of Pointers and Dynamic Allocation.
- Review of Arrays and Structures: Arrays
- Dynamic Allocated Arrays.
- Structures and Unions.
- Demonstration of representation of Polynomials.
- Sparse Matrices with arrays.
- Representation of Linear Arrays in Memory
- Strings
- STACKS: Stacks
- Stacks using dynamic Arrays
- Evaluation and Conversion of Expressions.

MODULE - 1

Data structures and Applications [BCS304]

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 an 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 structure includes:

* defining operations that can be performed on data.
* Representing data in memory.
* determining amount of memory required to store, and the time needed to process the data.

Need for Data Structures:

The computers are electronic data processing machines. In order to solve a particular problem, one must know

⇒ How to represent data in a computer?
⇒ How to access them?
⇒ What are the steps to be performed to get the needed output?

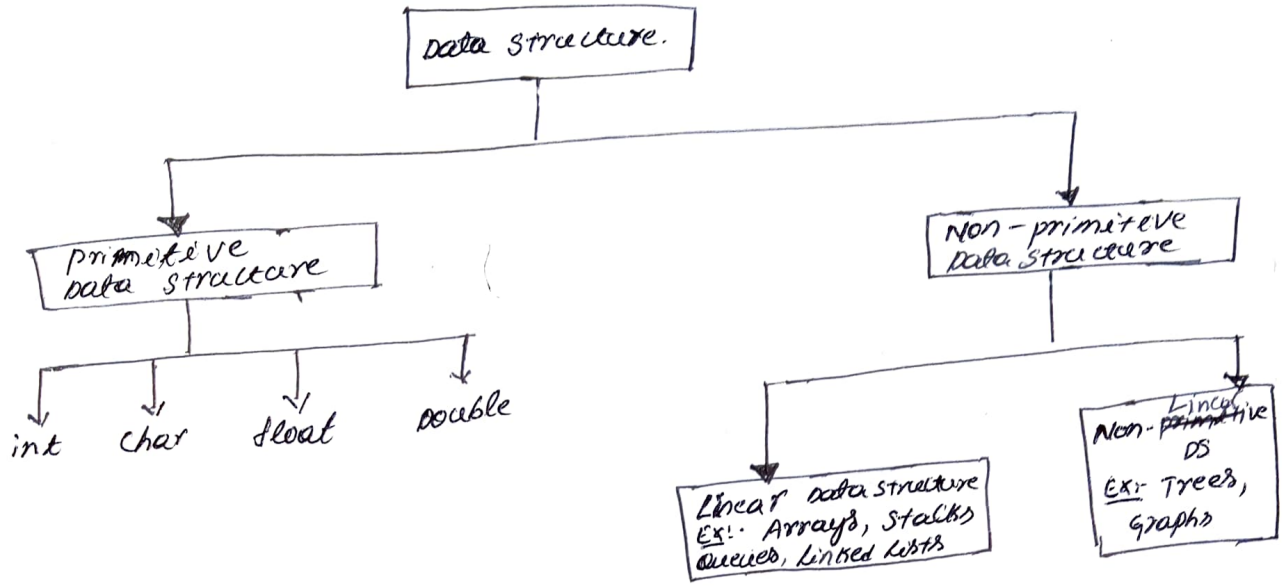
The above mentioned tasks can be achieved with the knowledge of data structures and Algorithm.

Classification of data structures

Data structures are classified as

i) primitive data structure.

and ii) Non-primitive data structure.



i) primitive data structures:-

These are the basic (or) fundamental standard data types, which are used to represent single values.

Example:- int, float, char, double.

ii) Non-primitive datastructures:-

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

Example:- Arrays, Stacks, Queues, Linked list, Trees etc.,

Non-primitive data structures are further classified into Linear and Non-linear datastructures.

Linear Datastructures:-

A data structure is said to be linear, if its elements are stored in a sequential order.

Example:- Arrays, Stacks, Queues, Linked list

Non-linear data structures:

A Non-linear data structure is one in which the data is not arranged in a sequential order.

Example :- Trees, Graphs.

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

Data structure operations:

Data stored in a memory of computer are processed by some operations listed below.

i) Traversing :-
Accesing every element exactly once so that certain elements may be processed.

ii) Searching :-
Finding the location of record with a given key value (or) finding the location of all records which satisfy one (or) more conditions.

iii) Insertion :-
Adding a new record into the data structure.

iv) Deletion :-
Removing a record from the structure

⇒ Sometimes two (or) more operations may be used in a given situation

Example :- When we want to delete the element with given key
i.e., ~~first~~ search for location and delete.

Other operations like:

Sorting:- Arranging the records in Ascending/descending order.

Merging:- Combining two data structures.

Review of Pointers

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

step	Steps to use pointers	example
1.	declare a variable.	int a;
2.	declare a pointer variable.	int *pa;
3.	Initialize a pointer variable.	pa = &a;
4.	Access the pointer.	printf("value of a is %d", *pa);

pointer declaration

data type * pointer_var;

Ex: int *pa;

pointer initialization

pa = &a;

Here the pointer pa, holds the address of ■ variable 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 an important task in computer programming.
* There are two types of memory management
  i) Static memory management
  ii) Dynamic memory management.

i) Static memory management:

The allocation & deallocation of memory is done during compilation time of the program.

Example:- int a[10];

ii) Dynamic memory management:

The memory management of 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 compilation time.	memory allocation is performed at run-time.
2.	prior to allocation, fixed size of allocation of memory has to be declared	No need to know size of memory prior to allocation.
3.	wastage/shortage of memory occurs. Example: Array	Memory is allocated as per requirement. Example:- Linked list

There are four functions used in dynamic memory allocation

i) **malloc()**: This function 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, therefore 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 = (data type *) malloc (size);

Here ptr is a pointer variable of type data type, data type - Any c datatype (or) user defined data type
size - Nor of bytes required.

Example:- int *ptr;
ptr = (int *) malloc (10);

calloc () :- Allocates multiple blocks of memory.

* It is similar to malloc, but it initializes the allocated memory to zero.

syntax:- ptr = (data type *) calloc (n, size)

n → no. of blocks to be allocated

Example:- ptr = calloc (20, sizeof (int));

This function computes memory required for 20 x 2 bytes (for int) = 40 bytes of memory blocks is allocated.

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~~ ptr = (data type *) realloc (ptr, size);
↓
new size

Example:- char *str;

str = (char *) malloc (10);

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

free () :- deallocates the allocated memory which was done using malloc (), calloc () and realloc ().

Syntax:- free (pointer name)

Example:- free (str);

Review of Arrays

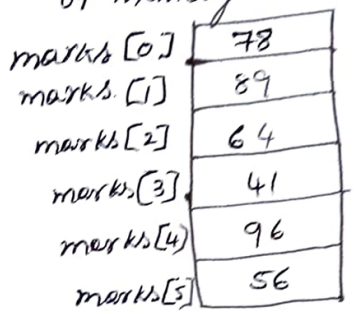
Definition of Array: An array is a collection of similar data items under a common name. The elements of an array are of same type and each element can be accessed using same name, with different index values.

[An array is defined as a set of pairs <index, value>, where index is position and value is data stored in position index]

Example: - Array of marks

<index, value>
<0, 78>
<1, 89>
<2, 64>
<3, 41>
<4, 96>
<5, 56>

Array implemented as consecutive set of memory locations.



Abstract Data type (ADT) for Array

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

ADT array is

object: A set of pairs <index, value> where value is data stored at index position in an array. For a single dimensional array index values are [0, 1, 2, 3, ... n-1]

For two dimensional array index values are

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

functions: $A \in$ Array, $i \in$ index, $x \in$ item, $n \in$ integer

return type	function name	operation performed
Array	create (n,a) ::=	returns array a size n
item	retreive (a,i) ::=	if $i \in$ index then return item stored in array a at index position i.
Array	store (a,i,x) ::=	if $i \in$ index then return array A, by storing x at index position x at 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};

Dynamically allocated Arrays:

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

similarly 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 number 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);
  }
```

Structures and Unions

A Structure is a user defined data type.
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 roll-no;
};
```

where struct - is a keyword (for structures)
type1, type2 ... typen - basic datatypes like int, char, float, double
var1, var2 ... varn - are members of the structures.
tag name - Any identifier.

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

```
struct student ai, ece, cse;
```
↓ keyword, ↓ struct name, ↓ structure variable.

@ once structure variables are declared, compiler allocates memory for the structure variables.

Two ways to declare variables

(1)
```
struct student
{
  - - -
  - - -
  - - -
};
```

(2)
```
struct student
{
  - - -
  - - -
  - - -
} ai, ece, cse;
```

Memory representation of Structure: ai, ece, cse

ai

name	30 bytes
age	2 bytes
roll-no	2 bytes

ece

name	30 bytes
age	2 bytes
roll-no	2 bytes

cse

name	30 bytes
age	2 bytes
roll-no	2 bytes

(*) 34 bytes of memory is allocated each for the structure variable ai, ece & cse. (*)

Accessing structure members

Using dot (.) operator, structure member can be accessed

Example: ① ai.name = "Kumar";

② printf ("%d", ai.roll-no);

Type-defined structure

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

* most powerful way of defining the structure.

2 methods

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

Example: typedef struct
{
char name[30];
int age;
int roll-no;
} student

create variables:
typedef_name variables;

EX: student ai, ece;

② struct structure name
{
type1 member1;
type2 member2;
:
};
typedef struct structure_name
typedef_name;

EX: struct student
{
char name[30];
int age;
int roll-no;
};

typedef struct st;
create variable:
st ai, ece;

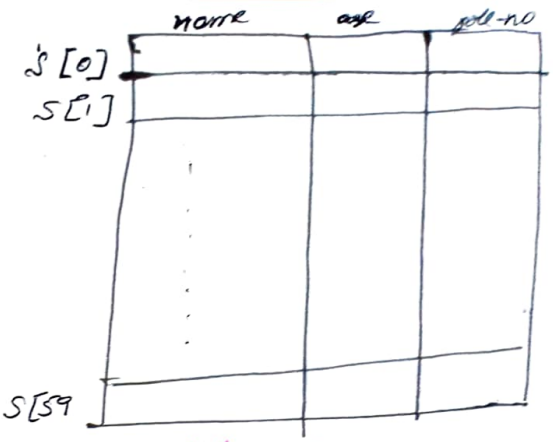
Array of structures

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

Example:- If there are 60 students in a class, then the records 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 rdl-no;
} s[60];
```

memory representation



Nested structures / structure within a structure

* Nested structure – embeds a structure within another structure

Ex :-

```
struct date
{
int month;
int date;
int year;
};

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

⇒ In this example a person who is born on 24, Sept 1978, would have the values for the struct dob set as:

```
p1.dob.month = 09;
p1.dob.day = 24;
p1.dob.year = 1978;
```

Self-referential structure :-

A self referential structure is a structure defenetion which includes at least one member that is a pointer to the structure of its own kind.

i.e., pointer stores the address of the structure of the same type.

Example :-

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

Advantages :-

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

(2) useful in applications that involve data structures like linked list and trees.

Unions :-

* Union is a user defined data type 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 seperately, whereas unions allocates one common storage space for all its members seperately, whereas unions allocates one common storage space for all its members.

Example :-

```
Union student
{
char name[20];
int age;
int roll-no;
};
```

⇒ Here union allocates only 20 bytes of memory, since the name field requires the maximum space (1 byte × 20) compared to age and roll-no (only 2 bytes each)

However, a structure would allocate 24 byte total (includes the size of all its members)

```
Union tagname
{
type1 member1;
type2 member2;
  :
typen member_n;
};
```

(OR)

```
typedef union
{
- - -
- - -
- - -
} tag-name;
```

Declaration of union variables

Union tagname var1, var2, --- varn

Ex:- Union student S1, S2, S3.

Polynomials

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

$a x^e$ — e → exponent, x → variable, a → co-efficient.

* The largest exponent (leading exponent) of a polynomial is called its degree.

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

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 like $35x^0$.

*. Abstract data type:- [ADT]

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: $p(x) = a_1 x^{e_i} + \cdots + a_n x^{e_n}$

where a_i and e_i are co-efficients and exponents respectively and are integers $>= 0$.

Functions:

① parameters used: poly, poly1, poly2 ∈ polynomials.
coef ∈ coefficients
expon ∈ exponents.

② Boolean IsZero(poly) ::= if polynomial poly does not exists return true, else return False.

③ coefficient Coef(poly, expon) ::= if expon ∈ poly, then return coefficient, else return 0

④ Exponent Lead exp(poly) ::= return the largest exponent of poly

⑤ polynomial Zero() ::= return empty polynomial

⑥ polynomial attach (poly, coef, expon) ::= if expon ∈ poly, then
return error,
else insert term <coeff, expon>
into poly, and return poly.

⑦ polynomial remove (poly, expon) ::=
if expon ∈ poly,
delete the term <coeff, expon> &
return poly.
else return error.

⑧ polynomial add (poly1, poly2) ::= return poly1 + poly2

⑨ 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

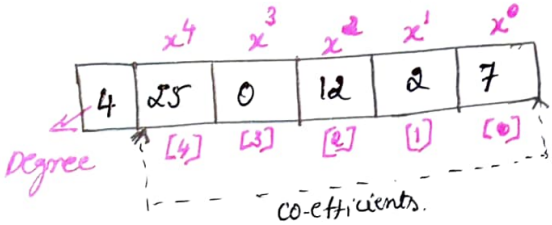
Polynomials can be represented in C using a structure as shown below:

```
#define MAX_DEGREE 101
typedef struct
{
  int degree;
  float coeff[MAX_DEGREE];
} polynomial;
polynomial a, b;
```

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

Example :-

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



Advantages :- ⊛ Simple

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

Disadvantages :-

If most of the terms are not present, then we store 0's in corresponding co-efficients, and occupy more space; This can be overcome using array of structures.

Second method of using Array of Structures :-

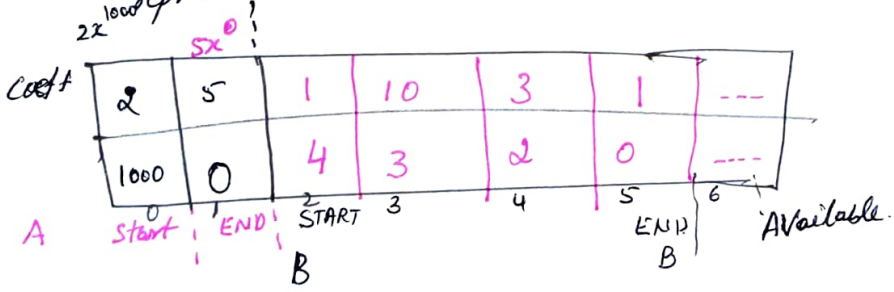
⊛ Can be used to store several polynomials.

⊛ This method is used to save space.

⊛ Each term of a polynomial with two fields : coeff and exponent can be stored in array location p[0], p[1], p[2] .....

```
# define MAX-DEG 100
  typedef struct
  {
    float coeff;
     int expo;
    } polynomial;
  polynomial p[MAX-DEG]
```

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



* START A : index of first term of a polynomial A
* END A : index of last term of polynomial A

START B - index of first term of polynomial B

END B - index of last term of polynomial B

Available - index of next free space, where a term of polynomial can be stored.

Disadvantage

If more nor of non-zero co-efficients are present, then it occupies double the memory occupied by previous method.

Polynomial Addition

① Design an algorithm to add two polynomials using ADT by considering various cases as shown.

Case 0; powers of two terms to be added are equal.

Assume that the first term of the following polynomials are to be added

$a = 25x^6 + 10x^5 + 6x^2 + 9$ ⇒ Lead expon(a) = 6

$b = 15x^6 + 5x^4 + 4x^3$ ⇒ Lead expon(b) = 6

$c = 40x^6$

∴ Lead Exp(a) and Lead Exp(b) are same, they can be added:

Scem = coeff(a, Lead exp(a)) + coef(b, Lead Exp(b))

= 25 + 15

= 40

② This can be done by calling attach function as.

If (Scem != 0) Attach(c, Scem, Lead Exp(a));

③ Now, move to the next term of poly a, and poly b, by removing the added terms i.e.,

a = Remove(a, Lead Exp(a)); i.e, $25x^6$

b = Remove(b, Lead Exp(b)); $15x^6$

$\therefore a = 10x^5 + 6x^2 + 9$ and

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

The complete code can be written as

```
if (Lead Exp (a) == Lead Exp(b))
  {
Sum = coef (a, Lead Exp (a)) + coef (b, Lead Exp(b)));
 if (Sum != 0)   Attach (c, Sum, Lead Exp(a));
   a = Remove (a, Lead Exp (a));
   b = Remove (b, Lead Exp (b));
  }
```

Case 1: power of 1st term of poly a is greater than power of 1st term poly b. i.e.,

$a = 10x^5 + 6x^2 + 9$ → Lead exp(a) = 5

$b = 5x^4 + 4x^3$ → Lead Exp(b) = 4

$c = 10x^5$

$\because 10x^5 > 5x^4$, we attach $10x^5$ into c, and then we remove it before moving to the next of poly a i.e.,

Complete code.

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

Default: If 1st term of poly B > 1st term of poly A

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

* All the above cases have to be repeated untill one (or) both polynomials 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 (LeadExpo(a), LeadExpo(b))
  {
  case 0 : sum = Coef (a, LeadExpo (a)) + Coef (b, LeadExpo(b));
           if (sum != 0) Attach (c, sum, LeadExpo (a));
           a = Remove (a, LeadExpo(a));
           b = Remove (b, LeadExpo (b));
           break;
  case 1 : Attach (c, Coef (a, LeadExpo(a)), LeadExp (a));
           a = Remove (a, LeadExp(a));
           break;
  default : Attach (c, Coef (b, LeadExp (b), LeadExp (b));
           b = Remove (b, LeadExp(b));
  }
 }
```

(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 (coeff, 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[start.B].coef, P[startB].expo P);
   startB++;
   }
    * endC = avail - 1
   }
```

Sparse Matrices:-

* A sparse matrix is a zero matrix which has more number of zero elements (or) a very few non-zero elements.
* It can be a single dimension, two dimension etc.,

Ex:-

a

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

end

	0	1	2	3
0	10	0	0	40
1	11	0	20	0
2	0	0	0	0
3	20	0	0	90

* 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 (i.e. variables) and functions as given 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

Sparse Matrix Create : (maxRow, MaxCol) – Creates TWO dimensional array with row and columns.

Sparse matrix Transpose (a) ::= Returns the transpose of a given matrix. The row elements ~~are~~ are change to column elements and vice-versa

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

sparse matrix multiply (a, b) ::= Let m x n and p x q are the size of matrix A and B respectively, If n and p are same then multiplication is possible. otherwise return 0.

The matrix C :

$$C[i][j] = \sum_{k=0}^{n-1} \left(a[i][k] * b[k][j] \right) \quad \text{for } i=0 \text{ to } m-1 \text{ and } j=0 \text{ to } q-1$$

Disadvantages of Sparse matrix

A Sparse matrix contains many 0's. If we are manipulating only non-zero values, then we are wasting the memory space by storing unnecessary 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 tripples 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 single dimensional array.

* The non-zero elements of the matrix along with Row and col positions 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

Sparse matrix various information are

→ The size of matrix : a[0].row , a[0].col
→ The number of non-zero values : a[0].Val
→ The row index of a non-zero : a[j].row element
→ 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.

To represent a sparse matrix into single dimensional 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

	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

	0	1	2	3	4
0	10	11		40	
1					15
2		22			
3	40			50	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++;
   }
}
```

Multidimensional Arrays

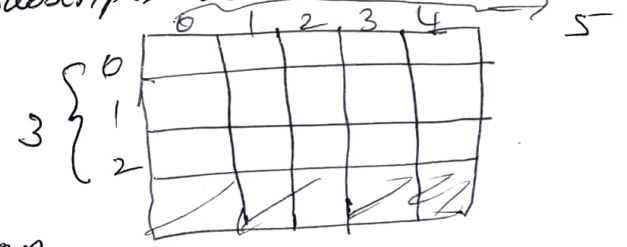
An array that has more than one subscripts are called as multidimensional arrays.

Example:- 2D Arrays, 3D Array 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 5 integers.
* The elements of this Array can be accessed by specifying a row number and a column number

Ex: the element in row 1 and column 3 can be represented as a[1][3]

(A) The no. of elements in the array is the product of ranges of all its dimensions i.e., the above array contains $3 \times 5 = 15$ elements.

(B) To implement a 2D array in a linear fashion, there are two methods: Row major and column major order. In row-major order; the elements are arranged row-by-row manner, where as 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]

Row-major elements

C

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]

Column-major elements

Representation of 2-D 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. i.e., 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.

(C) To calculate the address of an arbitrary element ar[i1][i2] apply the following equation

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

Example:- To calculate ar[0][2]

$$ar[0][2] = 200 + (0 * 3 + 2) * 2$$
$$= 200 + (0 + 2) * 2$$
$$= 204$$

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

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 character 'V' appears in an entered 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 S_1 and S_2 be strings. The string consisting of the characters of S_1 followed by the characters of S_2 is called concatenation of S_1 and S_2.
- A string Y is called a substring of string S, if there exists strings X and Z such that, $S = XYZ$.

Storing strings:-

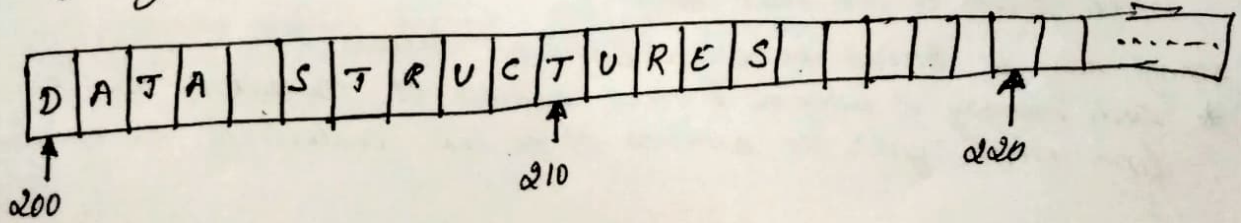
Strings are stored in 3 types of structures: Fixed-length, variable-length and linked structures.

i) Record-length, Fixed length storage.

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

Eg:- Since data are frequently input on terminals appear in memory as shown below

String → "DATA STRUCTURES"



Advantages :-

① Ease of accessing data from any given record.

② Ease of updating data on any given record (as long as it does not exceed the record length)

Disadvantages :-

✱ Time is wasted in reading an entire record if most of the storage is blanks.

✱ certain records may require more space than available.

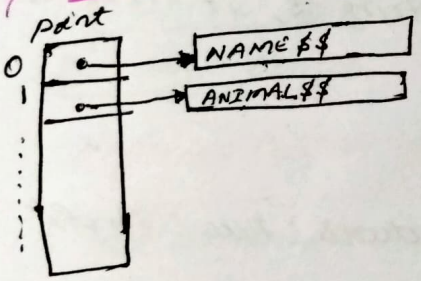
ii) Variable-length storage with fixed maximum.

This can be done in two ways

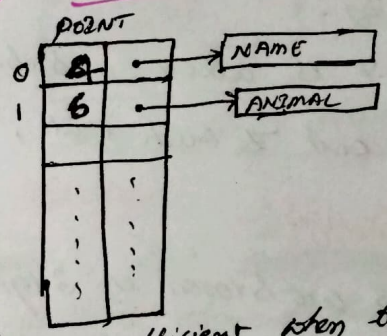
a) use of marker, like two dollar signs ($$), to indicate the end of the strings.

b) list the length of the string - as an additional item in the pointer array.

a) marker method



b) 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..

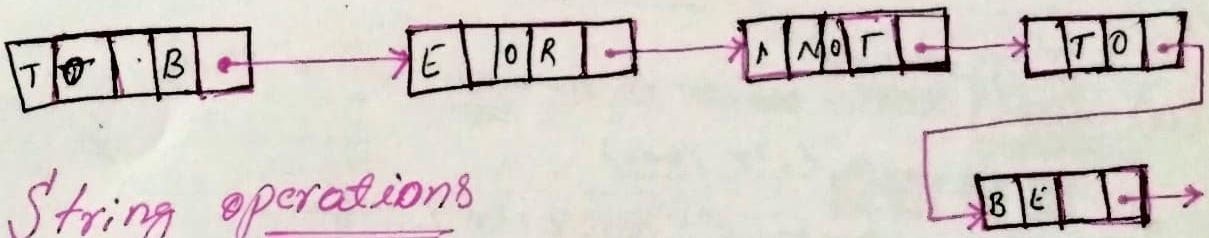
⇒ for most extensive word processing applications, strings are stored by means of linked list.

⇒ 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 number 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 information
a) Name of the string
b) position of first character of the substring in the given string.
c) length of the substring.

Syntax:- SUBSTRING (string, initial, length)

~~Accessing a substring from a given string requires 3 pieces of info~~

Example:

SUBSTRING ('TO BE OR NOT BE', 4, 7)

= 'BE OR N'

(ii) indexing:- This 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 appears in T, then index is assigned the value 0

Example:- T = 'HIS SHIRT IS BLUE', then

INDEX (T, 'BLUE') = 14

iii) concatenation:- (S_1||S_2)

If S_1 and S_2 are two strings. The concatenation of string S_1 and S_2, is denoted by S_1||S_2 - is the string consisting of the characters of S_1 followed by S_2.

Example:- S_1 = "SANDEEP" , S_2 = "KUMAR" , then

$S_1 || S_2$ = "SANDEEPKUMAR"

(iv) Length:- The number of characters in a string

=> Length (str-name)

Example:- 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 the length of P does not exceed length of T.

① This has Two Algorithms

⇒ characters are denoted as lowercase letters and exponents uses to denote repetation. (a, b, c, ...), a^2, b^3 ab^2

⇒ a a bbb abb

$(cd)^3$ ⇒ cd cd cd

② empty string ^ (Lambda)

③ 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.

First Algorithm

Let W_k = SUBSTRING $(T, k, \text{LENGTH}(P))$

W_k = Substring of T having same length as P and beginning with k^{th} character of T.

→ First we compare P, character by character, 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, the $P \neq W_1$ 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

Example:- 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 $W_{17} = T_{17}\ T_{18}\ T_{19}\ T_{20}$
$P = P_1\ P_2\ P_3\ P_4$
} If $P = W_{17}$, INDEX $(T, P) = 17$ else INDEX $(T, P) = 0$ i.e., 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 algorithm finds INDEX of P in T.

Step: 1. Initialize, set K=1 and MAX = S-R+1
2. Repeat steps 3 to 5 while K <= MAX
3. Repeat for L=1 to R [test each character of P]
If P[L] != T[K+L-1], then Goto step 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 i.e., 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

like 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~~.

LENGTH(T) - LENGTH(P) + 1

→ We write the table in the state diagram/graph as shown below

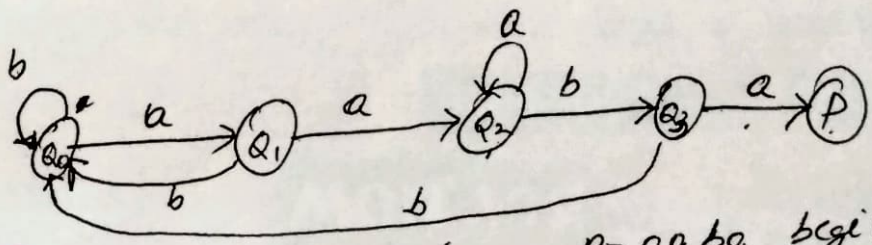
$t(Q_i, t)$

	a	b	x
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

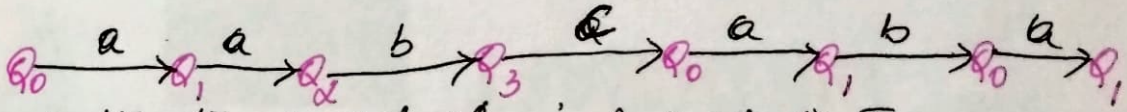
Q_0 = (empty)

$Q_1 = a$, $Q_2 = aa$ (a^2)

$Q_3 = aab$ (a) a^2b $Q_4 = aaba = a^2ba$.

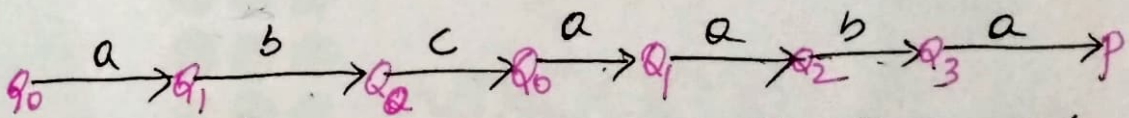


Example ①:- Suppose $T = aabcaba$, $P = aaba$ beginning with Q_0 we use the characters of T and graph (or table) to obtain following sequence of states.



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

Example ②:- Suppose $T = abcbabaca$, $P = aaba$



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 patter P is in memory and input is an N-character string $T = T_1, T_2, T_3 \cdots T_N$ This algorithm finds index of P in T.

1. [Initialize] set $K=1$ and $S_1 = Q_0$
2. Repeat step 3 to 5 while $S_K \neq P$ and $K \leq N$
3. Read T_K
4. Set $S_{K+1} = f(S_K, T_K)$ [Finds next state]
5. Set $K = K+1$ [update counter]
   [End of step 2 loop]
6. Is success?, then
   If $S_K = P$, then
   INDEX = K - LENGTH (P)
   else:
   INDEX = 0
7. EXIT.

Stacks

* A Stack is a linear list of elements in which an element may be inserted (or) deleted at only one end (i.e., called top of the Stack)

* Stack is a data structure which works on the principle "Last in First out LIFO". i.e, the last element inserted will be the first element to be deleted.

operations on Stacks

1. PUSH — inserting an element into the Stack
2. POP - Removing / deleting an element from the Stack.
3. overflow - check whether Stack is full (or) not.
4. under flow - Checks whether Stack is empty (or) not.

Abstract data type Stack (or) ADT Stacks.

* An ADT Stack is the one that shows the various operations that can be performed on Stacks.
* An ADT can be written as shown below.

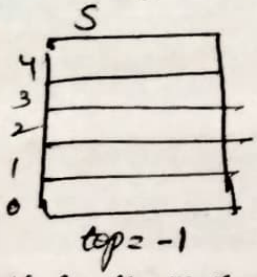
ADT Stack is:
objects: parameters used: Stack_SIZE ∈ positive integer
S ∈ Stack, item ∈ element

	Return-type	Function name	operations
1	Stack	Create (STACK_SIZE) ::=	Creates an empty Stack, whose MAX_SIZE is STACK_SIZE
2	Boolean	IsFull (S, STACK_SIZE)	if no. of elements in Stack S is STACK_SIZE returns TRUE else FALSE
3	STACK	PUSH (S, item)	If Stack is not full, insert item onto Stack
4	Boolean	IsEmpty (S)	if no elements, return TRUE else FALSE
5	Element	POP (S)	If Stack is empty, return ERROR condition as underflow, else returns the element on TOP of Stack.

Implementation of Stacks (using arrays)

1) **Create ()**: The create stacks function can be implemented as a single dimensional array i.e.,

```
#define SIZE 5
int S[SIZE];
int top = -1;
```



Here: SIZE → Symbolic constant, specifies the max. no. of elements that can be inserted into stacks.

S[SIZE] → An array, to hold the elements of the stacks.

② **IsFull ()**: (checks for overflow)

In order to perform a push operation, we need to check whether the stacks is completely filled with elements or not.

i.e.,

```
if (top == SIZE-1)
{
printf ("Stack is FULL - overflow \n");
return;
}
```

③ **PUSH ()**:-

- inserts an element onto stacks.
- First checks whether sufficient space is available in the stacks. If available then increments the value of top by 1, and then inserts the item into stacks. e.g.,

```
Void push ()
{
int item;
if (top == SIZE-1)
{
printf ("Stacks overflow \n");
return;
}
printf ("Enter item to insert \n");
scanf ("%d", &item);
S[++top] = item;
}
```

④ isempty(): Before we delete an item from the stack, we must first check whether there are any element in the stacks. i.e.,

```
If  top == -1
  {
  printf ("stacks is empty - underflow \n");
    return
    }
```

∵ top = -1, indicates that the stack is empty.

⑤ pop(): To delete an item from the stack.

```
Void  pop()
{
if (top == -1)
 {
  printf ("stacks underflow \n");
   return;
   }
printf ("deleted item = %d", s[top--]);
}
```

⑤ display(): Print the contents of the stacks.

```
Void display()
{
int i;
if (top == -1)
 {
  printf (" Stack is empty - underflow \n");
     return
   }
printf ("contents of the stacks are : \n");
 for (i = top ; i >=0; i--)
 printf ("%d \n", s[i]);
   }
```

"C" Program to implement Stacks using Arrays.

```
#include<stdio.h>
#include<conio.>
#define SIZE 10
int S[SIZE], top=-1;
Void PUSH();
Void POP();
Void display();
int main()
{
 int ch;
 for(;;)
{
 printf("1. PUSH  2. POP
        3. DISPLAY  4. EXIT\n");
 printf("Enter your choice:\n");
 scanf("%d",&ch);
 switch(ch)
 {
 case 1: push(); break;
 case 2; pop(); break;
 case 3: display(); break;
default: printf("Invalid choice");
         exit(0);
 }
}
 return(0);
}
```

Functions

```
Void push()
 {
 int item;
 if(top==SIZE-1)
 {
 printf("Stacks overflow\n")
  return;
 }
 printf("Enter an item\n");
 scanf("%d",&item);
 S[++top]=item;
 }
Void pop()
{
if(top==-)
{
printf("Stack under flow\n");
 return;
}
printf("deleted item=%d",
                S[top--]);
 }
Void display()
 {
 int i;
if(top==-1)
 {
printf("Stacks is empty-underflow\n");
 return;
}
printf("Contents of the Stacks are:\n");
for(i=top;i>=0;i--)
printf("%d\n",S[i]);
 }
```

Stacks implementation using structures

```
#include <stdio.h>
#include <stdlib.h>
#define SIZE 5

struct stack
{
int a[SIZE];
int top;
};
struct stacks s;
 s.top = -1

void push();
void pop();
void display();
void main()
 {
  int ch;
  for(;;);
  {
printf("1. PUSH  2. POP  3. DISPLAY  4. EXIT \n");
printf("enter your choice: \n");
scanf("%d", &ch);
switch(ch)
{
case 1: push();  break;
case 2: pop();  break;
case 3: display(); break;
default: printf("Invalid choice\n"); exit(0);
 }
 }
return 0;
}
```

```
Void push ()
{
int item;
if (s.top == SIZE -1)
 {
 printf ("stacks overflow \n");
 return;
 }
printf ("Enter item: \n");
scanf ("%d", &item);
s.top++;
s.a[s.top] = item;
}
```

```
Void pop()
 {
if (s.top == -1)
  {
  printf ("stacks underflow \n");
   return;
  }
printf ("deleted item = %d", s.a[s.top]);
   s.top--;
  }
```

```
Void display()
{
if (s.top == -1)
  {
printf ("stacks is empty - underflow \n");
   return;
  }
printf ("contents of the stacks are: \n")
for (i = s.top; i >= 0; i--)
printf ("%d \n", s.a[i]);
  }
```

Stacks using Dynamic Arrays

⇒ We can allocate (or) create a stacks dynamically during run-time, when we are not sure about the size of the stack, as follows.

Assume that the stacks size is 1

```
int stacksize=1;
int *s;
int top=-1;
s=(int *) malloc (stacksize *sizeof(int));
void push
{
if (top==stacksize-1)
  {printf ("stack is full : Increase by 1 \n");
   stacksize++;
   s=(int *) realloc (s, stacksize *sizeof(int));
  }
}
void pop()
{
  if (top==-1)
    {
     printf ("stack underflow \n");
      return;
     }
printf ("item deleted = %d \n", s[top--]);
printf ("stack size decreased by 1 \n");
  stacksize--;
  s=(int *) realloc (s, stacksize *sizeof(int));
}
  main()
  {
   int ch;
    for (;;)
    {
    printf ("1. PUSH  2. POP  3. Display  4. EXIT \n");
```

```
printf("enter your choice \n");
scanf("%d", &ch);
switch(ch):
 {
case 1: push (); break;
case 2; pop () ; break;
case 3; display(); break;
default : printf ("Invalid choice \n"); exit (0);
   }
   xtarn (0)
   }
```

Application of stacks

① Used in Function calls:

Example:

```
Void main ()
{
printf ("AIML \n");
  t1 ( );
printf (" schedule \n");
   }
```

```
void t1 ( )
 {
 printf ("students");
  t2 ( );
printf ("marks \n");
  }
void t2 ( )
 {
 printf ("Good \n");
  }
```

---

② used in conversion of expressions i.e., arithmetic expressions.

③ evaluation of expressions : An arithmetic expression represented in the form of either postfix (or) prefix.

④ used in Recursion.

Applications of Stacks.

Polish Notations

* An arithmetic expression can be of 3 types.

a) Infinix expression - placing the operator between the operands eg:- +ab
(Polish Notation)

b) postfix expression - placing the operator after the operands eg:- ab+
(Reverse polish notation) -

The computer usually evaluates arithmetic expression in 2 steps

a) First, it converts the expression (infinix) into a postfix notation.

b) second, it evaluates the postfix expression.

a) Conversion/Transform infinix expressions into postfix expressions

Algorithm

Step 1: push # onto stack.

Step 2: scan the expression from left to right and repeat step 3 to 6 for each element.

Step 3: If a '(' paranthesis is present, push it on to the stack.

Step 4: If an operand is encountered, add it into the postfix P.

Step 5: If an operator is encountered, then

i) checks whether the precedence of the current operator on top of the stacks is greater than (or) equal to the precedence of the scanned operator, If so then go on poping all the operators from the stacks and place them in the postfix P.

ii) otherwise (else) push the scanned operator on to stacks.

Step 6: If an ')' right paranthesis is encountered, then go on poping all the items from the stack and place them in postfix expression; till we get the matching left '(' paranthesis.

Note: Do not push the '(' left paranthesis on to postfix 'P'

Step 7: Check if any operator (or) symbols are present in the stack. If so, then pop them and place it onto postfix 'p'.

Example: convert the following expressions into postfix expressions using the above algorithm.

① $(a*b)+c$
0 1 2 3 4 5 6

initial top = -1 and post = 0

steps	Symbol	postfix (P)	Stacks	Variables
initial	—	—	#	top=0, post=0
i=0	(	—	#, (	top=1, post=0
1	a	a	#, (	top=1, post=1
2	*	a	#, (, *	top=2, post=1
3	b	ab	#, (, *	top=2, post=2
4	)	ab*	#,	top=0, post=3
5	+	ab*	#, +	top=1, post=3
6	c	ab*c	#, +	top=1, post=4

place the remaining operators i.e., + into the postfix:

∴ ab*c + → postfix expression.

② $(A-B)*(D/E)$
0 1 2 3 4 5 6 7 8 9

step	symbol	post fix (P)	Stacks	Variables
initial	—	—	#	top=0, post=0
i=0	(	—	#, (	top=1, post=0
1	A	A	#, (	top=1, post=1
2	-	A	#, (, -	top=2, post=1
3	B	AB	#, (, -	top=2, post=2
4	)	AB-	#	top=0, post=3
5	*	AB-	#, *	top=1, post=3
6	(	AB-	#, *, (	top=2, post=3
7	D	AB-D	#, *, (	top=2, post=4
8	/	AB-D	#, *, (, /	top=3, post=4
9	E	AB-DE	#, *, (, /	top=3, post=5
10	)	AB-DE/	#, *	top=1, post=6

∴ postfix expression = AB−DE/*

precedence of Arithmetic operator:

^ → p = 4

%, /, * → p = 3

+, − → p = 2

(,) → p = 1

→ p = 0

problem on postfix expression

1) 12, 7, 3, −, /, 2, 1, 5, +, *, +

⇒ 12, (7−3), /, 2, 1, 5, +, *, +

= 12/(7−3), 2, 1, 5, +, *, +

= 12/(7−3), 2, (1+5), *, +

= 12/(7−3), (2*(1+5)), +

= (12/(7−3) + (2*(1+5)))

= (12/4) + (2*6)

= 3 + 12

= 15

convert the infix expression into postfix.

(i) a+b * c/d

= a + bc* /d

= a + bc*d/

= abc*d/+

(ii) (A−B) * (D/E)

= (AB−) * (DE/)

= AB−DE/*

(iii) (A+B^D)/(E−F)+G

= (A+BD^)/(E−F)+G

= (A+BD^)/(EF−)+G

= (ABD^+)/(EF−)+G

~~= (ABD^+)/EF−~~

= (ABD^+EF−/)+G

= ABD^+EF−/G+

(iv) $A*(B+D)/E-F*(G+H/K)$

$A*(BD+)/E-F*(G+HK/)$

$(ABD+*)/E-F*(GHK/+)$

$(ABD+*E/)-F*(GHK/+)$

$(ABD+*E/)-(FGHK/+*)$

$(ABD+*E/FGHK/+*-)$

II Evaluating of postfix expressions

Algorithm

Step 1: Scan the given expressions from left to right.

Step 2

(i) If symbol is an operand then push its value onto the stack.

(ii) If the symbol is an operator, then pop out two values from the stack and assign them respectively to operand 1 and operand 2. Then perform the required operator i.e.,

res = operand 1 * operand 2

push the result back to the stack.

Step 3: Repeat step 2 until all the symbols are covered.

Step 4: Pop out the result and print.

Example:- Evaluate the following expression

(i) $78+65+*$

Step	Symbol	operand 1	operand 2	Stack
i=0	7	—	—	7
1	8	—	—	7, 8
2	+	7	8	15
3	6	—	—	15, 6
4	5	—	—	15, 6, 5
5	+	6	5	15, 11
6	*	15	11	165

Result = 165

(2) 5, 6, 2, +, *, 12, 4, /, −

Step	Symbol	operand 1	operand 2	Stack
i=0	5	—	—	5
1	6	—	—	5, 6
2	2	—	—	5, 6, 2
3	+	6	2	5, 8
4	*	5	8	40
5	12	—	—	40, 12
6	4	—	—	40, 12, 4
7	/	12	4	40, 3
8	−	40	3	37

Result = 37