

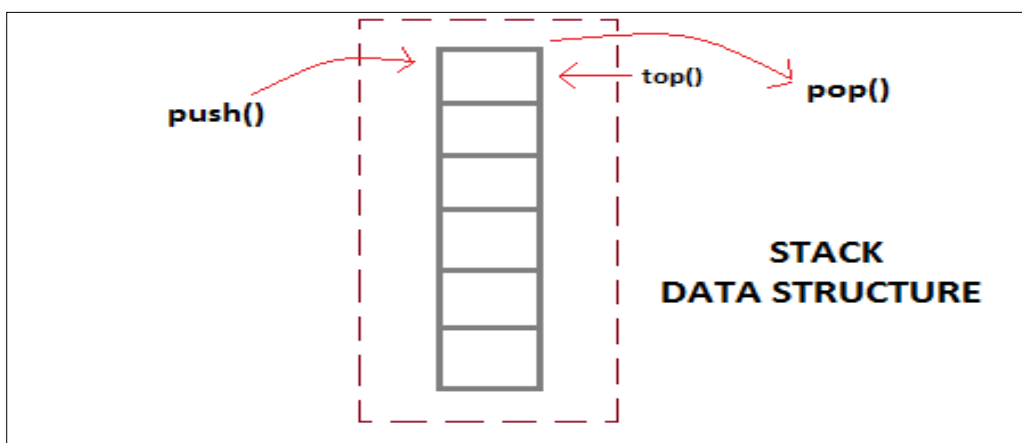
## Unit III : Stacks, Queues, Recursion

- Introduction, Stack
  - Array representation of stack
  - Push & Pop operations
  - Arithmetic Expression : Polish Notation – Infix, Postfix & Prefix Notations
  - Evaluation of postfix expression
  - Recursion : Factorial, Fibonacci
  - Queue : Memory Representation, Insertion, Deletion
  - Dequeue
  - Priority Queue
- 

### Introduction to Stack :

Stack is a linear data structure which follows a particular order in which the operations are performed. It is a simple data structure that allows adding and removing elements in a particular order.

Every time an element is added, it goes on the top of the stack and the only element that can be removed is the element that is at the top of the stack.



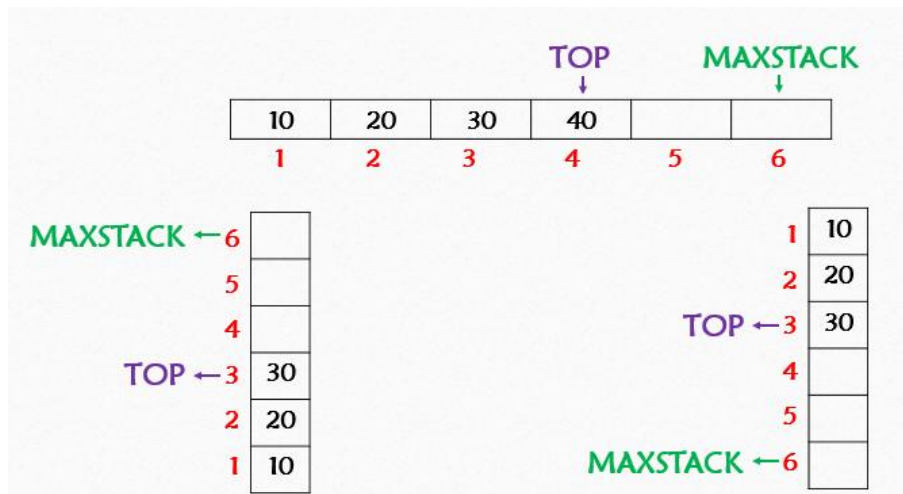
Stack is an ordered list of similar data type. Stack is a LIFO (Last in First out) structure or we can say FILO (First in Last out).

push() function is used to insert new elements into the Stack and pop() function is used to remove an element from the stack.

Both insertion and removal are allowed at only one end of Stack called Top. Stack is said to be in Overflow state when it is completely full and is said to be in Underflow state if it is completely empty.

## ✚ Array representation of Stack :

Stack data structure can be represented in memory in various forms. Linear array is the most representation of stack in memory as :



In the above representation, following are the two special pointers used, as :

- TOP** : This pointer indicates location of the last element added in stack and is useful for any push and pop operation.
- MAXSTACK** : It is a value that indicates size or maximum number of elements in stack.

When **MAXSTACK = NULL**, it means that the stack is not allocated.

If **TOP = NULL**, it means that the stack is empty.

If **TOP = MAXSTACK**, it means that the list is full.

✚ **PUSH Operation** : Inserting a new element in stack is called PUSH operation. As long as **TOP != MAXSTACK**, PUSH operation can be performed.

**Algorithm : STACK\_PUSH(STACK, TOP, MAXSTACK, ITEM)**

Let STACK be the stack in memory with TOP & MAXSTACK pointers. ITEM is the element to be inserted.

This algorithm confirms if the stack has space & then inserts new element.

**Step 1** : If **TOP = MAXSTACK** then


Write OVERFLOW & EXIT

else

Set  $TOP := TOP + 1$

Set  $STACK[TOP] := ITEM$

**Step 2 : EXIT**

 **POP Operation** : Removing an existing element from stack is called POP operation.

**Algorithm :  $STACK\_POP(STACK, TOP, MAXSTACK, ITEM)$**

Let STACK be the stack in memory with TOP & MAXSTACK pointers. ITEM is the element to be deleted.

This algorithm confirms that  $TOP \neq NULL$  & if it is the stack is underflow & hence no deletion is possible.

**Step 1 : If  $TOP = NULL$  then**

Write UNDERFLOW & EXIT

else

Set  $ITEM := STACK[TOP]$

Set  $TOP := TOP - 1$

**Step 2 : EXIT**

As like Linked LIST, STACK also allocates memory in two forms, as :

1. **Static Memory Allocation** : Static Memory Allocation does not use the concept of HEAP, thus it works similar to array where no insertion & deletion is possible during run-time.
2. **Dynamic Memory Allocation** : Dynamic Memory Allocation in STACK is done using the concept of HEAP.

## Arithmetic Expressions :

In any algorithm or program, each step, sentence or instruction is used for specific purpose. An instruction or step which performs an arithmetic operation is known as an Arithmetic Expression.

For e.g. Set  $A := B + C$

An expression may have single or multiple operators, hence an expression can also be defined as – “ An Instruction with collection of operators & operands in a specific order.”

When a single expression uses multiple operators, it is necessary to decide which operator executes first? The rule by which execution of operator is decided is called **operator precedence**.

According to mathematical rules, different operators have following precedence level, as

**Highest** -  $[\wedge]$

**Higher** -  $[ * // ]$

**Lower** –  $[ + / - / = ]$

If there are different operators in an expression, the operator with highest precedence is executed first.

$$A = 10 + 5 * 3$$

Here,

$$A = 10 + 15$$

$$\mathbf{A = 25}$$

If the expression contains multiple operators of same level, then execution follows “Left – to – Right” order, as :

$$A = 10 + 5 - 3$$

Here,

$$A = 15 - 3$$

$$\mathbf{A = 12}$$

The default precedence can be changed using parenthesis, as :

$$A = (10 + 5) * 3$$

Here,

$$A = 15 * 3$$

$$\mathbf{A = 45}$$

The most common arithmetic operations are represented using operator symbols such as

$\mathbf{A + B, \quad A - B, \quad A * B, \quad A / B}$

## Polish Notations :

Expressions with same level operators or single operators does not clearly specify the execution sequence, i.e. If  $A + B$  is the expression then it may be  $A+B$  or  $B+A$ .

But for processor it must be clear that what is the sequence & position of operand. For this purpose Polish Notations are recommended.

The operator representation rules designed by mathematician from POLAND are popularly known as Polish Notations.

According to polish notations, operators should be placed before operands to show the sequence & position as :

$+AB$ ,  $-CD$ ,  $*EF$ ,  $/BA$

Polish notations uses three different types of notations, as :

- Infix Notation
- Prefix Notation
- Postfix Notation

### 1. Infix Notation :

In Infix Notation the binary operator is between the two operands. It is a regular arithmetic expression given by parenthesis.

It converts the regular arithmetic expression into polish notation using square brackets[ ].

For e.g.

$$(A+B)*C = [A+B]*C$$

### 2. Pre-fix Notation :

In Pre-fix Notation the binary operator preceeds the two operands. It is a regular arithmetic expression where operators are placed before operands.

For e.g.  $+AB$ ,  $-AB$ ,  $*AB$ ,  $/AB$

### 3. Post-fix Notation :

In some cases, polish notations are represented in reverse order, hence called reverse polish notations.

In this operators are placed after operands, thus also called Post-fix notations.

For e.g.  $AB+$ ,  $AB-$ ,  $AB*$ ,  $AB/$

## Evaluation of Post-fix Expression :

The Postfix notation is used to represent algebraic expressions. The expressions written in postfix form are evaluated faster compared to infix notation as parenthesis are not required in Postfix expressions.

Following is algorithm for evaluation postfix expressions :

**Step 1 :** Create a stack to store operands (or values).

**Step 2 :** Scan the given expression and do following for every scanned element.

- a) If the element is a number, push it into the stack.
- b) If the element is a operator, pop operands for the operator from stack. Evaluate the operator and push the result back to the stack

**Step 3 :** When the expression is ended, the number in the stack is the final answer.

**Let the given expression be “2 3 1 \* + 9 -“. We scan all elements one by one, as**

- 1) Scan ‘2’, it’s a number, so push it to stack. Stack contains ‘2’
- 2) Scan ‘3’, again a number, push it to stack, stack now contains ‘2 3’ (from bottom to top)
- 3) Scan ‘1’, again a number, push it to stack, stack now contains ‘2 3 1’
- 4) Scan ‘\*’, it’s an operator, pop two operands from stack, apply the \* operator on operands, we get  $3*1$  which results in 3. We push the result ‘3’ to stack. Stack now becomes ‘2 3’.
- 5) Scan ‘+’, it’s an operator, pop two operands from stack, apply the + operator on operands, we get  $3 + 2$  which results in 5. We push the result ‘5’ to stack. Stack now becomes ‘5’.
- 6) Scan ‘9’, it’s a number, we push it to the stack. Stack now becomes ‘5 9’.
- 7) Scan ‘-’, it’s an operator, pop two operands from stack, apply the – operator on operands, we get  $5 - 9$  which results in -4. We push the result ‘-4’ to stack. Stack now becomes ‘-4’.
- 8) There are no more elements to scan, we return the top element from stack (which is the only element left in stack).

Post-fix Expression	Operators	Stack
2 3 1 * + 9 -		2
3 1 * + 9 -		2 3
1 * + 9 -		2 3 1
* + 9 -	3 * 1 = 3	2 3
+ 9 -	2 + 3 = 5	5
9 -		5 9
-	5 - 9 = -4	-4

## Recursion :

In programming, algorithm & programs are divided into smaller executable modules. Each such smaller executable module is called a method or a procedure.

It is very common that one procedure calls another procedure, Thus, there will be one caller procedure & called procedure.

From caller procedure execution control is transferred to called procedure.

But in some real life applications, a procedure also calls itself, i.e. the caller & the called procedure both are same.

**The technique of a procedure or a function calling itself is called recursion.**

The procedure which is calling itself is known as recursive procedure. To avoid the chances of infinite calls, a recursive function is defined such that it must have following two properties :

1. It must be using a base criteria (Condition)
2. In each calling, the counter should get closer to the base criteria.

Use of such recursive procedure is very common while calculating factorial of a number as :

$$n! = 1 * 2 * 3 * \dots * n$$

The factorial of a negative number doesn't exist. And the factorial of 0 is 1.

**Step 1 :** Start

**Step 2 :** Read number  $n$

**Step 3 :** Call factorial( $n$ )

**Step 4 :** Print factorial  $f$

**Step 5 :** Stop

**factorial( $n$ )**

**Step 1 :** if  $n < 1$  then

return 1

**Step 2 :** else

$f = n * \text{factorial}(n-1)$

**Step 3 :** Return  $f$

**Program to get input of a number & calculate its factorial using recursion.**

```
#include<iostream.h>
```

```
void main( )
```

```
{
```

```
int N, f;
```

```
clrscr();
```

```
cout<<"Enter a no. : ";
```

```
cin>>N;
```

```
f=FACT(N);
```

```
cout<<"Factorial : "<<f;
```

```
}
```

```
int FACT(int n)
```

```
{
```

```
if( $n \geq 1$ )
```

```
return  $n * \text{FACT}(n-1)$ ;
```

```
else
```

```
return 1;
```

```
}
```

**$5! = 5 * 4 * 3 * 2 * 1 = 120$**

**$5 * \text{FACT}(4) \quad 24 * 5$**

**$4 * \text{FACT}(3) \quad 6 * 4$**

**$3 * \text{FACT}(2) \quad 2 * 3$**

**$2 * \text{FACT}(1) \quad 1 * 2$**

**$1 * \text{FACT}(0) \quad \text{return } 1$**



## Introduction to Queues :

Queue is a linear data structure where the first element is inserted from one end called **REAR** and deleted from the other end called **FRONT**.

Queue is also called as **FIFO [ First In First Out]** data structure. According to its FIFO structure, element inserted first will also be removed first.

**Front pointer** is used to get the first data item from a queue.

**Rear pointer** is used to get the last item from a queue.

Like Stacks, Queues can also be represented in memory in two ways, as :

- Using the contiguous memory like an array
- Using the non-contiguous memory like a linked list

### Array representation of queue in memory :

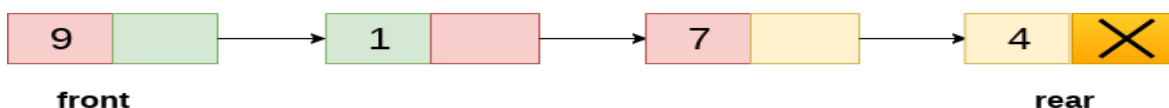
We can easily represent queue by using linear arrays. There are two variables i.e. front and rear, that are implemented in the case of every queue. Front and rear variables point to the position from where insertions and deletions are performed in a queue.

### Linked representation of queue in memory :

In a linked queue, each node of the queue consists of two parts i.e. data part and the link part. Each element of the queue points to its immediate next element in the memory.

In the linked queue, there are two pointers maintained in the memory i.e. front pointer and rear pointer. The front pointer contains the address of the starting element of the queue while the rear pointer contains the address of the last element of the queue.

Insertion and deletions are performed at rear and front end respectively. If front and rear both are NULL, it indicates that the queue is empty.



## Insertion in Queue :

Queue allows insertion operation that refers to adding new elements in queue. Queue in memory is represented using linear array with two exclusive terms 'REAR' & 'FRONT'.

Insertion in queue is always done at REAR position & generally after successful insertion, value of REAR is updated. While performing insertion following possibilities are to be considered :

1. Queue is full : When REAR pointer is equal to maximum number of elements in the queue, it is said to be in overflow condition.
2. Queue is empty : When REAR & FRONT pointers are NULL, the queue is said to be in underflow condition.

**Algorithm : INSERT\_QUEUE(FRONT, REAR, ITEM, N)**

**Description :** Let Queue be the array in memory with size N. REAR pointer points to the last element while Front pointer points to the first element. ITEM is the new element to be inserted.

**Step 1 :** If REAR=N, then

Write Overflow & Exit.

**Step 2 :** Else if REAR=NULL & FRONT=NULL then

Set REAR := REAR + 1

Set QUEUE[REAR] := ITEM

Set FRONT := FRONT + 1

**Step 3 :** Else

Set REAR := REAR + 1

Set QUEUE[REAR] := ITEM

**Step 4 :** EXIT

### **Deletion in Queue :**

Queue allows deletion operation that refers to deleting an existing element from the queue. Deletion in queue is always done using FRONT pointer and generally after successful deletion, FRONT pointer is to be updated.

While performing deletion, following possibilities are considered, as :

1. **Queue is empty :** When FRONT & REAR are NULL, it means queue contains no elements, i.e. underflow condition.
2. **Queue with one element :** When FRONT & REAR pointers points to the first element, after deletion of that element, both values become NULL.

**Before Deletion :**

10				
----	--	--	--	--

FRONT = 1  
 REAR = 1

1
2
3
4
5

**After Deletion :**

--	--	--	--	--

FRONT = 0  
 REAR = 0

1
2
3
4
5

3. **Queue with few elements :** When there are few elements in the queue, after deletion of first element, FRONT pointer is incremented by 1.

**Before Deletion :**

10	20	30	40	
----	----	----	----	--

FRONT = 1  
 REAR = 4

1
2
3
4
5

**After Deletion :**

	20	30	40	
--	----	----	----	--

FRONT = 2  
 REAR = 4

1
2
3
4
5

**Algorithm : DELETE\_QUEUE (QUEUE, FRONT, REAR, ITEM, N)**

**Description :** Let QUEUE be the array in memory with size N. FRONT pointer points to the first element & REAR pointer points to the last element in the queue.

This algorithm checks if the queue goes underflow condition & if not then deletes the first element from the queue & increments the size of FRONT pointer.

**Step 1 :** If FRONT = NULL then

Write UNDERFLOW & EXIT

**Step 2 :** Else if FRONT = 1 & REAR = 1 then

Set ITEM := QUEUE[FRONT]

Set FRONT := NULL

Set REAR := NULL

**Step 3 :** Else

Set ITEM := QUEUE[FRONT]

Set FRONT := FRONT + 1

**Step 4 :** EXIT

## Dequeue [Double Ended Queue] :

Queue applies restriction on insertion & deletion i.e. insertion only at REAR & deletion only at FRONT positions. Hence it is also called Single Ended Queue.

But in some cases, a specialized queue is required that allows insertion & deletion at both ends. Thus, DEQUEUE is used, also called as 'DECK'.

Dequeue allows insertion & deletion at both ends & generally represented in memory as circular array.

Dequeue uses two special pointers for its handling as "Left" & "Right". Dequeue in programming can be implemented in two ways :

1. **Input Restricted** : In this type, insertion is restricted to be done only at one end but deletion is allowed at both ends.
2. **Output Restricted** : In this type, deletion is restricted to be done only at one end but insertion is allowed at both ends.

While performing insertion & deletion following possibilities are considered, as :

- When LEFT pointer contains NULL it means DEQUEUE is empty & only insertion is possible.
- When RIGHT = N, it means DEQUEUE is full & only deletion is possible.

## Priority Queue :

Generally queue stores & processes data elements using FIFO rule. But in many cases, FIFO cannot be applied. In such cases, modified form of queue is used called Priority Queue.

A special implementation of queue which processes data elements on the basis of a priority value is called "Priority Queue". Priority is a relative importance of a data element as compared to others.

In priority queue, data element with highest priority will be executed first. If elements with the same priority occur, they are served according to their order in the queue.

In a queue, the first-in-first-out rule is implemented whereas, in a priority queue, the values are removed on the basis of priority. The element with the highest priority is removed first.

The common implementation of priority queue is found in time sharing multitasking operating system. Each element is assigned a positive number as priority number[PRN].

The processing principle of priority queue is based on following rules :

1. If elements of different priorities are available then element with highest priority will be processed first.
2. If elements of same priorities are present then processing is done according to FIFO.

**Memory Representation :** A priority queue can be represented in memory as linked list with three parallel arrays, as : INFO, LINK & PRN. INFO contains the data element, LINK contains address of next node & PRN contains priority number.

