Date- 4/09/19

UNIT-2

STACK

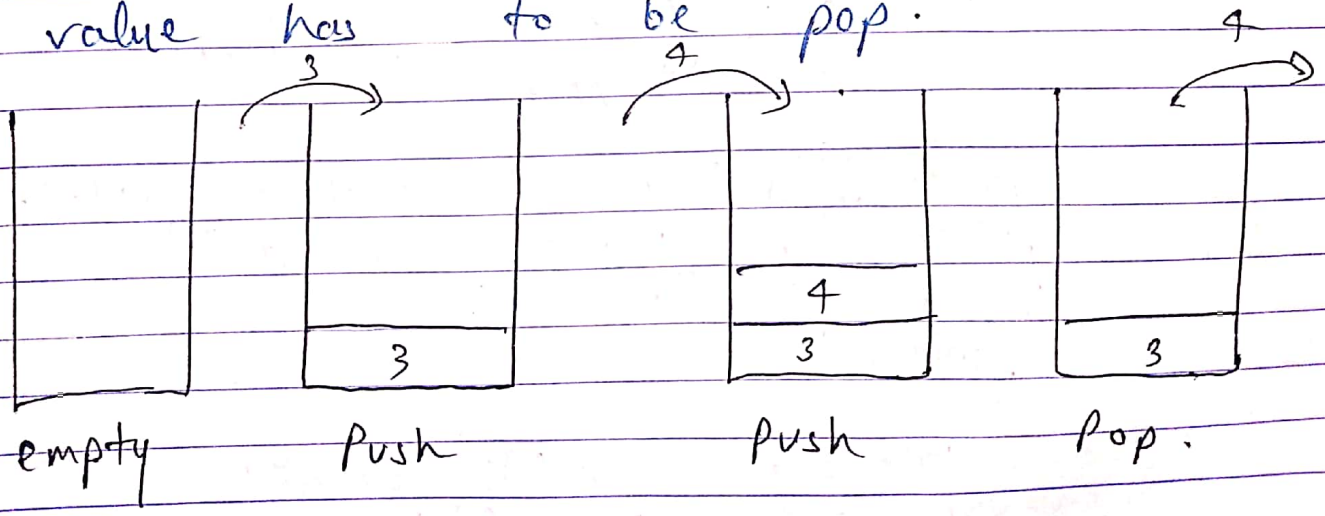
* Stack is the data structure which is used to restore data in LIFO order LIFO stand for Last In First out order.

* The element which is placed last is accessed first

* In stack terminology, insertion operation is called Push operation & removoral operation is called Pop operation.

* Both insertion & removal are allowed at only one end of the stack called TOP of the stack.

* A stack data structure has a special variable called StackTop whose value denotes the position of stack where the value has to be inserted or from the value has to be pop.



* Implementation of stack is done by using Array & LinkList.

* BASIC OPERATIONS OF STACK

(i) Push ()
(ii) Pop ()
(iii) Peep ()
(iv) Isempty ()
(v) Isfull ()

```
#include<stdio.h>

int isfull(void);
int isempty(void);
void push(void);
int peep(void);
int pop(void);
int a[50], top = 0;
void main(void)
  {
    int i,n
    char ch='y';
    while(toupper(ch)=='Y')
      {
    printf("\n 1: PUSH \n 2:POP \n 3: PEEP\n");
    printf("\n Enter your choice ");
    scanf("%d", &i);
    switch(i)
    {
      Case 1:  push(); break;
      Case 2:  n=pop();
```

```
                                    deleted
        printf("\n Value of top is %d", n);
            break;
    Case 3 :    n = peep();
        printf("\n Value at top is %d", n);
            break;
    default :
        printf("\n Wrong Choice");
    }
        printf("\n Do you want to continue
                Y/N");
        scanf("%c", &ch);
    }
}
    int isempty()
    {
        if (top == 0)
            return 1;
        else
            return 0;
    }.
    int is full()
    {
        if (top == 50) // Size of stack array
            return 1;
        else
            return 0;
    }
    void push (void)
    {
        int n;
        if (isfull())
```

```
printf("\n stack is full");
else
{
 printf("\n Enter the value ");
 scanf("%d", &n);
   a[top] = n;
    top++;
  }
}printf("\n Value inserted successfully");
}
void pop (void)
 {
   int n;
   if (isempty())
     printf("\n Stack is empty");
   else
     top--;
     n = a[top];
      return n;
  }
 printf("Value deleted successfully");
 }.
int peep (void)
 {
   int n;
   if (isempty())
     printf("\n Stack is empty");
   else
     n = a[top-1];
      return n;
 }.
```

REVERSE POLISH →

+ ÷ 18

(Imp)

INFIX TO POSTFIX CONVERSION

Priority
( )
↑ or ^
* or /
+ or -

Q (A+B/C * (D+E) - F)

Scan Symbol	Stack	Postfix Expression
(	(	
A	(	A
+	(+	A
B	(+	A B
/	(+ /	A B
C	(+ /	A B C
*	(+ * (	A B C /
(	(+ * (	A B C /
D	(+ * (	A B C / D
+	(+ * (+	A B C / D
E	(+ * (+	A B C / D E
)	(+ * (+)	A B C / D E +
-	(* - ÷	A B C / D E + * F
F	(-	A B C / D E + * F +
)	(	A B C / D E + * F - +

ABC/DE+*+F-+ is the postfix expression.

Q ((G * (F ↑ E/D) - C * B) + A)

Scan	Stack	Postfix Expression.
(	(	
(	((	
G	((	G

((G * (F ↑ E / D) - C * B) + A)

*	((*	G
(	((* (	G
F	((* (	GF
↑	((* (↑	GF
E	((* (↑	GFE
/	((* (/	GFE ↑
D	((* (/	GFE ↑ D
)	((* (/)	GFE ↑ D /
-	((* -	GFE ↑ D / *
C	((-	GFE ↑ D / * C
*	((- *	GFE ↑ D / * C
B	((- *	GFE ↑ D / * C B
)	((- *)	GFE ↑ D / * C B * -
+	(+	GFE ↑ D / * C B * -
A	(+	GFE ↑ D / * C B * - A
)	(+) empty	GFE ↑ D / * C B * - A +

GFE↑D/*CB*-A+ is the postfixpression

INFIX TO PREFIX CONVERSION -

Q ((G * (F ↑ E / D) - C * B) + A)
(A + (B * C - (D / E ↑ F) * G))

Reverse Infix → Postfix.
Postfix → Reverse.

(A + (B * C - (D / E ↑ F) * G)) . 20

Scan Symbol	Stack	Postfix Expression
(	(	
A	(	A
+	(+	A
(	(+ (	A
B	(+ (	A B
*	(+ (*	A B
C	(+ (*	A B C
-	(+ (* -	A B C *
(	(+ (- (	A B C *
D	(+ (- (	A B C * D
/	(+ (- (/	A B C * D
E	(+ (- (/	A B C * D E
↑	(+ (- (/ ↑	A B C * D E
F	(+ (- (/ ↑	A B C * D E F
)	(+ (- (/ ↑)	A B C * D E F ↑ /
*	(+ (- *	A B C * D E F ↑ /
G	(+ (- *	A B C * D E F ↑ / G
)	(+ (- *)	A B C * D E F ↑ / G * -
)	(+)	A B C * D E F ↑ / G * - +

Now Prefix is + - * G / ↑ F E D * C B A

POSTFIX TO INFIX.

Q GFE ↑ D / * C B * - A +

=

Scan	Top of Stack	Expression Stack.
G	G	G
F	F	F G
E	E	E F G
↑	E ↑ F	E↑F G
D	D	D E↑F G
/	D / (E↑F)	D/(E↑F) G
*	(D/(E↑F)) * G	(D/(E↑F)) * G
C	C	C (D/(E↑F)) * G
B	B	B C (D/(E↑F)) * G
*	B * C	B * C (D/(E↑F)) * G
-	(B * C) - ((D/(E↑F)) * G)	(B*C) - ((D/(E↑F)) * G)

A	A	A
+	A + (B*C) - ((D/(E↑F)) * G)	(B*C) - ((D/(E↑F)) * G)
		A+(B*C) - ((D/(E↑F)) * G)

PREFIX TO INFIX.

Q = + - * A B / C D E

Reverse → E D C / B A * - +

E	E	E
D	D	D E
C	C	C D E
/	C / D	C / D E
B	B	B C / D E
A	A	A B C / D E
*	A * B	A * B C / D E

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

(A*B)-(C/D)
E

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

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

Q Convert ^Postfix AB- DE+F*/ into infix.

A	A	A
B	B	B / A
-	(B-A)	(B-A)
D	D	D / (B-A)
E	E	E / D / (B-A)
+	(E+D)	(E+D) / (B-A)
F	F	F / (E+D) / (B-A)
*	F*(E+D)	F*(E+D) / (B-A)

F * (E+D) / (B - A) (F * (E+D) / (B-A))

II Sessional Start :-

RECURSION -

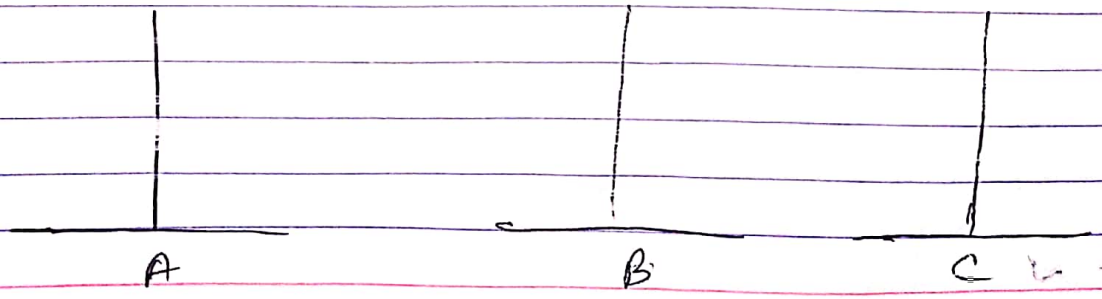
* TOWER OF HANOI

Tower of Hanoi is a mathematical puzzle where we have three poles or rod & n discs. The n disc is placed on one pole in decreasing order from bottom to top. The objective of puzzle is to move entire disc stack to another pole obeying the following rules-

Rule 1- Only one disc can be move from one pole to another pole at a time.

Rule 2- Each move consist of taking the upper disc from one of the stack & placing it on top of another stack it implies the disc can only be moved if it is upper most disc on the stack.

Rule 3- No disc may be placed on the top of smaller disc



In General the pattern of movement of n disc will be -

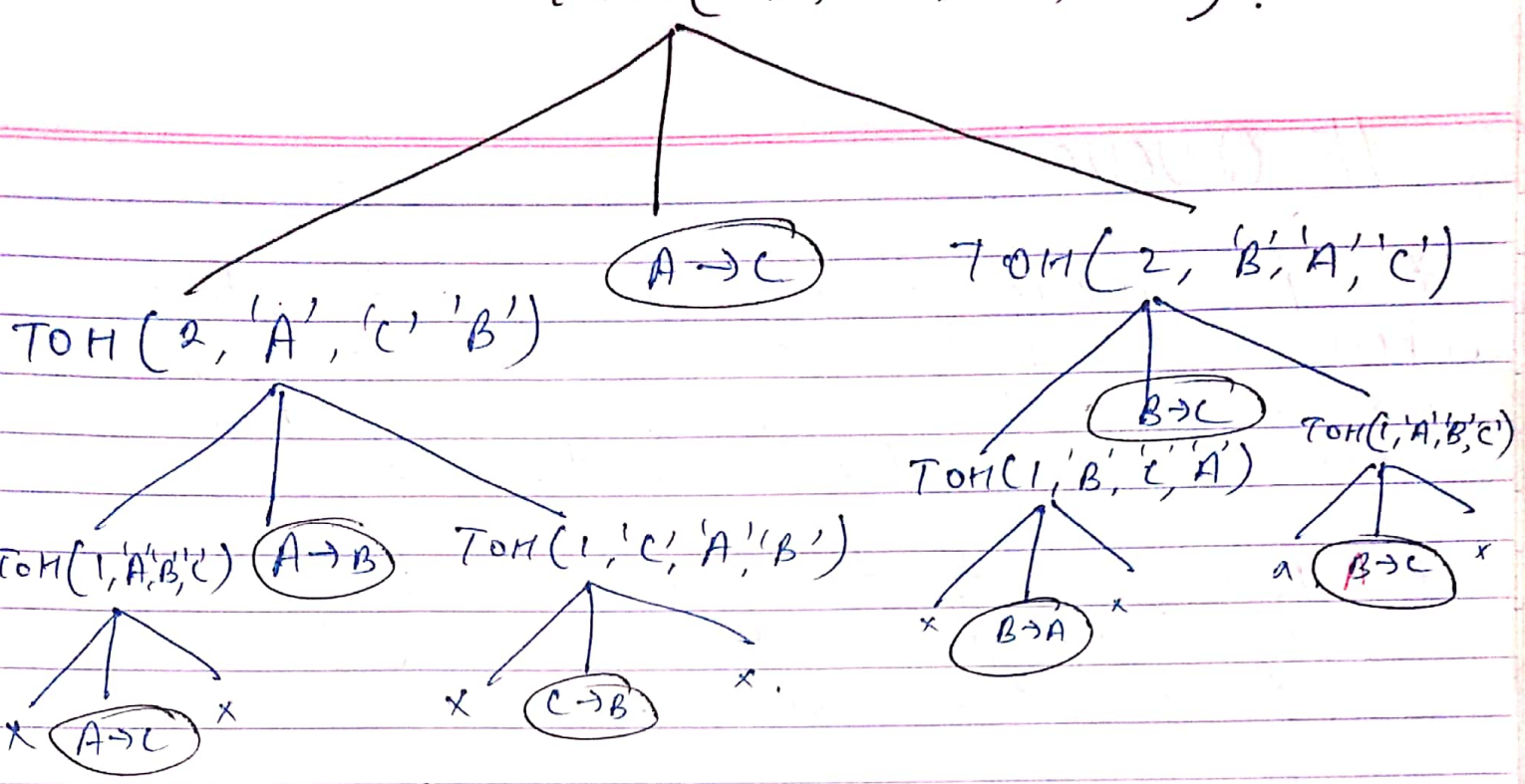
Step 1 - Move n-1 disc from A → B using C

Step 2 - Move a disc from A → C

Step 3 - Move n-1 disc from B → C using A.

By above movement we can see there is a recursive call in Step 1 & step 3. The algorithm / function of Tower of Hanoi will be -

```
void TOH(int n, char A, char B, char C).
{
    if (n > 0)
    {
        TOH(n-1, 'A', 'C', 'B')
        printf("\n Move Disc %c to %c ", A, C);
        TOH(n-1, 'B', 'A', 'C');
    }
}
```



Order of execution step. —

(A→C), (A→B), (C→B), (A→C), (B→A), (B→C), (B→C)

Note - Total no. of movement of n disc will be $2^n - 1$.

In every step we have to show poles.

QUEUE -

* A queue is a linear data structure where the ~~first~~ element is inserted from one end called rear & deleted from other end called front.

* front points to the beginning of the queue whereas rear points to the end of the queue.

* Queue follows the FIFO (First in First out)

* According to its FIFO structure element inserted first will also be removed first.

* In Queue, one end is always used to insert data known as enqueue & the other end is used to delete data known as dequeue. The enqueue & dequeue are the two important function for any insertion & deletion operation performed on queue.

FUNCTION ( ).

⇒
```
int a[5], front = -1, rear = -1;
int isempty (void)
{
  if( front == -1 || front > Rear).
```

```
    return 1;
  else
    return 0;
}

int isfull(void)
{
  if ( Rear >= (Size -1)   // Size → array
      return 1;
  else
      return 0;
}

void enqueue(int x)
{
  if (isfull())
    printf("\n Queue is full");
  else if( Rear == -1 || front == -1)
      Rear = front = 0;
  else
      Rear++;
      a[Rear] = x;
  }.a[Rear] = x;
}

int dequeue(void x)
{
  int t;
  if (isempty())
      printf("\n Queue is Empty");
  else
  {
```

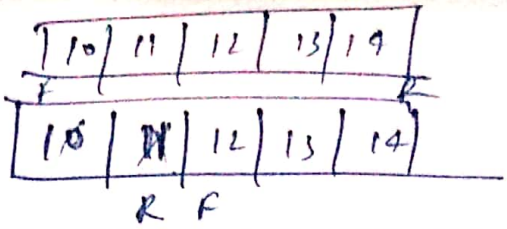
```
    t = a[front]
    f++;
    return t;
}

void display(void)
{
    int i;
    if(isempty(1)
        printf("Queue is Empty");
    else
    {
        for(i=front; i<=Rear; i++)
        {
            }.printf("%d", a[i]);
        }
    }
```

CIRCULAR QUEUE.

⇒ FUNCTION()

```
int isempty(void.)
{
    if(front == -1)
        return 1;
    else
        return 0;
}.
```

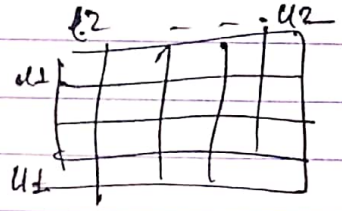



```
int isfull(void)
{
  if ( (front == 0 && R == size-1) || (front == Rear+1) )
      return 1;
  else
      return 0;
}

void enqueue (int x)
{
  if(isfull())
  {
    printf("\n Queue is full");   // Overflow
  }
  else
  {
    if( Rear == -1)
    {
      front = 0;
      Rear = 0;
    }
    else if( Rear == size-1)
    {
      Rear = 0;
    }
    else
    {
      R = R+1;
    }
    a[R] = x;
  }
}
```

int a[3] = {21, }

a[ ] = {21 }



a[l:5]

(u1-l1+1)(u2-l2+1)

a[l1 : u1]

[u1-l1+1]

5-1+1 = 5

5-0+1 = 6

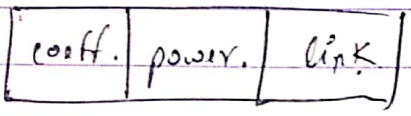
```
int cdequeue (void)
{
    int x;
    if( isempty ())
    {
        printf ("\n Underflow ");
        return;
    }
    else
    {
        x = a[front];
        if ( front == Rear) // Single element.
        {
            front = -1;
            Rear = -1;
        }
        else if ( front == size -1)
        {
            front = 0;
        }
        else
        {
            front = front + 1;
        }
        return x;
    }
}

void cdisplay (void)
{
    int i;
    if( isempty ())
```

```
{
}printf("\n Underflow ");
else
{
  for( i=front ; i!=Rear ; i=(i+1)%5.
  {
  } printf("%d", a[i]);
    printf(" %d", a[R]);
}
}
```

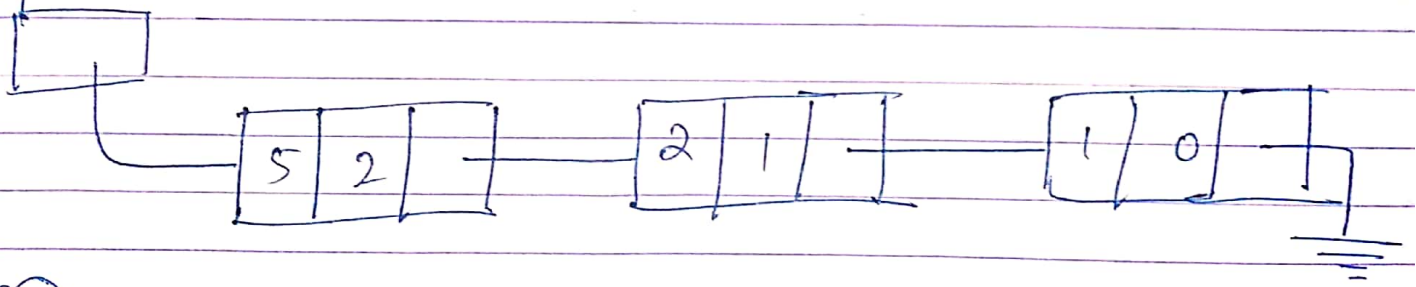
POLYNOMIAL.

* SINGLE VARIABLE



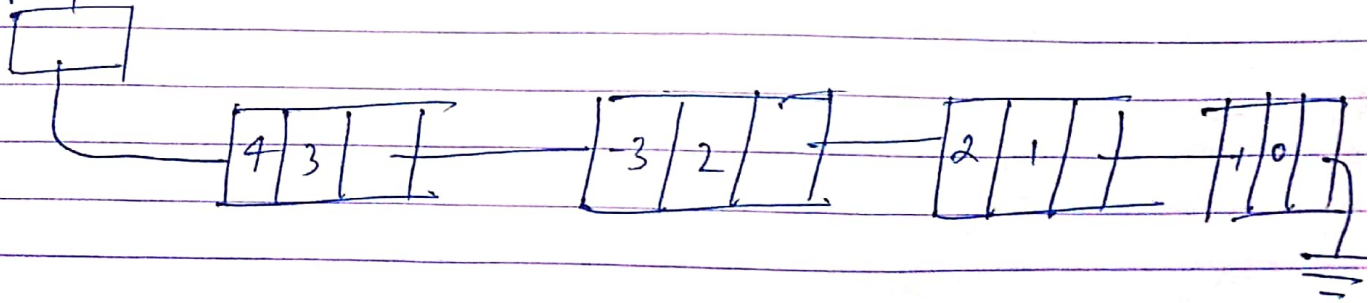
(i) $5x^2 + 2x + 1$.

Poly 1



(ii) $4x^3 - 3x^2 + 2x - 1$.

Poly 2



* TWO VARIABLE - | Coeff. | Power of x | Power of y | link. |

Q $7*x^3*y^2 - 8*x^2*y + 3*x*y + 11*x - 4$.

⇒ Poly