DATA STRUCTURE :

Arrangement of data inside the memory

Prog : Add 2 elements using functions :-

```
# include <stdio.h>
 int sum (int a, int b)
{    int add = a+b;
     return add;
}

 int main ()
{    int a, b;
     scanf ("%d%d", &a, &b);
     int result = sum (a, b);
     printf ("The sum is: %d", result);
}
```

Program to add 2 arrays using functions :-

```
# include <stdio.h>
 int sum (int [ ], int [ ]);
// or  int sum (int *, int *);
```

note :- acc to compiler no diff b/w array & pointer.

- name of array holds base address of allocated memory

∴ a ⇒ 1000 and &a ⇒ 1000

passing array as a param is call by reference by default

arr[i] and *arr(0+i) same thing

```
void sum (int a[20], int b[20], int n)
{    int add[20];
```

```
    for (int i=0; i<n; i++)
        add[i] = a[i] + b[i];
    for (int i=0; i<n; i++)
        printf("%d ", add[i]);
}

int main()
{
    int a[20];
    int b[20];
    printf("Enter the size of the arrays: ");
    scanf("%d", &n);
    printf("Enter the elements of the first array:");
    for (int i=0; i<n; i++)
        scanf("%d", &a[i]);
    printf("Enter the elements of the sec array:");
    for (int i=0; i<n; i++)
        scanf("%d", &b[i]);
    sum(a, b);
}
```

For any question:
→ definition
→ memory representation
→ working code

POINTER

Its a variable which holds the address of another variable.

* → dereferencing operator
& → address of operator
→ → structure to pointer

Name of an array is a constant pointer
↳ it holds base address of the array

Program to read & print array elements with pointers :

```
# include <stdio.h>
int main()
{
    int n;
    int a[20];
    printf("enter number of elements : \n");
    scanf("%d", &n);
    for(int i = 0; i < n ; i++)
        scanf("%d", (a+i));
    for(int i = 0; i < n; i++)
        printf("%d", *(a+i));
}
```

Add all elements of an array elements :

```
# include <stdio.h>
int main()
{
    int n, sum = 0;
    int a[100];
```

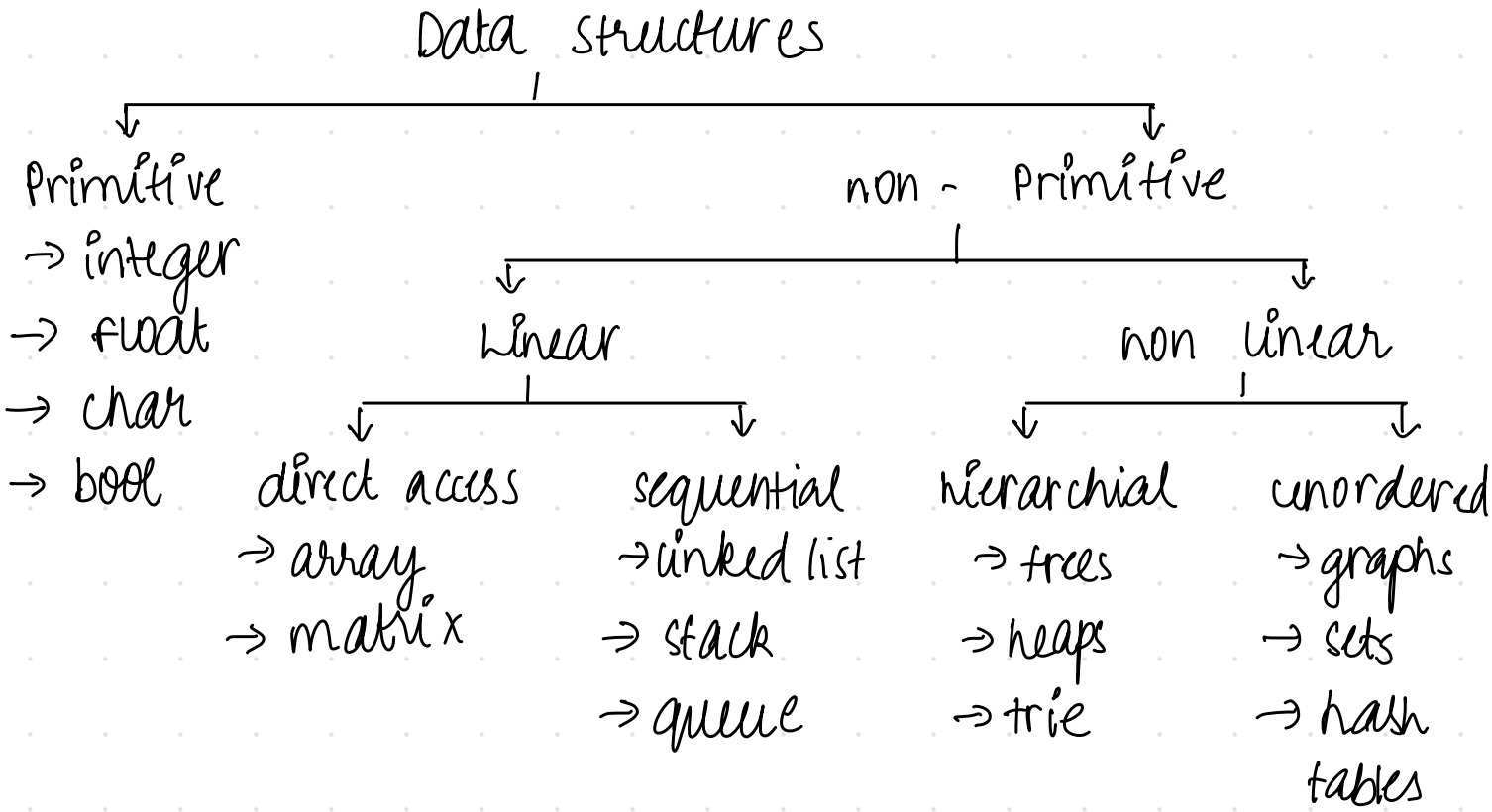
```
printf("enter no of elements:");
scanf("%d", &n);
for(int i=0; i<n; i++)
    scanf("%d", (a+i));
for(int i=0; i<n; i++)
    sum = sum + *(a+i);
printf("The sum is = %d", sum);
}
```

unit - 1

INTRODUCTION:

Data structures : arrangement of data in the memory

Classification of data structures :-



primitive data structures are defined within the language (compiler defined)

Linear data structure : follows sequential ordering / storing of data

Non linear : its hierarchial

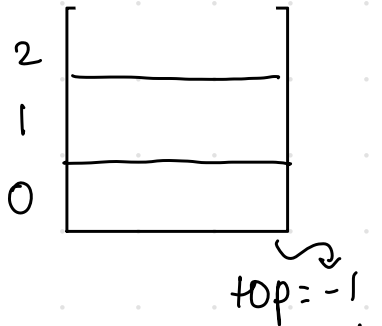
STACKS:

- It is a linear data structure
- LIFO data structure
- element inserted at last removed first

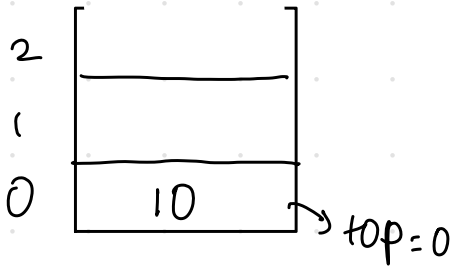
Representation :-

- It can be represented using arrays
- using structures
- using linked lists

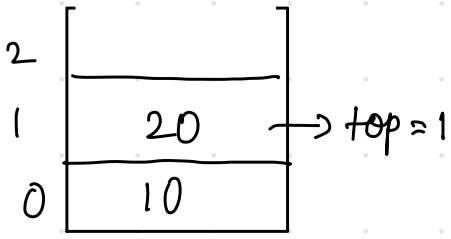
Concept of stack :



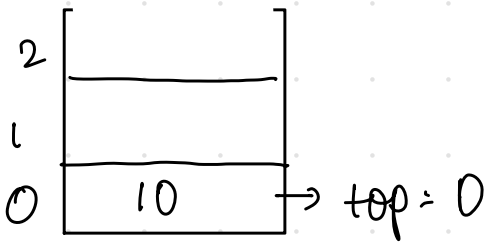
empty stack



push 1 element



push 2 elements



pop 1 element

conditions :

→ stack empty : underflow (x pop)

→ stack full : overflow (x push)

operations :

→ insertion of ele : push

→ deletion of ele : pop

top points to the recently inserted element

Push operation :

It's used to insert an element into stack

→ In push operation, stack overflow condition is to be checked

```
void push (int stack[], int *top, int ele)
{       if (*top == S_size -1)
        {       OVERFLOW
                return;
        }
        *top++;
        stak[*top] = ele;
}
```

Pop operation :

Its used to delete an element from the stack

→ stack underflow condition is to be checked

```
void pop (int stack[], int *top)
{       if (*top == -1)
        {   UNDERFLOW
            return;
        }
        printf("%d element is deleted", stack[top]);
        logical deletion
        *top--;
}
```

Display operation :

Its used to display elts of stack

→ stack empty condition to be checked

```
void display (int stack[], int top)
{
    if (top == -1)
    {
        EMPTY
        return;
    }
    for (int i = top; i >= 0; i--)
    {
        printf ("%d " , stack[i]);
    }
}
```

full implemented code →

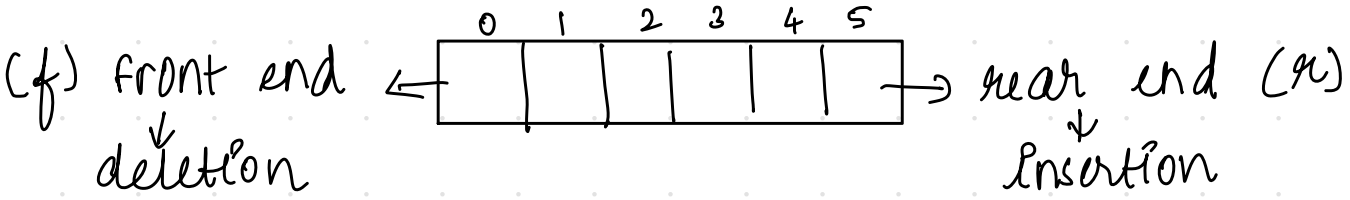
c Copy

```c
#include <stdio.h>
#define MAX 5  // maximum size of the stack

int stack[MAX];
int top = -1;

// Function prototypes
void push();
void pop();
void display();

int main() {
    int choice;

    while (1) {
        printf("\n--- STACK MENU ---\n");
        printf("1. Push\n");
        printf("2. Pop\n");
        printf("3. Display\n");
        printf("4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1: push(); break;
            case 2: pop(); break;
            case 3: display(); break;
            case 4: return 0;
            default: printf("Invalid choice! Please try again.\n");
        }
    }
}

// ============================
// PUSH OPERATION
// ============================
void push() {
    int item;
    if (top == MAX - 1) {
        printf("Stack Overflow! Cannot push element.\n");
    } else {
        printf("Enter the element to push: ");
        scanf("%d", &item);
        top++;
        stack[top] = item;
        printf("%d pushed into stack.\n", item);
    }
}

// ============================
// POP OPERATION
// ============================
void pop() {
    if (top == -1) {
        printf("Stack Underflow! No element to pop.\n");
    } else {
        printf("%d popped from stack.\n", stack[top]);
        top--;
    }
}

// ============================
// DISPLAY OPERATION
// ============================
void display() {
    if (top == -1) {
        printf("Stack is empty.\n");
    } else {
        printf("Stack elements are:\n");
        for (int i = top; i >= 0; i--) {
            printf("%d\n", stack[i]);
        }
    }
}
```

QUEUE

- linear data structure
- FIFO
- Its FCFS (first come first serve)

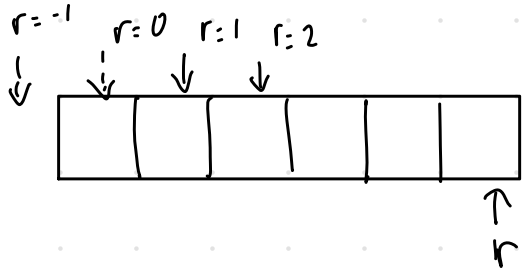
A queue is an ordered collection of items where items may be inserted to one end & deleted from the other end



initial: f = 0 & r = -1

Insertion:

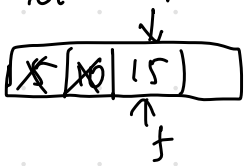
```
void insertion (int queue[], int *r, int item)
{       if (r == (MAX - 1))
        {   QUEUE FULL
            return;
        }
        (*r)++;
        queue[r] = item;
}
```



deletion :

→ it can be performed by incrementing f pointer

→ when f and r point to same location, it indicates only 1 element in the queue




```
void deletion (int queue[], int *f, int *r)
{       if (r == -1 && f == 0)
        {  QUEUE empty
           return;
        }
        printf ("%d deleted", queue[*f]);
       (*f)++;
       if (*f > *r)
       {    *f = 0;
            *r = -1;
        }
}
```

full implemented code →

QUEUE

```c
#include <stdio.h>
#define MAX 5  // maximum size of the queue

// Function prototypes
void enqueue(int queue[], int *front, int *rear);
void dequeue(int queue[], int *front, int *rear);
void display(int queue[], int front, int rear);

int main() {
    int queue[MAX];
    int front = -1, rear = -1;
    int choice;

    while (1) {
        printf("\n--- QUEUE MENU ---\n");
        printf("1. Enqueue\n");
        printf("2. Dequeue\n");
        printf("3. Display\n");
        printf("4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1: enqueue(queue, &front, &rear); break;
            case 2: dequeue(queue, &front, &rear); break;
            case 3: display(queue, front, rear); break;
            case 4: return 0;
            default: printf("Invalid choice! Try again.\n");
        }
    }
}

// ============================
// ENQUEUE OPERATION
// ============================
void enqueue(int queue[], int *front, int *rear) {
    int item;

    if (*rear == MAX - 1) {
        printf("Queue Overflow! Cannot insert element.\n");
    } else {
        printf("Enter the element to enqueue: ");
        scanf("%d", &item);

        if (*front == -1)
            *front = 0;

        (*rear)++;
        queue[*rear] = item;
        printf("%d enqueued successfully.\n", item);
    }
}

// ============================
// DEQUEUE OPERATION
// ============================
void dequeue(int queue[], int *front, int *rear) {
    if (*front == -1 || *front > *rear) {
        printf("Queue Underflow! No element to dequeue.\n");
    } else {
        printf("%d dequeued from queue.\n", queue[*front]);
        (*front)++;
        if (*front > *rear) {
            *front = *rear = -1;  // reset queue
        }
    }
}

// ============================
// DISPLAY OPERATION
// ============================
void display(int queue[], int front, int rear) {
    if (front == -1) {
        printf("Queue is empty.\n");
    } else {
        printf("Queue elements are:\n");
        for (int i = front; i <= rear; i++) {
            printf("%d ", queue[i]);
        }
        printf("\n");
    }
}
```

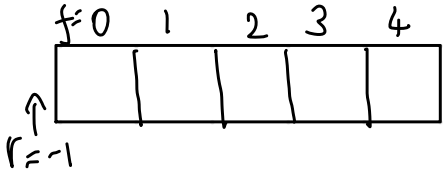
disadvantages:

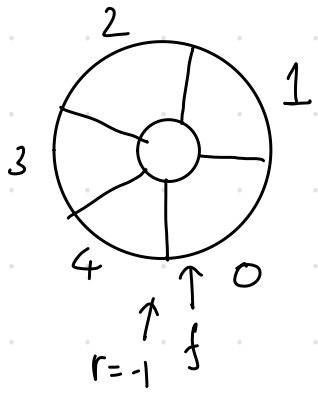
shows queue is not full the ordinary queue displays queue overflow.

CIRCULAR QUEUE

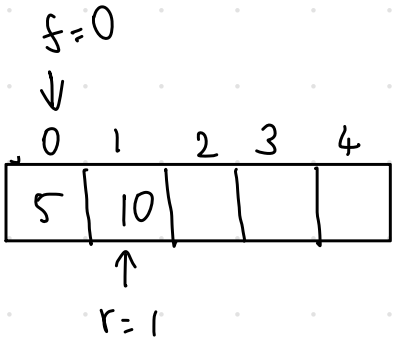
$r = (r+1) \% (\text{queue size})$

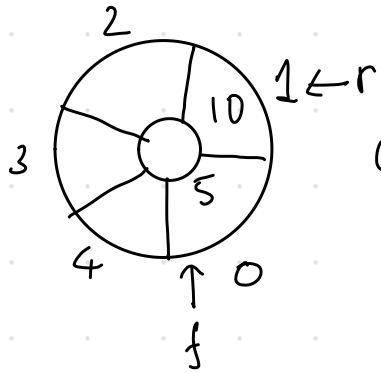
Notation:



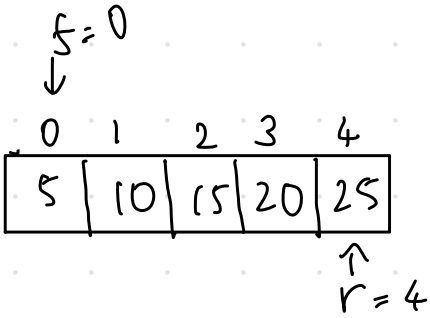


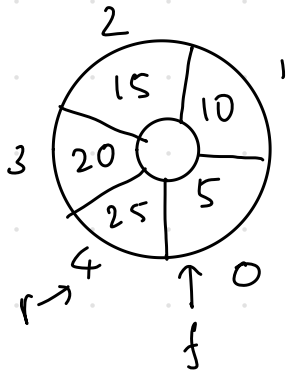
count = 0



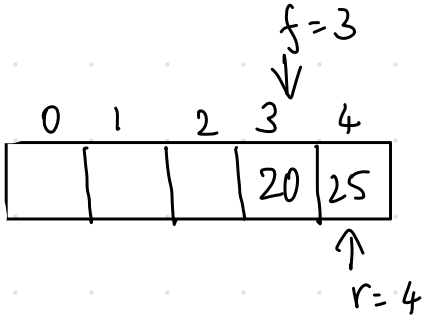


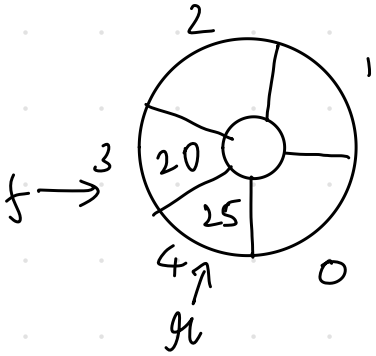
count = 2



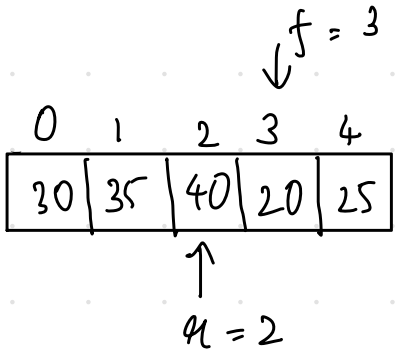


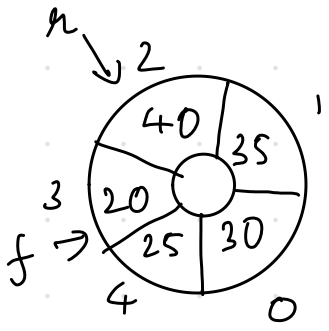
count = 5





count = 2





count = 5

Insert rear operation:

- Its performed at the rear end of the queue
- During insertion we've to check queue overflow condition → `if (count == queue_size)`

`r = (r + 1) % (queue_size)`

```
void insert (int cq[], int *r, int item, int *count)
{   if (*count) == queue_size)
    {  printf("queue full");
       return;
    }
   *r = (*r+1) % (queue_size);
   c_q[*r] = item;
  (*count)++;
}
```

Delete front operation:

- performed at the front end of the queue.
- we've to check queue underflow → `if(count == 0)`

`f = (f+1) % (queue_size)`

```
void delete ( int cq[], int *f , int *count)
{       if ((*count) == 0)
        {   printf ("queue underflow");
            return;
        }
        printf ("%d element is delleted", c_q[*f];
        *f = (*f +1) % (queue_size);
        (*count)--;
}
```

Display operation:

- Displays the contents of the queue

```
void display ( int cq[], int f, int count)
{       int temp, i;
        if ( count == 0)
        {   printf ("queue underflow");
            return;
        }
        for ( i= 1; i< count; i++)
        {   printf("%d ", c_q[f] );
            f = (f+1) % (queue_size);
        }
}
```

RECURSION:

- Its the capacity of the f^n to call itself
- Its an application of stack
- Its a name given to a technique of defining a program or process in terms of itself until a condition is true

recursive definition : It's a method of writing mathematical condition in a logical notation.

It consists of :-

→ recursive call

→ termination condition

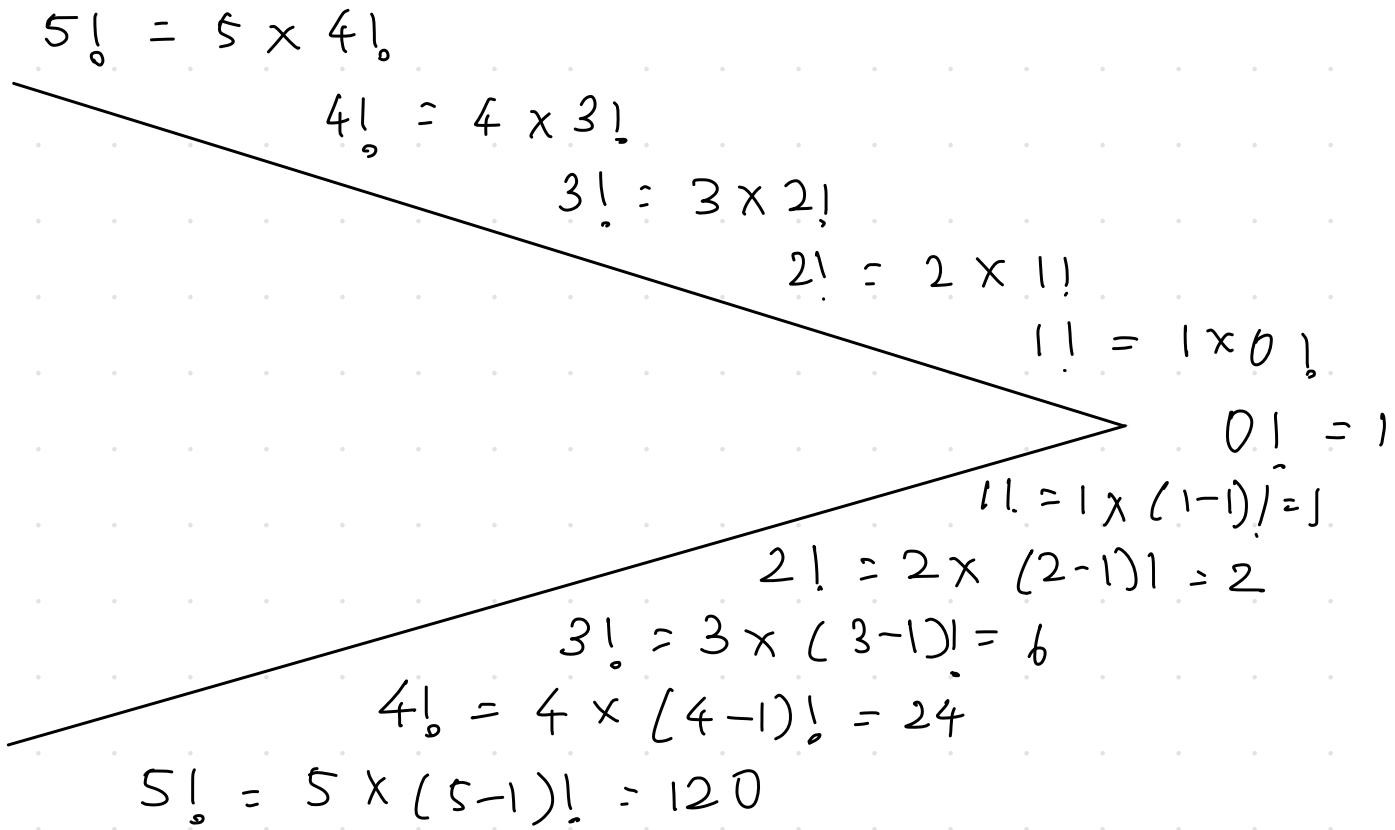
FACTORIAL :

n! = product of numbers from 1 to n.

The recursive definition is given below:

```
fact(n) {
    if (n == 0)
        return 1
    if (n == 1)
        return 1
    else
        return n * fact(n-1)
}
```

Assume we are finding fact(5)



code:

```
#include <stdio.h>
int fact(int n)
{   if (n==0)
        return 1;
    else
        return n * fact(n-1);
}

void main()
{   int n;
    printf("enter a number");
    scanf("%d", &n);
    int res = fact(n);
    printf("%d! = ", res);
}
```

fⁿ call : application of system stack

note : if stack not empty, control stays with recursion

first recursive call then basic operation executed

stack contents :

fact(0) 1

fact(1) 1 * 1

fact(2) 1 * 2

fact(3) 2 * 3

fact(4) 6 * 4

fact(5) 24 * 5 = 120 → returns to main

BINARY SEARCH

- searching technique based on divide & conquer
- preliminary condition : all ele to be sorted

working :

1. find mid position
   → compare key with middle ele
2. if key < a[mid]
   → perform search in left part of array
   else
   → perform search in right part of array
3. exit

recursive definition :

binary () =
- if (key == a[mid]
  - return mid
- if (key < a[mid]
  - return binary (a, key, low, mid-1)
- else
  - return binary (a, key, mid+1, high)

code :

```
# include <stdio.h>
 int binary ( int a[], key , low, high)
 {   if (low > high)
          return -1;
     int mid = (low + high)/2;
     if ( key == a[mid])
          return mid;
     else if (key < a[mid])
          return binary (a, key, low, mid-1);
     else
          return binary (a, key , mid+1 , high);
}
 int main ()
 {   int a[50];
     int n, key ;
     printf ( " enter num of eles : ");
     scanf ("%d", &n);
     printf ( " enter elements ");
```

```
for ( int i = 0; i < n ; i++)
        scanf ("%d", &a[i]);
printf ("ele to find ");
scanf ("%d", &key);
int res = binary (a, key, 0, n-1);
printf ("ele found at %d", res);
}
```

Stack : a : 5, 10, 15, 20, 25 key = 20

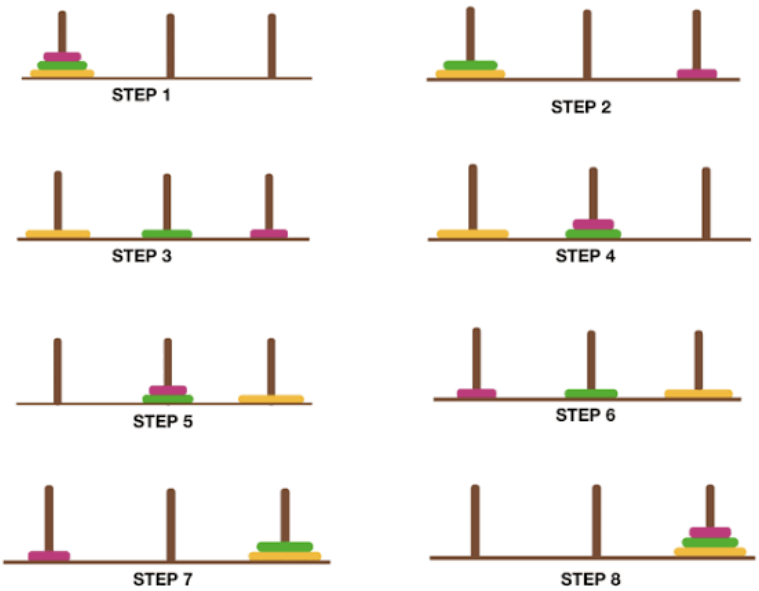
binary (a, 20, 3, 4)
binary (a, 20, 0, 4)

TOWER OF HANOI

- solved efficiently with recursion
- in this prob we need to move n disks from source to destination w/o violating :-

1. smallest disc must be at top of needle
2. only 1 disc can be transfered at a time

consider 3 discs & 3 needles :-



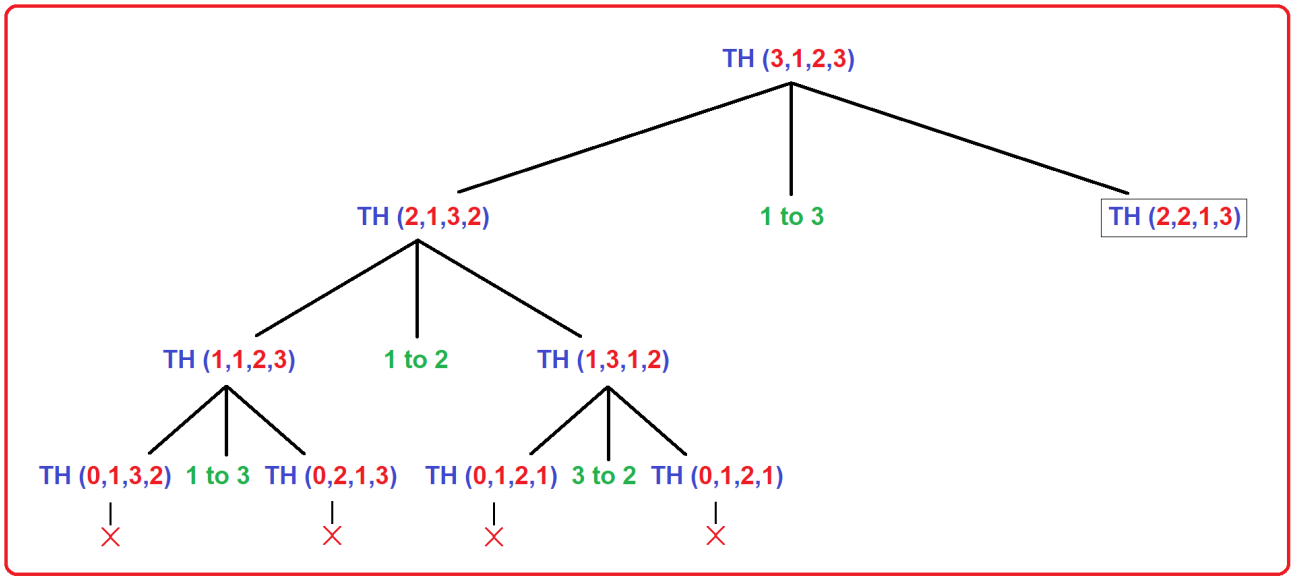
recursive def^n :-

$$TOH() = \begin{cases} \text{if } (n == 1) \text{ return} \\ \text{tower } (n-1, s, d, t) \\ \text{show the transfer of disc} \\ \text{tower } (n-1, t, s, d) \end{cases}$$

code :

```
# include <stdio.h>
void tower( int n, char s, char t, char d)
{       if (n == 1)
        {   printf("move disc %d from %c to %c",
             n, s, d);
             return;
        }
        tower (n-1, s, d, t);
        printf("move disc %d from %c to %c",
          n, s, d);
        tower (n-1, t, s, d);
}

void main()
{   int n;
    printf("enter the num of discs %d", &n);
    tower (n, 'a', 'b', 'c');
}
```

trace:



APPLICATIONS OF STACKS:

There are various applications of stacks:-

1. conversion of expression
2. evaluation of expression
3. recursion

EXPESSION NOTATIONS:

1. Infix expression
2. postfix expression
3. prefix expression

INFIX:

Its the one where operator is placed b/w 2 operands

eg: a + b

POSTFIX:

Its the one where operands precced the operator

eg: ab+c-

PREFIX

Its the one wher operands followo the operator

eg: +AB

+-ABC

CONVERSION OF EXPRESSIONS: -

1. INFIX TO POSTFIX

normally, mathematical expressions are denoted in infix but evaluation of postfix is much easier from compiler POV

consider infix expression:

((a + (b-c))*d)

= (a + (bc-)*d)

= ((a + (T_1))*d)

= ((a + T_1) *d)

= ((aT_1+) *d)

= (T_2 *d)

= T_2d*

⇒ aT_1+d*

⇒ abc-+d*

$$(((((a+(b-c))*d)-e)+f)*g)$$
$$(((((a+(bc-)*d)-e)+f)*g)$$
$$=((((a+T_1*d)-e)+f)*g)$$
$$=(((((a+T_1d*)-e)+f)*g)$$
$$=((((((a+T_2)-e)+f)*g)$$
$$=(((((aT_2+)-e)+f)*g)$$
$$=(((T_3-e)+f)*g)$$
$$=((T_3e-)+f)*g)$$
$$=((T_4+f)*g)$$
$$=(T_4f+)*g$$
$$=(T_5)*g$$
$$=T_5g*$$
$$=T_4f+g*$$
$$=T_3e-f+g*$$
$$=aT_2+e-f+g*$$
$$=aT_1d*+e-f+g*$$
$$=abc-d*+e-f+g*$$

recheck smth is off

program (infix to postfix)

preliminary req to convert infix to postfix :-

1. precedence f^n (F) : contains precedence value of the symbol on the top of the stack

2. precedence f^n(G) : contains precedence value of the symbol in the given string.

operator/ operand	F() stack precedence	G() i/p o/p precedence
+, -	2	1
x, /	4	3
$, ^	5	6
(	0	9
)		0
#	-1	
operand	8	7

Rule 1 :

until the precendence of ele on top of stack is greater than precendence of i/p symbol, then pop the ele of the stack and place them onto postfix expression.

```
while [F(s[top]) > G(symbol)
{   postfix[j++] = s[top--]
}
```

Rule 2:

precedence of ele which is located at top of stack ≠ precedence of i/p symbol then place the symbol onto the top of the stack

```
if ( F[S[top]) != G[symbol])
{
    S[++top] = symbol;
}
top--;
```

Rule 3:

precedence of ele at top of stack = precedence of i/p element then discard the element which is located at the top of the stack

consider the infix expression given below:

((A + (B - C)) * D)

	stack	S[top]	i/p symbol	F(S[top]) > G(symbol)		postfix
1.	#	#	(	-1 > 9	no	
2.	#(	(	(	0 > 9	no	
3.	#((	(	A	0 > 7	no	
4.	#((A	A	+	8 > 1	yes	A
5.	#((	(	+	0 > 1	no	
6.	#((+	+	(	2 > 9	no	
7.	#((+(	(	B	0 > 7	no	
8.	#((+(B	B	-	8 > 1	yes	AB
9.	#((+(	(	-	0 > 1	no	
10.	#((+(-	-	C	2 > 7	no	
11.	#((+(-C	C	)	8 > 0	yes	ABC

12. #CC+C-	-	)	2 > 0 yes	ABC-
13. #CC+C	C	)	0 > 0 no discard stack top	
14. #CC+	+	)	2 > 0 yes	ABC-+
15. #CC	C	)	0 > 0 no discard stack top	
16. #C	C	)	0 > 0 no discard stack top	
17. #	C	)	0 > 0 no discard stack top	
18. #	#	*	-1 > 3 no	
19. #*	*	D	4 > 7 no	
20. #*D	D	)	8 > 0 yes	ABC-+D
21. #*	*	)	4 > 0 yes	ABC-+D*

∴ postfix = ABC-+D*

program to convert infix to postfix

```
#include <stdio.h>

int F(char);
int G(char);
void infix_postfix(char[], char[]);

void main()
{
    char infix[20], postfix[20];
    printf("read infix expression");
```

```
    scanf ("%s", infix);
    infix-postfix (infix, postfix);
    printf ("The postfix expression %s", postfix);
}

int F (char symbol)
{
    switch (symbol)
    {   case '+':
        case '-':
            return 2;
        case '*':
        case '/':
            return 4;
        case '$':
        case '^':
            return 5;
        case '(':
            return 0;
        case ')':
            return;
        case '#':
            return -1;
        default:
            return 8;
}
```

```
int G(char symbol)
{
    switch (symbol)
    {   case '+':
        case '-':
                return 1;
        case '*':
        case '/':
                return 3;
        case '$':
        case '^':
                return 6;
        case 'C':
                return 9;
        case ')':
                return 0;
        case '#':
                return;
        default:
                return 7;

}
```

```
void infix_postfix (char infix[], char postfix[])
{   int top = -1;
    int s[30];
    int i, j;
    char symbol;
    s[++top] = '#';
    j = 0;
    for (i = 0; i < strlen(infix); i++)
    {   symbol = infix[i];
    }
    place the 3 conditions
}
```

complexity of this ↑

POSTFIX EPRESSION EVALUATION:

The following rules are req to evaluate postfix expression

Rule-1 : If the i/p symbol is an operand place it onto the stack

s[++top] = symbol

Rule-2 : If the i/p symbol is an operator then pop 2 operands from the stack, perform the operation on operands wrt operator

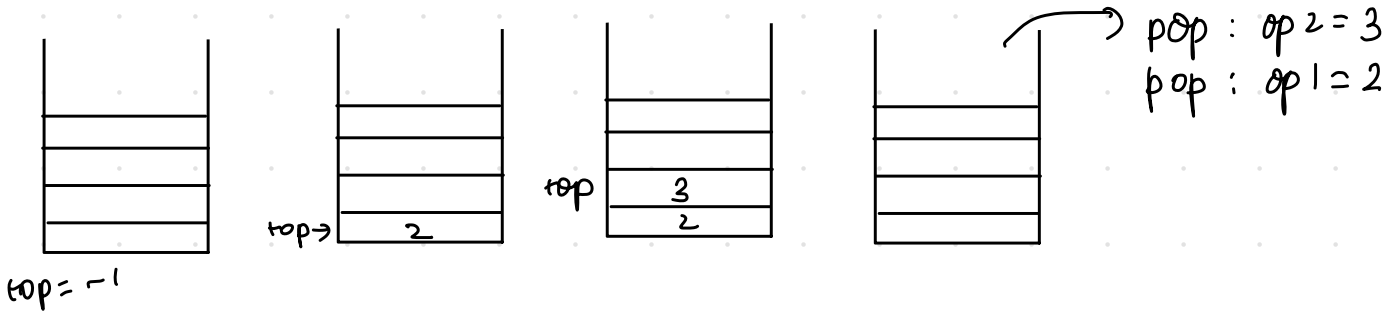
op2 = s[top--]; op1 = s[top--];

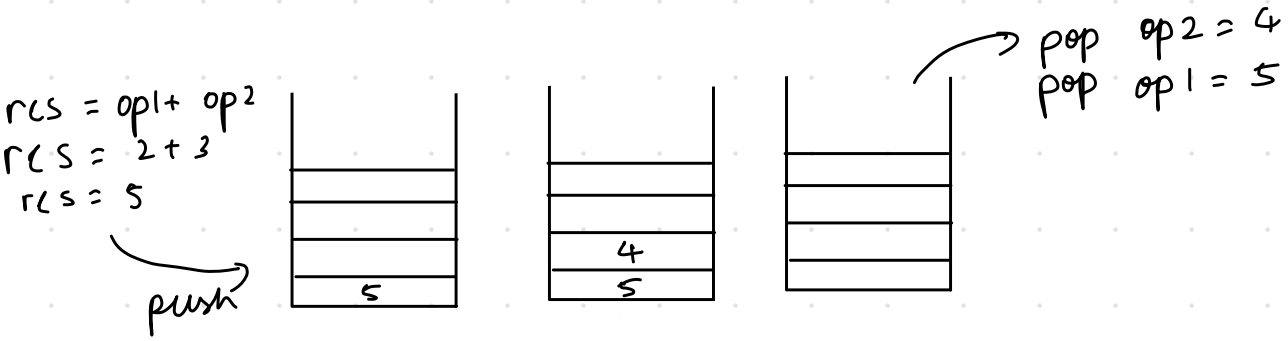
Rule-3 : place intermediate & final result on stack

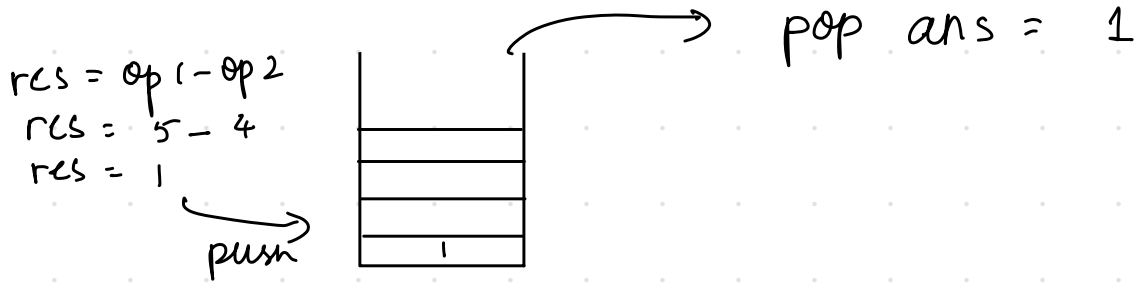
res = op1 operator op2

s[++top] = res

Working : 23+4-







(((4 + 3) - 7) * 9)

= (((43+) - 7) * 9)

= ((A) - 7) * 9)

= ((A7-) * 9)

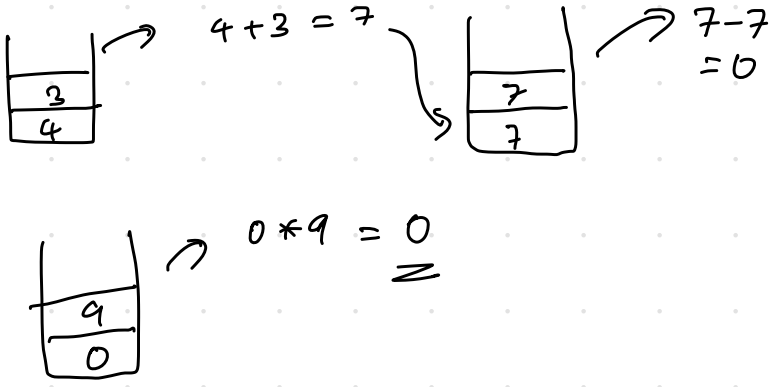
= (B * 9)

= B9*

= A7 - 9 *

= 43 + 7 - 9 *

evaluation :



code :

C program to implement evaluation of postfix expression :-

```
#include <math.h>
#include <stdio.h>
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);
        default:
            return ;
    }
}

void main()
{   double s[20];
    int top = -1;
    double op1, op2, res;
    int i;
    char postfix[20];
    char symbol;
    printf("enter a valid postfix expression");
    scanf("%s", postfix);
    for(i=0; i<strlen(postfix); i++)
    {   symbol = postfix[i];
        if(isdigit(symbol))
            s[++top] = symbol - '0';
        else
        {   op2 = s[top--];
            op1 = s[top--];
            res = compute(symbol, op1, op2);
            s[++top] = res;
        }
    }
    printf("The result is %lf", res);
}
```

INFIX TO PREFIX:

Consider the given infix expression:

(((A + B) - C) * D)

= * - + A B C D

symbol	F() - stack	G() - symbol i/p
+ -	1	2
* /	3	4
^ $	6	5
(	null	0
)	0	9
#	-1	null
operand	8	7

rules to convert infix to prefix:

preliminary req are F() & G()

note:

before applying rules, infix array must be reversed

rule - 1: until the precedence of ele which is located at top of stack > precedence of i/p symbol, pop the element from stack & place them onto prefix expression

rule-2: precedence of ele on top of the stack != precedence of i/p symbol then place the i/p symbol onto the stack

rule-3: precedence of ele on the top of the stack == precedence of i/p symbol then discard the ele which is located at top of the stack

```
while (fcs[top]) > G(symbol))
{       prefix[j++] = s[top--];
}
if ( F(s[top]) != G(symbol))
{       s[++top] = symbol;
}
else
{       top--;
}
```

)D*)C-)B+A((

stack	s[top]	i/p symbol	F[s[top]]>G(symbol)		prefix
#	#	)	-1 > 9	no	
#)	)	D	0 > 9	no	
#)D	D	*	8 > 4	yes	D
			0 > 4	no	
#)*	*	)	3 > 9	no	
#)*)	)	(	0 > 7	no	
#)*)(	(	-	8 > 2	yes	DC
			0 > 2	no	
#)*)-	-	)			

finish & reverse prefix array

program for infix to prefix

```
#include <stdio.h>
#include <string.h>
void infix_prefix(char[], char[]);
int F(char);
int G(char);

void main()
{
    char infix[20];
    printf("enter the valid infix expression");
    scanf("%s", infix);
    infix_prefix(infix, prefix);
    printf("prefix array is %s", prefix);
}

int F(char symbol)
{   switch(symbol)
    {
        case '+':
        case '-':
            return 1;
        case '*':
        case '/':
            return 3;
        case '^':
```

```
        case '$':
            return 6;
        case ')':
            return 0;
        case '#':
            return -1;
        default:
            return 8;
    }
}

int G(char symbol)
{   switch(symbol)
    {
        case '+':
        case '-':
            return 2;
        case '*':
        case '/':
            return 4;
        case '^':
        case '$':
            return 5;
        case ')':
            return 9;
        case 'C':
            return 0;
        default:
            return 7;
```

```
    }
}

void infix_prefix (char infix[], char prefix[])
{   char st[20];
    int top = -1;
    int i, j = 0;
    char symbol;
    s[++top] = '#';
    strrev(infix);
   for(i=0; i< strlen(infix); i++)
   {   symbol = infix[i];
       while( F(s[top]) > G(symbol))
       {   prefix[j++] = s[top--];
        }
       if ( F(s[top]) != G(symbol))
       {    s[++top] = symbol;
       }
       else
       {   top--;
       }
   }
    strrev(prefix);
}
```