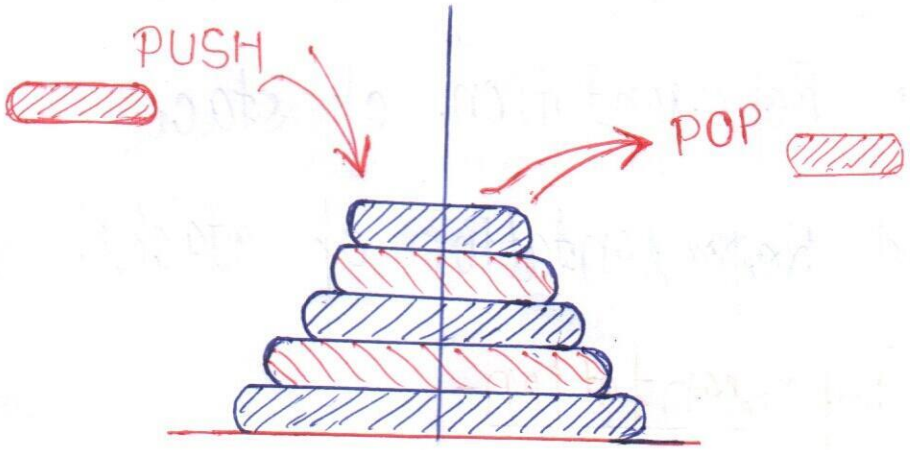
UNIT-2

STACKS:- A stack is a list of elements in which an element may be inserted or deleted at only one end, called the top of the stack.

- Stack is a linear data structure.
- Stack is follow the principle of LIFO (Last in First out) or FILO (First in last out).



OPERATIONS:-

- **PUSH()** - Push is the operation used to Insert an element into a stack.
- **POP()** - Pop is the operation used to delete an element from a stack.
- **Peek()** - Returns the value of top element of the stack.
  Top()

↳ isEmpty()- Return true if stack is empty.

↳ isfull()- Return true if stack is full.

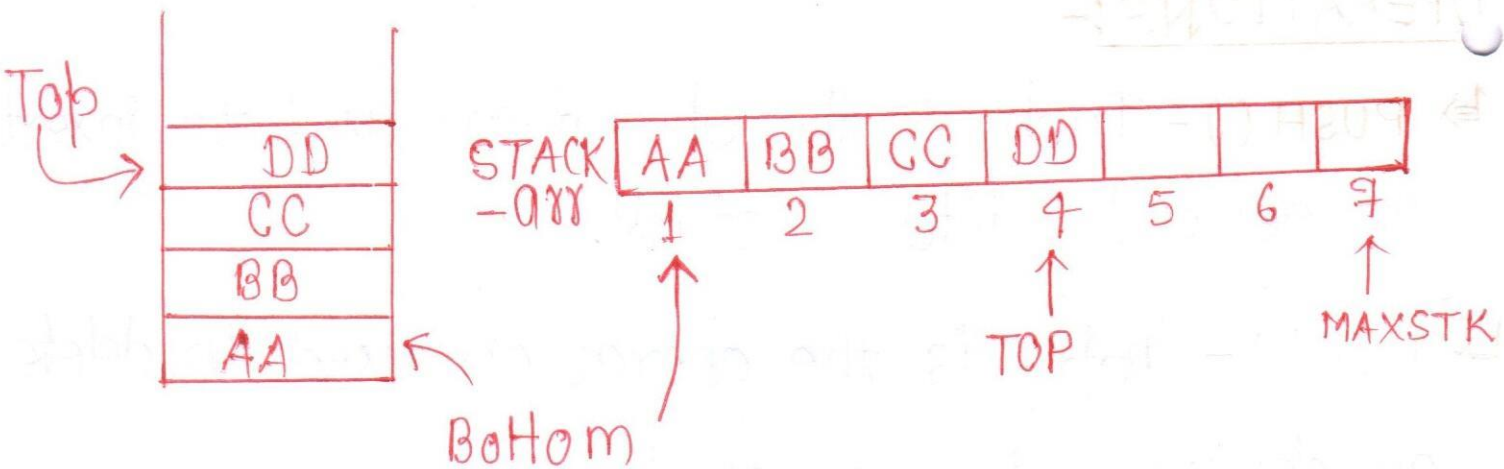
↳ size()- Return size of stack.

REPRESENTATION OF STACKS:- Stacks ~~has~~ represented in memory by two ways-

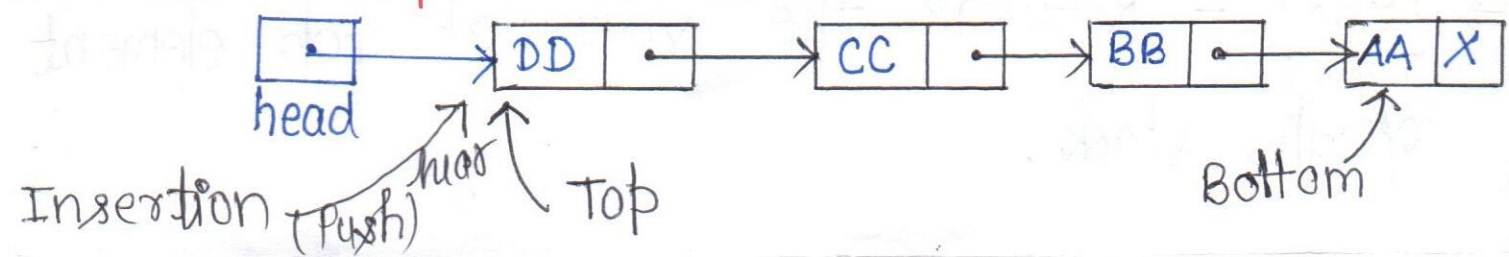
a) Arrays Representation of stacks.

b) Linked Representation of stacks.

a) Array Representation-



b) Linked Representation-



NOTE: ↳ TOP = 0 OR TOP = NULL (stack is empty)

↳ MAXSTK - maximum number of elements that can be held by the stack. MAXSTK = 8.

↳ TOP - Top contain the location of top element of the stack.

ALGORITHM - PUSH (STACK, TOP, MAXSTK, ITEM)

This procedure push an ITEM onto a STACK.

step 1. [stack already filled]
if TOP = MAXSTK, then PRINT: OVERFLOW

step 2. Set TOP = TOP + 1 [Increase Top by 1]

step 3. Set STACK[TOP] = ITEM [Insert ITEM at new TOP Pos^n]

step 4. Return.

ALGORITHM - POP (STACK, TOP, ITEM)

This procedure deletes the top element of STACK and assigns it to the variable ITEM.

step 1 - [stack has an ITEM to be removed]
if TOP = 0 then PRINT: UNDERFLOW.

step. 2. Set ITEM = STACK[TOP] [Assigns Top element to ITEM]

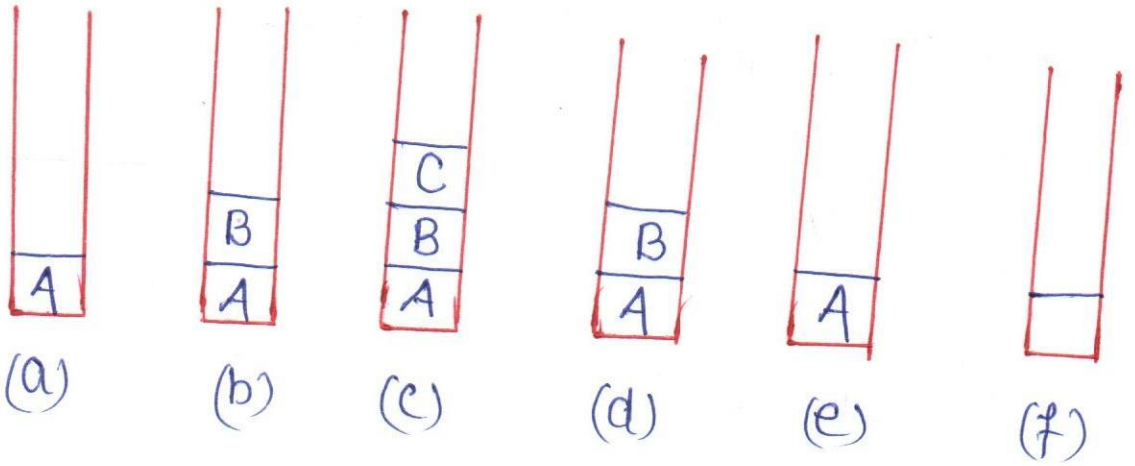
step 3. Set TOP = TOP - 1 [Decrease TOP by 1]

step 4. Return.

IMPLEMENTATION OF STACKS BY USING ARRAYS

```
#include <stdio.h>
#include <stdlib.h>
#define max 4
  int stack_arr[max];
  int top = -1;
  void push (int data)
    {
      if (top == max-1)
        {
          printf (" STACK OVERFLOW ");
          return;
        }
      top = top + 1;
      stack_arr[top] = data;
    }
    int main ()
      {
        push (1);
        push (2);
        push (3);
        push (4); return 0;
      }
```

POSTPONED DECISIONS- STACKS are frequently used to indicate the order of the processing of data when certain steps of the processing must be postponed until other condition are fulfilled.



ARITHMETIC EXPRESSION- Let Q be an arithmetic expression involving constant and operations. The binary operations in Q may have different levels of precedence.

Highest ⟶ (↑)

Next Highest ⟶ *, /

Lowest ⟶ +, −

1) Above three priority apply to parenthesis free exp.

2) The operation apply from left to right.

Q. Evaluate the following expression

$2 \uparrow 3 + 5 * 2 \uparrow 2 - 12 / 6$

step1. $8 + 5 * 4 - 12 / 6$

step2. $8 + 20 - 12 / 6$

step3. $8 + 20 - 2$

step4. $28 - 2$

step5. 26

NOTATION = There are three types of NOTATION used for mathematical expression.

a) INFIX NOTATION — A+B, C-D, E*F, G/H.

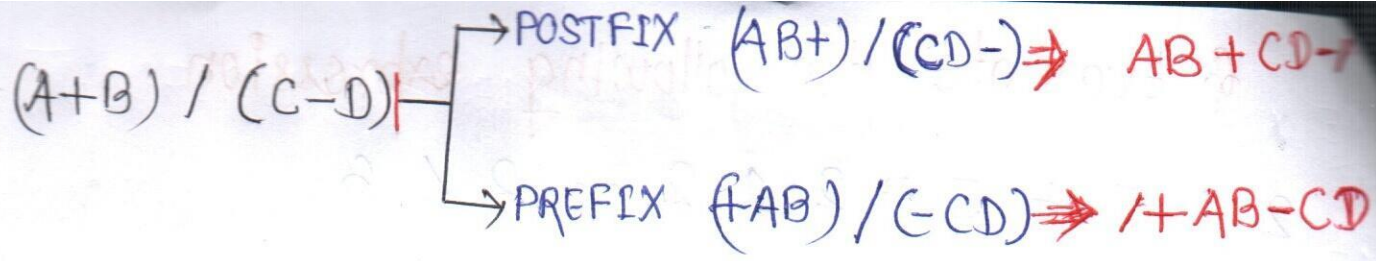
b) PREFIX NOTATION (POLISH NOTATION) — +AB, -CD, *EF, /GH

c) POSTFIX NOTATION — AB+, CD-, EF*, GH/.

Ex-

$(A+B) * C$
- POSTFIX: $(AB+) * C \Rightarrow AB+C*$
- PREFIX: $(+AB) * C \Rightarrow *+ABC$

$A + (B*C)$
- POSTFIX: $A+(BC*) \Rightarrow ABC*+$
- PREFIX: $A+(*BC) \Rightarrow +A*BC$



EVALUATION OF A POSTFIX EXPRESSION

Suppose P is an arithmetic expression written in postfix notation.

ALGORITHM- This algorithm find the value of an arithmetic expression P written in postfix notation.

Step 1. Add right parenthesis ")" at the end of P.

Step 2. Scan P from left to right and repeat step 3 and 4 for each elements of P untill right parenthesis ")" is encountered.

Step 3. IF an operand is encountered put it on STACK.

Step 4. IF an operator ⊛ is encountered then

a) Remove top two elements of stack, where A is the top elements and B is the next-to-top element

b) Evaluate B ⊛ A.

c) Place the result of (b) back on stack.

[End of IF structure]

[End of step 2 loop]

Step 5. Set value equal to the top element on STACK

Step 6. Exit

Ex. P: 5, 6, 2, +, *, 12, 4, /, −.

Equivalent infix exp: Q: 5 * (6+2)−12/4

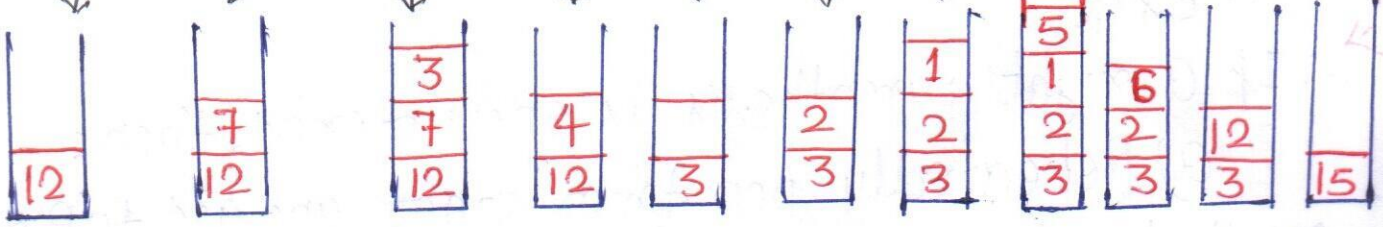
Soln: P: 5 (1) 6 (2) 2 (3) + (4) * (5) 12 (6) 4 (7) / (8) − (9)) (10)

I st Method.

SYMBOL SCANNED		STACK
(1)	5	5
(2)	6	5 6
(3)	2	5 6 2
(4)	+	5 8
(5)	*	40
(6)	12	40 12
(7)	4	40 12 4
(8)	/	40 3
(9)	−	37
(10)	)	37 ANS.

Ex. P: 12, 7, 3, −, /, 2, 1, 5, +, *, +

2nd Method.



TRANSFORMING INFIX EXPRESSION INTO POSTFIX EXP.

ALGORITHM- POSTFIX (Q, P): Suppose Q is an arithmetic expression written in infix notation. This algorithm finds the equivalent postfix expression P.

Step 1. Push "(" onto STACK, and add ")" to the end of Q.

Step 2. Scan Q from left to right and repeat steps 3 to 6 for each element of Q until the STACK is empty.

Step 3. IF an operand is encountered, add it to P.

step 4. IF a left paranthesis is encountered, push it onto STACK

step 5. IF an operator ⊗ is encountered, then -

a) Repeatedly pop from STACK and add to P each operator (on the top of STACK) which has the same precedence as or higher precedence than ⊗

Step 6 → b) Add ⊗ to STACK.

b) Remove the left parenthesis [Do not add the left Parenthesis to P]

[End of IF structure]

[End of step 2 loop]

Step 7. Exit

6. If a right parenthesis is encountered, then:

a) Repeatedly pop from STACK and add to P each operator (on the top of STACK) until a left parenthesis is encountered.

EX- Q: A+(B*C-(D/E↑F)*G)*H

Soln:

Q: A + (B * C — (D / E ↑ F) * G) * H)

(1) (2) (3) (4) (5) (6) (7) (8) (9) (10) (11) (12) (13) (14) (15) (16) (17) (18) (19) (20)

SYMBOL SCANNED	STACK	EXPRESSION: P
(1) A	(	A
(2) +	(+	A
(3) (	(+ (	A
(4) B	(+ (	A B
(5) *	(+ (*	A B
(6) C	(+ (*	A B C
(7) —	(+ (-	A B C *
(8) (	(+ (- (	A B C *
(9) D	(+ (- (	A B C * D
(10) /	(+ (- (/	A B C * D
(11) E	(+ (- (/	A B C * D E
(12) ↑	(+ (- (/ ↑	A B C * D E
(13) F	(+ (- (/ ↑	A B C * D E F
(14))	(+ (-	A B C * D E F ↑ /
(15) *	(+ (- *	A B C * D E F ↑ /
(16) G	(+ (- *	A B C * D E F ↑ / G
(17))	(+	A B C * D E F ↑ / G * -
(18) *	(+ *	A B C * D E F ↑ / G * -
(19) H	(+ *	A B C * D E F ↑ / G * - H
(20))	AMS→	A B C * D E F ↑ / G * - H * +

Q. Convert the given infix expression into postfix expression.

a) $(A-B) * (D / E)$

b) $(A+B \uparrow D) / (E-F) + F$

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

RECURSION:- The process in which a function calls itself directly or indirectly is called recursion and corresponding function is called a recursive function.

Syntax.

```
int fun()
{
...
fun();
}
```

Example.

a) FACTORIAL FUNCTION

b) FIBONACCI SEQUENCE

c) Ackermann FUNCTION

d) TOWERS OF HANOI

a) FACTORIAL :- Factorial of a non-negative integer is the multiplication of all positive integers smaller than or equal to n.

Ex. $n!_0 = n \times (n-1) \times (n-2) \times \cdots 1.$

$= 1 \times 2 \times 3 \times 4 \times \cdots (n-1) \times (n).$

$4!_0 = 1 \times 2 \times 3 \times 4 = 24$

$\boxed{0!_0 = 1}$

(a) if $n = 0$ then $n! = 1$

(b) if $n > 0$ then $n! = n \times (n-1)!$

Ex. $4! = 4 * 3!$

$3! = 3 * 2!$

$2! = 2 * 1!$

$1! = 1 * 0!$

Base Value. → $0! = 1$

$1! = 1 * 1 = 1$

$2! = 2 * 1 = 2$

$3! = 3 * 2 = 6$

$4! = 4 * 6 = 24$

Algorithm:- This procedure calculates $n!$ and returns the value in the variable FACT.

Step 1. IF N = 0 then FACT = 0 and Return

2. Call FACTORIAL (FACT, N-1)

3. Set FACT = N * FACT

4. Return.

```
#include <stdio.h>
int rec(int)
int main()
{
int a, fact;
printf("Enter any Number");
scanf("%d", &a);
fact = rec(a);
printf("Factorial value = %d", fact);
return 0;
}
int rec(int x)
{
int f;
if(x==1)
return 1;
else
f = x * rec(x-1);
return f;
}
```

a = 4
x = 4
4 == 1 F
f = 4 * rec(3)
x = 3
3 == 1 F
f = 3 * rec(2)
x = 2
2 == 1 F
f = 2 * rec(1)
x == 1
return 1

f = 2 × 1 = 2
f = 3 × 2 = 6
4 × 6 = 24

ITERATION:- Iteration means a set of intruction repeatedly executed.

Ex- for loop, while loop etc.

Note:- Recursion involves calling the same function again, and hence, has a very small length of code. But Howewer as we saw in the analysis the time complexity of recursion can get to be exponential.

⇒ Hence usage of recursion is advantageous in shorter code, but higher time complexity. (lesser number of recursive call)

⇒ Iteration involves larger size of code, but the time complexity is generally lesser than it is for recursion.

Fibonacci Sequence:- A series of numbers in which each number is the sum of the two preceding numbers.

a) if $n=0$ or $n=1$ then $Fn = n$

b) if $n > 1$ then $Fn = Fn-2 + Fn-1$

Ex- 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ------

ALGORITHM:- FIBONACCI (FIB, N) This procedure calculates FN and returns the value in the first parameter FIB

step1. IF N=0 or N=1 then Set FIB = N and Return

step2. Call FIBONACCI (FIBA, N-2)

Step3. Call FIBONACCI (FIBA, N-1)

step4. Set FIB = FIBA + FIBB.

step5. Return.

Tower of Hanoi →

Tower of Hanoi is a mathematical puzzle where we have three rods and n disks.

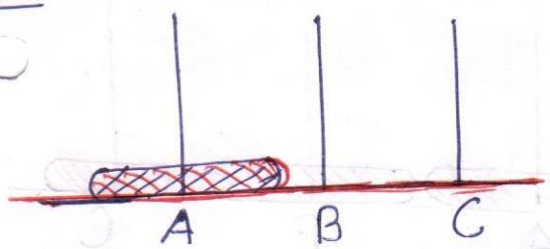
The objective of the puzzle is to move the entire disk to another rod. The rules of the game are as follows -

i) Only one disk may be moved at a time.

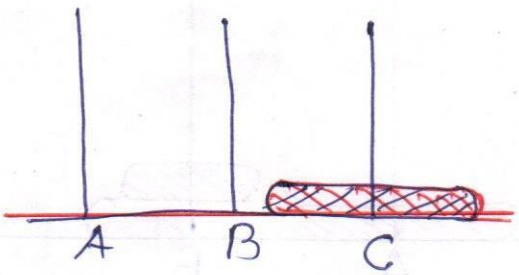
ii) Only the top disk on any peg may be moved to any other peg.

iii) No disk may be placed on top of a smaller disk.

Ex. 1. Take 1- disk. ($n = 1$).

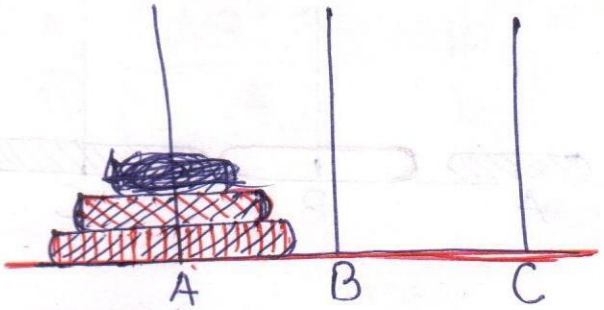


Move A → C

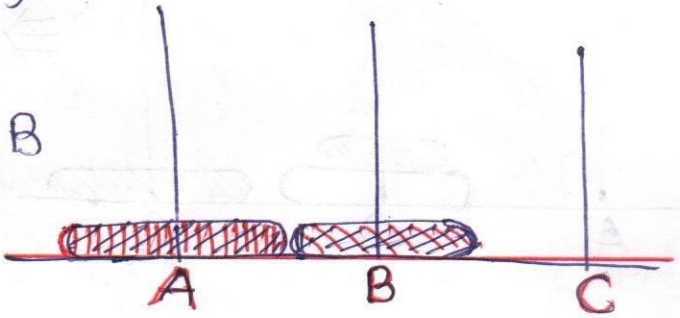


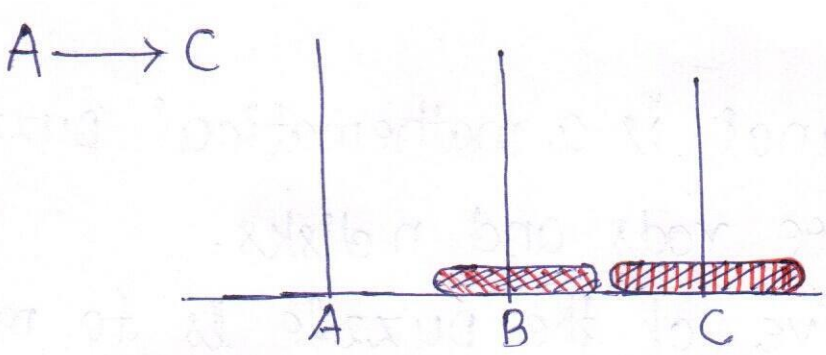
No of Move = $1 = 2^n - 1 = 2^1 - 1 = 2 - 1 = 1$.

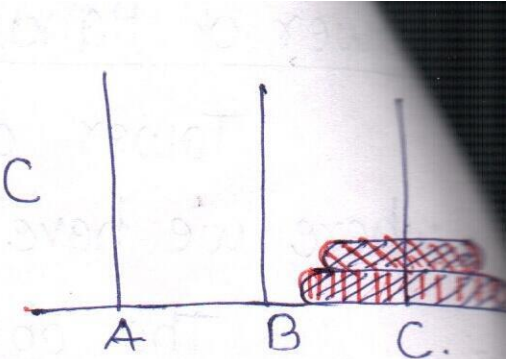
Ex. 2 Take 2- disk ($n = 2$)



A → B

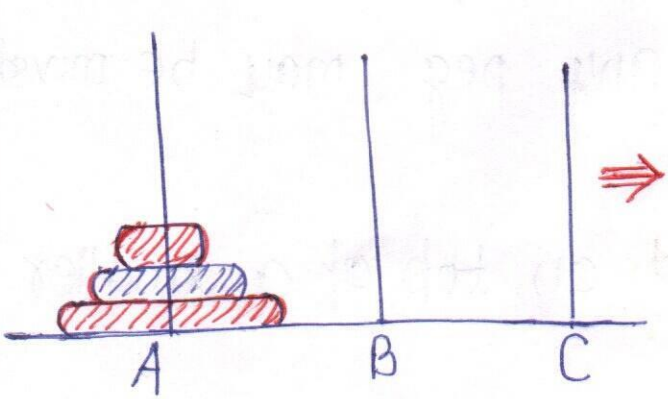




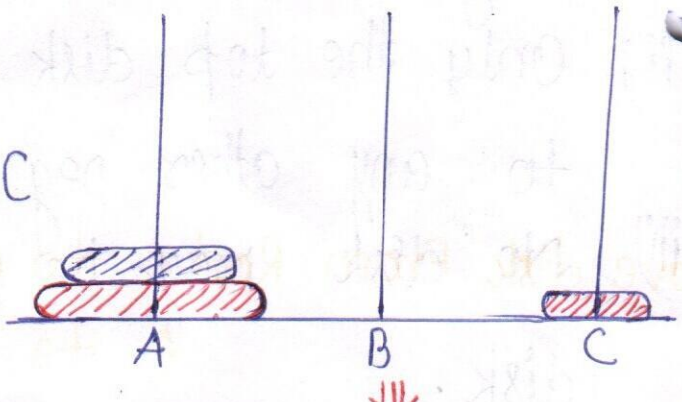


No of move = $3 = 2^n - 1 = 2^2 - 1 = 4 - 1 = 3$

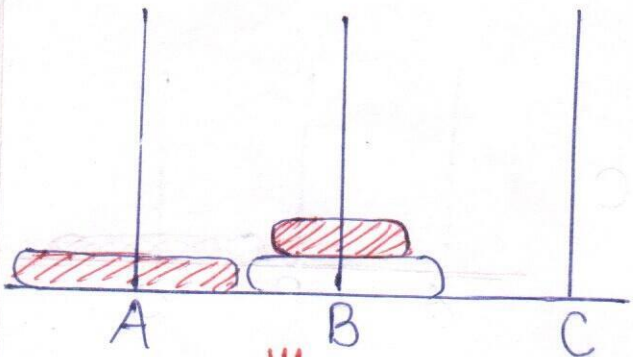
Ex- Take 3-disk:-



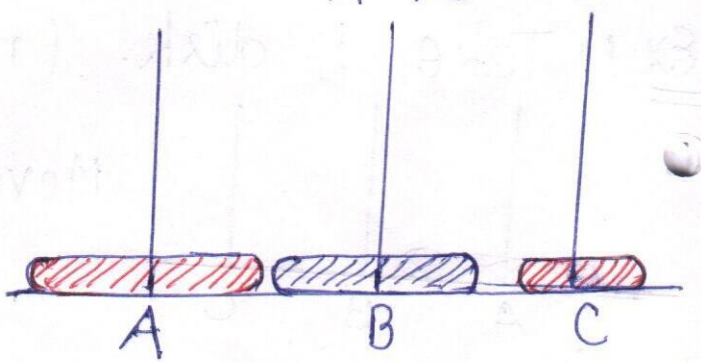
⇒ A → C



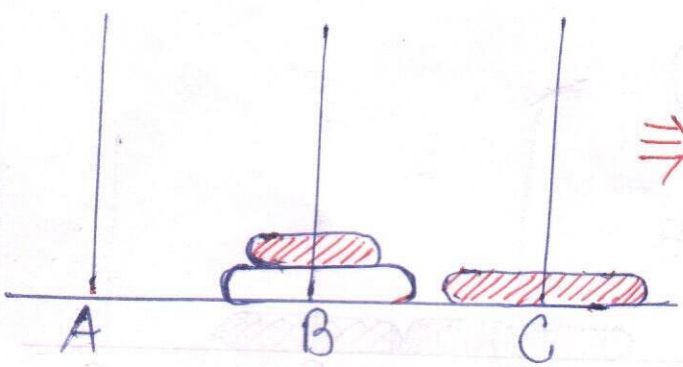
A → B



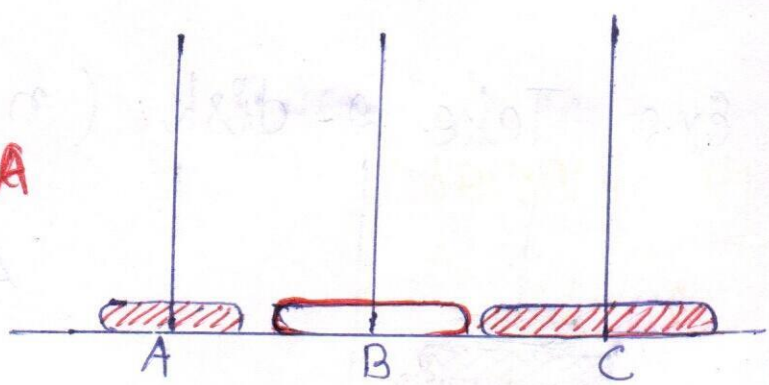
C → B



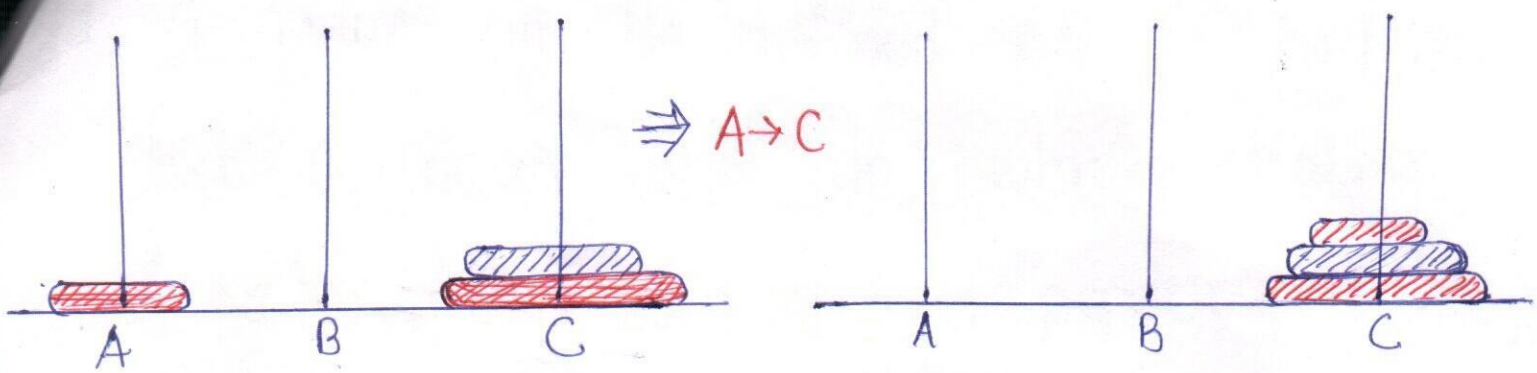
A → C



⇒ B → A



B → C



No of Moves = $7 = 2^n - 1 = 2^3 - 1 = 8 - 1 = 7$

Example- No of disk = 4

A[4, 3, 2, 1] B[ ] C[ ]

1. Move disk From Rod A to Rod B: A[4,3,2] B[1] C[ ]
2. A to C : A[4,3] B[1] C[2]
3. B to C : A[4,3] B[ ] C[2,1]
4. A to B : A[4] B[3] C[2,1]
5. C to A : A[4,1] B[3] C[2]
6. C to B : A[4,1] B[3,2] C[ ]
7. A to B : A[4] B[3,2,1] C[ ]
8. A to C : A[ ] B[3,2,1] C[4]
9. B to C : A[ ] B[3,2] C[4,1]
10. B to A : A[2] B[3] C[4,1]
11. C to A : A[2,1] B[3] C[4]
12. B to C : A[2,1] B[ ] C[4,3]
13. A to B : A[2] B[1] C[4,3]
14. A to C : A[ ] B[1] C[4,3,2]

15. B to C

A[ ] B[ ] C[4,3,2,1]

No of Moves:

$2^n - 1 = 2^4 - 1$

$= 16 - 1$

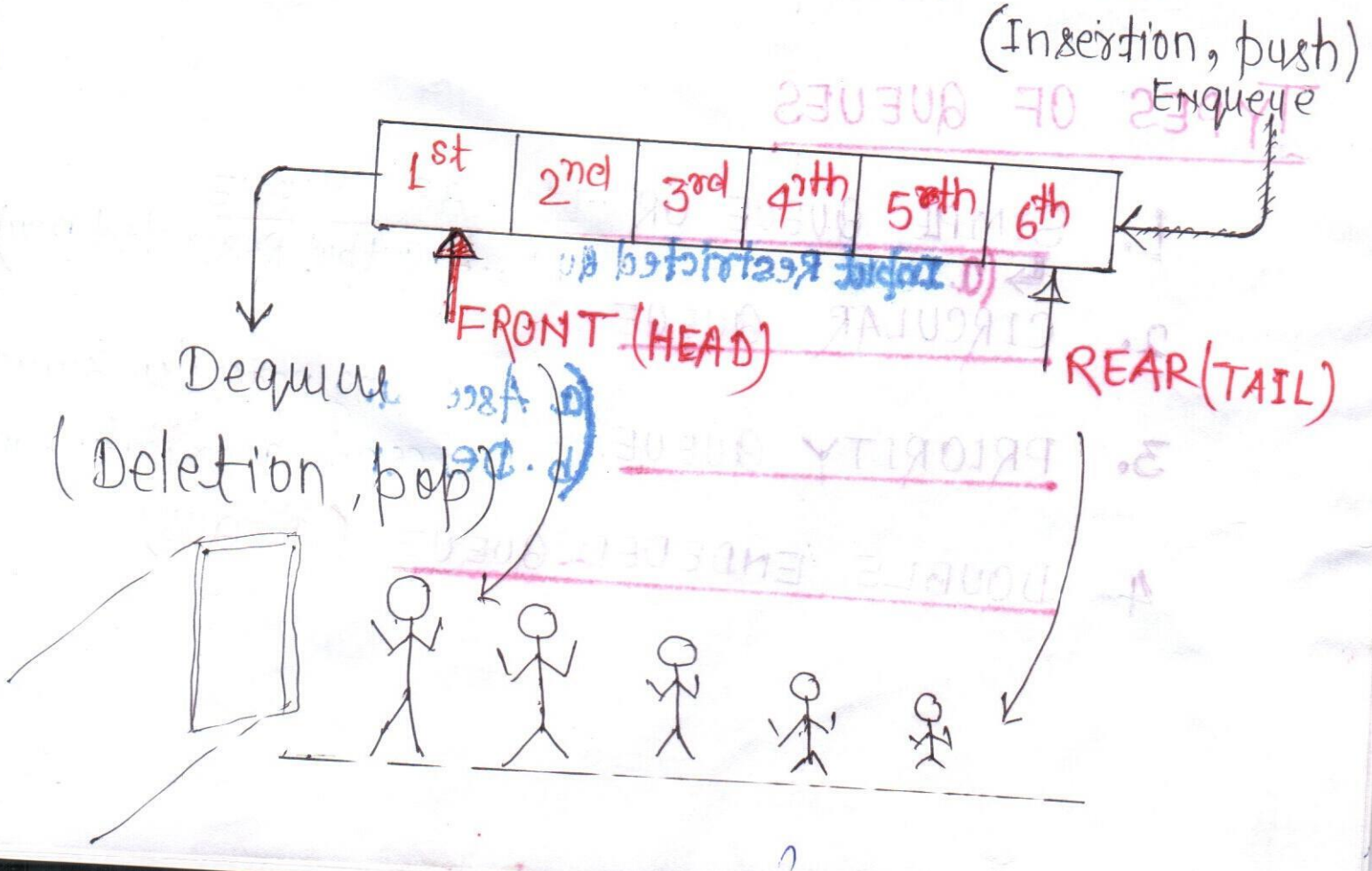
$= 15$ Ans

QUEUES

1- A queue can be defined as an ordered list (linear collection) which enables insert operations to be performed at one end called REAR and delete operation to be performed at another end called FRONT.

2. Queue worked on the principle of
— FIRST IN FIRST OUT (FIFO)

3- Queue is the linear Data Structure.



Application of Queue

1. Used as waiting list for a single shared resource like – printer, Disk, CPU.
2. Used in synchronous transfer of data. (pipes, file, sockets)
3. Used as buffer (MP3 Player, CD player)
4. Used to maintain the play list in media players.
5. Used in operating system for handling interrupts.

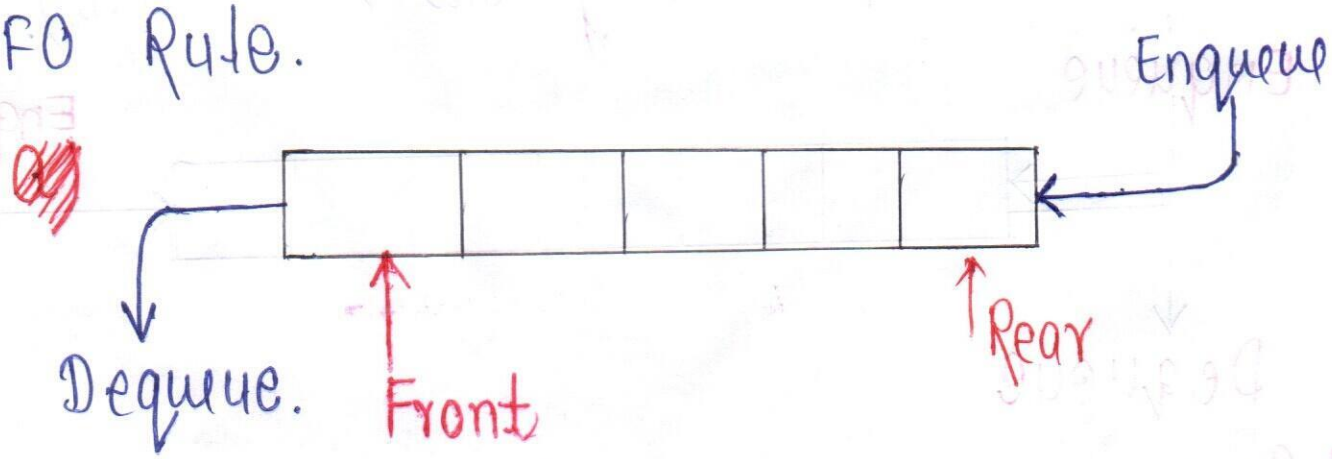
TYPES OF QUEUES

1. SIMPLE QUEUE OR LINEAR QUEUE
   - a. Input Restricted Queue, b. Output Restricted queue)
2. CIRCULAR QUEUE
3. PRIORITY QUEUE
   - a. Ascending priority Queue
   - b. Descending priority Queue
4. DOUBLE ENDED QUEUE (DEQUE)

TYPES OF OPERATIONS

1. Insertion → EnQueue()
2. Deletion → DeQueue()
3. Peek() → Used to get front element without removing it.

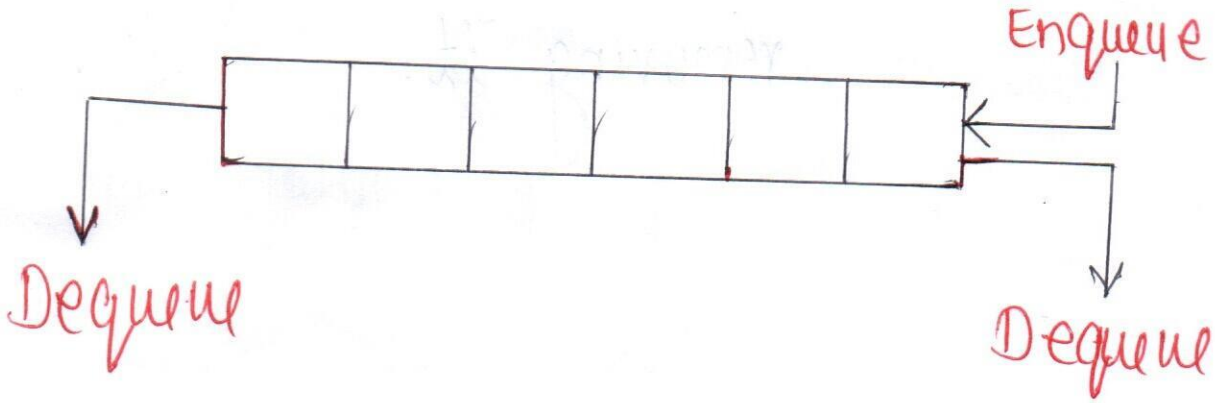
1. Simple Queue:- In this queue an insertion take place from one end (rear) while the deletion occurs from other end (front). It follow the FIFO Rule.



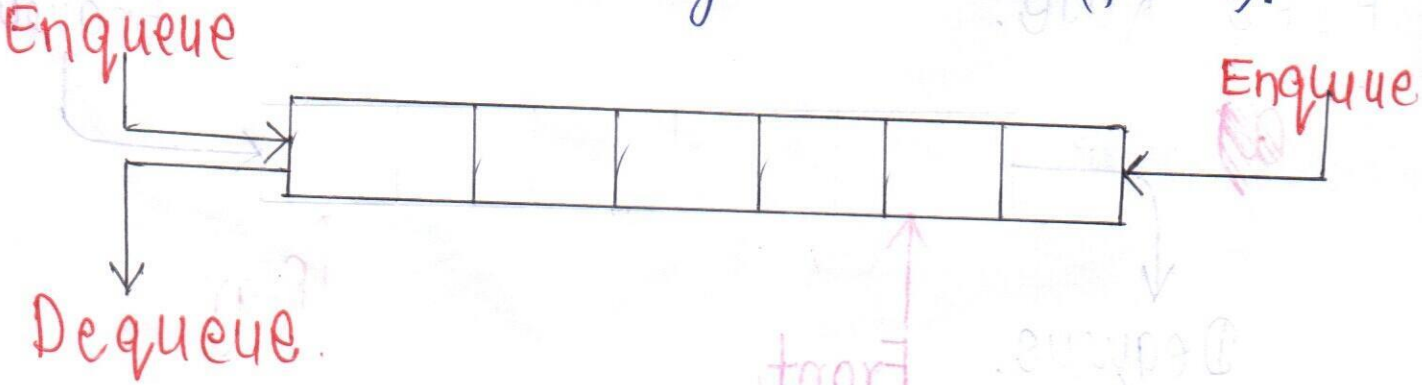
a) Input Restricted Queue:- In this type

of queue, the input can be taken from side (rear) only, and deletion of elements can be done from both sides (from & rear).

* This kind of queue does not follow FIFO

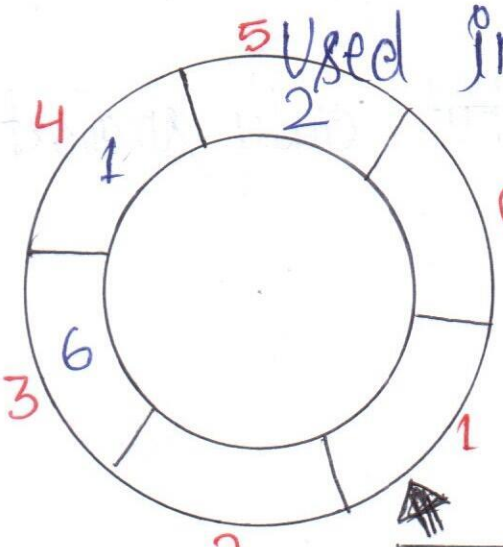


b) Output Restricted Queue. In this type of queue, the input can be taken from both sides (rear and front) and the deletion of elements can be done from only one side (front).

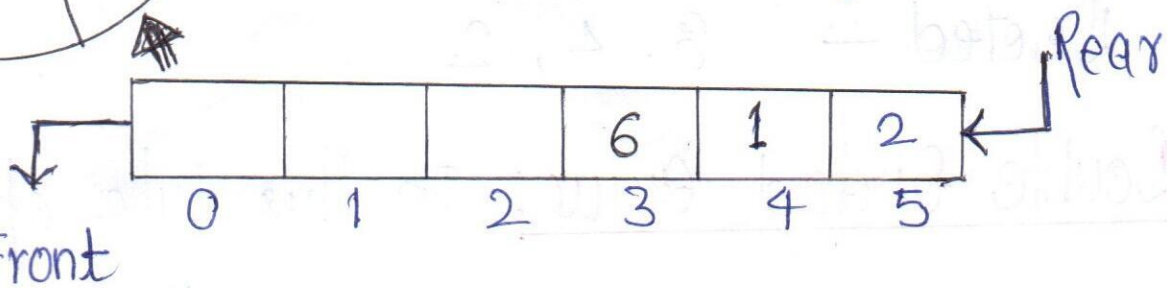


2.) CIRCULAR QUEUE :- In this type of queue, the

last position connected to back to the first position to make a circle. It is also called "RING BUFFER". It follow the principle of FIFO.



Used in — Memory Management
— Traffic System.
— CPU Scheduling.



3) PRIORITY QUEUE:- Is a special type of queue in which each element is associated with a priority and is served according to its priority.

a) Ascending Priority Queue:- Element can be inserted in any order but only smallest

element can be removed.

Ex. Inserted → 4, 2, 8

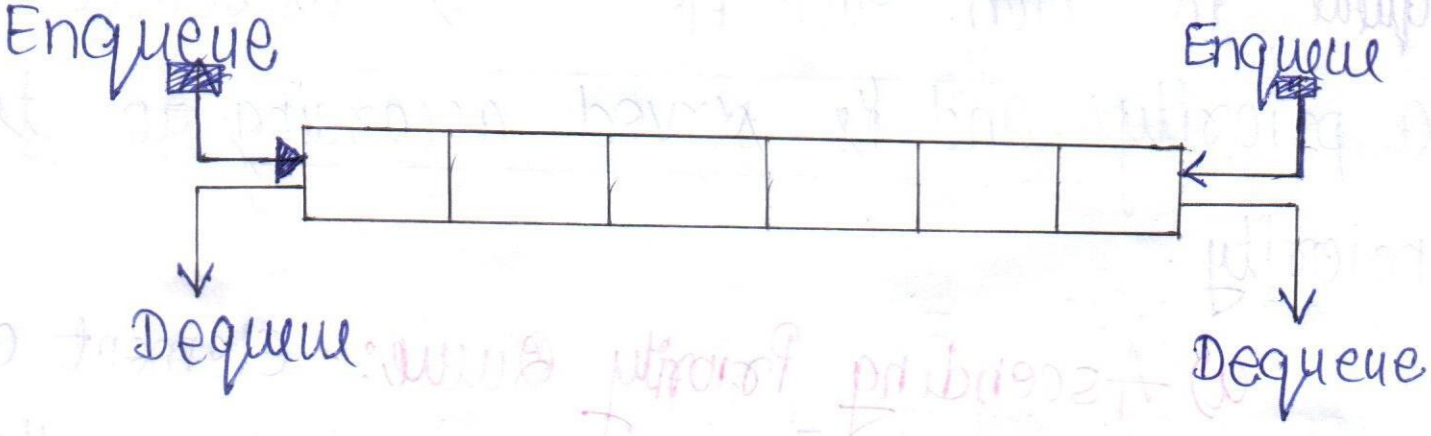
Deleted → 2, 4, 8.

b) Descending priority Order:- Element can be inserted in any order but only smallest element can be removed.

Ex. Inserted → 4, 2, 8

Deleted → 8, 4, 2

4) Double Ended Queue:- In this type of queue Insertion and deletion performed at both the end (front & rear).

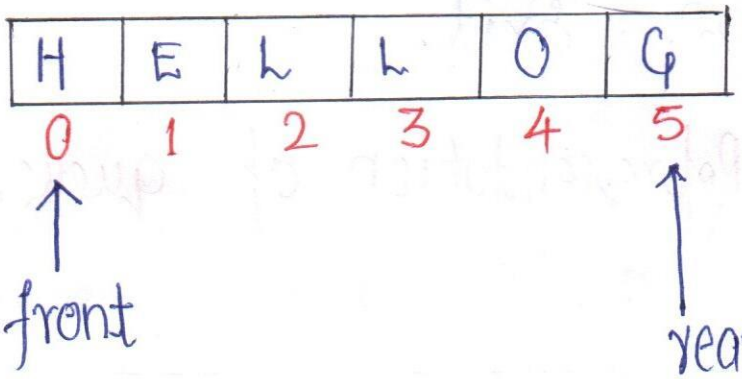


REPRESENTATION OF QUEUE :- The queue is represented in memory by two ways.

a) Array representation of queue.

b) Linked list representation of queue.

(a) Array Representation of queue :-



(i) INSERT OPERATION

Algorithm.

step1. IF rear = MAX - 1

WRITE : OVERFLOW.

step2. IF front = -1 and rear = -1.

Set front = rear = 0.

else

Set rear = rear + 1

Step 3. Set queue_arr [rear] = data;

step4. exit.

ii) Delete operation:.

step1. IF front = -1 or front > rear

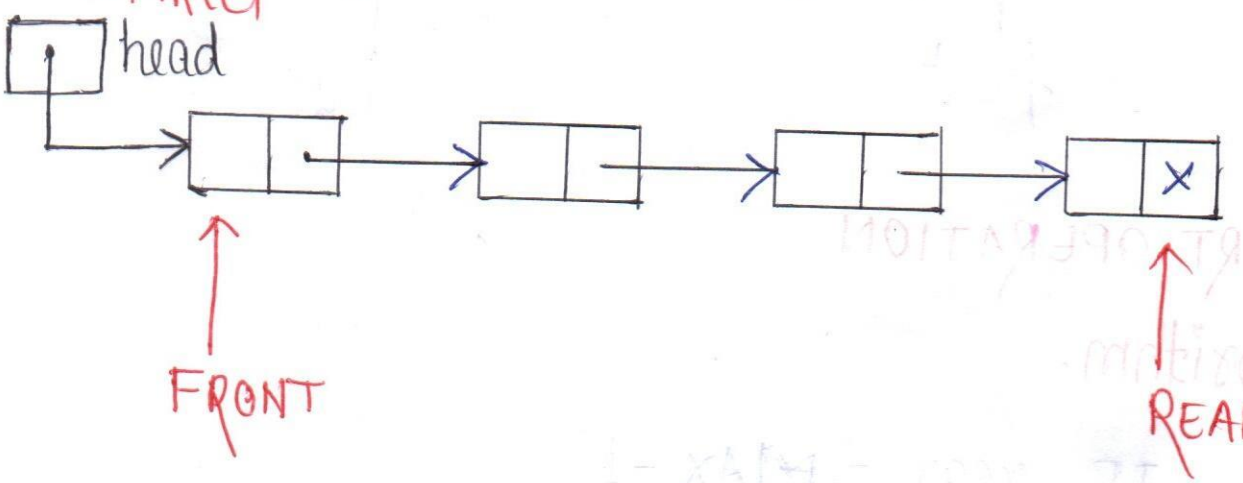
Write UNDERFLOW.

Set val = Queue_arr [front].

Set front = front + 1.

Step2. Exit.

b) ~~Queue~~ Linked Representation of queue.



i) INSERT OPERATION:-

Algorithm:.

step1. Allocate memory for the new node and name it as PTR.

Step2. SET PTR → DATA = VAL.

Step3. ~~SET~~ IF FRONT = NULL

SET FRONT = REAR = PTR

SET FRONT ⇒ NEXT = REAR → NEXT = NULL

ELSE

SET REAR → NEXT = PTR

SET REAR → PTR

SET REAR → NEXT = NULL

(END OF IF)

Step 4. END.

(b) Delete operation.

Algorithm. Step 1. IF FRONT = NULL

PRINT UNDERFLOW

Step 2. SET PTR = FRONT

Step 3. SET FRONT = FRONT → NEXT

Step 4. FREE PTR.

Step 5. Exit.

Program to Array Implementation of qu

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 5
 void int queue_arr [MAX];
int front = -1;
int rear = -1;
void insert();
void delete();
void display();
void main()
   {
    int choice;
    while (choice != 4)
      {
     printf("\n **** Main Menu ****\n");
     printf("\n Enter 1 for INSERT AN
                    ELEMENT \n ");
     printf("\n Enter 2 for DELETE AN
          ELEMENT \n");
```

```
printf("\n Enter 3 for DISPLAY QUEUE
                \n");
printf("\n Enter 4 for EXIT \n");
scanf("%d", &choice);
switch(choice)
  {
  case 1: insert();
          break;
  case 2: delete();
          break;
  case 3: display();
          break;
  case 4: exit(0);
          break;
  default: printf(" Enter valid choice\n");
  }
 }
}
```

```
void insert()
    {
     int item;
     printf("\n Enter the element \n");
     scanf("\n %d", &item);
     if (rear == MAX-1)
     {
      printf(" OVERFLOW");
      return;
      }
     if (front==-1 && rear==-1)
     {
      front = 0;
      rear = 0;
      }
     else
        {
         rear = rear+1;
        }
     queue_arr[rear] = item;
     printf("\n Value Inserted");
}
```

```
void delete()
{
int item;
if (front == -1 || front > rear)
    printf("\n UNDERFLOW");
else
    {
    item = queue_arr[front];
    if (front == rear)
        {
        front = -1;
        rear = -1;
        }
    else
        front = front + 1;
    }
}=
void display()
{
int i;
if (rear == -1)
printf("\n Empty queue");
```

```
else
{
    printf("\n The value of queue are:\n");
    for(i=0; i<=rear; i++)
    {
        printf("\n%d\n", queue_arr[i]);
    }
}
```