# PYTHON FOR COMPUETR SCIENCE

## DATA STRUCTURE

## FOR

## CLASS – XII

KAPIL SEHGAL

KAPIL SEHGAL (P.G.T. (CS) K.V. NO2 Delhi Cantt,

# ELEMENTARY DATA REPRESENTATION

Data can be in the form of raw data, data items and data structure.

Raw Data – Raw data are the facts or they are simply values or set of values

Data Item – Data items represents single unit of values of certain types.

Data Structure – A Data Structure is a named group of data of different data types which can be processed as a single unit. A data structure has well-defined operation behavior and properties. Record and Array is the example of the data structure.

# DIFFERENT DATA STRUCTURES

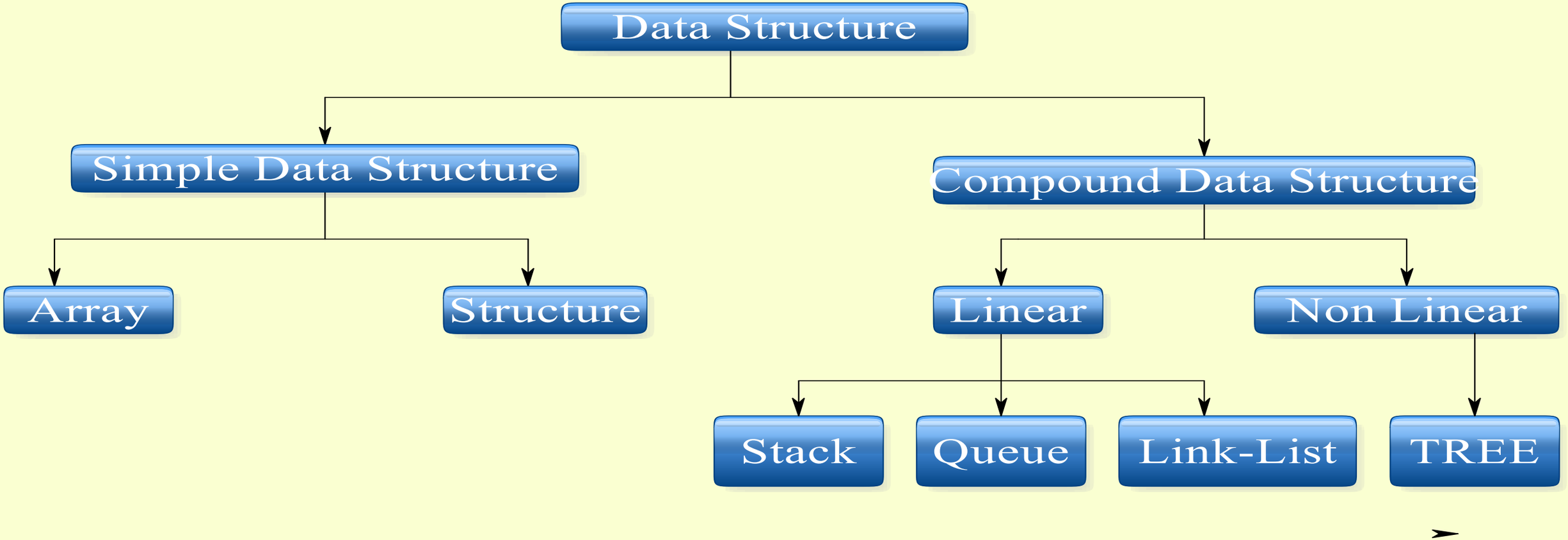
Data Structure is very important in Computer System. As these not only allow the user to combine various data types in a group but also allow processing of the group as a single unit. Data structure be in two types.

## Simple Data Structure

These data structure are normally built from primitive data types like integer, real char, Boolean etc These are
for ex.

LIST
TUPLE

# DATA STRUCTURE - ARCHITECTURE

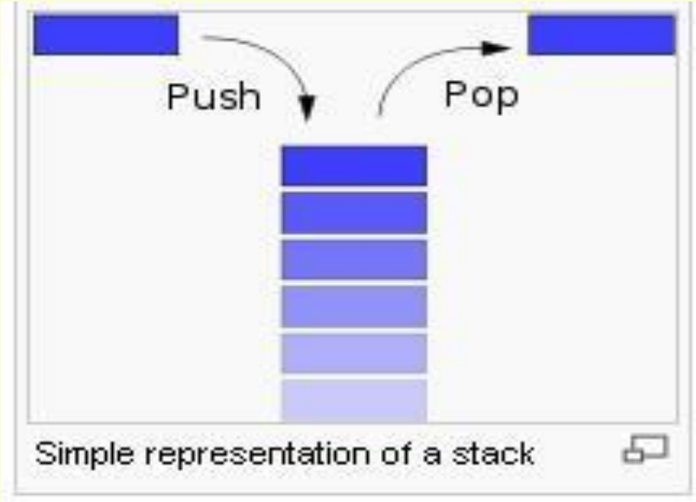



KAPIL SEHGAL

# STACK

Stack is a Linear Data Structure, in which elements are processed in the LAST IN FIRST OUT (LIFO) fashion.
In other words Stack is a linear list in which elements are add and removed at single end.
The process of add element into stack is called PUSH.
The process of delete element from stack is called POP

**The Recently added and that element which will remove first is called TOP element**




KAPIL SEHGAL

# STACK

A stack is an abstract data type ADT, commonly used in most programming languages. It is named stack as it behaves like a real-world stack, for example – deck of cards or pile of plates etc

A real-world stack allows operations at one end only. For example, we can place or remove a card or plate from top of the stack only. Likewise, Stack ADT allows all data operations at one end only. At any given time, We can only access the top element of a stack.

KAPIL SEHGAL

# STACK

This feature makes it LIFO data structure. LIFO stands for Last-in-first-out. Here, the element which is placed inserted or added last, is accessed first. In stack terminology, insertion operation is called PUSH operation and removal operation is called POP operation.

## Stack Representation

# STACK OPERATIONS

## Stack Basic Operations

Stack operations may involve initializing the stack, using it and then de-initializing it. Apart from these basic stuffs, a stack is used for the following two primary operations −

(a) **Push** : Storing an element at top of stack.

(b) **Pop** : Removing an element from stack.

(c) **Peak**: Get the top element of stack without removing it.

(d) **isFull**: Check if stack is full or not.

(e) **isEmpty**: Check if stack is empty or not.

# PEAK OPERATION

Peak operation returns the peak (top) element of the stack i.e. This operation return that element which is added latest.

Algorithm of peek function –

```
begin procedure peek

   return stack[top]

end procedure
```

## Stack peak() Operations

```
def peak(stack):
    return stack[top]
```

**Here top will be global variable**

# ISFULL() OPERATION

isFull() operation returns the True if the stack is full and return False if stack is not Full

**When stack is full : When top reaches at Maxsize**

Algorithm of isfull function –

```
begin procedure isfull

   if top equals to MAXSIZE
      return true
   else
      return false
   endif

end procedure
```

## Stack isfull() Operations

```
def isFull(stack):
    if (top==maxsize-1):
        return True
    else:
        return False
```

**Here top and maxsize will be global variable**

# ISEMPTY() OPERATION

isEmpty() operation returns the True if the stack is Empty (i.e. No element left in stack) and return False if stack is not Empty (i.e. at least one element remain in stack)

**When Stack is Empty : When Top will be at 0.**

**isempty**

Algorithm of isempty function –

```
begin procedure isempty

   if top less than 1
      return true
   else
      return false
   endif
```

## Stack isEmpty() Operations

```
def isEmpty(stack):
    if (top==-1):
        return True
    else:
        return False
```

**Here top and maxsize will be global variable**

# OVERFLOW AND UNDERFLOW

## OVERFLOW

**After full of stack, we inserted an element (i.e. applied push operation) at the top of stack then "Overflow" occurred**

## UNDERFLOW

**When Stack is Empty, we tried to remove element (i.e. applied pop operation) from the stack then "Underflow" occurred**

## Traverse Function

```
def Traverse(stack):
    if (isEmpty(stack)):
        print("Stack is Empty ")
    else:
        for i in stack:
            print(i,end="  ")
```

# PUSH() OPERATION

The process of putting a new data element onto stack is known as PUSH Operation. Push operation involves series of steps −

(a) **Step - 1** : Check if Stack is full

(b) **Step - 2** : If Stack is full produce error and exit.

(c) **Step – 3** : If Stack is not full, increment the top to the point next empty space

(d) **Step – 4** : Add data element to the stack location, where top is pointing

(e) **Step – 5** : Return Success

# PUSH() ALGORITHM

## Algorithm for PUSH operation

A simple algorithm for Push operation can be derived as follows –

```
begin procedure push: stack, data

   if stack is full
      return null
   endif

   top ← top + 1

   stack[top] ← data

end procedure
```

## PUSH Function

```
def Push(stack):
    global top
    if (isFull(stack)):
        print("\n Stack is OverFlow \n ")
    else:
        n=int(input("Enter An Element to Push "))
        top=top+1
        stack.append(n)
```

KAPIL SEHGAL

# POP() OPERATION

Accessing the content while removing it from stack, is known as pop operation.Pop operation involves series of steps –

(a) **Step - 1** : Check if Stack is empty

(b) **Step - 2** : If Stack is empty produce error and exit.

(c) **Step – 3** : If Stack is not empty, access the element from top of the stack

(d) **Step – 4** : Remove top element from memory

(e) **Step – 5** : Decrease the value of top by 1 and return success

# POP() ALGORITHM

## Algorithm for POP operation

A simple algorithm for Pop operation can be derived as follows −

```
begin procedure pop: stack

   if stack is empty
      return null
   endif

   data ← stack[top]

   top ← top - 1

   return data

end procedure
```

## POP Function

```
def Pop(stack):
    global top
    if (isEmpty(stack)):
        print("\n Stack is UnderFlow  \n")
    else:
        n=stack[top]
        print("Removed Element  ",n)
        top=top-1
        stack.pop()
```

# ENTIRE STACK PROGRAM WITH MAIN

```
global maxsize
global top
top=-1
****************** Main Program ***************
stack=[]
cont=0
while True:
    while True:
        print("1. Stack Push Operation ")
        print("2. Stack Pop Operation ")
        print("3. Show Peak / Top Position ")
        print("4. Traverse / Show Stack ")
        print("5. Exit ")
        ch=int(input("Enter Choice "))
        if (ch >= 1 and ch<=5):
            break
    if (ch == 1):
        Push(stack)
    if (ch == 2):
        Pop(stack)
    if (ch == 3):
        print("\n Peak Position ",Peak()+1, " \n ")
    if (ch == 4):
        Traverse(stack)
    if (ch == 5):
        cont=1
    if (cont == 1):
        break
```

# APPLICATION OF STACK

## Infix, Postfix and Prefix notations

**Infix**, Postfix and Prefix notations are three different but equivalent ways of writing **expressions**. It is easiest to demonstrate the differences by looking at examples of operators that take two operands. **Infix** notation: X + Y. Operators are written in-between their operands. This is the usual way we write **expressions**

### Infix Notations

Where operator is placed between two operands. Ex.

A+B

Here A and B are operand and + is operator

### Postfix Notations

Where operator is placed after two operands. Ex.

AB+

Here A and B are operand and + is operator

### Prefix Notations

Where operator is placed between two operands. Ex.

+ AB

Here A and B are operand and + is operator

# INFIX, POSTFIX AND PREFIX NOTATIONS

## Simple Example

| Infix Expression | Prefix Expression | Postfix Expression |
|---|---|---|
| A + B | + A B | A B + |
| A + B * C | + A * B C | A B C * + |

## Bracket Example

| Infix Expression | Prefix Expression | Postfix Expression |
|---|---|---|
| (A + B) * C | * + A B C | A B + C * |

## More Example

| Infix Expression | Prefix Expression | Postfix Expression |
|---|---|---|
| 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 * + |
| A + B + C + D | + + + A B C D | A B + C + D + |

## Infix to Postfix

A + B * C
A + BC*
ABC*+

(A + B) * C
AB+ * C
AB+C*

(A + B) * (C+D)
AB+ * (C+D)
AB+ * CD+
AB+CD+*

## Infix to Prefix

A + B * C
A + *BC
+ A *BC

(A + B) * C
+AB * C
*+ABC

(A + B) * (C+D)
+AB * (C+D)
+AB * +CD
*+AB+CD

KAPIL SEHGAL

# ALGORITHM : CONVERSION FROM INFIX TO POSTFIX

Algorithm to convert Infix To Postfix
Let, X is an arithmetic expression written in infix notation. This algorithm finds the equivalent postfix expression Y.
Push “(“onto Stack, and add “)” to the end of X.
Scan X from left to right and repeat Step 3 to 6 for each element of X until the Stack is empty.
If an operand is encountered, add it to Y.
If a left parenthesis is encountered, push it onto Stack.
If an operator is encountered ,then:

Repeatedly pop from Stack and add to Y each operator (on the top of Stack) which has the same precedence as or higher precedence than operator.
Add operator to Stack.
[End of If]

If a right parenthesis is encountered ,then:

Repeatedly pop from Stack and add to Y each operator (on the top of Stack) until a left parenthesis is encountered.
Remove the left Parenthesis.
[End of If]
[End of If]

END.

**KAPIL SEHGAL**

# CONVERSION FROM INFIX TO POSTFIX EXAMPLE

## Infix Expression : A * (B+C) * D

| Symbol No | Symbol | Stack | Postfix Expression |
|---|---|---|---|
| 1 | A | ( | A |
| 2 | * | (* | A |
| 3 | ( | (*( | A |
| 4 | B | (*( | AB |
| 5 | + | (*(+ | AB |
| 6 | C | (*(+ | ABC |
| 7 | ) | (* | ABC+ |
| 8 | * | (* | ABC+* |
| 9 | D | (* | ABC+*D |
| 10 | ) | EMPTY | ABC+*D* |

KAPIL SEHGAL

# CONVERSION FROM INFIX TO POSTFIX EXAMPLE

## Infix Expression : (A+B)* (C+D)

( A + B ) * ( C + D ) )

1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12

| Symbol No | Symbol | Stack | Postfix Expression |
|---|---|---|---|
| 1 | ( | (( | |
| 2 | A | (( | A |
| 3 | + | ((+ | A |
| 4 | B | ((+ | AB |
| 5 | ) | ( | AB+ |
| 6 | * | (* | AB+ |
| 7 | ( | (*( | AB+ |
| 8 | C | (*( | AB+C |
| 9 | + | (*(+ | AB+C |
| 10 | D | (*(+ | AB+CD |
| 11 | ) | (* | AB+CD+ |
| 12 | ) | Empty | AB+CD+* |

KAPIL SEHGAL

# ALGORITHM : CONVERSION FROM INFIX TO PREFIX

Algorithm to convert Infix To Prefix

**********  **Reverse Infix Expression**  **********

Let, X is an arithmetic expression written in infix notation. This algorithm finds the equivalent postfix expression Y.
Push “(“onto Stack, and add “)” to the end of X.
Scan X from left to right and repeat Step 3 to 6 for each element of X until the Stack is empty.
If an operand is encountered, add it to Y.
If a left parenthesis is encountered, push it onto Stack.
If an operator is encountered ,then:

Repeatedly pop from Stack and add to Y each operator (on the top of Stack) which has the same precedence as or higher precedence than operator.
Add operator to Stack.
[End of If]

If a right parenthesis is encountered ,then:

Repeatedly pop from Stack and add to Y each operator (on the top of Stack) until a left parenthesis is encountered.
Remove the left Parenthesis.
[End of If]
[End of If]

**** **Reverse Postfix Expression**  ******

END.

**KAPIL SEHGAL**

# CONVERSION FROM INFIX TO PREFIX EXAMPLE

**Infix Expression : A * ( B – C ) ^ E / F + G**

**Reverse Infix Expression and right parenthesis: G + F / E ^ ( C – B ) * A )**

| Symbol No | Symbol | Stack | Postfix Expression |
|---|---|---|---|
| 1 | G | ( | G |
| 2 | + | (+ | G |
| 3 | F | (+ | GF |
| 4 | / | (+/ | GF |
| 5 | E | (+/ | GFE |
| 6 | ^ | (+/^ | GFE |
| 7 | ( | (+/^( | GFE |
| 8 | C | (+/^( | GFEC |
| 9 | - | (+/^(- | GFEC |
| 10 | B | (+/^(- | GFECB |
| 11 | ) | (+/^ | GFECB- |
| 12 | * | (+* | GFECB-^/ |
| 13 | A | (+* | GFECB-^/A |
| 14 | ) | EMPTY | GFECB-^/A*+ |

**Again reverse this expression : (Prefix Expression) + * A / ^ - BCEFG**

KAPIL SEHGAL

# ALGORITHM :

**Algorithm:**

Iterate the given expression from left to right, one character at a time

If a character is operand, push it to stack.

If a character is an operator,

pop operand from the stack, say it's s1.

pop operand from the stack, say it's s2.

perform **(s2 operator s1)** and push it to stack.

Once the expression iteration is completed, initialize the result string and pop out from the stack and add it to the result.

Return the result.

KAPIL SEHGAL

# CONVERSION FROM POSTFIX TO INFIX

## Postfix Expression : ABC*+

| Symbol No | Symbol | Stack | Intermediate Operation |
|---|---|---|---|
| 1 | A | A | |
| 2 | B | A , B | |
| 3 | C | A, B, C | |
| 4 | * | A, B*C | Pop S1='C'<br>Pop S2='B'<br>Push = 'B*C' |
| 5 | + | A+B*C | S1='B*C'<br>S2= 'A'<br>PUSH = A+B*C |

## Postfix Expression : AB+C*

| Symbol No | Symbol | Stack | Intermediate Operation |
|---|---|---|---|
| 1 | A | A | |
| 2 | B | A , B | |
| 3 | + | A+B | Pop S1='B'<br>Pop S2='A'<br>Push 'A+B' |
| 4 | C | A+B,C | |
| 5 | * | A+B*C | S1='C'<br>S2= 'A+B'<br>PUSH = A+B*C |

# CONVERSION FROM POSTFIX TO INFIX

## Postfix Expression : AB+CD+*

| Symbol No | Symbol | Stack | Intermediate Operation |
|---|---|---|---|
| 1 | A | A | |
| 2 | B | A , B | |
| 3 | + | A+B | S1='B'<br>S2= 'A'<br>PUSH = A+B |
| 4 | C | A+B , C | |
| 5 | D | A+B, C, D | |
| 6 | + | A + B , C+D | S1='D'<br>S2= 'C'<br>PUSH = C+D |
| 7 | * | (A+B)*(C+D) | S1= 'C+D'<br>S2= 'A+B'<br>PUSH (A+B)*(C+D) |

KAPIL SEHGAL

# EVALUATION OF POSTFIX EXPRESSION

## 2,5,3,-,+

| Symbol No | Symbol | Stack | Intermediate Operation |
|---|---|---|---|
| 1 | 2 | 2 | |
| 2 | 5 | 2 , 5 | |
| 3 | 3 | 2, 5, 3 | |
| 4 | - | 2, 2 | Pop S1=3<br>Pop S2=5<br>Push = '5-3=2' |
| 5 | + | 4 | Pop S1=2<br>Pop S2= 2<br>PUSH = 4 |

## 6,9,+,3,/

| Sym9ol No | Sym9ol | Sta3k | Intermediate Operation |
|---|---|---|---|
| 1 | 6 | 6 | |
| 2 | 9 | 6 , 9 | |
| 3 | + | 15 | Pop S1=9<br>Pop S2=6<br>Push 15 |
| 4 | 3 | 15,3 | |
| 5 | / | 5 | S1=3<br>S2= 15<br>PUSH = 15/ 3 |

## 5,20,10,+,*,3,/

| Sym9ol No | Sym9ol | Sta3k | Intermediate Operation |
|---|---|---|---|
| 1 | 5 | 5 | |
| 2 | 20 | 5,20 | |
| 3 | 10 | 5,20,10 | |
| 4 | + | 5,30 | Pop S1=10<br>Pop S2=20<br>Push = 30 |
| 5 | * | 150 | Pop S1=30<br>Pop S2= 5<br>PUSH = 150 |
| 6 | 3 | 150,3 | |
| 7 | / | 50 | Pop S1=3<br>Pop S2= 150<br>PUSH = 150/3=50 |

KAPIL SEHGAL

# EVALUATION OF POSTFIX EXPRESSION

## Postfix Expression : 2,5,+,8,4,+,*

| Symbol No | Symbol | Stack | Intermediate Operation |
|---|---|---|---|
| 1 | 2 | 2 | |
| 2 | 5 | 2 , 5 | |
| 3 | + | 7 | S1='5'<br>S2= '2'<br>PUSH = 7 |
| 4 | 8 | 7 , 8 | |
| 5 | 4 | 7,8,4 | |
| 6 | + | 7,12 | S1=4<br>S2= 8<br>PUSH = 12 |
| 7 | * | 84 | S1= 12<br>S2= 7<br>PUSH 84 |

KAPIL SEHGAL

# EVALUATION OF POSTFIX EXPRESSION : BOOLEAN VALUE

## Postfix Expression : True, False, NOT, AND, True, True, AND,

| Step | Input Symbol | Action Taken | Stack status | Output |
|---|---|---|---|---|
| 1. | True | Push | True (↑ - top)<br>↑ | |
| 2. | False | Push | True, False<br>↑ | |
| 3. | NOT | Pop (1 element)<br>Push result (True) | True<br>↑<br>True, True<br>↑ | NOT False = True |
| 4. | AND | Pop (2 element)<br>Push result (True) | ↑ (empty stack)<br>True<br>↑ | True AND True = True |
| 5. | True | Push | True, True<br>↑ | |
| 6. | True | Push | True, True, True<br>↑ | |
| 7. | AND | Pop (2 element)<br>Push result (True) | True<br>↑<br>True, True<br>↑ | True AND True = True |
| 8. | OR | Pop (2 element)<br>Push result (True) | ↑ (empty stack)<br>True<br>↑ | True OR True = True |
| | | | Ans = True | |

KAPIL SEHGAL