STACK

it may be possible that there is a need of data structure which takes operations on only one end i.e beginning or end of the list.

Stack and queue are data structure which fulfil the requirements to take operations on only at one end.

Stack:

A stack is a linear data structure where insertion and deletion of items takes place at one end called top of the stack.

A stack is defined as a data structure which operates on a Last-in-first-out basis. So it is also referred as Last-in-first-out (LIFO).

example:

a stack of dishes, a stack of folded towels etc.

Operations on stack

Push:

Insertion of an element into the top of the stack is push.

Top:

Top is a pointer which indicates the top element of the stack.

when ever we want to insert the element into the stack we need to check whether the Stack is full or not.

Overflow : (Is full)

checks for stack is full.

for pushing the element into stack, first we will check the condition of overflow. then push the element as:

```
if (top == MAXSTK - 1);
        Printf ("Stack overflow \n");
else
    {
      top = top + 1;
      Stack [top] = item ;
    }
```

underflow (Is empty)

Pop operation

Deletes an element from the stack.

To delete the element from the stack we need to check whether there is an element in the stack to be deleted or not. This condition is known as "Underflow" condition.

underflow:

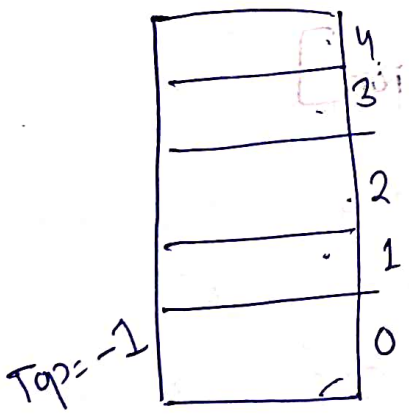
```
if (top == -1)
  Printf ("stack underflow \n"),
else
{ Printf ("popped element is : %d \n"
                    stack[top]);
  Top= Top-1
}
```

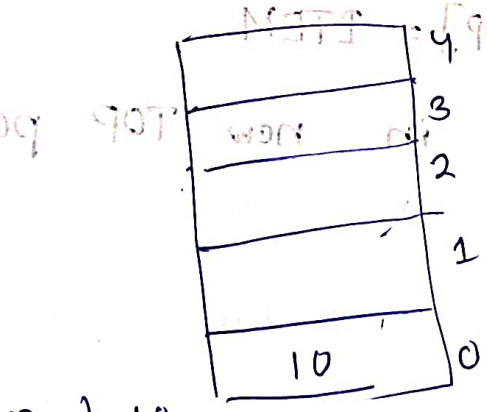
Memory Representation of Stack

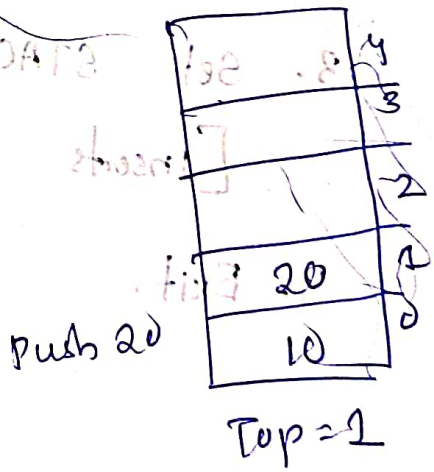
Since stack is a collection of same type of elements, So we can take array for implementing stack.

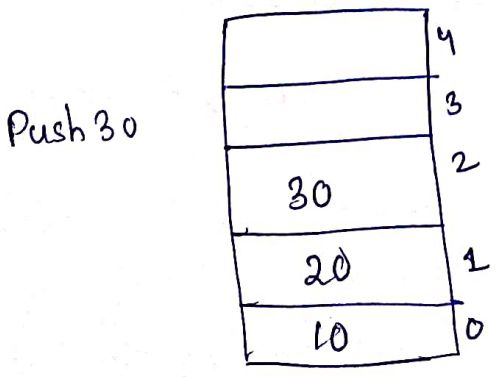
There are two ways of represent the stack in memory.

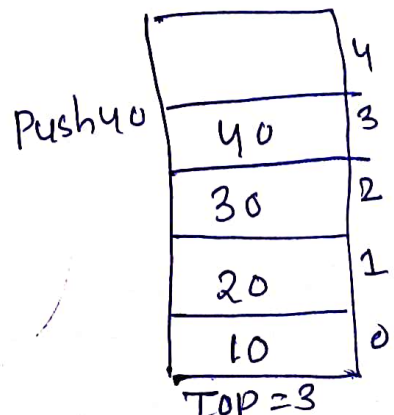
1. Using array.
2. using linked list

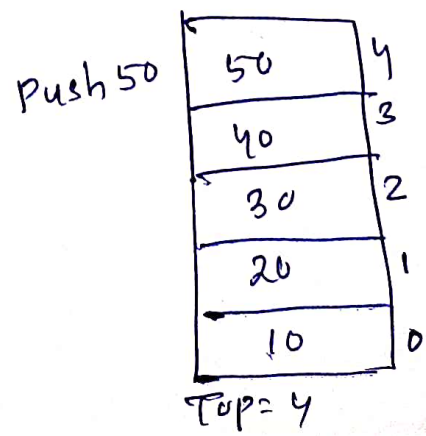












Push 60

Top = MAXSTK - 1

Overflow.

Algorithm for PUSH

PUSH (STACK, TOP, MAXSTK, ITEM)

This procedure pushes an ITEM onto a stack.

1. [Stack already full]

   if TOP = MAXSTK - 1,

   then Print : OVERFLOW,

   and

   Return.

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

3. Set STACK[TOP] = ITEM

   [Inserts ITEM in new TOP position]

4. Exit.

```
if (top == MAXSTK - 1)
    Printf ("Stack overflow \n");

else
  {
    top = top + 1;
    stack[top] = item;
  }
```

Algorithm for POP

POP (STACK, TOP, ITEM)

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

1. [Stack has an item to be removed ?]
   if top = -1, then
      Print "Underflow"
   and
      Return.
2. Set ITEM = STACK[TOP]
   [Assign TOP element to ITEM]
3. Set TOP = TOP - 1.
   [Decrease TOP by 1]
4. Return.

```
if (top == -1)
   Printf (" Stack underflow \n" );
else
 {
  Printf (" Popped element is %d \n", stack[top]);
  top = top - 1;
 }
```

display()

```
   if (top == -1)
       Printf(" Stack is empty \n");
   else
    {
     Printf (" Stack elements : \n" );
       for (i = top ; i >= 0; i--)
          Printf ("%d \n", stackarr[i]);
    }
```

^ , *

* / %

+ —

Applications Of Stack :

- Reversal Of a String.
- Checking validity of an expression containing nested Parentheses.
- Conversion of infix expression to postfix and Prefix forms.
- Evaluation of postfix and prefix forms.

Reversal Of String:

We can reverse a string by pushing each character of the string on the stack. when the whole string is Pushed on the stack we will pop the characters from the stack and we will get the reversed string.

```
#include
```

Conversion & Evaluation of Arithmetic Expressions

Arithmetic expression notations:

In any arithmetic expression, each operator is placed in between two operands of it. (i.e mathematical representation.

The arithmetic expression can be represented as:-

Infix notation:

The operator is placed between the operands is called infix notation.

A+B C-D E*F G/H

(A+B)*C

Prefix notation :- (Polish notation)

The notation in which the operator Symbol is placed before its operands, is referred as Polish or prefix notation.

example - +AB.

Postfix notation (Reverse polish notation)

The notation in which the operator symbol is placed after its operands, is referred as reverse polish or Postfix notation.

example:-

AB+

Mathematical Procedure for conversion

The possible conversions are:-

1. Infix to Prefix.
2. Prefix to infix.
3. Infix to Postfix.
4. Postfix to Infix.
5. Prefix to postfix.
6. Postfix to Prefix

Standard arithmetic Operators and Precedence Levels:

^ (exponential)	Higher level
*, /, %	Middle level
+, −	Lower level.

Infix to Prefix

(A+B * (C−D ^ E) / F

= A+B * (C − ^ DE) / F

= A + B * (− C ^ D E) / F

= A + * B − C ^ D E / F

= A + / * B − C ^ D E F

= + A / * B − C ^ D E F

Prefix to Infix

Identify the operator from right to left.

$$+ A / * B - C \wedge D E F$$

$$= + A / * B - [C \quad D \wedge E] F$$

$$= + A / * B [C - D \wedge E] F$$

$$= + A / B * (C - D \wedge E) / F$$

$$= + A \quad B * (C - D \wedge E) / F$$

$$= A + B * C - D \wedge E / F$$

→ Identify the operator from right to left order.
→ The two operands which immediately follows the operator are for evaluation.
→ Represent the operator and operands in infix notation.
→ continue this process until the equivalent infix expression is achieved.

Infix to Postfix:

→ Identify the innermost brackets.
→ Identify the operator according to the priority of evaluation.
→ Represent the operator and corresponding operator in Postfix notation.
→ continue this process until the equivalent postfix expression is achieved.

example:

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

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

= (A + B * [C D E ^ -] / F)

= A + B * C D E ^ - F /

= A + B C D E ^ - * F /

= A + B C D E ^ - * F /

= A + B C D E ^ - * F /

= A B C D E ^ - * F / +

Postfix to Infix

→ Identify the operator from left to right order.

→ The two operands which immediately preceeds the operator are for evaluation.

→ Represent the operator and operands in infix notation.

→ Continue this process until the equivalent infix expression is achieved.

→ example:

A B C D E ^ - * F / +

= A B C [D ^ E] - * F / +

= A B C - [D ^ E] * F / +

= ~~A [B * (C - [D ^ E) *~~

= A [B * C - (D ^ E)] F / +

= A [B * C - (D ^ E)] / F +

= A + B * C - D ^ E / F

Prefix to Postfix

$+ A / * B - C \wedge D E F$

$= + A / * B - C [D E \wedge] F$

$= + A / * B [C D E \wedge -] F$

$= + A / [B C D E \wedge - *] F$

$= + A B C D E \wedge - * F /$

$= A B C D E \wedge - * F / +$

→ Identify the operator from right to left order.
→ The two operands which immediately follows the operator are for evaluation.
→ Represent the operator and operands in postfix notation.
→ continue this process until the equivalent postfix expression is achieved.

Postfix to Prefix

A B C D E ^

Importance of postfix Expression

The processor represents the mathematical expression in postfix expression and uses it for evaluation. To do this it implements the concept of stack.

Evaluation Of Postfix Expression

Consider the following expression P written in postfix notation.

P : 5, 6, 2, +, *, 12, 4, /, −

convert it into infix expression and find the value of postfix notation.

Symbol Scanned	STACK
5	5
6	5, 6
2	5, 6, 2.
+	5, 8
*	40
12	40, 12
4	40, 12, 4
/	40, 3
−	37

5 6 2 + * 12 4 / −

~~5 (6+2)~~

~~= 5 6 2 + * 12 4~~

= 5 (6 + 2) * 12 4 / −

= 5 * 8 12 4 / −

= 5 * 8 3 / −

= 40 − 3 = 37.

Consider the following arithmetic expression P, written in postfix notation.

P : 12, 7, 3, −, /, 2, 1, 5, +, *, +

(1) convert it into its equivalent infix expression.

P = 12 [7 − 3] / 2, 1, 5, + * +

= [12 / (7 − 3)] 2, 1, 5, +, *, +

= [12 / (7 − 3)] 2, [1 + 5] * +

= [12 / (7 − 3)], [2 * (1 + 5)], +

= 12 / (7 − 3) + 2 * (1 + 5)

(2)

P.	Stack
12	12
7	12, 7
3	12, 7, 3
−	12, 4
/	3
2	3, 2
1	3, 2, 1
5	3, 2, 1, 5
+	3, 2, 6
*	3, 12
+	15

Transforming Infix Expressions into Postfix Expressions.

example:

P:- A + B C * D E F ↑ / G * − H * +

Symbol Scanned	STACK.	Expression P
A		A B C D E F ↑ * + G / * H * − +
B		
C		
*		
D		
E		
F		
↑		
/		
G		
*		
−		
H		
*		
+		
)		

evaluation of postfix expression.

4, 5, 4, 2, ^, +, *, 2, 2,

^ 9, 3, /, *, −

Ans - 72.

Converting infix expression into postfix expression

Scan from left to right.

operaters.

$$\wedge \quad + \quad - \quad * \quad /$$

$$A * (B + C \wedge D) - E \wedge F * (G / H)$$

Ans -

$$A \; B \; C \; D \; \wedge \; + \; * \; E \; F \; \wedge \; G \; H \; / \; * \; -$$

Q:- A + (B * C - (D / E ↑ F) * G) * H

Symbol Scanned	STACK	Expression P
A	(	A
+	(+	A
(	(+(	A
B	(+(	AB
*	(+(*	AB
C	(+(*	ABC
-	(+(-	ABC*
(	(+(-(	ABC*
D	(+(-(	ABC*D
/	(+(-(/	ABC*D
E	(+(-(/	ABC*DE
↑	(+(-(/↑	ABC*DE
F	(+(-(/↑	ABC*DEF
)	(+(-	ABC*DEF↑/
*	(+(-*	ABC*DEF↑/
G	(+(-*	ABC*DEF↑/G
)	(+	ABC*DEF↑/G*-
*	(+*	ABC*DEF↑/G*-
H	(+*	ABC*DEF↑/G*-H
)		ABC*DEF↑/G*-H*+

Algorithm:

Evaluation of a postfix expression:

Suppose P is an arithmetic expression written in Postfix notation.

This algorithm which uses a STACK to hold operands and evaluates P.

Algorithm:

This algorithm finds the VALUE of an arithmetic expression P written in Postfix notation.

1. Add a right parenthesis ")" at the end of P.
2. Scan P from left to right and repeat Steps 3 and 4 for each element of P until ")" is encountered.
3. if an operand is encountered, put it on STACK.
4. if an operator (⊗) is encountered, then:

   (a) Remove the two elements of STACK, where A is the top element 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].
5. Set VALUE equal to the top element on STACK.
6. Exit.

Transforming Infix expressions into Postfix Expressions.

Let Q be an arithmetic expression written in infix notation.

This algorithm transforms the expression Q into its equivalent postfix expression P.. The algorithm uses a STACK to temporarily hold operators and left parentheses. The postfix expression P will be constructed from left to right using the operands from Q and the operators which are removed from STACK.

Algorithm:

POLISH (Q, P)

Suppose Q is an arithmetic expression written in infix notation. This algorithm findsthe equivalent postfix expression P.

1. Push "(" onto STACK, and add ")" to the end of Q.
2. Scan Q from left to right, and repeat Steps 3 to 6 for each element of Q until the STACK is empty:
3. if an operand is encountred, add it to P.
4. if a left parenthesis is encountered, push it onto STACK.
5. if an operator ⊗ is encountered, then:
   (a) Repeatedly pop from STACK and add it to P each operator (on the top of STACK) which has the Same precedence as or higher precedance than ⊗
   (b) Add ⊗ to STACK.

   [End of if structure].
6. If a right parenthesis is encountered, then:
   (a) Repeatedly pop from STACK and add it to P each operator (on the top of STACK) until left Parenthesis is encountered.
   (b) Remove the left parenthesis [Do not add left parenthesis to P].

[End of if structure]

[End of step-2 loop]

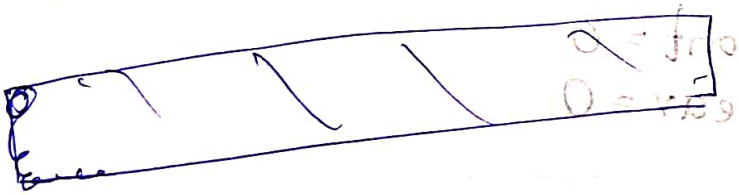
7. Exit.

QUEUES

→ Queue is a linear list of elements in which deletion can take place only at one end called the front, and insertion can take place only at the other end called the rear.

→ Queues are also called first-in-first-out (FIFO) lists. Since the first element in a queue will be the first element out of the queue. In other words, the order in which elements enter a queue is the order in which they leave.

→ example -
Queue of people,
Queue of cars etc.



Representation of Queue

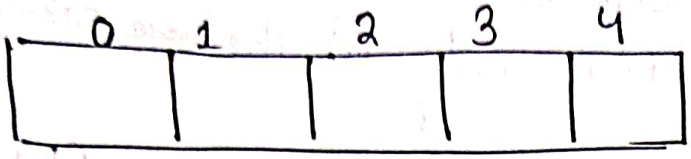
Queues may be represented in the computer in various ways, usually by means of one-way list or linear arrays.

QUEUES will be maintained by a linear array QUEUE and two pointer variables:

FRONT: containing the location of the front element of the queue.

REAR: containing the location of the rear element of the queue.

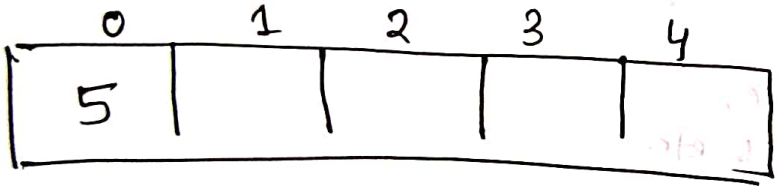
The condition (FRONT = NULL) will indicate that the Queue is EMPTY.



front = -1 or front = NULL
rear = -1

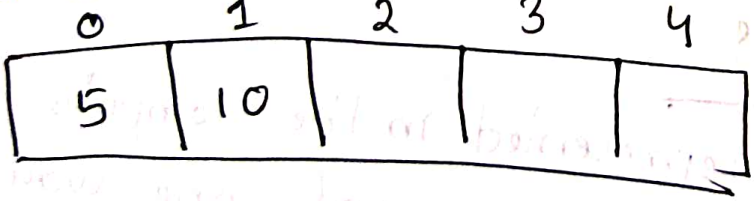
(Empty Queue)

Add 5 to Queue.



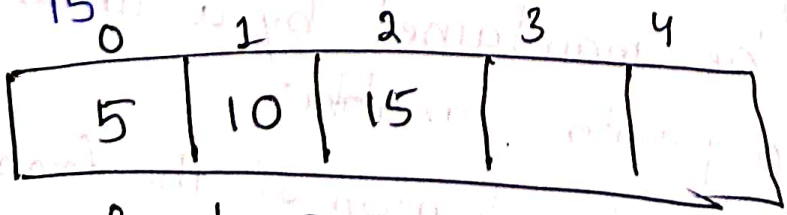
~~forant~~ = front = 0
rear = 0

Add 10



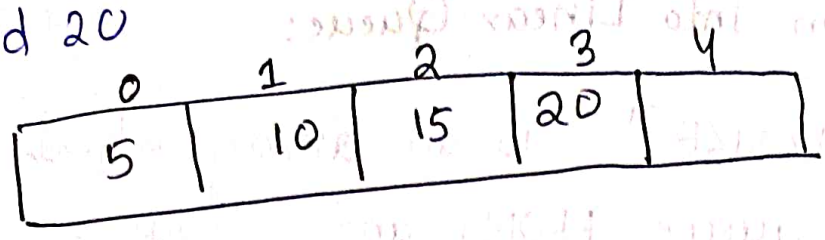
front = 0
rear = 1

Add 15



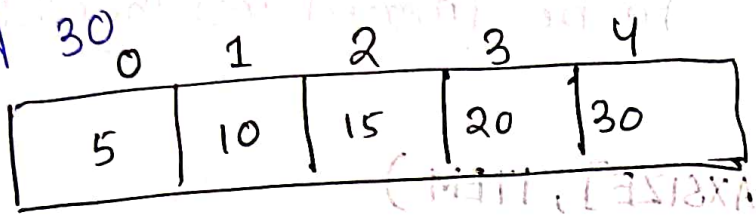
front = 0
rear = 2

Add 20



front = 0
rear = 3

Add 30



front = 0
rear = 4

when element is added rear is increased by 1.

REAR = REAR + 1

Add 40 Overflow" can not be added. Queue

rear = MAXSIZE - 1

Overflow

Algorithm for insertion into Linear Queue:

Assume QUEUE [MAXSIZE] is an array which represents a linear queue. FRONT and REAR identifies the index of FRONT and REAR respectively. ITEM is the element to be inserted into the QUEUE.

QINSERT (QUEUE[MAXSIZE], ITEM)

1. if REAR = MAXSIZE - 1,
   then.
   Print "Overflow"
   Exit.

2. Else if FRONT = -1 or REAR = -1
   then Set: FRONT = 0
   REAR = 0

3. Else
   REAR = REAR + 1.
   [End of IF].

4. QUEUE [REAR] = ITEM

5. Exit.

```
void qinsert(int Q[ ], int item
{
if (rear == MAXSIZE - 1)
 {
     Printf ("In Queue Overflow"
     exit(0);
 }
else if (front== -1 || rear
                    == -1)
   {
     front=0;
     rear=0;
   }
   else
     { rear = rear+1;
     Q[rear] = item
```

Algorithm for Deletion in a Linear Queue:

Assume QUEUE[MAXSIZE] is an array which represents a queue,
FRONT, and REAR identifies the index of front and rear end respectively.
ITEM holds the element to be deleted from the QUEUE]

QDELETE (QUEUE [MAXSIZE])

1. Let ITEM := element deleted from the Queue.
2. if FRONT = -1 OR REAR = -1, // front > rear
   Then [Queue has no elements to be deleted.
   Display "Underflow (or) Queue Empty"
   Exit.
   [End of if]
3. ITEM = QUEUE[FRONT]
4. Print ITEM (deleted element)
5. Else if FRONT = REAR [then QUEUE contains only one element]

   set FRONT = -1
   and REAR = -1

6. Else

   FRONT = FRONT + 1.

7. Exit.

```
void Qdeletion(int Q[])
{
  int item;
  if (front == -1 || rear == -1)
  {
    Printf("\n Queue Underflow");
    exit(0);
  }
  item = Q[front];
  if (front == rear) //front = 0
  {                  //rear = 0
    front = -1;
    rear = -1;
  }
  else
  {
    front = front + 1;
  }
  Printf("\n deleted item is %d", item);
}
```

```
display()
{
  int i;
  if (front == -1 || rear == -1)
      Printf("Queue is empty\n");
  else
  {
      Printf("Queue is :\n");
      for (i = front; i <= rear; i++)
          Printf("%d", queue[i]);
  }
}
```

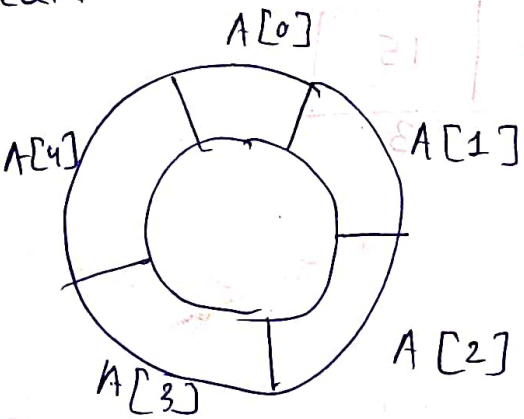
CIRCULAR QUEUE

it is a queue of circular nature, in which it is possible to insert a new element at the first location, when the last location of the queue is occupied.

In other words, if we have an array Q[MAXSIZE] representing a circular queue, then after inserting an element at (MAXSIZE - 1)th location of the array, the next element will be inserted at the very first location (i.e location with index 0)

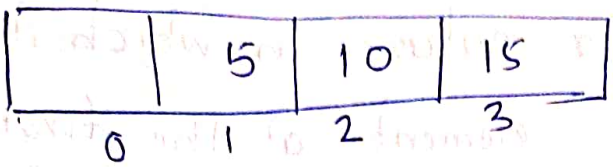
Circular Queue Representation

Let an array A[5] is maintained in the form of circular queue.



Here after n-1th element 0th element occurs.

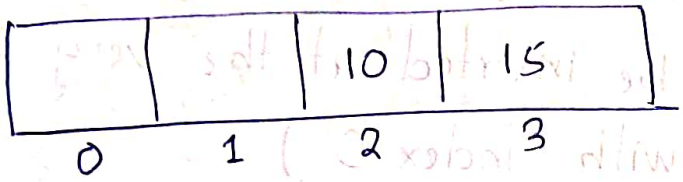
Initial queue



front = 1
rear = 3

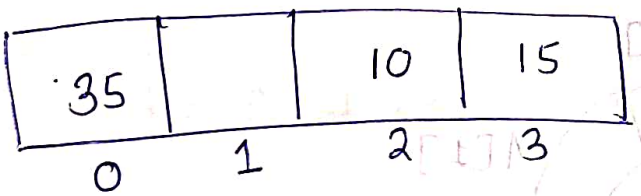
~~deletion in queue~~

Delete from queue



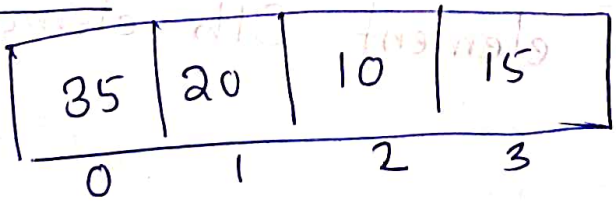
front = 2
rear = 3

Add 35



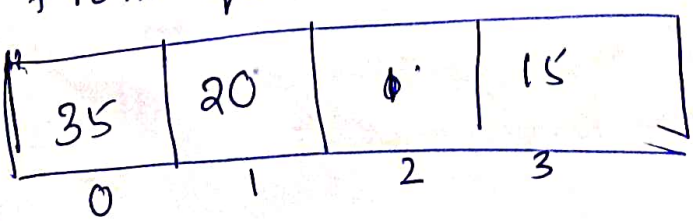
front = 2
rear = 0

Add 20



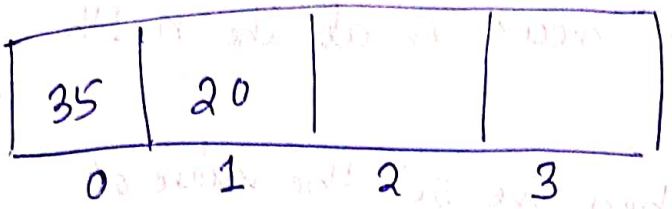
front = 2
rear = 1

Delete from queue



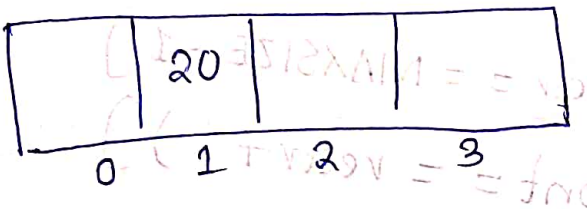
front = 3
rear = 1

Delete from queue



front = 0
rear = 1

Delete from queue.



front = 1
rear = 1

Delete from queue

front = -1
rear = -1

[Queue is empty]

front = 0
rear = Max - 1

Overflow.

front = -1
rear = -1

underflow.

Add Operation in Circular Queue :

We check that the rear is at the n-1th Position.

If rear = N-1, then we set the value of rear to 0. and add the element at the 0th position of the array. otherwise element will be added same as in Simple queue.

Overflow condition

```
if(front == 0 && rear == MAXSIZE -1 )
                || (front == rear+1) )

{
  Printf("Queue Overflow \n");
  return;
}
```

if Queue is initially empty then we set the value 0 to front and rear and then add the element in queue otherwise we increase the value of rear only and then element will be added in queue.

```
if (front == -1) (//queue initially empty)
{
  front = 0;
  rear = 0;
}
```

```
else
{
    if (rear == MAXSIZE -1)   // rear is at last
                              Position of queue
        rear = 0;
    else
        rear = rear + 1;

    cqueue [rear] = item;
}
```

Deletion Operation in Circular queue

We check that the front is at the N-1th Position. if front = N-1, then delete the element from the queue and set the value of front to 0.

```
if (front == Maxsize - 1)
    front = 0;
```

if there is only one element in queue then we delete the element from queue and set the value of front and rear to -1.

```
if (front == rear)
{
    front = -1;
    rear = -1;
}
```

Algorithm for Circular Queue insertion

Assume queue[MAXSIZE] is an array representing a circular queue.

FRONT and REAR identifies FRONT and REAR element of circular queue.

ITEM is the new element to be inserted.

A QINSERT(QUEUE[MAXSIZE], ITEM)

1. if (FRONT = 0 and REAR = MAXSIZE - 1) or (FRONT = REAR + 1) then

   Print "OVERFLOW"

   exit.

2. Else if FRONT == -1 or REAR == -1
   then set FRONT = 0
   REAR = 0

3. Else if ~~REAR = REAR + 1~~
   ~~(End of IF)~~ REAR = MAXSIZE - 1
   set REAR = 0
   Else set REAR = REAR + 1
   [End of IF]

4. Set QUEUE[REAR] = ITEM.

5. Exit

Algorithm for deletion in Circular Queue:

Assume queue[MAXSIZE] is an array representing a circular queue.
FRONT and REAR identifies FRONT and REAR element of the Circular queue.
ITEM is the element to be deleted.

CDELETE (QUEUE, FRONT, REAR)

1. Let ITEM
2. If FRONT = -1 or REAR = -1
   then Print "queue is empty or Underflow"
   Exit
   [End of IF
3. ITEM = QUEUE [FRONT]
4. Display ITEM
5. IF FRONT = REAR
   Set FRONT = -1
   REAR = -1
6. Else
   if FRONT = MAXSIZE - 1
   Set FRONT = 0
   else
   Set FRONT = FRONT + 1.
7. End.

DOUBLE ENDED QUEUE (DEQUE)

Deque or Double ended queue we can add or delete the element from both side.

Types of Deque:

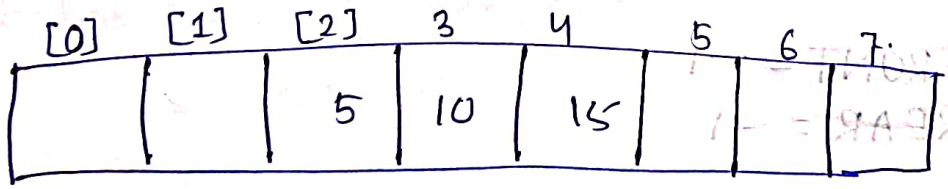
1. Input restricted
2. Output restricted.

In Input restricted dequeue, element can be added at only one end but we can delete the element from both side.

In Output restricted dequeue, element can be added from both side. but deletion is allowed only at one end.

Array Implementation Of Dequeue:

dequeue can be implemented using Array.
Two pointers LEFT and RIGHT indicate the left and right position of the dequeue.



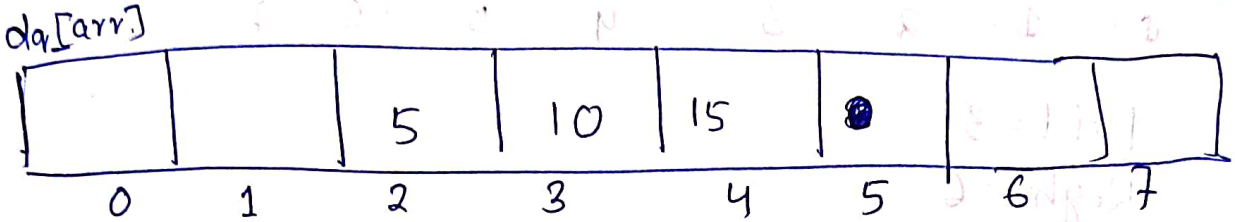
~~left~~ LEFT = 2
RIGHT = 4

we will assume this is circular array for queue operation.

Add and delete Operation in Dequeue:

Input restricted dequeue.

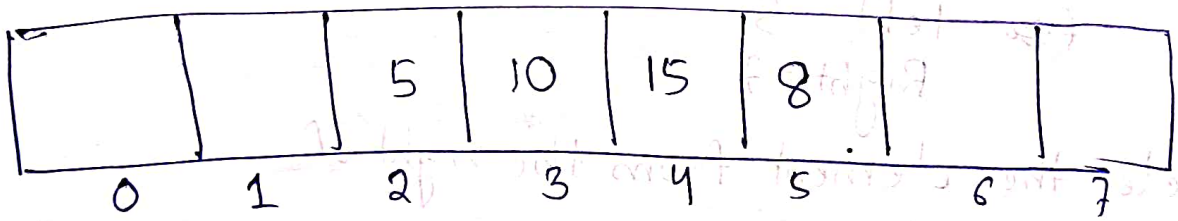
We can add element only on the right side of queue but we can delete from both sides.



Initial.

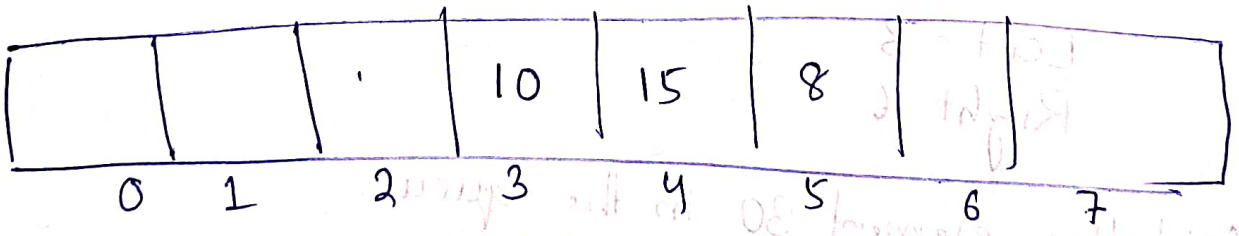
Left = 2
Right = 4

1. Add element 8 in the queue.



Left = 2
Right = 5

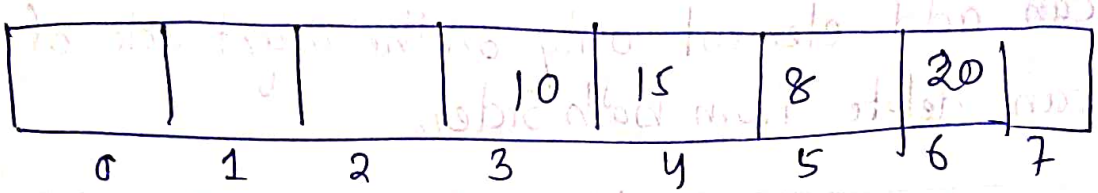
2. Delete the element from the left of the queue.



5 will be deleted.

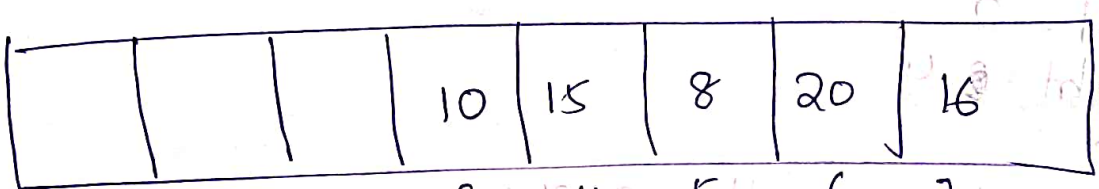
Left = 3
Right = 5

3. Add the element 20 in the queue.



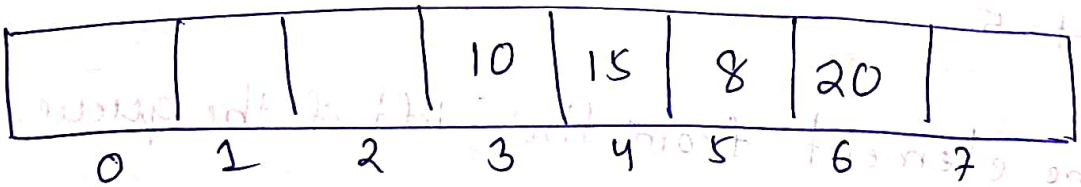
left = 3
Right = 6

4. Add 16 in the queue



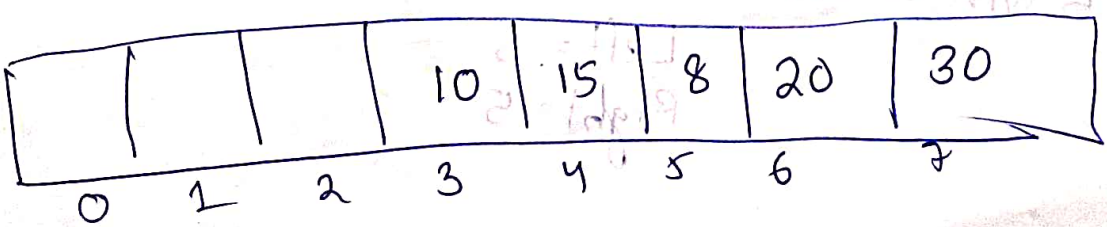
left = 3
Right = 7

5. Delete the element from the right of the queue.



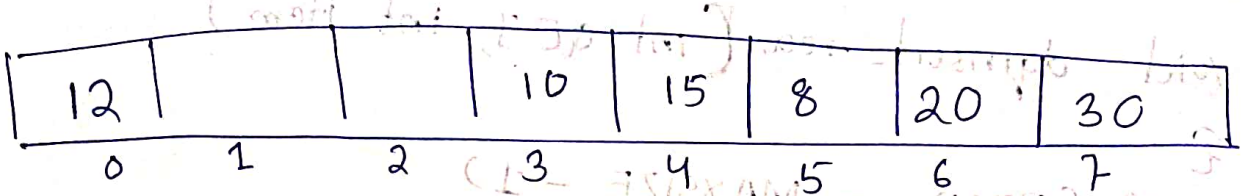
Left = 3
Right = 6

6. Add the element 30 in the queue.



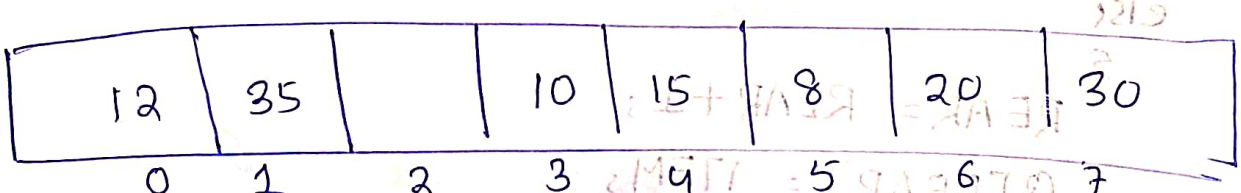
Left = 3
Right = 7

7. Add the element 12 in the queue.



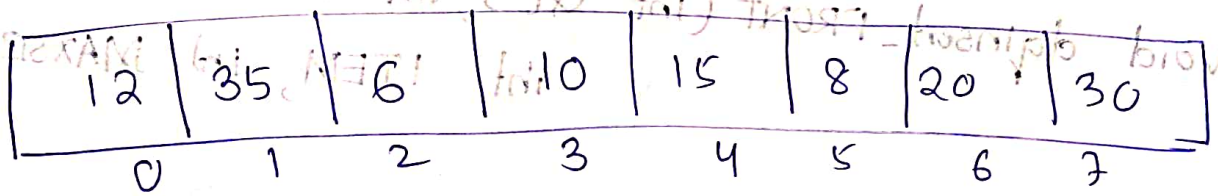
Right = 0
Left = 3

8. Add 35 in the queue.



Left = 3
Right = 1

9. Add 6 in the queue.



Left = 3
Right = 2

10. Add 45.

"OVERFLOW"

Insertion at the rear end of deque.

```
void dqinsert_rear (int Q[ ], int item)
{
  if (REAR == MAXSIZE - 1)
  {
    Printf("Dequeue is FULL");
  }
  else
  {
    REAR= REAR+1;
    Q[REAR] = ITEM;
  }
}
```

Function for Insertion at front end of dequeue

```
void dqinsert_FRONT (int Q[ ], int FRONT, int REAR
                     int ITEM, int MAXSIZE)
{
  if (FRONT == 0)
  {
    Printf ("Dequeue is FULL");
  }
  else
  {
    FRONT= FRONT -1;
    Q[FRONT] = ITEM;
  }
}
```

c function for deletion at front end of dequeue

```
void dqdelete_FRONT(int Q[ ])
{
  int ITEM;
  if (FRONT == -1)
  {
    Print ("Dequeue is empty");
  }
  else if (FRONT == REAR)
    {
      ITEM = Q[FRONT];
      FRONT = REAR = -1;
      Printf ("deleted item is %d", ITEM);
    }
  else
    {
      ITEM = Q[FRONT];
      FRONT = FRONT + 1;
      Printf ("Deleted item is %d", ITEM);
    }
}
```

'C' function for deletion at rear end of deque.

```
void dqdelete_REAR(int Q[])
{
  int ITEM;
  if (FRONT == -1 || REAR == -1)
    {
      Printf(" Dequeue is empty ");
      exit(0);
    }
  ITEM = Q[REAR];
  if(FRONT == REAR)
    {
      FRONT= REAR= -1;
    }
  else
    {
      REAR= REAR-1;
    }
  Printf (" Deleted Item is %d", ITEM );
}
```

Priority Queue :

As we know queue is an ordered list of elements in which we can add the elements only at one end called the rear of the queue. and delete the element only at the other end called front of the queue.

But in priority queue every element of queue has some priority and based on that priority it will be processed. So the element which has more Priority will be processed before the element which has less priority.

Suppose two elements have same priority then in this case FIFO rule will follow, means the element which comes first in the queue will be Processed first.

In computer implementation, priority queue is used in the CPU scheduling algorithm