

## Chapter 3 :- Stack & Queue

---

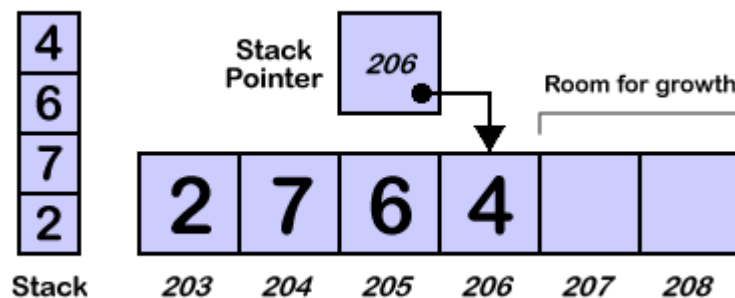
### STACK

A Stack is data structure in which elements are added and removed from one end, Last in first out structure (LIFO).

In this list insertion and deletion are made at one end, called the top of stack.

For Example, if your favorite green shirts are underneath of old blue one in the stack of shirts, if you want to wear this green shirt today, you must remove the first blue shirt from the stack, only then you can take the desired green shirt, which will now be on the top of the stack.

Most microprocessors use stack-based architecture. Whenever a member function is called, it returns an address and argument pushed into a stack, and when the function returns they are popped off, the stack operation is built into the microprocessor.



### STACK PUSH

In PUSH operation we can add an element from top of the stack, so before push operation user must check the stack should not be empty.

If a stack is full and when we try to push (insert) an element, then a stack overflow error occurs. It is called 'stack overflow' condition.

#### Algorithm: PUSH(S, TOP X)

Step 1: [Check for stack is overflow]

    If  $Top \geq N$

        Then write ("Stack is over flow")

    Exit

Step 2: [Increment pointer by value one]

$Top \leftarrow Top + 1$

Step 3: [Perform Insertion]

$S[\text{Top}] \leftarrow X$

Step 4: [Finished]

Exit

The first step of the algorithm checks for an overflow condition, if stack is full means Top pointer value reach at size of stack, then insertion cannot be performed.

In second & third step, if it is not full a top pointer value increment by one and insert a value to top pointer element.

## **STACK POP**

In POP operation we can delete or remove an element from top of the stack, so before pop operation user must check the stack should not be empty.

If a stack is empty and we can try to pop (delete) an element, then a stack under flow error occurs. It is called 'stack under flow' condition.

**Algorithm: POP (S, Top)**

Step 1: [Check for under flow or check whether the stack is empty]

    If Top=0

        Then write (“Stack is under flow”)

    Exit

Step 2: [Decrement pointer/remove the top information]

    Value <- S [top]

    Top <- Top-1

Step 3: [Return former top element of the stack]

    Write (value)

Step 4: [Finished]

    Exit

## **POLISH NOTATIONS**

The process of writing the operators of an expression either before their operands or after operands are called the POLISH NOTATION. The polish notation was discovered by mathematician named ‘Jan LuksieWciz.

The polish notations are classified into three categories:

1. Infix
2. Prefix
3. Postfix

When the operators exist between two operands then the expression is called Infix Expression.

When the operators are written before their operands then the resulting expression is called Prefix Expressions.

When operators come after their operands the resulting expression is called Postfix Expression.



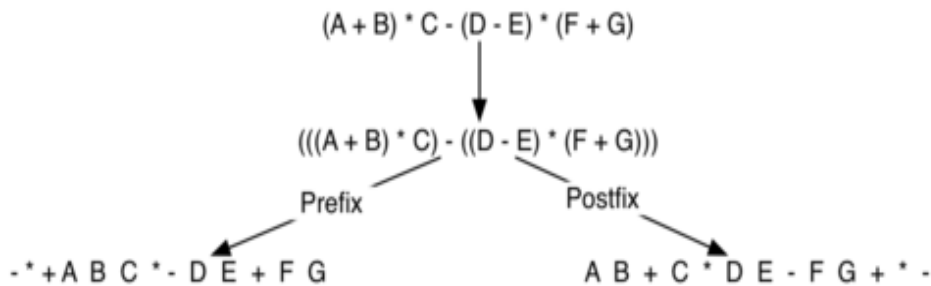
Moving Operators to the Right for Postfix Notation.

Infix Expression	Prefix Expression	Postfix Expression
$A+B*C+D$	$++A*BCD$	$AB\ C*+D+$
$(A+B)*(C+D)$	$*+AB+CD$	$AB\ +CD+*$
$A*B+C*D$	$+*AB*CD$	$AB\ *CD*+$
$A+B+C+D$	$+++ABCD$	$AB\ +C+D+$

### Rules for converting Infix notation to the Postfix/Prefix notation:-

1. The operation with highest precedence are converted first and then remaining portion of the expression is to be converted to postfix.
2. It is to be treated as single operand.
3. We consider five Binary operation, such are (+) addition, subtraction (-), multiplication (\*), division (/) and exponential(^).
4. Take only two operands at a time to convert in a postfix form like  $A + B$  to  $AB+$
5. Always, convert the parenthesis first.
6. Convert postfix exponential second if there are more than one exponential in sequence then take order from right to left.
7. Convert into postfix as multiplication and division operator, left to right.
8. Convert in postfix as addition and subtraction left to right.

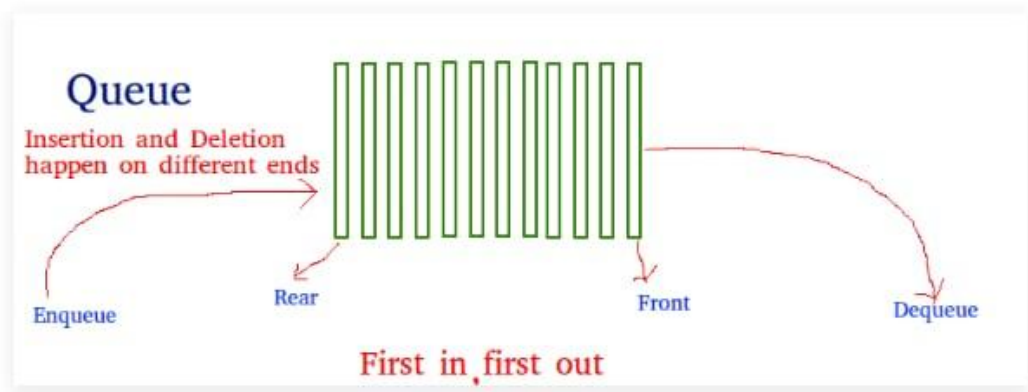
□ Converting Complex Expression :-



**Converting into Postfix expression:-**

- $A+B-C$   
 $= AB+-C$   
 $= AB+C-$
- $(A+B)*(C-D)$   
 $= (A + B)(C - D)*$   
 $= (A+B)CD-*$   
 $= AB+CD-*$
- $((A+B) * C - (D-E)) \$ (F+G)$   
 $= ((AB+) * C - (DE - ) ) \$ (FG+)$   
 $= (AB+C*DE--)$FG+$   
 $= AB+C*DE--FG+$$
- $A-B/(C*D\$E)$   
 $= A-B/(C*DE\$)$   
 $= A - B/ (CDE\$*)$   
 $= A - BCDE\$*/$   
 $= ABCDE\$*/ -$

**QUEUE** – a queue is a linear structure which follows a particular order in which the operations are performed. The order is first in first out(FIFO). A good example of queue is any queue of consumers for a resources where the consumer that came first is served first. The difference between stacks and queue is in removing. In a stack we remove the item added; in a queue ,we remove the item the least recently added.



here are basic operations for queues are insertions and deletions

For writing algorithm we can define following variables

S -> it is the array having an element

Rear -> it is the pointer which will points to last end of queue

Front -> it is the pointer which will points to first end of queue

X -> it is the variable which will store the value

**AIM: Write an algorithm and C program to perform Push in Queue.**

**(A)PUSH Operation**

**In push operations , if we push or insert an element in a queue at rear end rear pointer increment by 1.**

**Algorithm:**

Step1: [Check for queue Over flow]

If  $\text{rear} \geq \text{size}$

then

write("Queue is over flow")

Exit

Step2: [Increment Rear Pointer]

$\text{rear} = \text{rear} + 1$  Step3: [Insert an element at rear of Queue]

$S[\text{rear}] = x$

Step4: If  $\text{front} = -1$

then

$\text{front} = 0$

Step5: Exit.

**PROGRAM:**

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

```
#define MAX 50
```

```
void insert();
```

```
void delete();
```

```
void display();
```

```
int queue_array[MAX];
```

```

int rear = - 1;
int front = - 1;
main()
{
int choice;
while (1)
{
printf("1.Insert element to queue \n");
printf("2.Delete element from queue \n");
printf("3.Display all elements of queue \n");
printf("4.Quit \n");
printf("Enter your choice : ");
scanf("%d", &choice);
switch (choice)
{
case 1:
insert();
break;
case 2:
delete();
break;
case 3:
display();
break;
case 4:
exit(1);
default:
printf("Wrong choice \n");
}
}
}

void insert()
{
int add_item;
if (rear == MAX - 1)
printf("Queue Overflow \n");
else
{

```

```

if (front == - 1)
/*If queue is initially empty */
front = 0;
printf("Inset the element in queue : ");
scanf("%d", &add_item);
rear = rear + 1;
queue_array[rear] = add_item;
}
}

void delete()
{
if (front == - 1 || front > rear)
{
printf("Queue Underflow \n");
return ;
}
else
{
printf("Element deleted from queue is : %d\n", queue_array[front]);
front = front + 1;
}

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

```

## **OUTPUT**

### **PUSH:**



```

1.Insert element to queue
2.Delete element from queue
3.Display all elements of queue
4.Quit
Enter your choice : 1
Inset the element in queue : 35
1.Insert element to queue
2.Delete element from queue
3.Display all elements of queue
4.Quit
Enter your choice : 1
Inset the element in queue : 23
1.Insert element to queue
2.Delete element from queue
3.Display all elements of queue
4.Quit
Enter your choice : 3
Queue is :
35 23
1.Insert element to queue
2.Delete element from queue
3.Display all elements of queue
4.Quit
Enter your choice : _

```

## QUEUE POP

In pop operation, if we delete or pop a element in queue at front end, front pointer value increment by one.

here are basic operations for queues are insertions and deletions

For writing algorithm we can define following variables

S -> it is the array having an element

Rear -> it is the pointer which will points to last end of queue

Front -> it is the pointer which will points to first end of queue

X -> it is the variable which will store the value

**Algorithm: pop (s, front, rear x)**

Step 1: [check for Queue 'underflow']

```
    If (front = 0)

    then

    write ("Queue is under flow")

    Exit

Step 2: [Remove an element from Queue]

    X    S[front]

Step 3: [Print the popped element]

    Write("X")

Step 4: [Check for empty queue]

    If front = rear

    then

        front    0

        rear    0

    else

        front    front+1

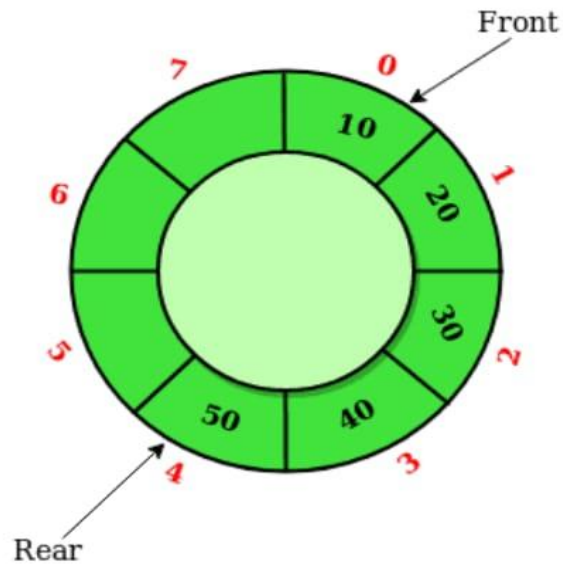
Step 5: [Finished]

    Exit
```

The first step in algorithm is to check the under flow condition, if queue is empty(front=0) and if we want to delete an element from a queue, it will indicate an 'under flow' errors.

In fourth step, if queue is not empty and we want to delete an element from a queue, so front pointer value will increment by one.

**Circular Queue** Circular Queue is a linear data structure in which the operations are performed based on FIFO (First In First Out) principle and the last position is connected back to the first position to make a circle. It is also called 'Ring Buffer'.



Front: Get the front item from queue.

Rear: Get the last item from queue.

Value: This function is used to insert an element into the circular queue. In a circular queue, the new element is always inserted at Rear position

**AIM :- Write an algorithm and C program to perform Push and Pop operation in circular queue.**

**(A)PUSH Operation**

**Algorithm:**

Step1: [Check for circular Queue 'Over flow']

if front=1 and rear=n

then

write("circular queue is over flow")

Exit

Step2: [Insert elements in the circular Queue]

Else if front=0

Front=1

rear=1

S[rear]=x

Step3: Else if rear=n

Rear=1

S[rear]=x

Step4: [Insert the element in circular queue]

Else

rear=rear+1

S[rear]=x

Step5: Exit

**PROGRAM:**

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

```
#define size 5
```

```
void insertq(int[], int);
```

```
void deleteq(int[]);
```

```
void display(int[]);
```

```
int front = - 1;

int rear = - 1;

int main()

{

    int n, ch;

    int queue[size];

    do

    {

printf("\n\n Circular Queue:\n1. Insert \n2. Delete\n3. Display\n0. Exit");

printf("\nEnter Choice 0-3? : ");

scanf("%d", &ch);

        switch (ch)

        {

            case 1:

printf("\nEnter number: ");

scanf("%d", &n);

insertq(queue, n);

                break;

            case 2:

deleteq(queue);

                break;

            case 3:

                display(queue);
```

```
        break;

    }

}while (ch != 0);

}

void insertq(int queue[], int item)

{

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

    {

printf("queue is full");

        return;

    }

    else if (rear == - 1)

    {

        rear++;

        front++;

    }

    else if (rear == size - 1 && front > 0)

    {

        rear = 0;

    }

    else

    {

        rear++;
```

```
    }

    queue[rear] = item;}

void display(int queue[])

{

    int i;

    printf("\n");

    if (front > rear)

    {

        for (i = front; i < size; i++)

        {

            printf("%d ", queue[i]);

        }

        for (i = 0; i <= rear; i++)

            printf("%d ", queue[i]);

    }

    else

    {

        for (i = front; i <= rear; i++)

            printf("%d ", queue[i]);

    }

}

void deleteq(int queue[])

{
```

```

if (front == - 1)

{
    printf("Queue is empty ");
}

else if (front == rear)

{

    printf("\n %d deleted", queue[front]);

    front = - 1;

    rear = - 1;

}

else

{

    printf("\n %d deleted", queue[front]);

    front++;

} }

```

## OUTPUT

```

Circular Queue:
1. Insert
2. Delete
3. Display
0. Exit
Enter Choice 0-3? : 1

Enter number: 2

Circular Queue:
1. Insert
2. Delete
3. Display
0. Exit
Enter Choice 0-3? : 3

2

Circular Queue:
1. Insert
2. Delete
3. Display
0. Exit
Enter Choice 0-3? : _

```

### Difference between circular queue and simple queue

#### Simple queue

- 1 first elements in the queue doesn't follow the last element.
- 2 problem of overflow in a queue frequently.
- 3 there is wastage of memory space.



### **Circular queue**

- 1 first element in the queue follows last element.
- 2 there is no overflow problem.
- 3 there is no wastage of memory space.