DATA STRUCTURES

A data structure is a specialized format for organizing, processing, retrieving and storing data.

A data structure may be selected or designed to store data for the purpose of working on it with various algorithms. Each data structure contains information about the data values, relationships between the data and functions that can be applied to the data.

DATA TYPES/DATA STRUCTURES can be classified into 2 types
Primitive data structures
Non-primitive data structures

**Primitive data structures**

Data structures that can be manipulated directly by machine instructions are termed as primitive data structures.
Ex: int, float char, long int etc.,

```
int a=10,b=20,c;
c=a+b; // no CTE
st p={“aa”,”11”,11}, q={“bb”,”22”,22), r;
r = p + q; //generates CTE;
```

**Non-primitive data structures**

Data structures that cannot be manipulated directly by machine instructions are termed as non-primitive data structures.

NP DS are created or constructed using primitive DS.
Ex: arrays, structures, stack, queues, linked list, tree, graphs, file etc.,

NP DS is categorized into 2 types
1. Linear DS Ex: arrays, stacks, queues, linked lists
2. Non-linear DS Ex: Trees, graphs, file, etc.,

**Linear DS**

Linear DS is one where its values or elements are stored in a sequential or linear order. Elements can be accessed sequentially one after the other in a linear order.

**Non-linear DS**

Non-linear Ds is one where elements are not stored in a sequential or linear order.

Logical adjacency between the elements is not maintained and elements cannot be accessed sequentially.
In NL DS, a data item could be attached to several other data items.

**Array - Traversing**

Accessing each item in the array is called traversing the array. Each element is accessed in linear order either from left to right or from **right to left**.
Ex:

```
int  a=10;  int *p=&a;  printf("%x",p);
p=p-1;  printf("%x",p);

int q[ ]={1,2,3};
int *r = q;   r = r+ 2;  printf("%d",*r);

for( ; q<=r; r--)
  printf("%d",*r);
```

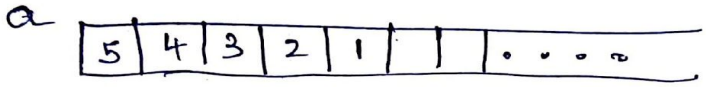
**Inserting an item into an array**

Given an array 'a' consisting of n elements, it may be required to insert an item at the specified position, say *pos*.

Value of pos cannot be: pos > n OR pos <= 0
Because user considers the element in 0th position as the 1st element
pos value for n elements will be in the range pos >=1 or pos <=n

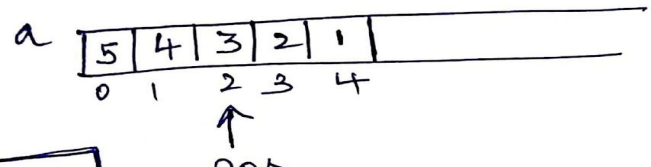
int a[100], n;

1)



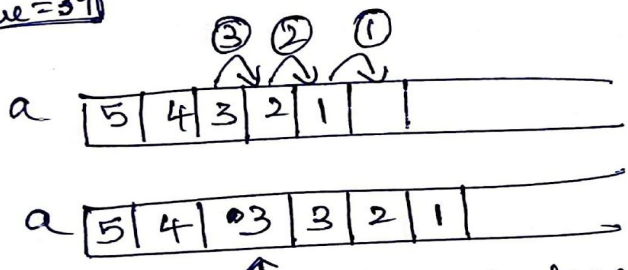

n = 5

2)

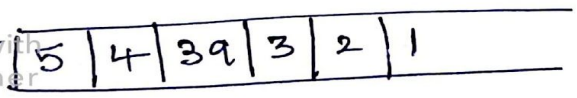



POS = 3

new value = 39

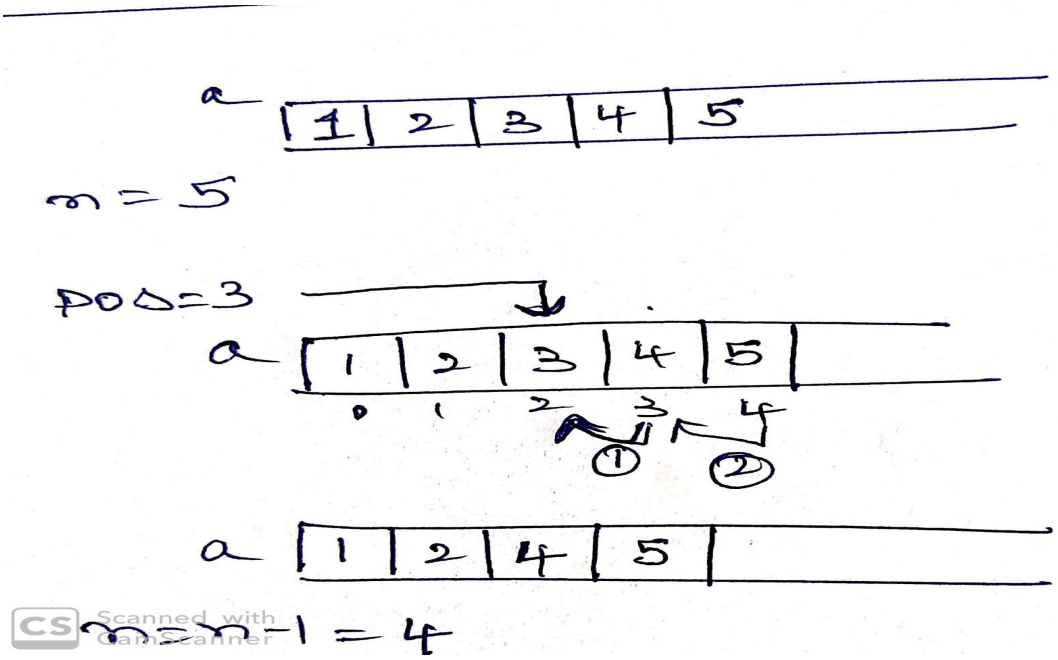



n = n+1 = 6 ← new value can be copied here



```
printf("enter the value pos");
scanf("%d",&pos);

if (pos <= 0 || pos > n)
{ printf("Invalid position\n"); return; }

printf("enter the value ele");
scanf("%d",&ele);
for(i=n-1;i>=(pos-1);i--)
   p[i+1]=p[i];
p[pos-1]= ele;
n++;
```

**Deleting an item from an array**






**Sorting an array - Bubble sort, quick sort**
**Searching an array for a key element - linear search, binary search**

**LAB PROGRAM 1**

**Design, Develop and Implement a menu driven Program in C for the following array**
**operations.**

**a. Creating an array of N Integer Elements**
**b. Display of array Elements with Suitable Headings**
**c. Inserting an Element (ELEM) at a given valid Position (POS)**
**d. Deleting an Element at a given valid Position (POS)**
**e. Exit.**

```
#include<stdio.h>
void accept(int *,int *);
void display (int *,int );
void insert_at_pos(int *,int*);
void delete_at_pos(int *,int*);

int main()
{
   int a[20],n=0,ch;

   for(;;)  //infinite loop
   {
    printf("1. enter the value for array\n");
    printf("2. display\n");
    printf("3. insert at pos\n");
    printf("4. delete at pos\n");
    printf("5. exit \n");
    printf("enter the choice =");
    scanf("%d",&ch);
    switch(ch)
    {
         case 1 :
                accept (a,&n);
                break;

        case 2 :
                display (a,n);
                break;
        case 3 :
```

```
                insert_at_pos(a,&n);
                break;
        case 4 :
                delete_at_pos(a,&n);
                break;
        case 5 :    return (0);
    }

        }// end of for loop
}

   void accept(int *p,int *n)
   {
    int i;

    /*
```

AFter execution begins, n will be 0, and it forces to accept value for n (which is in main scope), as execution continues, we will insert n values to array and to the task of inserting one or more values followed by deleting the values from array.

Before deleting all the values from the array if user makes a choice of 1

**Then the array still has some values left in it, in this scenario we will not accept any values from the user.**

Consider, we delete all the values from the array using delete_at_pos function call.

Then array will be totally empty or n will be equal to zero,

```
     */
    if (*n==0)  {
     printf("enter the value of n");
     scanf("%d",n);  }
    else
    {  printf("N value is not zero"); return;}

     for(i=0;i<*n;i++)
      scanf("%d", (p+i) );  // &p[i]   == p+i
    }


void display (int *p,int n)
{
```

```
        int i;

        if (n==0) { printf("Array is empty\n"); return;}
    printf(“Contents of array are\n”);
    for(i=0;i<n;i++)
      printf("%d ",p[i]);
}
```

```
void insert_at_pos(int *p,int *n)
{
    int pos,ele,i;
    printf("enter the value pos");
    scanf("%d",&pos);

    if (pos <= 0 || pos > (*n))
    { printf("Invalid position\n"); return; }

    printf("enter the value ele");
    scanf("%d",&ele);
    for(i=(*n)-1;i>=(pos-1);i--)
      p[i+1]=p[i];
    p[pos-1]= ele;
    (*n)++;
}

void delete_at_pos(int *p,int *n)
{
    int pos,i;
    printf("enter the value pos");
    scanf("%d",&pos);

    if (pos <= 0 || pos > (*n))
    { printf("Invalid position\n"); return; }

    for(i=pos-1;i<(*n);i++)
        p[i]=p[i+1];
    (*n)--;
}
```

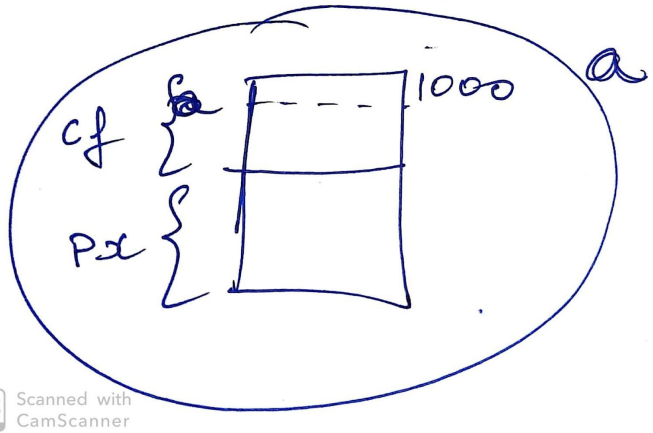
**Polynomials**

A polynomial is a mathematical expression consisting of a sum of terms, where each term is made up of a coefficient multiplied by a variable raised to a power.

Ex: x^4 + 10x^3 + 3x^2 + 1

To represent a polynomial in a program, there is no primitive data type and hence an non-primitive data type will be created using structure.

```
struct term
{
    int   cf;   // used to hold coefficient of a term
    int  px;  // used to hold power of x
};
typedef struct term tr;

tr   a;
```

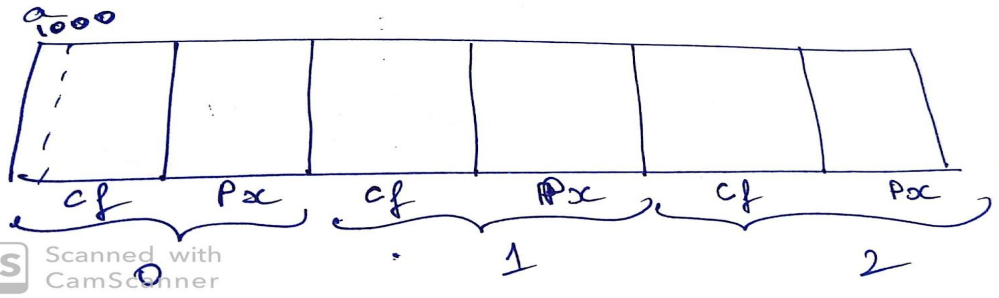





```
printf("%d", sizeof(a));
printf("%x\n", &a);
```

Since, a polynomial is made up of many terms, we need an array of tr;

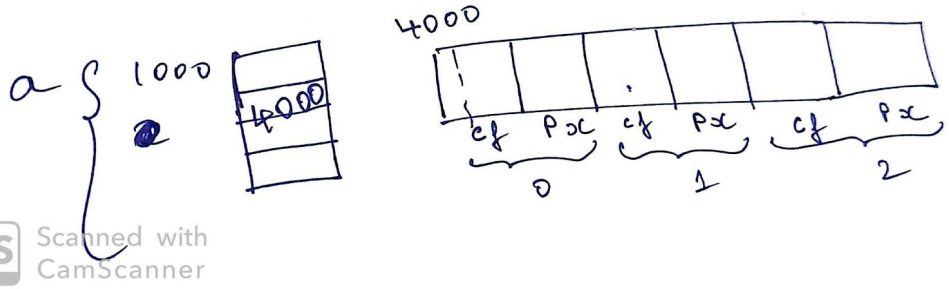
tr a[3]; **printf("%d",sizeof(a) );**






# OR

```
int  n;  tr *a;
scanf("%d",&n);  //n=3
a = (tr *) malloc(sizeof(tr)*n);
```






**printf("%d",sizeof(a));**

PGM TO ACCEPT A POLYNOMIAL AND TO DISPLAY.

```
struct term
{
    int   cf;   // used to hold coefficient of a term
    int  px;  // used to hold power of x
};
typedef struct term tr;

int main() {
     tr *a; int n,i;
     printf("Enter number of terms");
     scanf("%d", &n);
  a = (tr *) malloc(sizeof(tr)*n);
  for(i=0;i<n;i++)
       scanf("%d%d",   &(a[i].cf) , &(a[i].px) );

 for(i=0;i<n;i++)
        printf("%dx^%d+", a[i].cf, a[i].px);
 printf("\b");
}
```

PGM TO ADD TWO POLYNOMIALS

**********

Poly1 = 6x^6 + 7x^4 + 8x + 8  no of terms = 4

Poly2 = 8x^7 + 4x^6 + 7x^3 + 3x^2 + 4x  no of terms = 5

```
struct term
{
    int   cf;   // used to hold coefficient of a term
    int  px;  // used to hold power of x
};
typedef struct term tr;

void accept( int,tr *);
void display(int, tr *);
int main() {
        int n, n1;
```

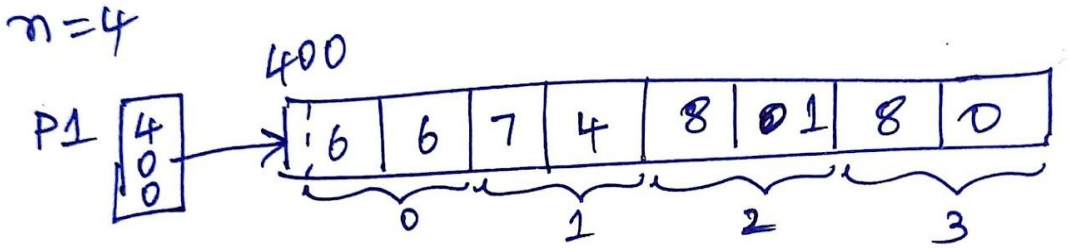
```
        tr  *p1, *p2, *p3;
       scanf("%d",&n);
       p1 = (tr *) malloc(sizeof(tr)*n);

      scanf("%d",&n1);
      p2 = (tr *) malloc(sizeof(tr)*n1);

      accept(n,p1);      accept(n1,p2);

  }
  void display(int n, tr * z)
 {
      int i;
      for(i=0;i<n;i++)
          printf("%dx^%d+", z[i].cf, z[i].px );
      printf("\b");
}
 void  accept(int n, tr * z)
 {
      int i;
      for(i=0;i<n;i++)
          scanf("%d%d", &(z[i].cf), &(z[i].px) );
}
```

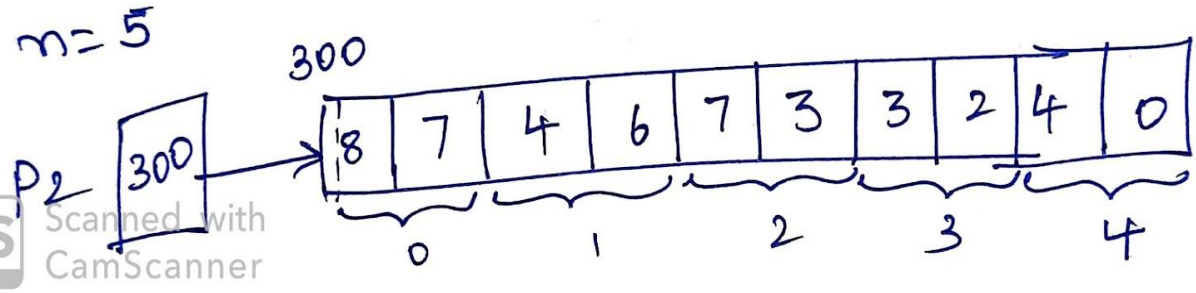
PGM TO ADD TWO POLYNOMIALS

**********

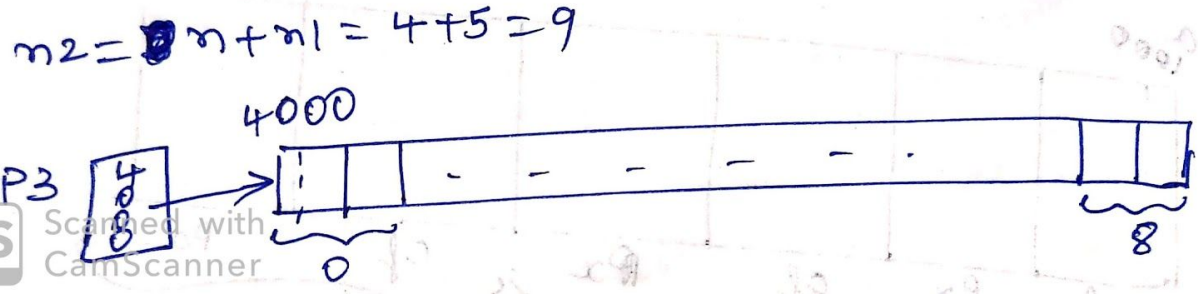
Poly1 = 6x^6 + 7x^4 + 8x + 8  no of terms = 4

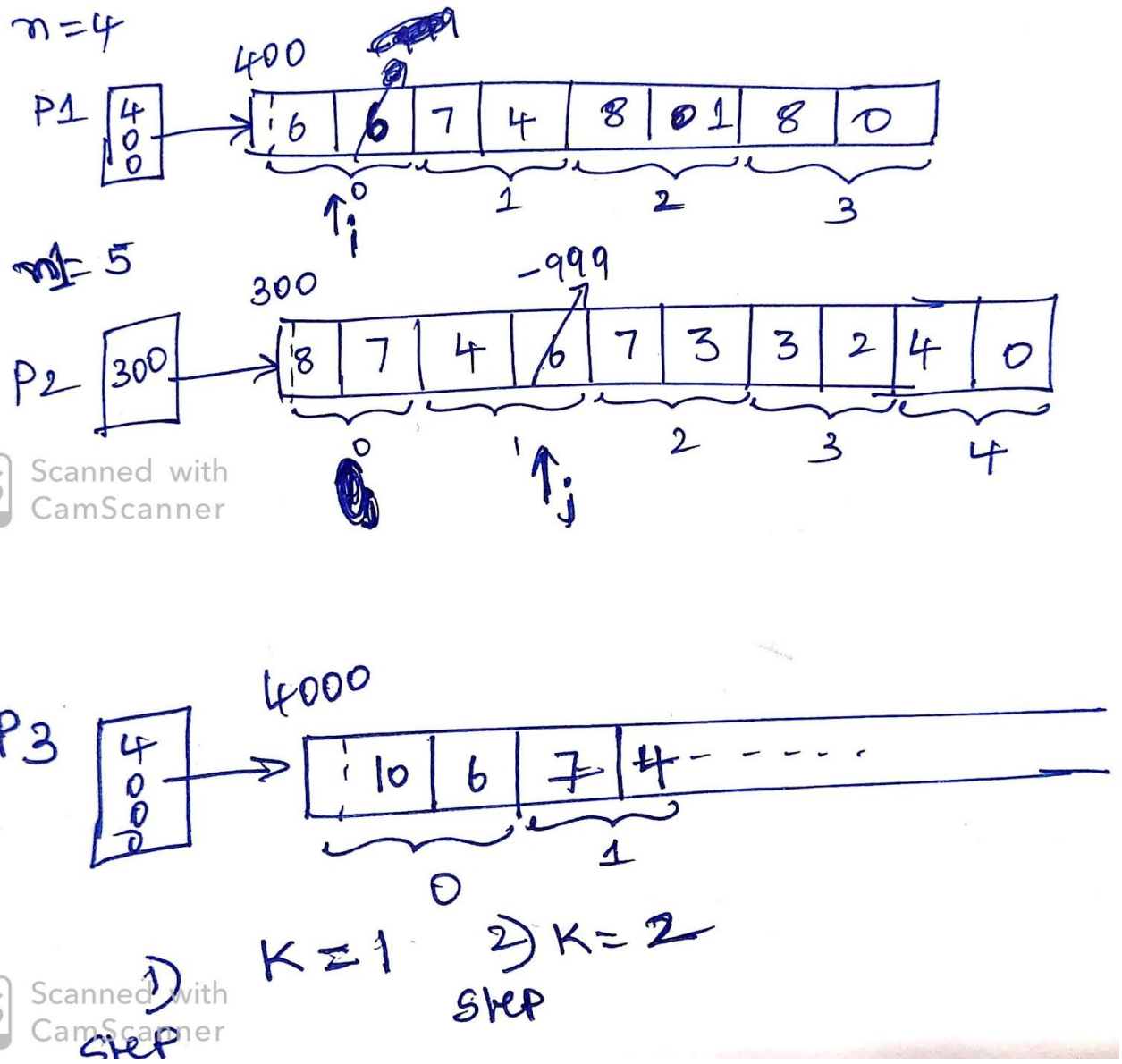
Poly2 = 8x^7 + 4x^6 + 7x^3 + 3x^2 + 4x  no of terms = 5








```
# include <stdio.h>
# include <stdlib.h>
struct term
{
    int   cf;   // used to hold coefficient of a term
    int  px;  // used to hold power of x
};
typedef struct term tr;

void accept( int,tr *);
void display(int, tr *);
int add(int, tr *, int, tr *, tr *);
```

```
int main()
{
        int n, n1,n2;   tr z;
        tr  *p1=0, *p2=0, *p3=0;
       scanf("%d",&n);
       p1 = (tr *) malloc(sizeof(tr)*n);

      scanf("%d",&n1);
      p2 = (tr *) malloc(sizeof(tr)*n1);

      accept(n,p1);      accept(n1,p2);

      p3 = (tr *) malloc(sizeof(tr)*(n+n1));
//assuming that each term in one polynomial will not match any of the term
//in other polynomials.  Later if less no of locations is used up, realloc
//will be used to reduce the memory.

      n2=0;
// right now no terms are there in resultant polynomial
      n2 = add(n,p1,n1,p2,p3);
      p3 = (tr *) realloc(p3,sizeof(tr)*n2);
      printf("Resultant polynomial is\n");
      display(n2,p3);
     free(p1);  free(p2);  free(p3);
  }
```

```
 int add(int n, tr *p1, int n1, tr *p2, tr *p3)
 {
     int i,j,sum=0,k=0;
//k used to index

     for(i=0;i<n;i++)
     {
         for(j=0;j<n1;j++)
          if (p2[j].px!=-999 && p1[i].px == p2[j].px)
            break;

         if (j<n1)
         {
           sum = p1[i].cf + p2[j].cf;
           p3[k].cf = sum;  p3[k].px = p1[i].px;
           p2[j].px = -999;
         }
         else
         {
           p3[k].cf = p1[i].cf;  p3[k].px = p1[i].px;
         }
        //  p1[i].px = -999;
          k++;
      }
//adding remaining terms in p2 to p3
        for(i=0;i<n1;i++)
         if (p2[i].px != -999)
           p3[k++] = p2[i];

        return k;
 }
```

```
 void display(int n, tr * z)
 {
     int i;
     for(i=0;i<n;i++)
         printf("%dx^%d+", z[i].cf, z[i].px );
     printf("\b");
}
 void  accept(int n, tr * z)
 {
     int i;
     for(i=0;i<n;i++)
         scanf("%d%d", &(z[i].cf), &(z[i].px) );
}
```

Input

Poly1 = 6x^9 + 7x^8 + 8x^7 + 8x^6 no of terms = 4

Poly2 = 8x^5 + 4x^4 + 7x^3 + 3x^2 + 4x^1 no of terms = 5

SPARSE MATRIX

A matrix has m rows and n columns

A matrix which contains more zeros compared to non-zero values.

**0 5 0 4**
**0 0 0 0**
**1 0 0 2**
**0 9 0 0**

AIM: To represent sparse matrix in an optimal way to minimize wastage of memory.
To use the same representation for matrix operation such as transpose, addition, multiplication etc.,

More space is wasted if the sparse matrix is stored in a 2-d array.
To minimize the wastage of space, a triplet structure will be used
<row, col, value>

```
# define MAX 100
struct term {
        int row, col, value;
};
typedef struct term T;
```

T a[MAX];  printf("%d", sizeof(a));

**a[0] : holds row size, col size and number of non-zero value entries.**

a[0].row = 4  a[0].col = 4  a[0].value = 5 (for the above matrix)

Position 1 through 5(inclusive) stores the triplets representing non-zero entries.

| i | a[i].row | a[i].col | a[i].value |
|---|---|---|---|
| **0** | **4** | **4** | **5** |
| 1 | 0 | 1 | 5 |
| 2 | 0 | 3 | 4 |
| 3 | 2 | 0 | 1 |
| 4 | 2 | 3 | 2 |
| 5 | 3 | 1 | 9 |

**row major order is maintained in triplet entries**

NOTE: row major order helps to build 2d-matrix from these triplets, if necessary.

ALGORITHM to perform transpose on triplets
  for each row i
        take element<i,j,value> and store it
        As elements <j,i,value> of the transpose

**[1, 0, 5]**
**[3, 0, 4]**
**[0, 2, 1]**
**[3, 2, 2]**
**[1, 3, 9]**

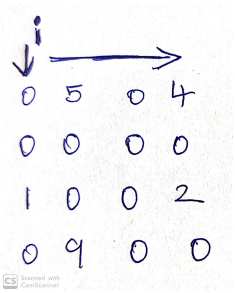
Row major order is lost if the algorithm is applied

ALGORITHM  (modified)
   for all elements in column j
            place elements <i, j, value> in
            element <j, i, value>

i.e  find all elements from column 0 and store it first
     find all elements from column 1 and store next ….

| | | |
|---|---|---|
| **[4,4,5]** | | **[4,4,5]** |
| **[0,1,5]** | | **[0,2,1]** |
| **[0,3,4]** | **⇒** | **[1,0,5]** |
| **[2,0,1]** | | **[1,3,9]** |
| **[2,3,2]** | | **[3,0,4]** |
| **[3,1,9]** | | **[3,2,2]** |

```
void transpose(T *a ,  T  b[ ])
{
// b holds transpose of a
int  n,i,j,currentb;

n = a[0].value; // total number of elements
b[0] = a[0];

if ( n>0 )
{
   currentb=1;

   for(i=0; i<a[0].col;  i++)
     // transpose by columns in a
```




```
     for(j=1; j<=n;  j++)

        // find elements from the current column
        if (  a[j].col  == i)
        {
         // elements are in current column, copy it to b
          b[currentb].row = a[j].col;
          b[currentb].col = a[j].row;
          b[currentb].value = value;
          currentb++;
         }
    }
}
```

```
# include <stdio.h>
struct sparse
{
  int row, col, value;
};
typedef struct sparse T;

void transpose(T [ ], T[ ]);

int main() {
 T a[10];
  T b[10]; int i;
  scanf(“%d%d%d”, &a[0].row, &a[0].column, &a[0].value);
  for(i=1; i<=a[0].value;i++)
     scanf(“%d%d%d”, &(a[i].row), &(a[i].column), &a[i].value);


  transpose(a, b);

  printf("Transposed sparse matrix\n");
  for(i=0;i<=b[0].value;i++)
         printf("%d %d %d\n",b[i].row,b[i].col,b[i].value);
}
```

**String**

String is a collection of characters, which ends with a null value.

char p[ ] = {“abc” };

| ‘a’ | ‘b’ | ‘c’ | ‘\0’ |
|---|---|---|---|

**Operations on String**

**size_t strlen(const char *s);**

char p[ ]={“abc”};

strlen(p+1) - prints 3 excluding null character

size_t is a synonym for unsigned integer

**char a[]={“abc”}; // a is constant pointer**

**const char *s=a ; // s is a pointer to constant**

**s++; //no CTE**

**(*s)++; // CTE**

**a++; // CTE**

**(*a)++ // no CTE**

**char * strcpy( char *dest, const char *src);**

```
char a[ ] = {“abc”}, b[5], c[={“abc”};
if ( strcmp( c, strcpy(b,a)) == 0)
    printf(“ equal”);
 else
       printf(“ ! equal”);
 printf(“%s”, b);
```

**char * strncpy (char *dest, const char *src, int n);**

n chars from src string will be copied to dest; and returns starting address of dest.

```
char a[] = {“rnsit engineering college”}, b[30];
 strncpy(b,(a+6),18);
```

**char * strtok( char *s, char *delimiters);**

Used to tokenize the string.

The strtok( ) function breaks a string into a sequence of zero or more nonempty tokens. On the first call to strtok( ) the string to be parsed should be specified in str. In each subsequent call that should parse the same string, str must be NULL.

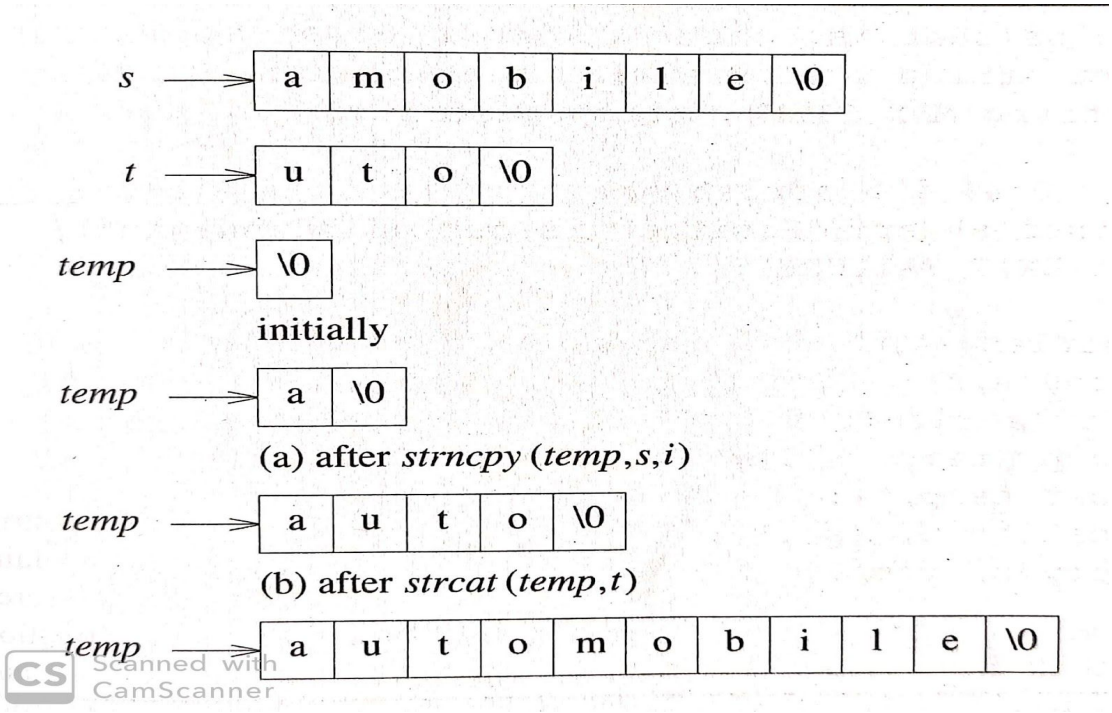
```
# include <string.h>
# include <stdio.h>
int main()
{
   char a[ ]={"https://www.google.com"};
   printf("%s\n",strtok(a,":"));
  printf("%s\n",a);      // "//www.google.com"
  printf("%s\n",strtok(NULL,"."));
  printf("%s\n",strtok(NULL,"."));
  printf("%s\n",strtok(NULL,"."));
}
```

| Function | Description |
|---|---|
| *char *strcat(char *dest, char *src)* | concatenate *dest* and *src* strings; return result in *dest* |
| *char *strncat(char *dest, char *src, int n)* | concatenate *dest* and *n* characters from *src*; return result in *dest* |
| *char *strcmp(char *str1, char *str2)* | compare two strings; return < 0 if *str1* < *str2*; 0 if *str1* = *str2*; > 0 if *str1* > *str2* |
| *char *strncmp(char *str1, char *str2, int n)* | compare first *n* characters return < 0 if *str1* < *str2*; 0 if *str1* = *str2*; > 1 if *str1* > *str2* |
| *char *strcpy(char *dest, char *src)* | copy *src* into *dest*; return *dest* |
| *char *strncpy(char *dest, char *src, int n)* | copy *n* characters from *src* string into *dest*; return *dest*; |
| *size_t strlen(char *s)* | return the length of a *s* |
| *char *strchr(char *s, int c)* | return pointer to the first occurrence of *c* in *s*; return *NULL* if not present |
| *char *strrchr(char *s, int c)* | return pointer to last occurrence of *c* in *s*; return *NULL* if not present |
| *char *strtok(char *s, char *delimiters)* | return a token from *s*; token is surrounded by *delimiters* |
| *char *strstr(char *s, char *pat)* | return pointer to start of *pat* in *s* |
| *size_t strspn(char *s, char *spanset)* | scan *s* for characters in *spanset*; return length of span |
| *size_t strcspn(char *s, char *spanset)* | scan *s* for characters not in *spanset*; return length of span |
| *char *strpbrk(char *s, char *spanset)* | scan *s* for characters in *spanset*; return pointer to first occurrence of a character from *spanset* |



**Figure 2.8:** C string functions

## String insertion



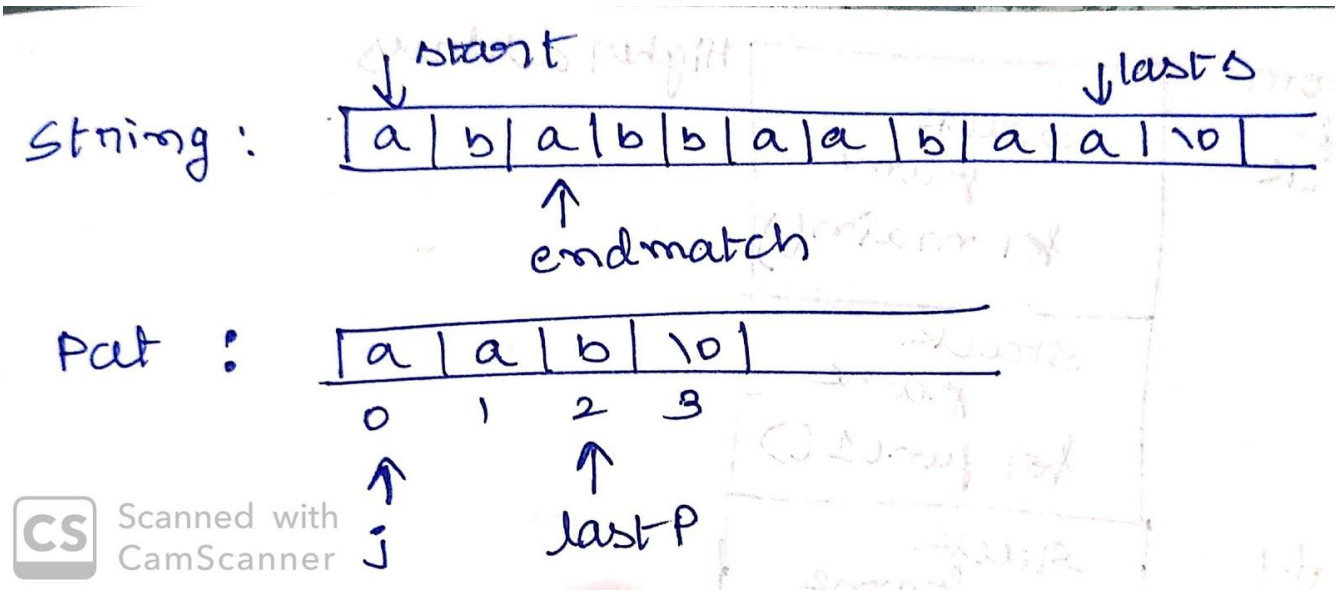


```
char s[30] = { "amobile"};
char t[20] = {"uto"};

void string_ins(char *s, char *t, int i)
{
  char temp[30];
  if (  i<0  ||  i>strlen(s) )
  {   printf("POS out of bounds\n");  return; }

 if ( !strlen(s) )  // if string s is empty
        strcpy(s,t);
 else
         if ( strlen(t) )
         {
           strncpy(temp,s,i);
           strcat(temp,t);
           strcat(temp, (s+i) );
           strcpy(s, temp);
         }
}
```

**PATTERN MATCHING**

String: ababbaabaa Index: start=0 lasts=9

pat: aab Index: j=0 lastp=2






**Algorithm**

1. Compare string[endmatch] & pat[lastp]
2. If they match, function uses i and j to move through the two strings until a mismatch occurs or until all of the characters in pat has been matched.
3. Variable start is used to reset i if a mismatch occurs.

```
int   nfind(char * string, char *pat) {
  int i, j, start=0;
  int lasts = strlen(string)-1;  lastp = strlen(pat)-1;
/*
-1 for strlen because:  strlen counts the character in the given string.
                        Counting starts from 1 and not from 0.
                        Counting stops when strlen encounters '\0' value.
                        Hence -1.
*/
  int endmatch = lastp;
 j=0;
for(i=0;  endmatch <= lasts;  endmatch++, start++ )
 {
            if (string[endmatch]  == pat[lastp] )
                 for(j=0, i=start;  j<lastp && string[i] == pat[j]; i++, j++);
  if ( j == lastp)
    return start;
 }
   return -1;
}
```

**First Pattern Matching Algorithm**

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

$W_k$ = Substring of T, whose length is equal to P
T = Main string whose length is greater than P
P = Substring to be searched in T

$W_k$ = SUBSTRING(T, K, LENGTH(P))
Let,
P = P[1] P[2] P[3] P[4]  and T = T[1] T[2] T[3].....T[20]
$W_1$ = T[1] T[2] T[3] T[4]
$W_2$ = T[2] T[3] T[4] T[5]
$W_3$ = T[3] T[4] T[5] T[6]
……
$W_{17}$ = T[17] T[18] T[19] T[20]
Maximum substring is MAX = 20-4+1 = 17

First, characters in P are compared one by one, with the substring, W1.
If P = $W_1$, P appears in T and INDEX(T, P) = 1

If P != $W_1$, P is compared with $W_2$, if P != $W_2$ procedure is continued until P is found in T or not found in T.

**ALGORITHM**

**********

P and T are strings with lengths R and S respectively
P and T are stored as arrays with one character per element.
Algorithm finds the INDEX of P in T

1. [initialize] Set K :=0 and MAX = S - R + 1.
2. Repeat Steps 3 to 5 while K <= MAX.
3. Repeat for L = 0 to R: [Tests each character of P.]
   If P[L] != T[K + L ], then Go to Step 5.
   [End of inner loop.]
4. [Success] Print "PATTERN FOUND" and Exit.
5. Set K := K + 1.
   [End of Step 2 outer loop]
6. [Failure] Print "PATTERN NOT FOUND".
7. Exit.

```
# include <stdio.h>
# include <string.h>
int main( ) {
   char T[40] = {"abcaabade"}, P[20] = {"ba"};
//W1=ab   w2=bc   w3=ca  w4=aa w5=ab w6=ba  w7=ad  w8=de
   int k=0, s = strlen(T), r = strlen(P), l;
   const int MAX = s-r+1;

   for(;k<=MAX;k++) {
          for(l=0;l<r;l++)
             if  (  P[l] != T[k+l] )
                break;
          if ( l==r)
          {
              printf("Substring found at %d \n", k);
              return 0;
          }
    }
    printf("Substring not found\n");     }
```

**LAB PROGRAM - 2**

**Design, Develop and Implement a Program in C for the following operations on Strings.**

**a. Read a main String (STR), a Pattern String (PAT) and a Replace String (REP)**

**b. Perform Pattern Matching Operation: Find and Replace all occurrences of PAT in**

**STR with REP if PAT exists in STR. Report suitable messages in case PAT does not exist in STR.**

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

int main()   {
  char T[40]={"bapqrababbzzba"},P[20]={"ba"},REP[20]={"BA"},FIN[50];

  //char T[40],P[20],REP[20],FIN[50];

  //printf("Enter Text, Pattern-to-find, Replacement-string\n");
  //gets(T);  gets(P);  gets(REP);
void replace(char *, char *, char *, char *);
  replace(T,   P,  REP,  FIN);

  printf("Output %s\n",FIN);
}

int slen(char *s)
{
  int len=0;

  for(;s[len] != '\0';len++);

  return len;
}
```

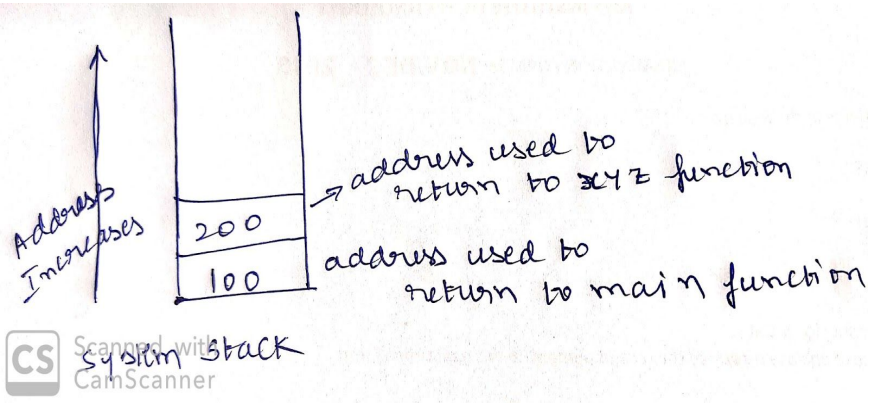
```
void replace(char *T, char *P, char *REP, char *FIN)
{
  int k=0,s = slen(T),r=slen(P),l,q=0,z;
  const int MAX = s-r+1;

 for(;k<=MAX;)
 {
  for(l=0;l<r;l++)
        if (P[l] != T[k+l])
          break;

  if (l==r)
  {
        //printf("Substring found at %d \n",k);
        for(z=0;z<strlen(REP);z++)
          FIN[q++] = REP[z];
        k = k + r;
  }
  else
   FIN[q++] = T[k++];
 }
 FIN[q]='\0';
}
```

**MODULE - 2**

Stack : Certain operations in computers require a restriction of inserting and deleting values from one end.

Ex: Resuming the execution of suspended function.

```
main( )              150: xyz( )          250: pqr( )
{                    {                    {
   xyz( );              pqr();               return;
100: stat1;          200: stat2;          }
}                    }
                     PC: 150              PC: 250
```






Stack is a list of elements, where-in elements are added and deleted from only one end termed TOS (Top Of Stack)

Stack operation is also termed as LIFO (Last In First Out order)

LIFO : Last Information that is added will be removed first.

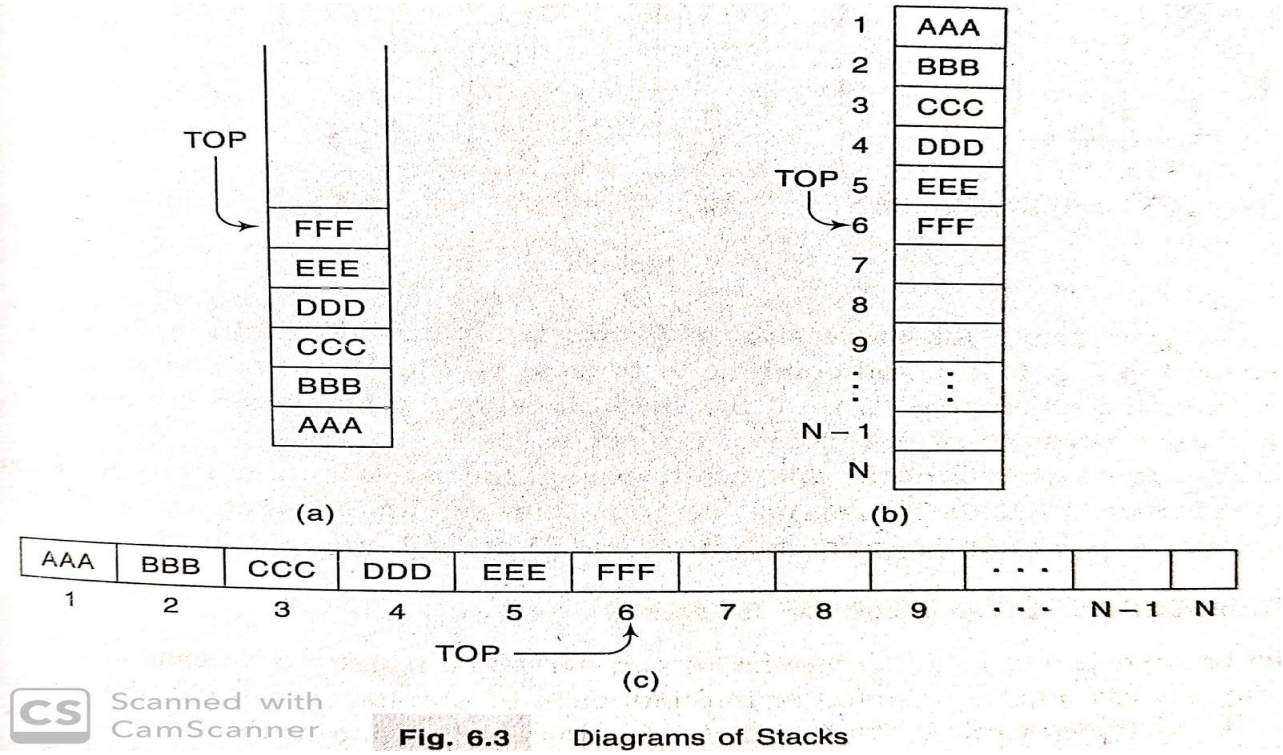





**Fig. 6.3** Diagrams of Stacks

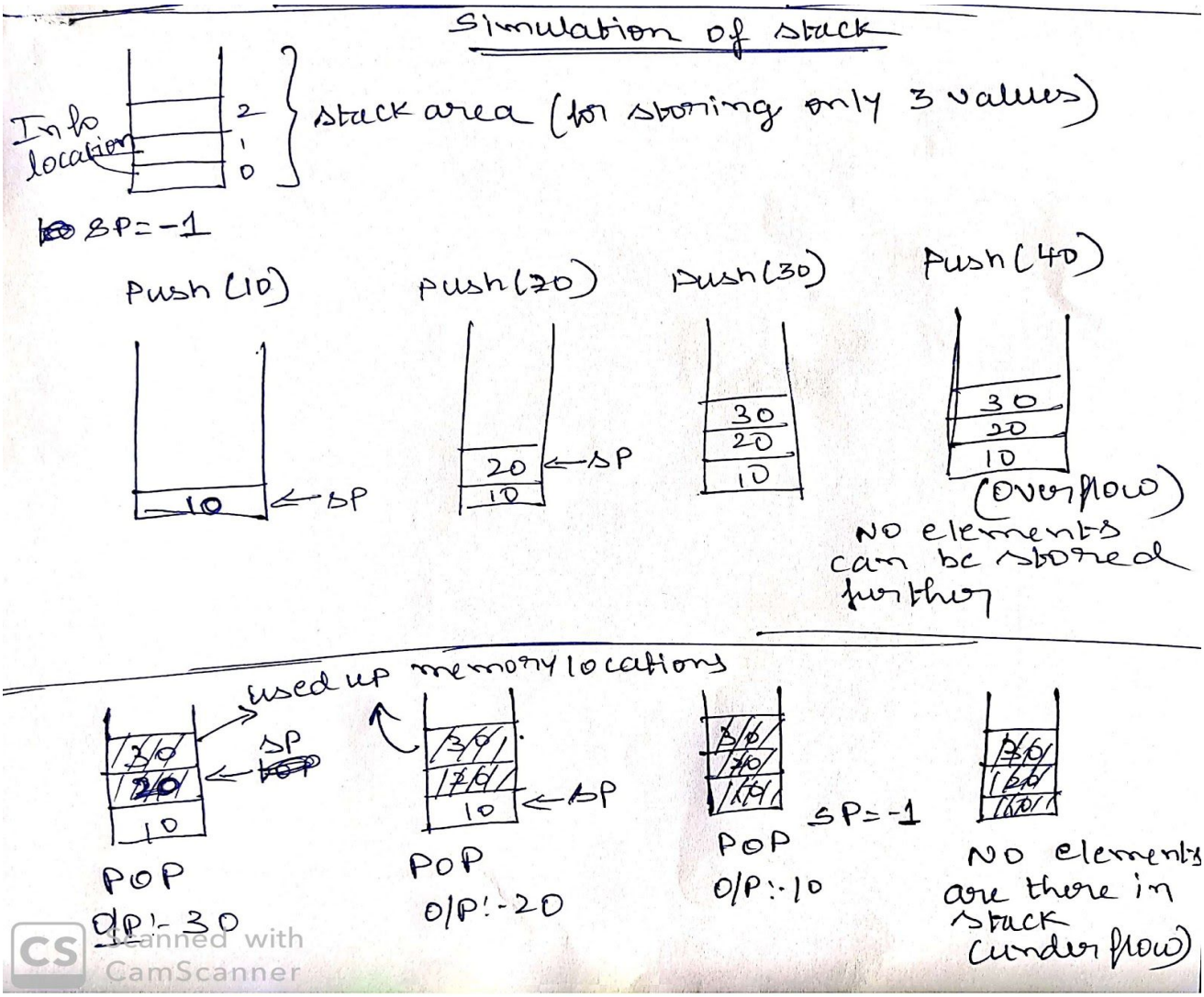
**Operations on stack**

PUSH : Term used to insert an element/information into stack.

POP: Term used to delete an element from stack.

An index is used in stack to point to the position, where PUSH & POP are performed termed as **stack pointer(sp).** Initially, when the stack is empty **sp** will not be pointing to the **stack area**.

One way of realizing stack in C is using array data structure and an integer index variable termed as sp.



**LAB PROGRAM 3**

**Design, Develop and Implement a menu driven Program in C for the following operations on STACK of Integers (Array Implementation of Stack with maximum size MAX)**

**a. Push an Element on to Stack**

**b. Pop an Element from Stack**

**c. Demonstrate how Stack can be used to check Palindrome**

**d. Demonstrate Overflow and Underflow situations on Stack**

**e. Display the status of Stack**

**f. Exit**

**MACRO -> REPLACEMENT -> BEFORE COMPILATION**

**FUNCTION -> CALL -> DURING EXECUTION**

**https://gcc.gnu.org/onlinedocs/cpp/Macros.html**

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAX_ELE 4

void push(int [],int *);
void pop(int [],int *);
void display(int [],int *);
void poly( );
```

```
int main()
{
   int ch,top=-1; // sp, TOS
   int s[MAX_ELE];
   for(;;)
   {
        printf("1:push\n2:pop\n3:display\n4:Palindrome\n5.Exit\n");
         printf("Enter choice: ");
         scanf("%d",&ch);
         switch(ch)
         {
                  case 1:  push(s,&top);  break;
                  case 2:  pop(s,&top);   break;
                  case 3:  display(s,&top); break;
                 case 4: palin(); break;
                  default: exit(0);
         }    }   }

void palin()
{
  char a[20];
  int start=0,top;
  printf("Enter any integer value\n");
  scanf("%s",a);
 //“1221”
  top = strlen(a)-1;

  for(;start<top; )
    if ( a[top--] != a[start++])
    {
        printf("Not a palindrome number\n");
        return;
    }
  printf("Palindrome number\n");
}
```

```
void display(int s[],int *top)
{
   int i;

   if( (*top) == -1)
         printf("stack empty\n");
   else
   {
         printf("stack elements are\n");
         printf("TOS is: ");
         for(i=(*top); i>=0; i--)
           printf("%d\n",s[i]);
   }
}
```

```
void push(int *s,int *top)
{
   int ele;
   if(  (*top)==MAX_ELE-1)
   {
     printf("stack overflow\n");
     return;
    }
    (*top)++;
    printf("enter the element\n");
    scanf("%d",&ele);
    s[*top]=ele;
}
void pop(int s[],int *top)
{
   if( (*top) == -1)
         printf("stack underflow\n");
   else
   {
         printf("element popped is\n");
         printf("%d\n",s[*top]);
         (*top)--;
   }
}
```

**STACKS USING DYNAMIC ARRAYS**

Shortcoming of the stack implementation using automatic array is the need to know at compile time, the size of the array.

This shortcoming can be resolved using a dynamically allocated array for the elements and then increasing the size of this array as needed.

Ex:

```
int n,*a;
scanf("%d",&n);
a = (int *) malloc(sizeof(int)*n);
```

N locations can be used to store information in stack.

Next if more values has to be stored, realloc can be used to increase the size of the stack.

Further, the size of the stack will be increased after changing the value of n.

```
n=n*2;  //doubling the size of the stack
a = (int *) realloc(a, sizeof(int)*n);
```

Stack underflow will be experienced provided 0 elements are stored in stack.
Stack overflow message will be printed, but elements can be inserted into the stack after increasing the size of the stack, using realloc.

```
int main( ) {
  int n,*a,top=-1;
 printf("Initial size of stack\n");
 scanf("%d",&n); // n=2;
a = (int *) malloc(sizeof(int)*n);
for(;;) {
        printf("DYNAMIC STACK\n");
        printf("1. Push\n2. Pop\n3. Display\n4. Exit\nEnter choice: ");
        scanf("%d",&ch);
        switch(ch) {
          case 1: push(a, &top, &n); break;
          case 2: pop(a, &top); break;
           case 3: display(a, top, n); break;
          default: return;
      }
}
} // end of main

void display(int *s, int top, int  n)
{
 int i;
  if ( top == -1)
    { printf("Stack underflow\n"); return; }
 for(i=top; i >=0; i--)
   printf("%d\n", i[s]);
}
```

```
void pop (int *s, int *top)
{
  if ( (*top) == -1)
  { printf(“Stack underflow\n”); return; }

  printf(“Popped element is %d”,s[(*top)]);
  (*top)--;
}
int * push(int *s, int *top, int *n)
{
    int ele;
    if(  (*top)==(*n)-1)
    {
       printf("stack overflow...stack size is being incremented\n");
       (*n) = (*n) * 2;
       s  = (int *) realloc(s, sizeof(int) * (*n));
     }
      (*top)++;
      printf("enter the element\n");
      scanf("%d",&ele);
      s[*top]=ele;
     return s;
}
```

**PROGRAM TO SIMULATE A STACK OF EMPLOYEE (EMP-ID, NAME, SALARY)**

**Applications of Stack**

*1. Conversion of expression*

Arithmetic, Relational and Logical expressions are represented in one format while coding and will be converted to another format by the compiler, which suits for execution.

Expression can be represented in the forms like

1. Infix expression
   Ex: a+b, a and b are termed as operands and + is operator.
2. Postfix expression Ex: ab+
3. Prefix expression Ex: +ab

In order to convert an expression from one form to another stack can be used.

*2. Evaluation of expression*

An arithmetic expression represented in the form of either postfix or prefix can be easily evaluated.

Stack can be used to evaluate the same.

*3. Recursion*

A function which calls itself is called a recursive function.

Stack data structure will be used to effectively evaluate a recursion call.

**Infix to Postfix**

In Expression a+b a,b are operands and + is operator.

Expressions can be classified as

1. Infix expression a+b operator is in between operands and can be parenthesized expression i.e (a+b) .
2. Postfix expression ab+ operator follows operands and **no parentheses** is used.
3. Prefix expression +ab operator precedes operands and no parentheses is used.

**PRECEDENCE & ASSOCIATIVITY OF OPERATORS**

Precedence hierarchy determines the order in which operators have to be evaluated.

Precedence hierarchy in C:

| Token | Precedence | Associativity |
|---|---|---|
| *, /, % | 13 | left-to-right |
| +,- | 12 | left-to-right |

a*b/c

| Token | Operator | Precedence[1] | Associativity |
|---|---|---|---|
| ()<br>[]<br>→ .<br>struct or union member | function call<br>array element | 17 | left-to-right |
| −− ++ | increment, decrement[2] | 16 | left-to-right |
| −− ++<br>!<br>~<br>−+<br>& *<br>sizeof | decrement, increment[3]<br>logical not<br>one's complement<br>unary minus or plus<br>address or indirection<br>size (in bytes) | 15 | right-to-left |
| (type) | type cast | 14 | right-to-left |
| * / % | multiplicative | 13 | left-to-right |
| + − | binary add or subtract | 12 | left-to-right |
| << >> | shift | 11 | left-to-right |
| > >=<br>< <= | relational | 10 | left-to-right |
| == != | equality | 9 | left-to-right |
| & | bitwise and | 8 | left-to-right |
| ^ | bitwise exclusive or | 7 | left-to-right |
| \| | bitwise or | 6 | left-to-right |
| && | logical and | 5 | left-to-right |
| \|\| | logical or | 4 | left-to-right |
| ?: | conditional | 3 | right-to-left |
| = += −= /= *= %=<br><<= >>= &= ^= \|= | assignment | 2 | right-to-left |
| , | comma | 1 | left-to-right |

1. The precedence column is taken from Harbison and Steele.
2. Postfix form
3. Prefix form



**Figure 3.12:** Precedence hierarchy for C

Ex: x = a / b - c + d * e - a * c  let a=4, b=c=2, d=e=3
= 4 / 2 - 2 + 3 * 3 - 4 * 2
= 2 - 2 + 9 - 8 = 1
Result is 1 because of Precedence Hierarchy of operators

**Associativity conveys, how operators have to be evaluated, if precedence level are same.**

/ and * Associativity is from left-to-right.
Ex: a/b*d  let a=4 b=3 d=2
= 4/3*2
= 2

**INFIX TO POSTFIX**
Infix expression is used by programmer

Computers use postfix expressions to execute an expression.

Infix may be parenthesised or non-parenthesised, but postfix expressions are always non-parenthesised.
Ex: (a+b) parenthesised valid infix expression
a+b non-parenthesised valid infix expression.

**Algorithm to convert infix to postfix**
Operator's with highest precedence must be converted to postfix form, followed by operator's with lowest precedence level.

If parenthesised expression (PE) is a part of expression, then PE must be first converted to postfix form and then the remaining part of the expression has to be considered.

Ex: a * b + d  T1 = a * b = a b *
T1 + d  T2 = T1 + d = T1 d +
T2
********
T1 d +
a b * d +

a*(b+d)  T1 = b + d = b d +
a *  T1  T2 = a * T1 = a T1 *
T2
**********
a T1 *
a b d + *

(a / (b - c + d) ) * (e - a) * c  $T_1$ = b - c = b c -
(a / $T_1$ + d ) *  (e - a) * c  $T_2$ = $T_1$ + d  = $T_1$ d +
a / $T_2$ * (e - a) * c  $T_3$ = a / $T_2$ = a $T_2$ /
$T_3$ * (e - a) * c  $T_4$ = e - a  = e a -
$T_3$ * $T_4$ * c  $T_5$ = $T_3$ * $T_4$ = $T_3$ $T_4$ *
$T_5$ * c
$T_5$ c *
**********
$T_5$ c *
$T_3$ $T_4$ * c *
$T_3$ e a - * c *
a $T_2$ / e a - * c *
a $T_1$ d+ / e a - * c *
a b c - d + / e a - * c *

a / b - c + d * e - a * c  T1 = a / b = a b /
T1 - c + d * e - a * c  T2 = d * e = d e *
T1 - c + T2 - a * c  T3 = a * c = a c *
T1 - c + T2 - T3  T4 = T1 - c = T1 c -
T4 + T2 - T3  T5 = T4 + T2 = T4 T2 +
T5 - T3
T5 T3 -
**********
T5  a  c * -  T3 replacement
T4  T2  + a c * -  T5 replacement
T1  c - T2 + a c * -  T4 replacement
a b / c - T2 + a c * -  T1 replacement
a b / c - d e * + a c * -  T2 replacement

( ( a + ( b - c ) * d ) ^ e + f )
( ( a + ( b - c ) * d ) ^ e + f )  T1 = b - c = b c -
( ( a + T1 * d ) ^ e + f )  T2 = T1 * d = T1 d *
( ( a + T2 ) ^ e + f )  T3 = a + T2 = a T2 +
( T3 ^ e + f )  T4 = T3 ^ e = T3 e ^
T4 + f
**********
T4 f +
T3 e ^ f +
a T2 + e ^ f +
a T1 d * + e ^ f +
a b c - d * + e ^ f +

PROGRAM TO CONVERT INFIX TO POSTFIX
Input: expression is entered in a string format
Output: postfix expression in string format

DATA STRUCTURE used to convert infix to postfix expression is STACK.
PRECEDENCE value & ASSOCIATIVITY are required to convert an expression from infix to postfix.

| SYMBOLS | STACK PRECEDENCE (function named F) | INPUT PRECEDENCE (function named G) | ASSOCIATIVITY |
|---|---|---|---|
| +, - | 2 | 1 | Left-to-right |
| *, / | 4 | 3 | -"- |
| $ or ^ | 5 | 6 | Right-to-Left |
| operands | 8 | 7 | Left-to-right |
| ( | 0 | 9 | -"- |
| ) | - | 0 | - |
| # | -1 | - | - |

**If operands or operators are having L-R associativity and if one of them is already in stack then it will have highest precedence.**

**If operands or operators are having R-L associativity and if one of them is already in stack then it will have lowest precedence.**

Stack precedence values, specify the precedence values for the symbols that are present in the stack. Stack has an initial value of '#', precedence value of which is -1.

Input precedence values, specify the precedence values for the symbols that are available in the string.

**[NOTE: in a single scan of the input infix expression, it will be converted as postfix expression by considering the above precedence table. Initial value of stack will be filled by '#']**

ALGORITHM
Step 1:
As long as precedence symbol on TOS is greater than the precedence on input symbol, pop an item from the stack and place it in a postfix array.

Step 2:
If the precedence symbol on TOS is not equal to the precedence of the current input
symbol, push the current symbol on to the stack. Otherwise, delete an item from the stack.

**LAB PROGRAM 4**
**Design, Develop and Implement a Program in C for converting an Infix Expression to PostfixExpression. Program should support for both parenthesized and free parenthesized expressions with the operators: +, -, *, /, % (Remainder), ^ (Power) and alphanumeric operands.**

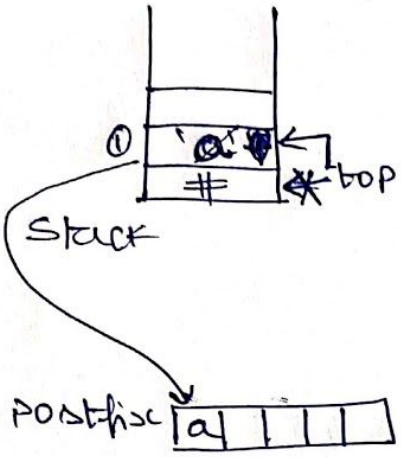
```
#include<stdio.h>
#include<string.h>
#define MAX_ELE 30

int f(char s)   {
  switch(s)   {
        case '+':
        case '-': return 2;
        case '*':
        case '/': return 4;
```

```
        case '$':
        case '^': return 5;
        case '(': return 0;
        case '#': return -1;
        default: return 8;
  }   }


int g(char s)  {
  switch(s)   {
        case '+':
        case '-': return 1;
        case '*':
        case '/': return 3;
        case '$':
        case '^': return 6;
        case '(': return 9;
        case ')': return 0;
        default: return 7;
  }  }
```

```
int main()  {
 char c, s[MAX_ELE]={'#'};
 char inf[MAX_ELE]={"a/b-(c+d)"};
 char pf[MAX_ELE];
  int top=0,i,j=0;  // top = 0  because '#' is already stored in stack

  printf("enter the infix expression\n");
  //scanf("%s",inf);
  for(i=0;i<strlen(inf);i++)   {
        c= inf[i];

//STEP 1
        while( f( s[top] ) > g(c) )
        {
          pf[j]=s[top];
          j++;
         top--;
        }
//STEP 2
        if( f( s[top] )  !=  g( inf[i] ) )
           s[++top]=inf[i];
        else
           top--;
  }
  for(;s[top]!='#';top--)
        pf[j++]=s[top];
  pf[j]='\0';
  printf("the postfix expression:%s\n",pf);
}
```

Infix

| a | + | d | # |
|---|---|---|---|
| 0 | 1 | 2 | 3 |




1) s = 'a'

while ( F('#') > G('a') )

-1 > 7 False

if ( F('#') != G('a') )

-1 != 7

Push ('a')

2) s = '+'

while ( F('a') > G('+') )

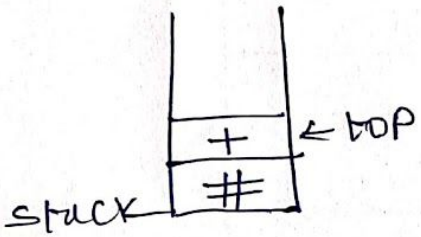
8 > 1

F('#') > G('+')

-1 > 1

if ( F('#') != G('+') )

Push ('+')




3) s = 'd'

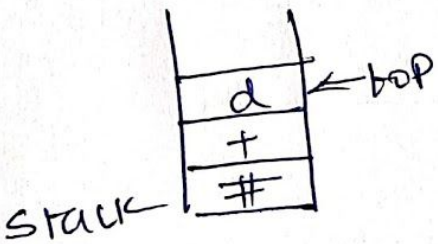
while ( F('+') > G('d') )

2 > 7

If ( F('+') != G('d') )

2 != 7

Push ('d')




LAST FOR LOOP

Postfix | a | d | + | 10 | ⟹ ad+

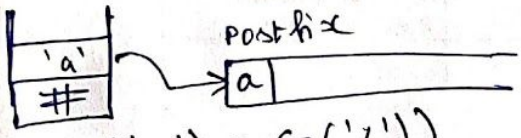
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| a | / | b | - | ( | c | + | d | ) | $ |

Infix

1) s = 'a'
won (F('#') > G('a'))
-1 > 7
if (-1 != 7)
Push 'a'

stack


2) s = '/'
won (F('a') > G('/'))
8 > 3

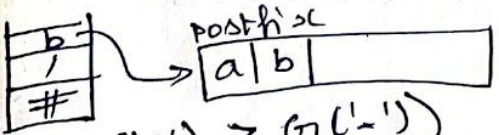



won (F('#') > G('/'))
-1 > 3
if (-1 != 3)
Push '/'

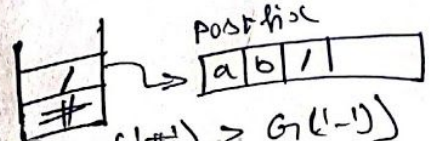
stack


3) s = 'b'
won (F('/') > G('b'))
4 > 7
if (4 != 7)
Push 'b'

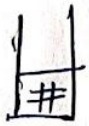

4) s = '-'
won (F('b') > G('-'))
8 > 1



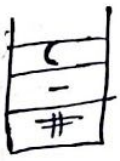

won (F('/') > G('-'))
4 > 1




won (F('#') > G('-'))
-1 > 1



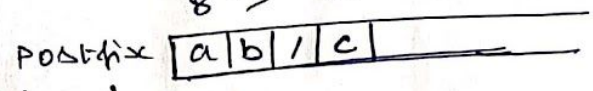

5) s = '('
won (F('-') > G('('))
2 > 9
if (2 != 9)
Push '('

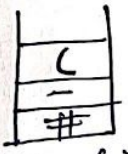



6) s = 'c'
won (F('(') > G('c'))
0 > 7
if (0 != 7)
Push 'c'


7) s = '+'
won (F('c') > G('+'))
8 > 1

postfix



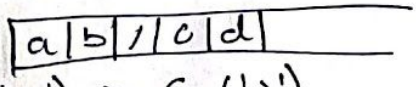




won (F('c') > G('+'))
8 > 9
if (8 != 9)
Push '+'

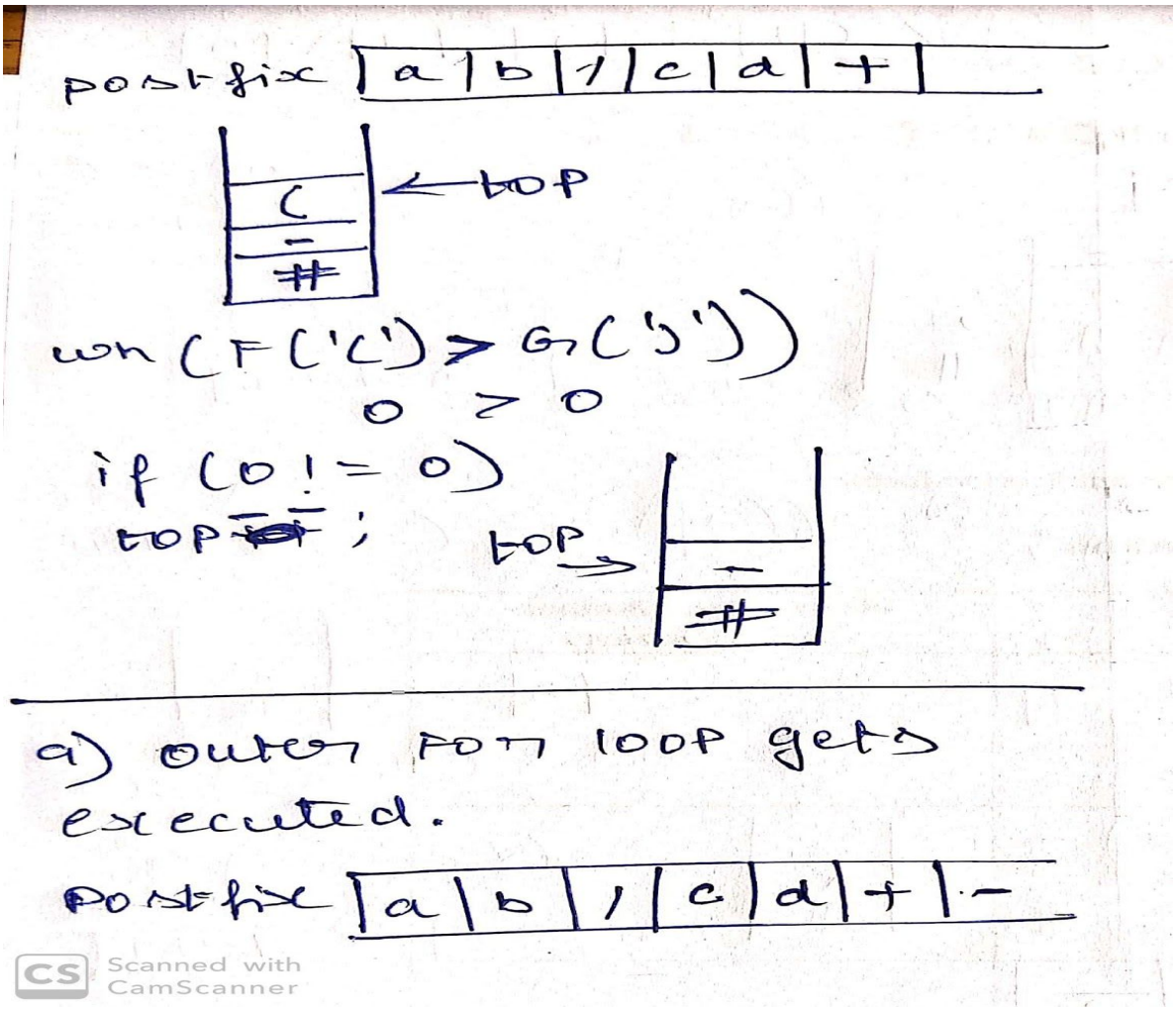

8) s = 'd'
won (F('+') > G('d'))
2 > 7
if (2 != 7)
Push 'd'


9) s = ')'
won (F('d') > G(')'))
8 > 0

Postfix




F('+') > G(')')
2 > 0




**POSTFIX EVALUATION**

System cannot execute expressions in infix form. Infix expression requires many traversals of R to L and L to R for execution and infix expression can contain parentheses as a part of it.

Postfix expressions will not have parentheses and can be executed in one scan.

System considers expressions in postfix form for evaluation.

**ALGORITHM**

Input: String value - Expression in postfix form

Operand length cannot be more than 1 digit.

Ex: "12+" (1 and 2 are operands, + is operator)

Output: double value - After evaluating expression.

1. Scan the symbol from left to right.
2. If the scanned symbol is an operand, push it on to stack.
3. If the scanned symbol is an operator, pop two elements from stack.
   First popped element is operand2 (RHS of operator).
   Second popped element is operand1(LHS of operator) .
4. Perform the indicated operation.
5. Push the result back to stack.
6. Repeat 1-5 until null value is encountered.

```
# include <stdio.h>
# include <string.h>  // for strlen function
# include <math.h>          // for pow function
# include <ctype.h>   // for isdigit function
//int isdigit(int c);
double compute(char symbol,double op1, double op2)
{
 switch(symbol)
 {
        case '+': return op1 + op2;
        case '-': return op1 - op2;
        case '*': return op1 * op2;
        case '/': return op1 / op2;
        case '$':
        case '^': return pow(op1,op2);
 }
}

int main()
{
 char postfix[20]={"56+437-*/"},symbol;
 double st[20],op1,op2;
 int top=-1,i;

 printf("Enter a valid postfix expression\n");
// scanf("%s",postfix);
```

```
 for(i=0;postfix[i] != '\0'; i++)
 {
// int isdigit(int c);
// return value is nonzero or zero
   if ( isdigit(postfix[i]) )
         st[++top] = postfix[i] - '0';
   else
   {
         op2 = st[top--];
         op1 = st[top--];

         st[++top] = compute(postfix[i],op1,op2);
   }
 }
 printf("Result is %lf\n",st[top--]);
}
```