

**UNIT – I CHAPTER-I****1.INTRODUCTION TO DATA STRUCTURES**

Data structure refers to the organization of data elements and the inter-relationships among them. Data structure is a method of organizing large amount of data more efficiently, so that any operation on that data becomes easy.

**i)Data:** Data is nothing but a piece of information. Data can be a number, a string, or a set of many numbers and strings. Data is of Atomic data or Composite data.

**Atomic data:** It is non-decomposable entity, i.e., a single unit. Atomic data is also known as Scalar data because of its numeric properties. Ex: 1234

**Composite data:** It can be broken into sub-fields that have meaning. Composite data is also known as Structured data, can be implemented using a structure or a class in C++.

Ex: Student record consists of Rollno, Name, Branch, Year and so on.

**ii)Data Type:** Data type refers to the kind of data that a variable may store.

Built-in data types: Generally programming languages have their own built-in data types. In C++, int, float, char, double, void are the built-in data types.

User-defined and Derived data types: Using built-in data types, user can define and derive different data types. C++ supports arrays, structures, unions, classes.

**iii) Data Object:** A data object represents a container for data values. It is characterised by a set of attributes, one of the most important is its data type.

A data object is a set of elements,  $D=\{A,B, \dots, Z, a, b, \dots, z\}$ ,  $D=\{\dots, -2,-1,0, 1, 2, 3, \dots\}$ .

A data structure may be finite or infinite.

A data object is a runtime instance of data structure.

**iv)Data Structure:** A Data Structure is a collection of data elements and the interrelationships among them. A DS is

a) a combination of elements, each of which is either as a data type or a data structure.

b) A set of associations or relationships involving the combined elements

A data structure is a set of Domains(D), a set of Functions(F), and a set of Axioms(A), i.e., in the form of an ADT.

**Domain(D):** is the range of values that the data may have.

**Functions(F):** is the set of operations for the data.

**Axioms(A):** is the set of rules with which different operations of F can be implemented.

**v) Abstract Data Type:** An ADT is a theoretical set of specifications of data set and the set of operations that can be performed on the data within a set.

A data type is termed as abstract when it is independent of various concrete implementations. **Different features of data specification methods are:**

**Abstract:** It should help the programmer to organise data by focusing on its logical properties rather than on the implementation details, it allows user to hide the complexity of a task.

**Safe:** It should control the manipulation of the representation of data, so that malfunctioning can be avoided.

**Modifiable:** It should make it relatively easy to modify the representation.

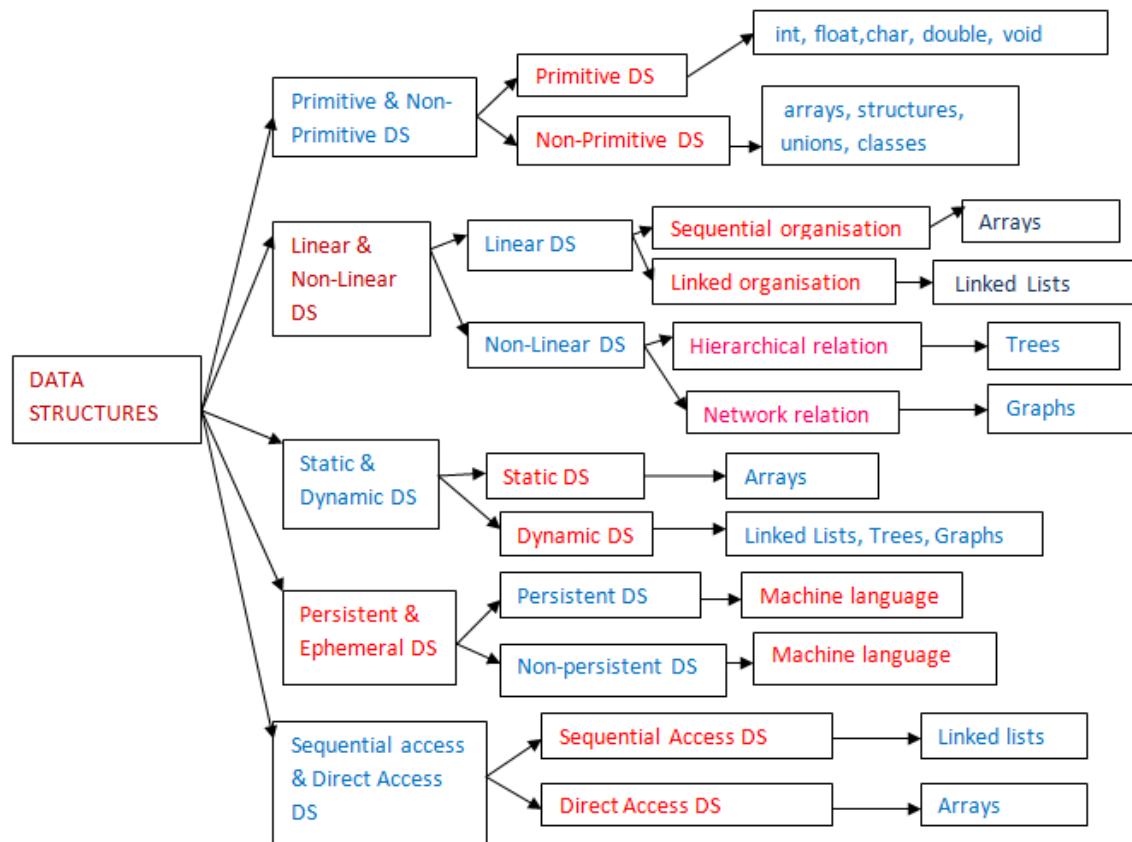
**Reusable:** The DS should be such that it is a reusable product for others.

ADT includes Domain(D), Functions(F), and Axioms(A).

**2. TYPES OF DATA STRUCTURES**

A DS is defined as a way of organising a set of data elements, i.e., data object and a set of operations that are applied on data object. Various types of DS are:

- 1) Primitive & Non-Primitive Data Structures
- 2) Linear & Non-Linear Data Structures
- 3) Static & Dynamic Data Structures
- 4) Persistent & Ephemeral Data Structures
- 5) Sequential & Direct Data Structures



### 1. Primitive & Non-Primitive Data Structures:

Primitive DS define a set of primitive elements that do not involve any other element as sub-parts. These are generally Primary/Built-in data types, int, float, char, double, void.

Non-Primitive DS are those that define a set of derived elements such as arrays, structures, unions, and classes.

### 2. Linear & Non-Linear Data Structures:

A data structure is said to be linear if its elements form a sequence or linear list. In this every element has a unique successor and predecessor. Array and Linked lists are Linear DS.

Non-Linear DS are used to represent the data containing Hierarchical or Network relationship among the elements. Trees and Graphs are Non-Linear DS.

### 3. Static & Dynamic Data Structures:

A DS is referred to as a Static DS if it is created before program execution begins. Array is an example to Static DS.

A DS that is created at runtime is called Dynamic DS, referred by pointer. Linked List, Trees, and Graphs are Dynamic DS.

### 4. Persistent & Ephemeral Data Structures:

A DS that supports operations on most recent version as well as on previous version are known as Persistent DS.

An Ephemeral DS supports operations on the most recent version, also known as Non-Persistent DS. Machine language supports both Persistent and Ephemeral Data Structures.

### 5. Sequential & Direct Data Structures:

Sequential Access means, to access the nth element, the user must access the preceding n-1 data elements. Linked List is a Sequential access DS. Direct Access means, any data element can be accessed directly. An array is an example to Direct Access DS.

### 3.INTRODUCTION TO ALGORITHMS

- Computer is a data processor or black box, accepts input and generates output.
- A program is a set of instructions written in a computer language.
- An algorithm is named after the 9<sup>th</sup> century mathematician Abu Jafar Mohammad bin Musa al-Khwarizmi, it is a set of rules for carrying out some task, either by hand or on a machine.
- A step by step details of a problem logic in general language is known as an algorithm. An algorithm is independent of system and programming language. Each algorithm includes input, processing and output.
- Software depends on
  - 1) algorithm chosen
  - 2) suitability & efficiency of layers of implementation

#### Characteristics of an algorithm:

- a) **Input:** every algorithm must take zero or more number of input values.
- b) **Output:** every algorithm must produce an output as result.
- c) **Unambiguous (definiteness) steps:** Every statement or instruction of algorithm must be clear and unambiguous.
- d) **Finiteness:** For all different cases, the algorithm must produce result with in finite number of steps.
- e) **Effectiveness:** Every instruction must be basic enough to be carried out and also must be feasible.

#### Algorithmics:

- It is a field of Computer Science, study of algorithms, goal is to understand complexity of algorithms includes design and analysis of algorithms.
- Algorithmics include
  - a) **How to devise algorithms:** Devising an algorithm is an art that can never be fully automated. By studying various techniques, i.e., design strategies, it becomes easier to devise new and useful algorithms.
  - b) **How to validate algorithms:** Once an algorithm is devised, it is necessary to show that it computes the correct answer for all possible inputs. The methods used for validation include contradiction and mathematical induction.
  - c) **How to analyse algorithms:** Analysis of algorithms refers to the task of determining how much computing time and storage, an algorithm requires.

#### Algorithm design tools:

- The two popular tools used to design algorithm are i) **Pseudo code** ii) **Flow-Chart**
- **Pseudo code** is a method of describing algorithms using natural language and programming language.
- **Flow-Chart** is a pictorial or graphical representation of problem logic.

#### **4. PSEUDO CODE**

- A tool which used to define algorithm is the Pseudo code. It is partly English and partly Computer language code.
  - A pseudo code is an English like representation of the code required for an algorithm.
  - **Pseudo code notations:** The pseudo code uses various notations such as header, purpose, pre-post conditions, return, variables, statement numbers and sub-algorithms.
- i) **Header:** It includes the name of the algorithm, the parameters, and the list of pre-post conditions. It makes the pseudo code readable.
  - ii) **Purpose:** It is brief description about algorithm work, describing the general algorithm processing.
  - iii) **Pre-condition:** It states the pre requirements for the parameters if any. The pre condition may state group of elements, for specifying no precondition, just use it as nothing. (Pre Nothing).
  - iv) **Post-Condition:** Identifies action taken and status of output performance, No post-condition use Nothing. (Post Nothing)
  - v) **Return statement:** Returns output or result.
  - vi) **Statement Numbers:** The statements in an algorithm are numbered sequentially. For conditional or non-conditional jumps and also for iteration statements, numbering helps to identify statements uniquely. Decimal, Roman numbers or even alphabets can be used to label the statements. If decimal notation is used, then within loops the lines can be numbered as 4.1, 4.2, and so on. Ex: 4 while(i<10)
 

```
begin
    4.1 x=x*y
    4.2 i=i+1
End
```
  - vii) **Variables:** Meaningful variable names make the code easier to understand, debug and modify. Rules are
    - a) It is easier to use descriptive names instead of single character names.
    - b) It is suggested to avoid usage of short forms and generic names.
    - c) It is expected to use variable names so that the data type of the variable can be indicated.
  - viii) **Statement Constructs:** There are three statement constructs used for developing an algorithm. a) Sequence b) Decision c) Repetition
 

The use of these constructs makes an algorithm easy to understand, debug and modify.

    - a) **Sequence:** Algorithm statements are in a linear order.
 

Ex: Area of a circle  
Pre none  
Post none  
1. Read radius  
2. Carea=3.142\*radius\*radius  
3. Print Carea  
4. Stop
    - b) **Decision:** Depends on condition, if it is true, one sequence of statements, if it is false, another sequence of statements are executed.
 

Ex: Biggest of two numbers  
Pre none  
Post none  
1. Read num1, num2

2. If num1 > num2 then
  - 2.1 Print num1
 Else
  - 2.2 Print num2
3. Stop

c) **Repetition:** Repeats a set of statements, uses looping constructs.

Ex: Factorial of a given number

Pre none

Post none

1. Read num
2. fact=1
3. While(num>0)
  - Do
    - 3.1 fact=fact\*n
    - 3.2 num=num-1
  - Done
4. Print fact
5. Stop

ix) **Sub-Algorithms:** In structured programming, the problem solution is described in the form of modules, also called functions, sub routines, procedures, methods & modules.

## **5.RELATIONSHIP AMONG DATA, DATA STRUCTURES & ALGORITHMS**

- There is an intimate relationship between the structuring of data and analysis of algorithms.
- A data structure and an algorithm should be thought as one single unit, neither one making sense without the other.
- Let us consider the example of searching for a person phone number in a directory. This search depends on how phone number and names are arranged in the directory.
- There are two ways of organising the data in the directory.
  - The data is organised randomly, then searching starts from the first name till the last name in the directory or until founding the required name.
  - If the data is organised by sorting the names, then the search is much easier.
- As the data is in sorted order, both the binary search and a typical directory search methods work.
- Hence our ideas for algorithms become possible when we realise that we can organise the data as we wish.
- We can say that there is a strong relationship between the structuring of data, data structure and the operations to process data(algorithms).

## **6.IMPLEMENTATION OF DATA STRUCTURES**

- A data structure is an aggregation of atomic and composite data types into a set with the relationship among them defined.
- A data structure DS is a triplet, i.e.,  $DS=(D,F,A)$ , where D is a Domain, a set of data object, F is a set of functions and A is a set of rules(Axioms) to implement the functions.
- Ex:  $D=(..., -2, -1, 0, 1, 2 ...)$        $F=(+, -, *, /, \%)$   
 A=(a set of binary arithmetic rules to perform addition, subtraction, multiplication, division, and modular division)

- Implementation of a data structure is a process of defining an ADT until all operations are expressed effectively, so that they are defined in terms of directly executable functions.
- Implementation of data structure can be viewed in terms of two phases.
  - i) Specification
  - ii) Implementation
- **Phase I: Specification:** A data structure should be designed. So that we know what it does and not necessarily how it will do it.
- **Phase II: Implementation:** We define all functions with respect to the description of how to manipulate data. This can be done with algorithms, and implement them easily and effectively with the help of any programming language. Algorithms and Flow Charts are used as design tools at this stage.

## **7. ANALYSIS OF ALGORITHMS**

- Algorithms depend on the organization of data. Analysis of algorithms involves measuring the performance of an algorithm.
- Performance is measured in terms of the following parameters:
  - i) Programmers time complexity – very rarely taken into account as it is to be paid for once.
  - ii) Time complexity – the amount of time taken by an algorithm to perform the task.
  - iii) Space complexity – the amount of memory needed to perform the task.

### **Complexity of algorithms:**

- Algorithms are measured in terms of time and space complexity.
- The time complexity of an algorithm is a measure of how much time is required to execute an algorithm for a given number of inputs and is measured by its rate of growth relative to standard functions.
- The space complexity of an algorithm is a measure of how much storage is required by the algorithm.
- It is possible to design an algorithm that uses more space and less time or less space and more time.
- Computer scientists are interested in minimizing the time complexity of algorithms.
- An algorithm can be characterised by a timing function  $T(n)$ .  $T(n)$  is a measure of how much time is required to execute an algorithm with the given 'n' data values.
- An algorithm  $O(n^2)$  (pronounced as 'oh of n squared'), indicates that its timing function will grow no faster than the square of the number of data values it processes.

### **Space Complexity:**

- Space complexity is the amount of memory required during the execution. Space complexity is the space requirement of an algorithm, can be performed at two different times:
  - i) Compile time
  - ii) Run time
- **Compile time space complexity:** It is defined as the storage requirement of a program at compile time. This can be computed during compile time, it can be determined by summing up the storage size of each variable using declaration statements.

***Compile-time Space Complexity = Space needed at Compile time***

- **Run time space complexity:** If the program is recursive or uses dynamic variables or dynamic data structures, then there is a need to determine space complexity at run time. This dynamic storage size is dependent on some parameters used in program. It is difficult to estimate the memory requirement accurately, as it is also determined by the efficiency of compiler.

***Runtime Space Complexity = Program Space + Data Space + Stack Space***

- *Program Space* is the memory occupied by the program.
- *Data Space* is the memory occupied by the data members, such as variables, constants.
- *Stack Space* is the stack memory needed to save the functions runtime environment while another function is called. Stack space cannot be accurately estimated since it depends on run time call stack, which can depend on the program data. This is important for recursive functions.

#### Time Complexity:

- Time complexity  $T(p)$  is the time taken by a program  $P$ , i.e., the sum of its compile and execution times. This is system dependent.
- Another way to compute time complexity is to count the number of algorithm steps.
- There are two ways to determine the number of steps needed by a program:
  - i) Introduce a new variable, count as global variable with initial value 0. statements to increment count are introduced in the program at appropriate locations, so that each time the statement in the program is executed, the count is incremented. Then measure the runtime of an algorithm by counting the number of steps.
  - ii) Manually compute the number of times each statement will be executed, it is the frequency count. Get the sum of frequency counts of all statements. This sum is the number of steps needed to solve the given problem.
- The best complexity of an algorithm is the function defined by the minimum number of steps taken on any instance of size  $n$ .
- The worst case complexity of an algorithm is the function defined by the maximum number of steps taken on any instance of size  $n$ .
- The average case complexity of an algorithm is the function defined by an average number of steps taken on any instance of size  $n$ .
- Each time of these complexities defines a numerical function time versus size.
- The total time taken by the algorithm, or program is calculated using the sum of the time taken by each of the executable statements in an algorithm or a program.
  - i) The time required for executing it once.
  - ii) The number of times the statement is executed.
- The product of these two statements gives the time requirement for that particular statement.
- Compute the execution time of all executable statements. The simulation of all the execution times is the total time required for that algorithm or program.

#### Big O Notation

- The simplification of efficiency is known as the Big-O Analysis.
- The Big-O notation can be derived from  $f(n)$  using the following steps:
  - i) In each term, set the coefficient of the term to 1.
  - ii) Keep the largest term in the function and discard the others. The terms are ranked from lowest to highest as follows:  $\log_2 n \dots n \dots n \log_2 n \dots n^2 \dots n^3 \dots n^k \dots 2^n \dots n!$
- Ex: calculate Big O notation for  $f(n) = n * (n+1)/2$ 

$$f(n) = n*(n+1)/2 = \frac{1}{2} n^2 + \frac{1}{2} n$$

we first remove all coefficients. This gives us  $n^2 + n$   
 which, after removing the smaller factors, give us  $n^2$   
 which, in Big-O notation, is stated as

$$O(f(n)) = O(n^2)$$

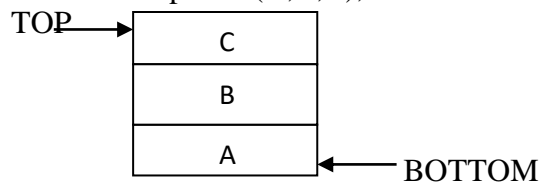
## CHAPTER : II

### 1.CONCEPT OF STACKS & QUEUES

- Stacks and Queues are the two data structures where insert and delete operations are applied at specific ends only. These are also called controlled linear lists.
- A **Stack** is a container of objects that are inserted and removed according to the **Last In First Out (LIFO)** principle.
- In a Stack insertions and deletions are made only at one end, called **Top**.
- Handling function calls in programs are based on stack operations.
- Queue** is a linear data structure in which insertions and deletions are performed at two different ends.
- In Queue, the operations are performed based on **First In First Out(FIFO)** principle.
- Queues are widely used in applications that maintain a list of printing jobs waiting at a network printer.

### 2.STACKS

- A stack is defined as a restricted list where all insertions and deletions are made at only one end, called 'top'.
- Elements may be added or removed from only one end, called the top of the stack.
- Each stack ADT has a data member, top, which points to the top most element in the stack.
- The two basic operations that can be performed on a stack are **push** and **pop**.
- Insertion of an element in to the stack is called push, and deletion of an element from the stack is called pop.
- In stacks, we cannot access data elements from any intermediate positions other than the top position.
- Given a stack  $S=(A1, A2, \dots, An)$ , where  $A1$  is the bottommost element,  $An$  is the topmost element of the stack.
- In our everyday life there are many examples of stacks, a stack of chairs, a stack of books, a stack of dishes etc.
- For example  $S=(A,B,C)$ , where  $A$  is the bottommost element and  $C$  is the topmost element.



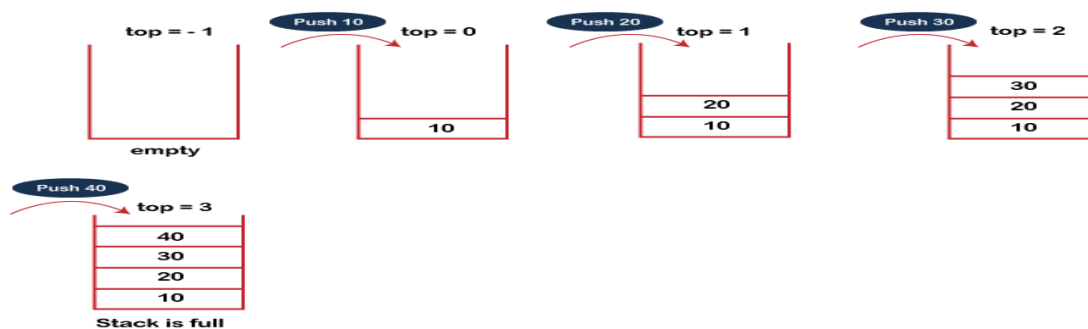
#### **Stack operations:**

- Different stack operations are Stack\_Initialization, Stack\_Empty, Stack\_Full, Push, Pop and Get\_Top. The three basic operations are Push, Pop and Get\_Top.
- Stack\_Initialization operation prepares the stack for use and set it to a vacant state.
- The Stack\_Empty operation tests whether the stack is empty or not. This operation is used when deleting an element from the stack, it also known as **stack underflow**.
- The Stack\_Full operation checks whether the stack is full or not, this is used when inserting an element into the stack. Pushing an element into a full stack is also an error, called **stack overflow**.
- The Get\_Top operation returns the topmost element of the stack.

- Push operation inserts an element on the top of the stack.
- Pop operation deletes an element from the top of the stack.

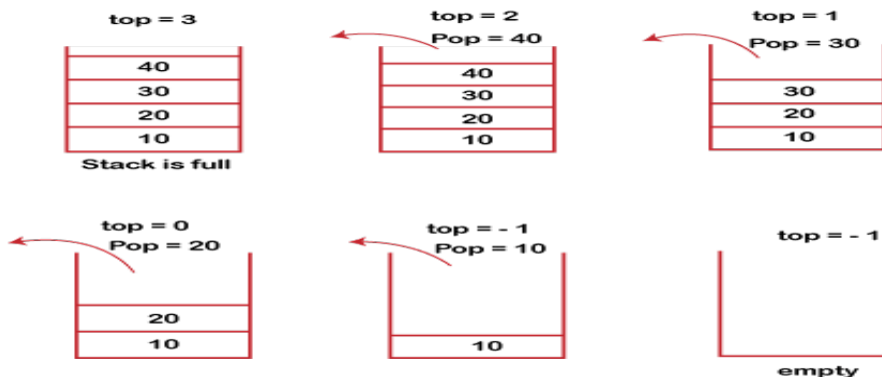
**i) PUSH:** *The steps involved in the PUSH operation is given below:*

- Before inserting an element in a stack, we check whether the stack is full.
- If we try to insert the element in a stack, and the stack is full, then the **overflow** condition occurs.
- When we initialize a stack, we set the value of top as -1 to check that the stack is empty.
- When the new element is pushed in a stack, first, the value of the top gets incremented, i.e., **top=top+1**, and the element will be placed at the new position of the **top**.
- The elements will be inserted until we reach the **max** size of the stack.

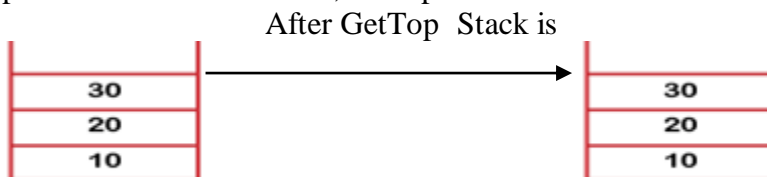


**POP:** *The steps involved in the POP operation is given below:*

- Before deleting the element from the stack, we check whether the stack is empty.
- If we try to delete the element from the empty stack, then the **underflow** condition occurs.
- If the stack is not empty, we first access the element which is pointed by the **top**.
- Once the pop operation is performed, the top is decremented by 1, i.e., **top=top-1**.



**GetTop:** The getTop operation returns the topmost element of the stack. In this only a copy of the topmost element is returned, the top is still set to the same element.



- As both push and pop operations are allowed only at one end of the stack, It retrieves the data in the reverse order in which data is stored.

- Suppose that the order of the operations is Push(A), Push(B), and Push(C). When removing, the order of operations are Pop() for C, Pop() for B and Pop for A.
- Elements are taken out in reverse order of the insertion sequence. So, a stack is called as **Last In First Out(LIFO)** data structure.

### 3.STACK ABSTRACT DATA TYPE

- Any set of elements that are of same data type can be used as data object for stacks.
- For example, Stack of integers, stack of characters, stack of names, stack of employee records, stack of operating system processes etc.
- The following functions comprise a functional definition of a stack:
  1. Create()—creates an empty stack.
  2. Push(ele) – inserts the element ele in to the stack, and returns the modified stack.
  3. Pop() – removes the top most element from the stack, and returns the modified stack.
  4. Get\_Top()—returns the topmost element of the stack.
  5. isEmpty() – returns true if the stack is empty, otherwise returns false.
  6. isFull() – returns true if the stack is full, otherwise returns false.
- When we choose to represent a stack, build the stack as an ADT.
- ADT Stack(element)
  1. Declare Create()  $\rightarrow$  Stack
  2. Push(element)  $\rightarrow$  Stack
  3. Pop()  $\rightarrow$  Stack
  4. getTop()  $\rightarrow$  Element
  5. isEmpty()  $\rightarrow$  Boolean, True or False
  6. isFull()  $\rightarrow$  Boolean, True or False
  7. For all  $S \in \text{Stack}$ ,  $ele \in \text{element}$ . Let
  8. isEmpty(Create())  $\rightarrow$  True
  9. isEmpty(Push(ele))  $\rightarrow$  False
  10. Pop(Create())  $\rightarrow$  Error
  11. Pop(Push(ele))  $\rightarrow$  S
  12. getTop(Create())  $\rightarrow$  error
  13. getTop(Push(ele))  $\rightarrow$  ele
  14. end
  15. end stack
- The six functions with their domains and ranges are declared in 1 to 6 lines, 7 to 14 lines are the set of Axioms.
- To implement the ADT Stack in C++, the operations are implemented as functions to provide data abstraction.

### 4.REPRESENTATION OF STACKS USING SEQUENTIAL ORGANIZATION(ARRAYS)

- A stack can be implemented using both a static data structure (Array) and a dynamic data structure (Linked List).
- The simplest way to represent a stack is by using a one-dimensional array. A Stack implementation using an array is known as a Contiguous Stack.
- An array is used to store an ordered list of elements. A Stack is an ordered collection of elements. Hence it is very simple to implement using an array.

- The only difficulty with an array is its static memory allocation, once declared, the size can not be modified during the runtime. This leads to either poor utilization of the space or inability to accommodate all possible data elements.
- Let Stack[n] be a one dimensional array, one of the two sides of an array can be considered as the top of the stack and the other as the bottom of the stack.
- The elements are stored in stack from the first location onwards. The first element is stored at 0<sup>th</sup> location of the array stack, stack[0], second element is at stack[1], and nth element is at stack[n-1].
- 'top' is an integer variable, which points to the top element in the stack. The initial value of top is '-1', stack is empty. It can hold the elements from index 0 and can grow upto a maximum of n-1.
- The simplest way to implement a Stack ADT is using array. Initialize top to -1 using a constructor to denote an empty stack.
- **Create():** The stack when created is empty. For each and every stack, the topmost element is pointed by top, this top is an integer variable which holds the index of the array.
- The constructor must initialize the stack top to -1, to represent an empty stack
- Each Push operation increments the top by 1, each Pop operation decrements the top by 1.

```
class Stack
{
    int top;
    int stk[MaxSize];
public:
    Stack()
    {
        top=-1;
    }
    void Push(int ele);
    void Pop();
    int isFull();
    int isEmpty();
    void getTop();
};
```

- **isEmpty():** isEmpty() checks whether the stack is empty or not, and returns either true(1) or false(0). When top is -1 then the stack is said to be empty and returns 1 otherwise returns 0.
 

```
if (top== -1)
    return 1;
else
    return 0;
```
- **isFull():** isFull() checks whether the stack is full or not, and returns either true(1) or false(0).
- When top is MaxSize-1 then the stack is said to be full and returns 1 otherwise returns 0.
 

```
if (top==MaxSize-1)
    return 1;
else
    return 0;
```
- **getTop():** it returns the topmost element of the stack, if the stack is not empty. It returns a copy of the topmost element of the stack.
 

```
if(!isEmpty())
    return(stack[top]);
```



4. Reversing a string
5. Processing function calls
6. Parsing of computer programs
7. Simulating recursion
8. In computations such as decimal to binary conversions
9. In backtracking algorithms( optimizations and games).

## **7.EXPRESSION EVALUATION AND CONVERSION**

An arithmetic expression is made of operands, operators and delimiters. Consider the following expression  $x=a/b*c-d$ . Let  $a=1$ ,  $b=2$ ,  $c=3$ ,  $d=4$ ;

1. one of the meanings of expression  $x=(1/2)*(3-4)=-1/2$
2. another is  $x=(1/(2*3))-4 = -23/6$

To avoid more than one meaning of expression, specify the order of operation by using parenthesis.  $x=(a/b)*(c-d)$ . In this also, we still get confused, i.e., whether evaluate  $(a/b)$  first or  $(c-d)$  first. To avoid this use operator priority.

To fix the order of evaluation, assign each operator a priority. Once priorities are assigned, then with in any pairs of parenthesis the operators with highest priority are to be evaluated.

<b><u>Operators</u></b>	<b><u>Priority</u></b>	<b><u>Associativity</u></b>
$\wedge$ , unary $+$ , unary $-$ , $\sim$	1	$\wedge$ is right to left, all other are left to right
$*$ , $/$	2	Left to Right
$+$ , $-$	3	Left to Right
$<$ , $<=$ , $>$ , $>=$ , $==$ , $!=$	4	Left to Right
AND ( $\&\&$ )	5	Left to Right
OR( $\ \ $ )	6	Left to Right

The order of evaluation from Right to Left, or Left to Right is called “Associativity”.

Ex:  $A+B-C \rightarrow (A+B)-C$        $A^B \wedge C \rightarrow A \wedge (B \wedge C)$

## **8.POLISH NOTATION & EXPRESSION CONVERSION**

The polish mathematician Han Lukasiewicz suggested a notation called polish notation, which gives two alternatives to represent an arithmetic expression,

- i) postfix notation
- ii) prefix notation

The conventional way of writing expression is called infix notation, because binary operators occurs between the operands, unary operators precede their operand.

In polish notations, the operations are to be performed is determined by the positions of the operators and operands in the expression. Advantage is that parenthesis is not required in polish notations.

<b><u>Infix</u></b>	<b><u>Prefix</u></b>	<b><u>Postfix</u></b>
Operand operator operand	operator operand operand	operand operand operator
$A+B$	$+AB$	$AB+$
$(A+B)*C$	$*+ABC$	$AB+C*$
$(A+B)*(C-D)$	$*+AB-CD$	$AB+CD-*$

### **Advantages of prefix , postfix expressions**

The prefix and postfix expressions possess many advantages as follows:

1. Parentheses are not used in pre and postfix expressions
2. The priority of operators is no longer relevant

3. The order of evaluation depends on position of the operator, but not on priority and associativity.
4. Expression evaluation is much simpler than infix evaluation.

The prefix notation is called as “polish notation”, the postfix notation is called “suffix notation” and also referred as “reverse polish notation”. The expression in one form can be converted to other two forms:

- |                                 |                                 |
|---------------------------------|---------------------------------|
| 1. Infix to Postfix conversion  | 2. Postfix to Infix conversion  |
| 3. Infix to Prefix conversion   | 4. Prefix to Infix conversion   |
| 5. Prefix to Postfix conversion | 6. Postfix to Prefix conversion |

### **1. Infix to Postfix Conversion**

#### **Procedure:**

1. Scan the infix expression from left to right.
2. A) if the scanned symbol is ‘(’, push it on to the stack.  
B) if the scanned symbol is ‘)’, pop the symbols from stack and place them in postfix expression till ‘)’.
- C) if the scanned symbol is operand, place it in postfix expression
- D) if the scanned symbol is operator, if its priority is  $\leq$  to the priority of stack operator, then pop the operator from the stack and place it in postfix expression and push the scanned operator on to the stack. If the priority  $>$  the priority of stack operator, then push it on to the stack.
3. Repeat 2<sup>nd</sup> step till the end of the infix expression.
4. pop all the symbols from the stack and place them in postfix expression and display the postfix.

#### **Ex: (A+B)\*(C-D)**

Scanned Symbol	Stack Contents	Postfix Expression
(	(	
A	(	A
+	(+	A
B	(+	AB
)		AB+
*	*	AB+
(	*(	AB+
C	*(	AB+C
-	*(-	AB+C
D	*(-	AB+CD
)	*	AB+CD-
End of infix	pop all symbols from stack and place it in postfix	<b><u>AB+CD-*</u></b>
<b><u>(A+B)*(C-D) postfix is AB+CD-*</u></b>		

**Ex:  $(A+B)*C$** 

Scanned Symbol	Stack Contents	Postfix Expression
(	(	
A	(	A
+	(+	A
B	(+	AB
)		AB+
*	*	AB+
C	*	AB+C
End of infix pop all symbols from stack&place it in postfix		<u><b>AB+C*</b></u>
<u><b><math>(A+B)*C</math> in postfix is <math>AB+C*</math></b></u>		

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

Scanned Symbol	Stack Contents	Postfix Expression
A		A
^	^	A
B	^	AB
*	*	AB^
C	*	AB^C
-	-	AB^C*
C	-	AB^C*C
+	+	AB^C*C-
D	+	AB^C*C-D
/	+/	AB^C*C-D
A	+/	AB^C*C-DA
/	+/	AB^C*C-DA/
(	+/ (	AB^C*C-DA/
E	+/ (	AB^C*C-DA/E
+	+/ (+	AB^C*C-DA/E
F	+/ (+	AB^C*C-DA/EF
)	+/	AB^C*C-DA/EF+
End of infix pop all symbols from stack and place it in postfix		<b>AB^C*C-DA/EF+/+</b>
<u><b><math>A^B*C-C+D/A/(E+F)</math> in postfix is <math>AB^C*C-DA/EF+/+</math></b></u>		

**2.Postfix to infix conversion****Procedure:**

1. Scan the postfix expression from left to right
2. If the scanned symbol is an operand, then push it on to the stack.
3. If the scanned symbol is an operator, then pop two symbols from stack, create an expression by placing the operator between the two popped symbols, use parentheses if need and push that expression on to the stack

4. Repeat 2<sup>nd</sup> and 3<sup>rd</sup> steps till the end of the postfix expression  
 5. pop the stack to get infix expression.

Ex:  $AB+C^*$

Scanned symbol	Stack	t1	t2	expression
A	A			
B	A B			
+		A	B	(A+B)
	(A+B)			
C	(A+B) C			
*		(A+B)	C	(A+B)*C
	(A+B)*C			
End of postfix	pop stack to get infix expression			
<u>AB+C* in Infix is (A+B)*C</u>				

Ex:  $AB^AC^*C-DA/EF+/+$

Scanned symbol	Stack	t1	t2	expression
A	A			
B	A B			
^		A	B	A^B
	A^B			
C	A^B C			
*		A^B	C	A^B*C
	A^B*C			
C	A^B*C C			
-		A^B*C	C	A^B*C-C
	A^B*C-C			
D	A^B*C-C D			
A	A^B*C-C D A			
/		D	A	(D/A)
	A^B*C-C (D/A)			
E	A^B*C-C (D/A) E			
F	A^B*C-C (D/A) E F			
+	A^B*C-C (D/A)	E	F	(E+F)
	A^B*C-C (D/A) (E+F)			
/	A^B*C-C	(D/A)	(E+F)	(D/A)/(E+F)
	A^B*C-C (D/A)/(E+F)			
+		A^B*C-C (D/A)/(E+F)		A^B*C-C+D/A/(E+F)
	A^B*C-C+D/A/(E+F)			
End of postfix	pop expression from stack to get infix expression			
AB^C*C-DA/EF+/+ infix is A^B*C-C+D/A/(E+F)				

Ex: AB+CD-\*

Scanned symbol	Stack	t1	t2	expression
A	A			
B	AB			
+		A	B	(A+B)
	(A+B)			
C	(A+B) C			
D	(A+B) C D			
-	(A+B)	C	D	(C-D)
	(A+B) (C-D)			
*		(A+B)	(C-D)	(A+B)*(C-D)
	(A+B)*(C-D)			
end of postfix pop expression from stack to get infix expression				
<u><b>AB+CD-* in infix is (A+B)*(C-D)</b></u>				

**3.infix to prefix conversion****Procedure:** It requires two stacks, stack1, stack2(prefix stack)

1. Scan the infix expression from Right to Left.
2. a) if the scanned symbol is an operand then push it on to the prefix stack, stack2  
b) if the scanned symbol is ')' then push it on to stack1.
- c) if the scanned symbol is '(' then pop all symbols from stack1 and push them on to the stack2 until '('
- d) if the scanned symbol is an operator, if its priority is  $\geq$  stack1 operator priority then push it on to stack1. Otherwise if its priority is  $<$  stack1 operator priority then pop from stack1 and push it on to stack2 repeatedly, push the scanned operator on stack1.
3. Repeat 2<sup>nd</sup> step until the end of the infix expression
4. pop all symbols from stack1 and push them on to stack2.
5. to get prefix expression pop all symbols from stack2.

Ex: (A+B)\*C

Scanned symbol	stack1	prefix stack (stack2)
C		C
*	*	C
)	* )	C
B	* )	C B
+	* ) +	C B
A	* ) +	C B A
(	*	C B A +
End of infix pop all symbols from stack1 and push on to stack2		C B A + *
<b>To get prefix pop all symbols from stack2, expression is</b>		<b>*+ABC</b>

Ex:  $(A+B)*(C-D)$ 

Scanned symbol	stack1	prefix stack (stack2)
)	)	
D	)	D
-	) -	D
C	) -	D C
(	EMPTY	D C -
*	*	D C -
)	* )	D C -
B	* )	D C - B
+	* ) +	D C - B
A	* ) +	D C - B A
(	*	D C - B A +
End of infix	pop all , push on to stack2	D C - B A + *
<b>Pop all symbols from Stack2 to get prefix expression</b>		<b>*+AB-CD</b>

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

Scanned symbol	stack1	prefix stack (stack2)
)	)	
F	)	F
+	) +	F
E	) +	F E
(		F E +
/	/	F E +
A	/	F E + A
/	//	F E + A
D	//	F E + A D
+	+	F E + A D //
C	+	F E + A D // C
-	+ -	F E + A D // C
C	+ -	F E + A D // C C
*	+ - *	F E + A D // C C
B	+ - *	F E + A D // C C B
^	+ - * ^	F E + A D // C C B
A	+ - * ^	F E + A D // C C B A
End of infix	pop all, push on to stack2	F E + A D // C C B A ^ * - +
<b>To get prefix, pop all from stack2</b>		<b>+ -*^ABCC//DA+EF</b>

**4.Prefix to Infix expression conversion**

Procedure:

- 1.Scan the prefix expression from right to left
- 2.If the scanned symbol is an operand, then push it on to the stack.

3. If the scanned symbol is an operator, then pop two symbols from stack, create it as an expression by placing the operator between popped symbols and push the expression on to the stack.

4. Repeat 2<sup>nd</sup> and 3<sup>rd</sup> steps till the end of the prefix expression.

5. Pop the stack to get infix expression

Ex:  $*+ABC$

Scanned symbol	stack	t1	t2	expression
C	C			
B	C B			
A	C B A			
+	C	A	B	(A+B)
	C (A+B)			
*		(A+B)	C	(A+B)*C
	(A+B)*C			
<b>Pop the stack to get infix expression (A+B)*C</b>				

Ex:  $+-*^ABCC//DA+EF$

S.symbol	stack	t1	t2	expression
F	F			
E	F E			
+		E	F	(E+F)
	(E+F)			
A	(E+F) A			
D	(E+F) A D			
/	(E+F)	D	A	D/A
	(E+F) D/A			
/		D/A	(E+F)	D/A/(E+F)
	D/A/(E+F)			
C	D/A/(E+F) C			
C	D/A/(E+F) C C			
B	D/A/(E+F) C C B			
A	D/A/(E+F) C C B A			
^	D/A/(E+F) C C	A	B	A^B
	D/A/(E+F) C C A^B			
*	D/A/(E+F) C	A^B	C	A^B*C
	D/A/(E+F) C A^B*C			
-	D/A/(E+F)	A^B*C	C	A^B*C-C
	D/A/(E+F) A^B*C-C			
+		A^B*C-C	D/A/(E+F)	A^B*C-C+D/A/(E+F)
	A^B*C-C+D/A/(E+F)			
End of prefix	<b>pop the stack to get infix A^B*C-C+D/A/(E+F)</b>			

Ex:  $*+AB-CD$ 

Scanned symbol	stack	t1	t2	expression
D	D			
C	D C			
-		C	D	(C-D)
	(C-D)			
B	(C-D) B			
A	(C-D) B A			
+	(C-D)	A	B	(A+B)
	(C-D) (A+B)			
*		(A+B)	(C-D)	(A+B)*(C-D)
	(A+B)*(C-D)			
End of prefix	<b>pop stack to get infix expression</b>		<b>(A+B)*(C-D)</b>	

**5.prefix to postfix conversion****Procedure:**

- 1.Scan the prefix from right to left
2. If the scanned symbol is an operand, push it on to the stack
- 3.if the scanned symbol is an operator, pop two symbols from stack and create expression by placing operator after two popped symbols and push it on to the stack.
- 4.Repeat 2<sup>nd</sup> and 3<sup>rd</sup> steps till the end of the prefix
5. pop the stack to get postfix

Ex:  $*+ AB-CD$ 

S.symbol	stack	t1	t2	expression
D	D			
C	D C			
-		C	D	CD-
	CD-			
B	CD- B			
A	CD- B A			
+	CD-	A	B	AB+
	CD- AB+			
*		AB+	CD-	AB+CD-*
	AB+CD-*			
End of prefix	<b>To get postfix pop the stack</b>		<b>AB+CD-*</b>	

Ex:  $*+ABC$ 

S.symbol	stack	t1	t2	expression
C	C			
B	C B			

A	C B A			
+	C	A	B	AB+
	C AB+			
*		AB+	C	AB+C*
	AB+C*			
End of prefix	<b>pop stack to get postfix</b>	<b>AB+C*</b>		

Ex: +-\*^ABCC//DA+EF

S.symbol	stack	t1	t2	expression
F	F			
E	F E			
+		E	F	EF+
	EF+			
A	EF+ A			
D	EF+ A D			
/	EF+	D	A	DA/
	EF+ DA/			
/		DA/	EF+	DA/ EF+ /
	DA/EF+ /			
C	DA/EF+ / C			
C	DA/ EF+ / C C			
B	DA/EF+ / C C B			
A	DA/EF+ / C C B A			
^	DA/EF+ / C C	A	B	AB^
	DA/EF+ / C C AB^			
*	DA/EF+ / C	AB^	C	AB^C*
	DA/EF+ / C AB^C			
-	DA/EF+ /	AB^C*	C	AB^C*C-
	DA/EF+ / AB^C*C-			
+		AB^C*C-	DA/EF+ /	AB^C*C-DA/EF+ / +
	AB^C*C-DA/EF+ / +			
End of prefix	<b>pop the stack to get postfix</b>	<b>AB^C*C-DA/EF+ / +</b>		

## 6.Postfix to Prefix conversion

### **Procedure:**

- 1.Scan the postfix expression from left to right.
- 2.if the scanned symbol is an operand push it on to the stack
- 3.if the scanned symbol is an operator, then pop two symbols from stack, create expression by placing scanned operator in front of the operands, and push the expression on to the stack.
- 4.Repeat 2<sup>nd</sup> and 3<sup>rd</sup> steps till the end of the postfix expression
5. Pop the stack to get prefix expression .

Ex: AB+CD-\*

S.symbol	stack	t1	t2	expression
A	A			
B	A B			
+		A	B	+AB
	+AB			
C	+AB C			
D	+AB C D			
-	+AB	C	D	-CD
	+AB -CD			
*		+AB	-CD	*+AB-CD
	*+AB-CD			
End of postfix to get prefix pop the stack <b>*+AB-CD</b>				

Ex: AB^C\*C-DA/EF+/+

S.symbol	stack	t1	t2	expression
A	A			
B	B			
^		A	B	^AB
	^AB			
C	^AB C			
*		^AB	C	*^ABC
	*^ABC			
C	*^ABC C			
-		*^ABC	C	-*^ABCC
	-*^ABCC			
D	-*^ABCC D			
A	-*^ABCC D A			
/	-*^ABCC	D	A	/DA
	-*^ABCC /DA			
E	-*^ABCC /DA E			
F	-*^ABCC /DA E F			
+	-*^ABCC /DA	E	F	+EF
	-*^ABCC /DA +EF			
/	-*^ABCC	/DA	+EF	//DA+EF
	-*^ABCC //DA+EF			
+		-*^ABCC	//DA+EF	+-*^ABCC//DA+EF
	+-*^ABCC//DA+EF			
End of postfix To get prefix pop the stack <b>+-*^ABCC//DA+EF</b>				

Ex:  $AB+C^*$

S.symbol	stack	t1	t2	expression
A	A			
B	A B			
+		A	B	+AB
	+AB			
C	+AB C			
*		+AB	C	*+ABC
	*+ABC			
End of postfix	<b>to get prefix pop the stack</b>		<b>*+ABC</b>	

### 9.EVALUATION OF POSTFIX EXPRESSION

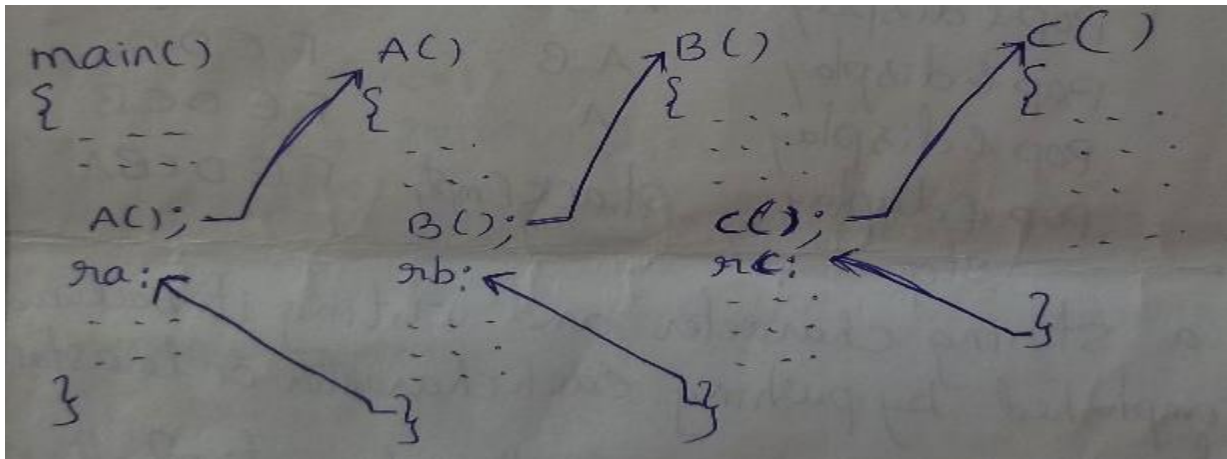
1. Scan the postfix expression from left to right
2. If the scanned symbol is an operand, push it on to the stack.
3. If the scanned symbol is an operator (O), then pop two values from the stack as t2 and t1 respectively, form as an expression (t1 O t2), the result of the expression is pushed on to the stack.
4. Continue 2<sup>nd</sup> and 3<sup>rd</sup> steps till the end of the postfix expression.
5. To get final result, pop the stack.

Ex:  $AB+C^*$  let  $A=5$ ,  $B=4$ ,  $C=3$  then

Scanned symbol	Stack	t1	t2	expression
A	5			
B	5 4			
+		5	4	$(5+4)=9$
	9			
C	9 3			
*		9	3	$9*3=27$
	27			
End of postfix	pop stack to get result = 27			

### 10. PROCESSING OF FUNCTION CALLS

- One natural application of stacks, which arises in computer programming is the processing of function calls and their termination.
- The program must remember where the call was made, so that it can return to there after completing the function.
- Suppose we have three functions, A, B, and C and one main() function.
- Let the main() calls function A, A calls B, B calls C.
- Then B will not have finished its work until C has finished and returned. Function A will not finished its work until B has finished and returned, similarly main() will not finished its work until A has finished and returned.
- This sequence shows LIFO technique. main() is the first to start work and main() is the last to be finished.
- In this case, the return addresses ra, rb, rc are stored in stack data structure.



### 11. REVERSING A STRING WITH STACK

- The LIFO property of the stack access guarantees the reversal of a string.
- Suppose, the sequence ABCDEF is to be reversed, with a stack, simply scans the sequence, pushing each element on to the stack, until the end of the sequence.
- The stack is then popped repeatedly, each popped element sent to the output, until the stack is empty.

<u>INPUT SEQUENCE</u>	<u>ACTION</u>	<u>STACK</u>	<u>OUTPUT</u>
ABCDEF	Push A	A	
BCDEF	Push B	A B	
CDEF	Push C	A B C	
DEF	Push D	A B C D	
EF	Push E	A B C D E	
F	Push F	A B C D E F	
End Of String	Pop & Display	A B C D E	F
	Pop & Display	A B C D	FE
	Pop & Display	A B C	FED
	Pop & Display	A B	FEDC
	Pop & Display	A	FEDCB
	Pop & Display	Stack is empty	FEDCBA
	Stop		

#### **ABCDEF in reverse is FEDCBA**

- Reading a string character and writing it backward can be accomplished by pushing each character on to the stack as it is read.
- When the string is finished, pop the characters of the stack, and they will come out in the reverse order.

### 12. RECURSION

- A function calling itself is called recursion.
- In C++, a function can call itself, i.e., one of the statement of that function calls itself. Such functions are called recursive functions.
- Recursion is a technique that allows us to break down a problem into one or more sub problems that are similar in the form to the original problem.

Ex: //factorial of a given number

```
#include<iostream.h>
long int factorial(unsigned int n)
{
    if(n==0)
        return 1;
    else
        return(n*factorial(n-1));
}
void main()
{ int n;
  long int f;
  cin>>n;
  f=factorial(n);
  cout<<n<<" factorial= "<<f;
}
```