UNIT-III

Data Structures

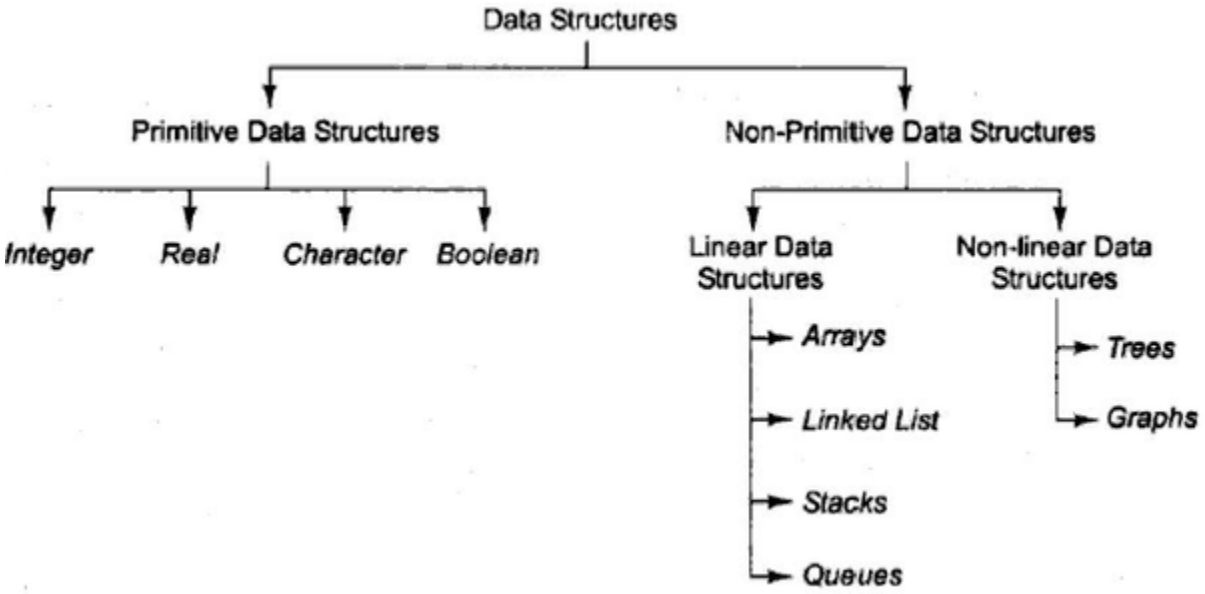
Topics:-

1. Overview of data structures
2. stacks and queues
3. Representation of a stack
4. Stack related terms, operations on a stack
5. Implementation of a stack
6. Evaluation of arithmetic expressions
7. Infix, prefix, and postfix notations
8. Evaluation of postfix expression
9. Conversion of expression from infix to postfix
10. Recursion
11. Queues - representation of queue, insertion, deletion, searching operations.

TOPIC 1: OVERVIEW OF DATA STRUCTURES

Data Structures in C are used to store data in an organised and efficient manner.

Types of Data Structures



There are two types of data structures:

- Primitive data structure
- Non-primitive data structure

Primitive Data structure

The primitive data structures are primitive data types. The int, char, float, double, and pointer are the primitive data structures that can hold a single value.

Non-Primitive Data structure

The non-primitive data structure is divided into two types:

- Linear data structure
- Non-linear data structure

Linear Data Structure

A linear data structure is a structure in which the elements are stored sequentially, and the elements are connected to the previous and the next element. As the elements are stored sequentially, so they can be traversed or accessed in a single run. The implementation of linear data structures is easier as the elements are sequentially organized in memory. The data elements in an array are traversed one after another and can access only one element at a time.

The types of linear data structures are Array, Queue, Stack, Linked List.

A non-linear data structure is also another type of data structure in which the data elements are not arranged in a contiguous manner. As the arrangement is nonsequential, so the data elements cannot be traversed or accessed in a single run. In the case of linear data structure, element is connected to two elements (previous and the next element), whereas, in the non-linear data structure, an element can be connected to more than two elements.

Trees and Graphs are the types of non-linear data structure.

Data structures can also be classified as:

- Static data structure: It is a type of data structure where the size is allocated at the compile time. Therefore, the maximum size is fixed.
- Dynamic data structure: It is a type of data structure where the size is allocated at the run time. Therefore, the maximum size is flexible.

Major Operations

The major or the common operations that can be performed on the data structures are:

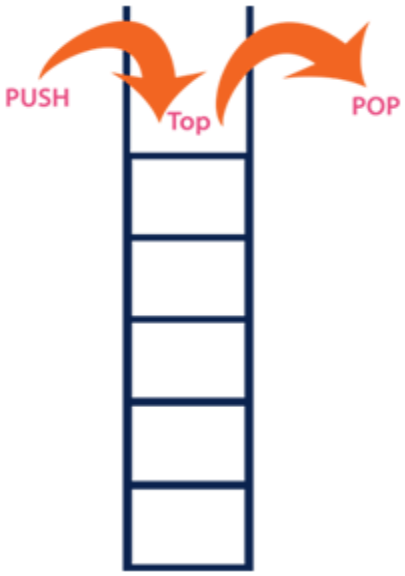
- Searching: We can search for any element in a data structure.

- Sorting: We can sort the elements of a data structure either in an ascending or descending order.
- Insertion: We can also insert the new element in a data structure.
- Updation: We can also update the element, i.e., we can replace the element with another element.
- Deletion: We can also perform the delete operation to remove the element from the data structure.

---***---

TOPIC 2,3,4,5,11 : STACKS AND QUEUES

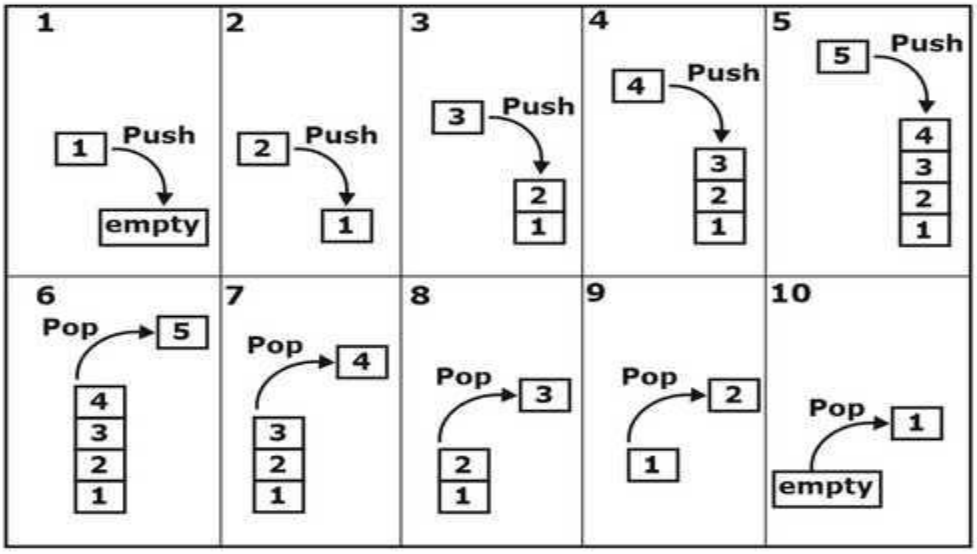
Stack:- Stack is a linear data structure in which the insertion and deletion operations are performed at only one end. In a stack, adding and removing of elements are performed at a single position which is known as "**top**". That means, a new element is added at top of the stack and an element is removed from the top of the stack. In stack, the insertion and deletion operations are performed based on **LIFO** (Last In First Out) principle.



In a stack, the insertion operation is performed using a function called **"push"** and deletion operation is performed using a function called **"pop"**.

In the figure, PUSH and POP operations are performed at a top position in the stack. That means, both the insertion and deletion operations are performed at one end (i.e., at Top)

A stack can be visualized as follows:



Operations on a Stack

The following operations are performed on the stack...

1. **Push (To insert an element on to the stack)**
2. **Pop (To delete an element from the stack)**
3. **Display (To display elements of the stack)**

Stack data structure can be implemented in two ways. They are as follows...

1. **Using Array**
2. **Using Linked List**

When a stack is implemented using an array, that stack can organize an only limited number of elements. When a stack is implemented using a linked list, that stack can organize an unlimited number of elements.

A stack data structure can be implemented using a one-dimensional array. But stack implemented using array stores only a fixed number of data values. This implementation is very simple. Just define a one dimensional array of specific size and insert or delete the values into that array by using **LIFO principle** with the help of a variable called **'top'**. Initially, the top is set to -1. Whenever we want to insert a value into the stack, increment the top value by one and then insert. Whenever we want to delete a value from the stack, then delete the top value and decrement the top value by one.

Stack Operations using Array

A stack can be implemented using array as follows...

Before implementing actual operations, first follow the below steps to create an empty stack.

- Step 1 - Include all the header files which are used in the program and define a constant 'SIZE' with specific value.
- Step 2 - Declare all the functions used in stack implementation.
- Step 3 - Create a one dimensional array with fixed size (int stack[SIZE])
- Step 4 - Define a integer variable 'top' and initialize with '-1'. (int top = -1)
- Step 5 - In main method, display menu with list of operations and make suitable function calls to perform operation selected by the user on the stack.

push(value) - Inserting value into the stack

In a stack, push( ) is a function used to insert an element into the stack. In a stack, the new element is always inserted at top position. Push function takes one integer value as parameter and inserts that value into the stack. We can use the following steps to push an element on to the stack...

- Step 1 - Check whether stack is FULL. (top == SIZE-1)
- Step 2 - If it is FULL, then display "Stack is FULL!!! Insertion is not possible!!!" and terminate the function.
- Step 3 - If it is NOT FULL, then increment top value by one (top++) and set stack[top] to value (stack[top] = value).

pop( ) - Delete a value from the Stack

In a stack, pop( ) is a function used to delete an element from the stack. In a stack, the element is always deleted from top position. Pop function does not take any value as parameter. We can use the following steps to pop an element from the stack...

- Step 1 - Check whether stack is EMPTY. (top == -1)
- Step 2 - If it is EMPTY, then display "Stack is EMPTY!!! Deletion is not possible!!!" and terminate the function.
- Step 3 - If it is NOT EMPTY, then delete stack[top] and decrement top value by one (top--).

display( ) - Displays the elements of a Stack

We can use the following steps to display the elements of a stack...

- Step 1 - Check whether stack is EMPTY. (top == -1)
- Step 2 - If it is EMPTY, then display "Stack is EMPTY!!!" and terminate the function.
- Step 3 - If it is NOT EMPTY, then define a variable 'i' and initialize with top. Display stack[i] value and decrement i value by one (i--).

- Step 3 - Repeat above step until i value becomes '0'.

Queue :- Queue is a linear data structure in which the insertion and deletion operations are performed at two different ends. In a queue data structure, adding and removing elements are performed at two different positions. The insertion is performed at one end and deletion is performed at another end. In a queue data structure, the insertion operation is performed at a position which is known as 'rear' and the deletion operation is performed at a position which is known as 'front'. In queue data structure, the insertion and deletion operations are performed based on FIFO (First In First Out) principle.

Operations on a Queue

The following operations are performed on a queue data structure...

1. insert(value) - (To insert an element into the queue)
2. delete() - (To delete an element from the queue)
3. display() - (To display the elements of the queue)

Queue data structure can be implemented in two ways. They are as follows...

1. Using Array
2. Using Linked List

When a queue is implemented using an array, that queue can organize an only limited number of elements. When a queue is implemented using a linked list, that queue can organize an unlimited number of elements.

A queue data structure can be implemented using one dimensional array. The queue implemented using array stores only fixed number of data values. The implementation of queue data structure using array is very simple. Just define a one dimensional array of specific size and insert or delete the values into that array by using FIFO (First In First Out) principle with the help of variables 'front' and 'rear'. Initially both 'front' and 'rear' are set to -1. Whenever, we want to insert a new value into the queue, increment 'rear' value by one and then insert at that position. Whenever we want to delete a value from the queue, then delete the element which is at 'front' position and increment 'front' value by one.

Queue Operations using Array

Queue data structure using array can be implemented as follows...

Before we implement actual operations, first follow the below steps to create an empty queue.

- Step 1 - Include all the header files which are used in the program and define a constant 'SIZE' with specific value.
- Step 2 - Declare all the user defined functions which are used in queue implementation.
- Step 3 - Create a one dimensional array with above defined SIZE (int queue[SIZE])

- Step 4 - Define two integer variables 'front' and 'rear' and initialize both with '-1'. (int front = -1, rear = -1)
- Step 5 - Then implement main method by displaying menu of operations list and make suitable function calls to perform operation selected by the user on queue.

insert(value) - Inserting value into the queue

In a queue data structure, insert( ) is a function used to insert a new element into the queue. In a queue, the new element is always inserted at rear position. The insert( ) function takes one integer value as a parameter and inserts that value into the queue. We can use the following steps to insert an element into the queue...

- Step 1 - Check whether queue is FULL. (rear == SIZE-1)
- Step 2 - If it is FULL, then display "Queue is FULL!!! Insertion is not possible!!!" and terminate the function.
- Step 3 - If it is NOT FULL, then increment rear value by one (rear++) and set queue[rear] = value.

delete( ) - Deleting a value from the Queue

In a queue data structure, delete( ) is a function used to delete an element from the queue. In a queue, the element is always deleted from front position. The delete( ) function does not take any value as parameter. We can use the following steps to delete an element from the queue...

- Step 1 - Check whether queue is EMPTY. (front == rear)
- Step 2 - If it is EMPTY, then display "Queue is EMPTY!!! Deletion is not possible!!!" and terminate the function.
- Step 3 - If it is NOT EMPTY, then increment the front value by one (front ++). Then display queue[front] as deleted element. Then check whether both front and rear are equal (front == rear), if it TRUE, then set both front and rear to '-1' (front = rear = -1).

display( ) - Displays the elements of a Queue

We can use the following steps to display the elements of a queue...

- Step 1 - Check whether queue is EMPTY. (front == rear)
- Step 2 - If it is EMPTY, then display "Queue is EMPTY!!!" and terminate the function.
- Step 3 - If it is NOT EMPTY, then define an integer variable 'i' and set 'i = front+1'.
- Step 4 - Display 'queue[i]' value and increment 'i' value by one (i++). Repeat the same until 'i' value reaches to rear (i <= rear)

---***---

TOPIC 6: EVALUATION OF ARITHMETIC EXPRESSIONS

An expression is a collection of operators and operands that represents a specific value.

In above definition, **operator** is a symbol which performs a particular task like arithmetic operation or logical operation or conditional operation etc.,

Operands are the values on which the operators can perform the task. Here operand can be a direct value or variable or address of memory location.

Evaluation of Arithmetic Expressions:-

Parentheses may be used in expressions to specify the order of evaluation. Expressions within parentheses are evaluated first. When parentheses are nested, the innermost set of parentheses is evaluated first, and then successively more inclusive parentheses are evaluated.

When an expression contains no parentheses, the expression evaluates the arithmetic operators in the following hierarchical order:

1. unary plus and minus
2. exponentiation
3. multiplication and division
4. addition and subtraction

When the sequence of execution is not specified by parentheses and two or more operators exist at the same hierarchical level, the order of evaluation is from left to right.

An arithmetic expression can begin with only a left parenthesis, a plus sign, a minus sign, an identifier, or a literal. It can end only with a right parenthesis, an identifier, or a literal. Each left parenthesis in an expression must have a matching right parenthesis, and each right parenthesis must have a matching left parenthesis. If the first operator is unary, it must be preceded by a left parenthesis if the expression immediately follows an identifier or another arithmetic expression.

---***---

TOPIC 7: INFIX, PREFIX, AND POSTFIX NOTATIONS

Expression Types:- Based on the operator position, expressions are divided into THREE types. They are as follows...
Infix Expression
Postfix Expression
Prefix Expression

Infix Expression

In infix expression, operator is used in between the operands.

The general structure of an Infix expression is as follows...

Operand1 Operator Operand2

Postfix Expression

In postfix expression, operator is used after operands. We can say that "Operator follows the Operands".

The general structure of Postfix expression is as follows...

Operand1 Operand2 Operator

Prefix Expression

In prefix expression, operator is used before operands. We can say that "Operands follows the Operator".

The general structure of Prefix expression is as follows...

Operator Operand1 Operand2

Infix Expression	Prefix Expression	Postfix Expression
A + B	+ 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 +
(A + B) * (C + D)	* + A B + C D	A B + C D + *
A + B + C + D	+ + + A B C D	A B + C + D +

---***---

TOPIC 8: EVALUATION OF POSTFIX EXPRESSION

Postfix Expression Evaluation using Stack Data Structure

A postfix expression can be evaluated using the Stack data structure. To evaluate a postfix expression using Stack data structure we can use the following steps...

1. Read all the symbols one by one from left to right in the given Postfix Expression
2. If the reading symbol is operand, then push it on to the Stack.
3. If the reading symbol is operator (+ , - , * , / etc.,), then perform TWO pop operations and store the two popped oparands in two different variables (operand1 and operand2). Then perform reading symbol operation using operand1 and operand2 and push result back on to the Stack.
4. Finally! perform a pop operation and display the popped value as final result.

Example #1: 4 5 + 7 2 - *

---

4 5 + 7 2 - *
The first character scanned is "4", which is an operand, so push it to the stack.

Stack Expression

---

4 5 + 7 2 - *
The next character scanned is "5", which is an operand, so push it to the stack.

Stack Expression

---

4 5 + 7 2 - *
The next character scanned is "+", which is an operator, so pop its two operands from the stack. Pop 5 from the stack for the right operand and then pop 4 from the stack to make the left operand.

4 + 5 = 9

Stack Expression

Next, push the result of 4 + 5 (9) to the stack.

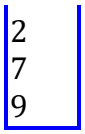
Stack Expression

---

4 5 + 7 2 - *
The next character scanned is "7", which is an operand, so push it to the stack.

Stack Expression

4 5 + 7 2 - *
The next character scanned is "2", which is an operand, so push it to the stack.



Stack Expression

---

4 5 + 7 2 - *
The next character scanned is "-", which is an operator, so pop its two operands from the stack. Pop 2 from the stack for the right operand and then pop 7 from the stack to make the left operand.

7 - 2 = 5

Stack Expression

Next, push the result of 7 - 2 (5) to the stack.

Stack Expression

---

4 5 + 7 2 - *
The next character scanned is "*", which is an operator, so pop its two operands from the stack. Pop 5 from the stack for the right operand and then pop 9 from the stack to make the left operand.

9 * 5 = 45

Stack Expression

Next, push the result of 9 * 5 (45) to the stack.

Stack Expression

Since we are done scanning characters, the remaining element in the stack (45) becomes the result of the postfix evaluation.
Postfix notation: 4 5 + 7 2 - *
Result: 45

Example #2: 4 2 3 5 1 - + * +

---

4 2 3 5 1 - + * +
The first character scanned is "4", which is an operand, so push it to the stack.

Stack Expression

---

4 2 3 5 1 - + * +

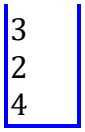
The next character scanned is "2", which is an operand, so push it to the stack.

Stack Expression

---

4 2 3 5 1 - + * +

The next character scanned is "3", which is an operand, so push it to the stack.

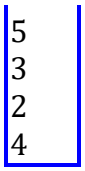


Stack Expression

---

4 2 3 5 1 - + * +

The next character scanned is "5", which is an operand, so push it to the stack.

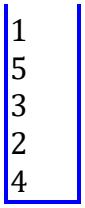


Stack Expression

---

4 2 3 5 1 - + * +

The next character scanned is "1", which is an operand, so push it to the stack.

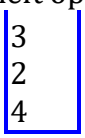


Stack Expression

---

4 2 3 5 1 - + * +

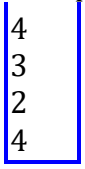
The next character scanned is "-", which is an operator, so pop its two operands from the stack. Pop 1 from the stack for the right operand and then pop 5 from the stack to make the left operand.



5 - 1 = 4

Stack Expression

Next, push the result of 5 - 1 (4) to the stack.



Stack Expression

---

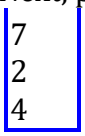
4 2 3 5 1 - + * +

The next character scanned is "+", which is an operator, so pop its two operands from the stack. Pop 4 from the stack for the right operand and then pop 3 from the stack to make the left operand.

3 + 4 = 7

Stack Expression

Next, push the result of 3 + 4 (7) to the stack.



Stack Expression

---

4 2 3 5 1 - + * +

The next character scanned is "*", which is an operator, so pop its two operands from the stack. Pop 7 from the stack for the right operand and then pop 2 from the stack to make the left operand.

2 * 7 = 14

Stack Expression

Next, push the result of 2 * 7 (14) to the stack.

Stack Expression

---

4 2 3 5 1 - + * +

The next character scanned is "+", which is an operator, so pop its two operands from the stack. Pop 14 from the stack for the right operand and then pop 4 from the stack to make the left operand.

4 + 14 = 18

Stack Expression

Next, push the result of 4 + 14 (18) to the stack.

Stack Expression

Since we are done scanning characters, the remaining element in the stack (18) becomes the result of the postfix evaluation.

---***---

TOPIC 9: CONVERSION OF INFIX TO POSTFIX EXPRESSION

Algorithm for conversion of infix to postfix expression using stack data structure:-

1. Scan the infix expression from left to right.

2. If the scanned character is an operand, output it.

3. Else,

1 If the precedence of the scanned operator is greater than the precedence of the operator in the stack(or the stack is empty or the stack contains a '('), push it.

2 Else, Pop all the operators from the stack which are greater than or equal to in precedence than that of the scanned operator. After doing that Push the scanned operator to the stack. (If you encounter parenthesis while popping then stop there and push the scanned operator in the stack.)

4. If the scanned character is an '(', push it to the stack.

5. If the scanned character is an ')', pop the stack and and output it until a '(' is encountered, and discard both the parenthesis.

6. Repeat steps 2-6 until infix expression is scanned.

7. Print the output

8. Pop and output from the stack until it is not empty.

Example 1:-

Infix Expression: **A+ (B*C-(D/E^F)*G)*H**, where **^** is an exponential operator.

Symbol	Scanned	STACK	Postfix Expression	Description
1.		(		Start
2.	A	(	A	
3.	+	(+	A	
4.	(	(+(	A	
5.	B	(+(	AB	
6.	*	(+(*	AB	
7.	C	(+(*	ABC	
8.	-	(+(-	ABC*	'*' is at higher precedence than '-'
9.	(	(+(-(	ABC*	
10.	D	(+(-(	ABC*D	
11.	/	(+(-(/	ABC*D	
12.	E	(+(-(/	ABC*DE	
13.	^	(+(-(/^	ABC*DE	
14.	F	(+(-(/^	ABC*DEF	
15.	)	(+(-	ABC*DEF^/	Pop from top on Stack, that's why '^' Come first
16.	*	(+(-*	ABC*DEF^/	
17.	G	(+(-*	ABC*DEF^/G	
18.	)	(+	ABC*DEF^/G*-	Pop from top on Stack, that's why '^' Come first
19.	*	(+*	ABC*DEF^/G*-	
20.	H	(+*	ABC*DEF^/G*-H	
21.	)	Empty	ABC*DEF^/G*-H*+	END

Resultant Postfix Expression: ABC*DEF^/G*-H*+

Example 2:-

Infix expression **((A * (B + D)/E) - F * (G + H / K)))**

Label No.	Symbol Scanned	Stack	Expression
1	(	(	
2	(	((	
3	A	((	A
4	*	((*	A
5	(	((*(	A
6	B	((*(	AB
7	+	((*(+	AB
8	D	((*(+	ABD
9	)	((*	ABD+
10	/	((*/	ABD+
11	E	((*/	ABD+E
12	)	(	ABD+E/*
13	-	(-	ABD+E/*
14	(	(-(	ABD+E/*
15	F	(-(	ABD+E/*F
16	*	(-(*	ABD+E/*F
17	(	(-(*(	ABD+E/*F

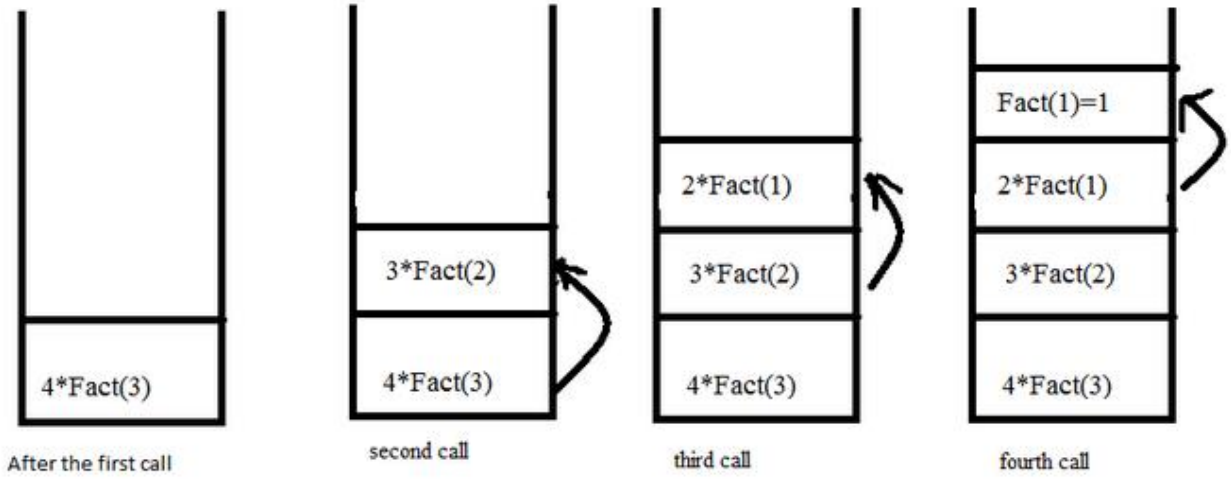
18	G	(-(*(	ABD+E/*FG
19	+	(-(*(+	ABD+E/*FG
20	H	(-(*(+	ABD+E/*FGH
21	/	(-(*(+/	ABD+E/*FGH
22	K	(-(*(+/	ABD+E/*FGHK
23	)	(-(*	ABD+E/*FGHK/+
24	)	(-	ABD+E/*FGHK/+*
25	)		ABD+E/*FGHK/+*-

Resultant Postfix Expression: ABD+E/*FGHK/+*-

TOPIC 10: RECURSION

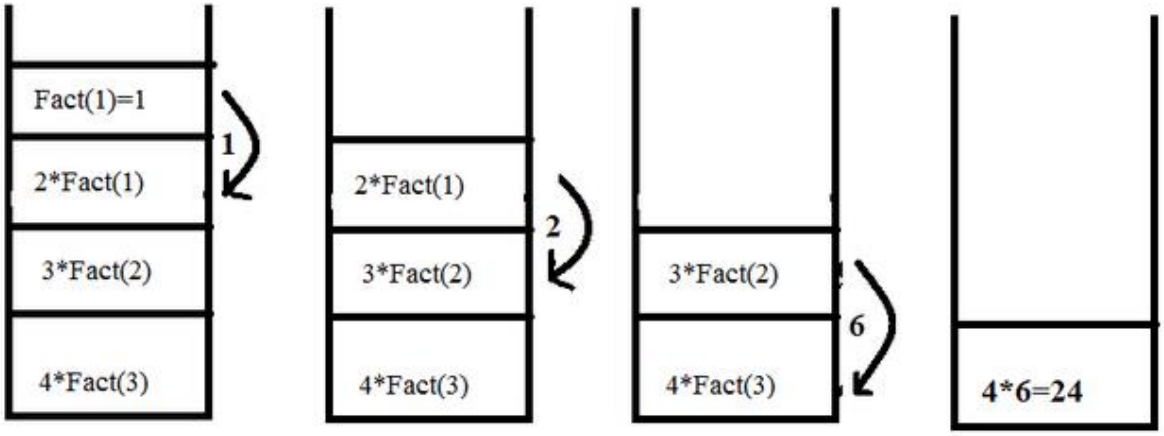
1. "Recursion" is technique of solving any problem by calling same function again and again until some breaking (base) condition where recursion stops and it starts calculating the solution from there on. For eg. calculating factorial of a given number
2. Thus in recursion last function called needs to be completed first.
3. Now Stack is a LIFO data structure i.e. (Last In First Out) and hence it is used to implement recursion.

When function call happens previous variables gets stored in stack



After the first call | second call | third call | fourth call

Returning values from base case to caller function



----***THE END***-----