# Data Structure with Python

# Queue

Mr. V. M. Vasava

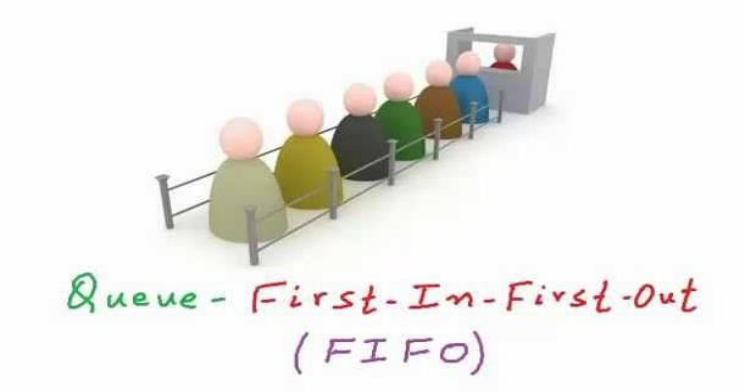
GPG, IT Dept. Surat

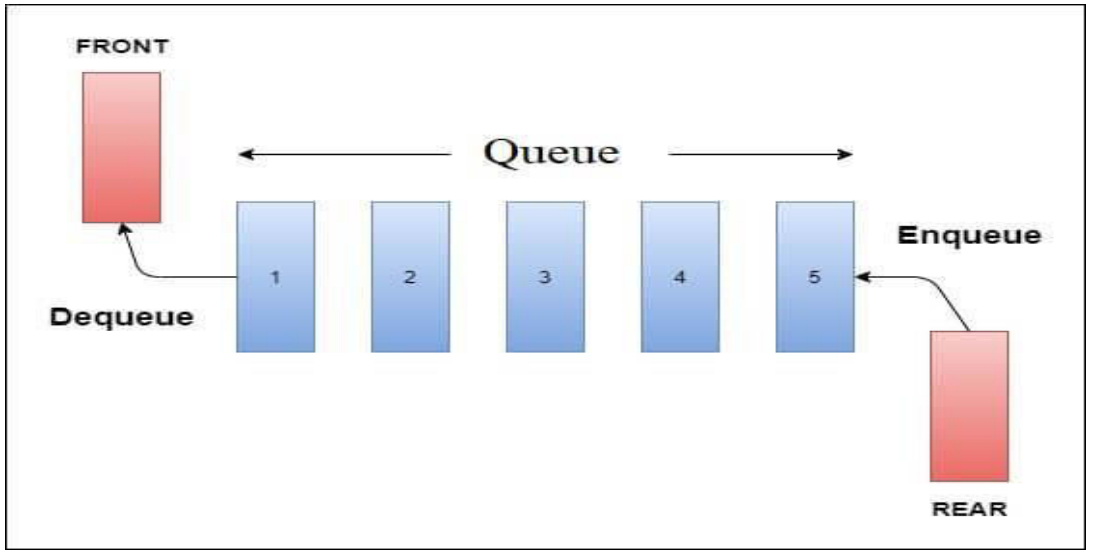
# Agenda

- Introduction about Queue
- Types Queue Data Structure
- Operation of Queue
- Representation of Queue

# Queue

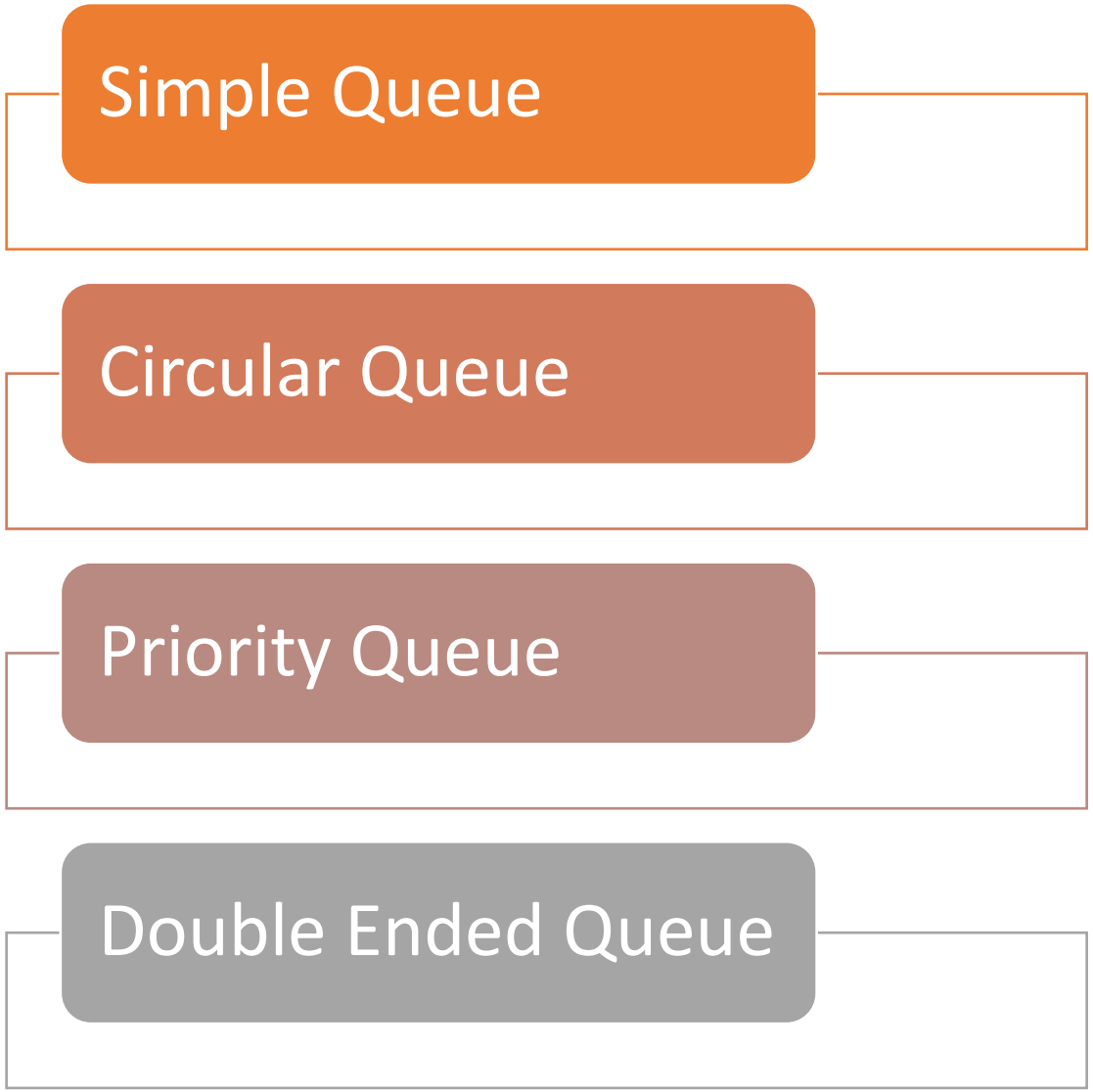
- In python, Queue is a linear data structure in which insertion can done at one end called rear **(R)**and deletion can done other side called front**(F)**. Hence, this structure is also known as FIFO (first in first out).




- A good example of a queue is any queue of consumers for a resource where the consumer that came first is served first.

# Types of Queue

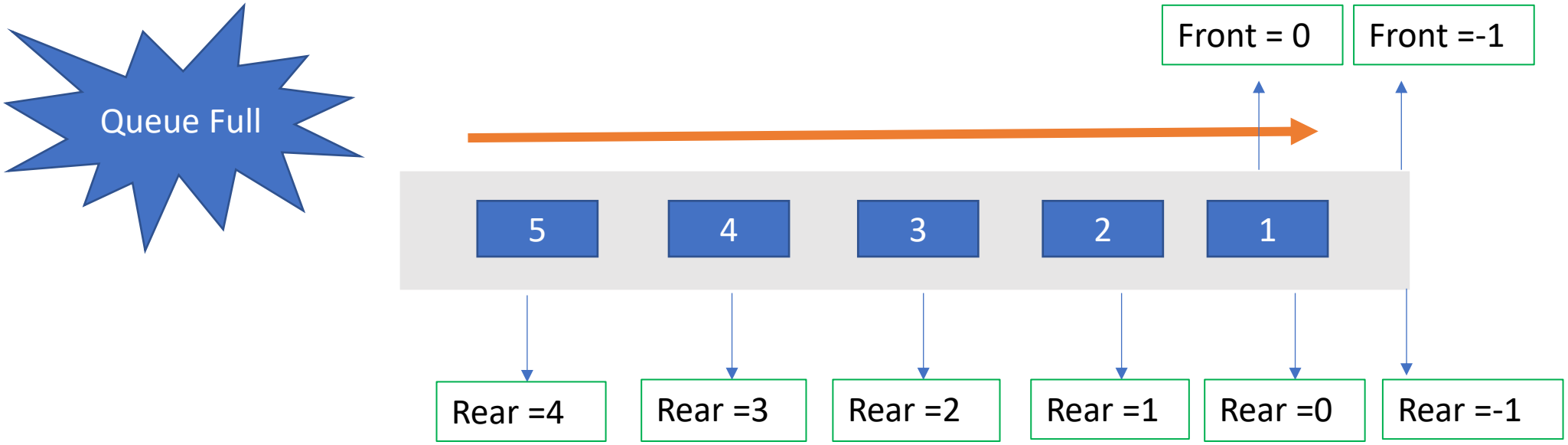


# Operation of Queue

1. **Enqueue**: Add an element to the end of the queue
2. **Dequeue**: Remove an element from the front of the queue
3. **IsEmpty**: Check if the queue is empty
4. **IsFull**: Check if the queue is full
5. **Peek**: Get the value of the front of the queue without removing it

# Operation of Queue

- Enqueue : The process to add an element into queue is called **Enqueue.** Assume Max size of queue =5

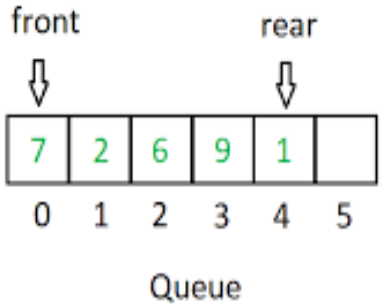


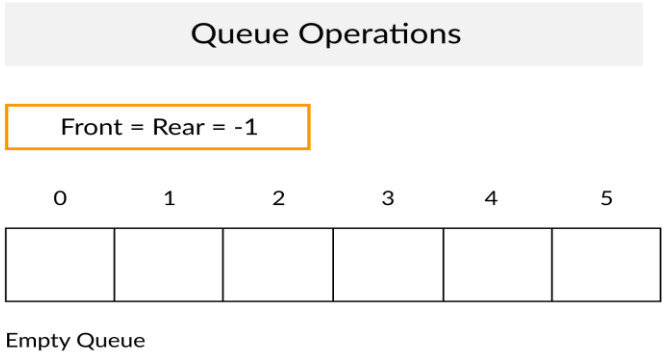

- Rear =Rear + 1 (Insert the data)

# Algorithm: INSERT (Q[], ele)

- Insert the element at rear side in queue.

1. if self.rear==self.msize-1: **# check queue is full or not**
   print("Queue is Full!!!!")
   else:
2. Read Data
3. **[Check possibility while insert element]**
   if self.rear==-1: [Empty]
   self.front=0
   self.rear=0
   else: [More Element]
   self.rear=self.rear+1
   self.que[self.rear]=ele [Added Element in queue]
   print(ele,"is added to the Queue!")
4. Return




faceprep

# Display

**Algorithm: Display ()**

1. if self.front==-1:

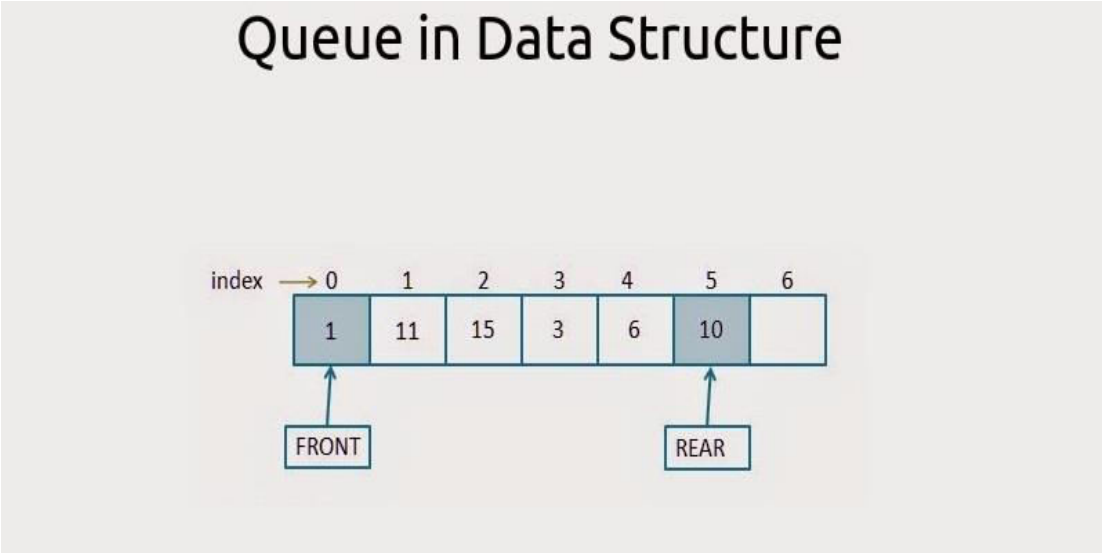
        print("Queue is Empty!!!")

2. else:

        for a in range(self.front,self.rear+1,1):

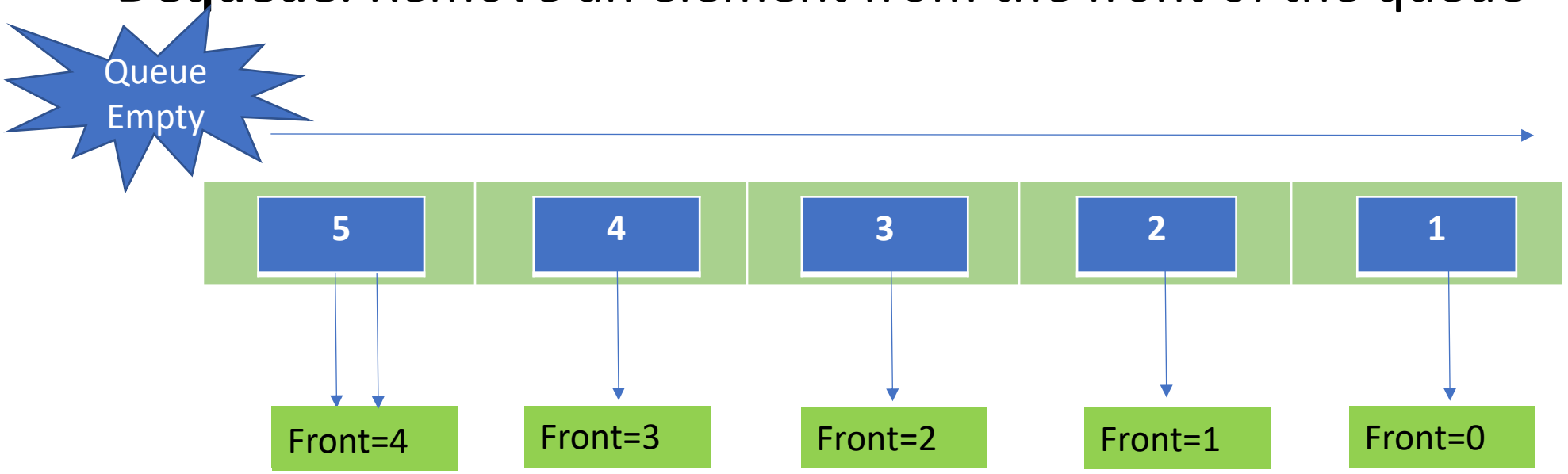
            print(self.que[a])

3. Exit



# Dequeue: Consider static queue size =5

- **Dequeue**: Remove an element from the front of the queue




- Front =Front +1

# **Dequeue(**Delete):

**Algorithm: DELETE ()**

Delete the element from queue using Front pointer.

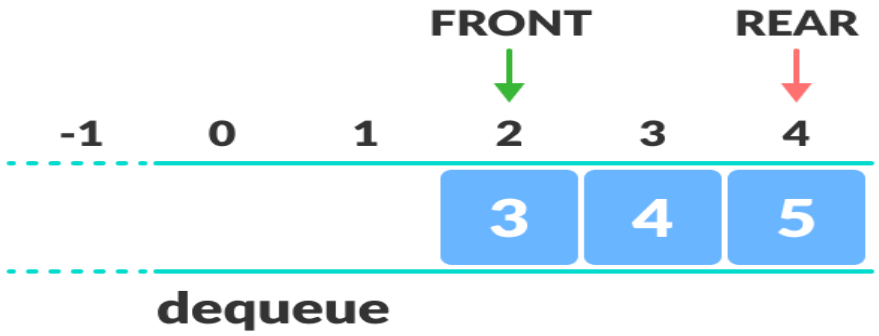
1. [Queue Empty]

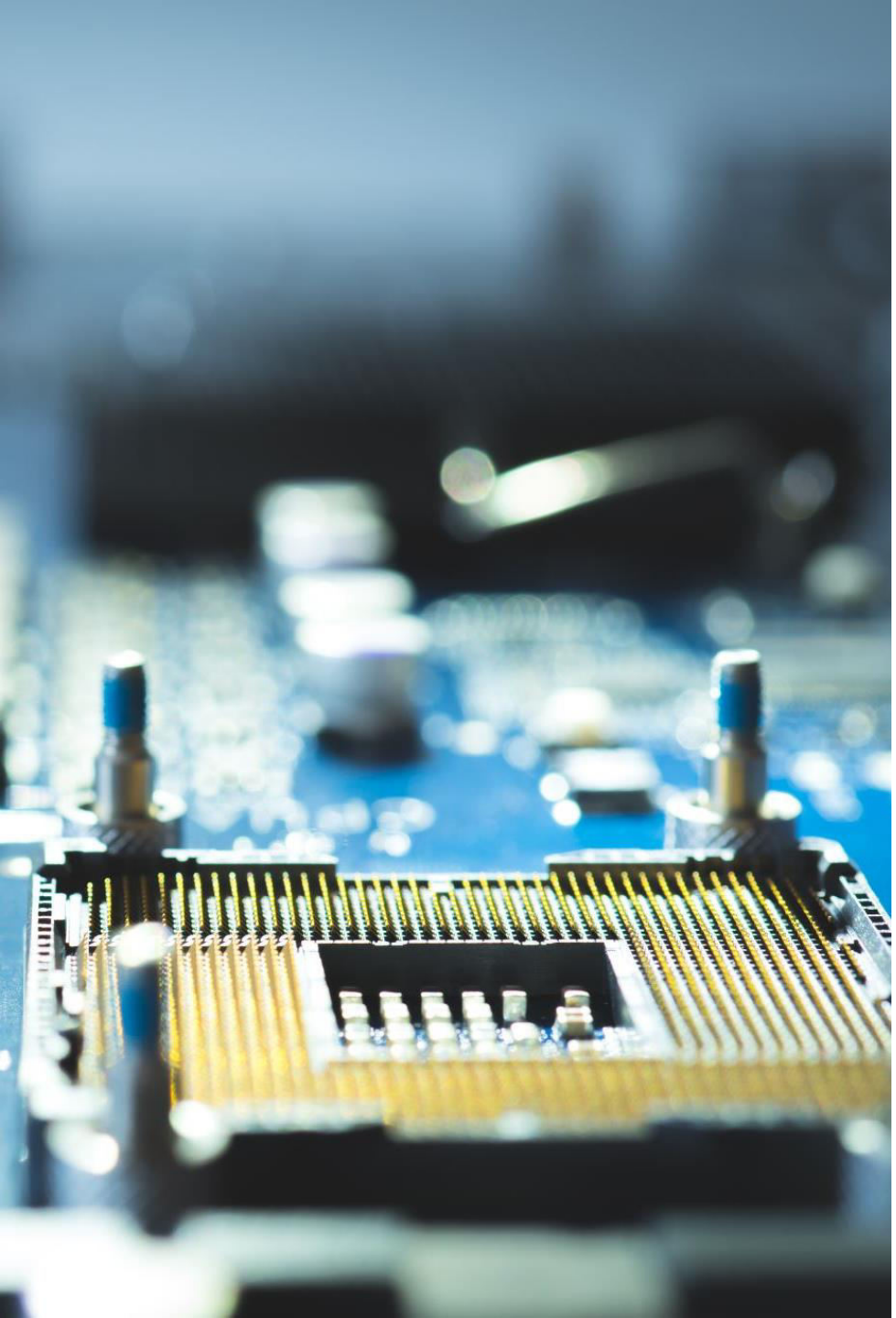
```
  if self.front==-1:# or if len(stack)==0
        print("Queue is Empty!!!")
```

2.else: [Check possibility]

```
    if self.front==self.rear: [Only one element]
          self.front=-1
          self.rear=-1
     else: [Two or more element]
          self.element=self.que[self.front]
          print("element removed!!:",self.element)
          self.front=self.front+1
```

3. Exit





# Application of Queue

1. Job Scheduling Algorithm

2.Handles multi-user, multi-programming environment, and time-sharing environment.

3.To implement a printer spooler.

4.In real-world queue is used for customer service like railway reservation, etc.

# Advantages & disadvantages

- Advantages:
- Insertion can done at Rear and Deletion can done at Front side.
- Queue follows FIFO structure.
- It can easily implement and understand.

- Disadvantages:
- Wastage of Memory
- Less efficient

# Time Complexity

- Best Case
- Average Case
- we know exactly, on which position new element will be added and from where an element will be removed, hence both these operations requires a single step.
- Enqueue: **O(1)**
- Dequeue: **O(1)**
- Size: **O(1)**

# Any Questions???

# Data Structure with Python

## Circular Queue

Mr. V. M. Vasava

GPG, IT Dept. Surat

# Agenda

- Limitation of Simple Queue
- Introduction about Circular Queue
- Representation of CQ
- Operation of CQ
- Advantages & Disadvantages

# Limitation of Simple Queue

- On deletion of an element  from existing queue, front pointer shifted to the next position.
- This result into virtual deletion of an element.
- By doing so memory space which was occupied by deleted element is wasted and hence  inefficient memory utilization is occur.


front=2, rear=6

This queue is considered full, even though the space at beginning is vacant.

- Requires more memory spaces.

# Circular Queue

- What is Circular queue ?
- A circular queue is a linear data structure in which the operations are performed based on FIFO (First In First Out) principle and the last position is connected back to the first position to make a circle.
- It is also called **‘Ring Buffer’**.

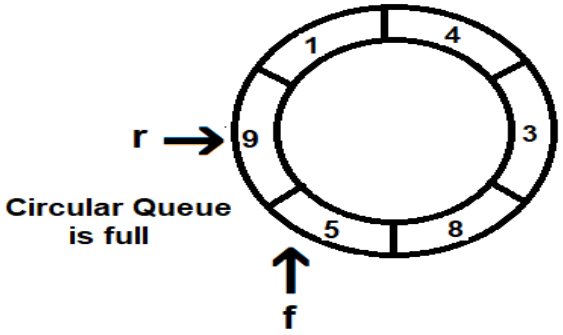
# Characteristics of Circular Queue

- Circular queue is linear data structure
- Based on FIFO principle
- Has two ends front and Rear
- Insertion from Rear
- Deletion from Front

# Working of CQ

- Circular Queue works by the process of circular increment i.e. when we try to increment the pointer and we reach the end of the queue, we start from the beginning of the queue.
- Here, the circular increment is performed by modulo division with the queue size. That is,
- `if F==(R +1)%Max :`

  `overflow`



**Algorithm: INSERT (Q[size], F, R, Data)**

Insert the element at rear side in circular queue. Where **k** =Max size of queue

1 if ((self.rear + 1) % self.k == self.front):**# check queue is full or not**

print("The circular queue is full”)

else:

2. Read Data

3. **[Check possibility while insert element]**

if self.rear==-1: [Empty]

self.front=0

self.rear=0

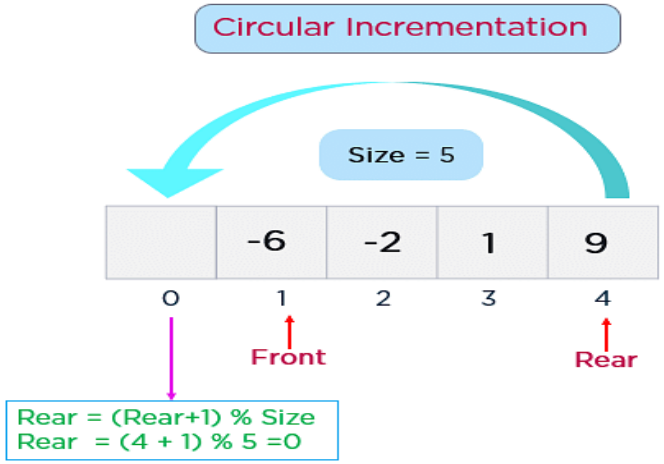
else: [More Element]

self.rear=(self.rear+1)%self.k

self.que[self.rear]=ele [Added Element in queue]

print(ele,"is added to the Queue!")

4.Return



# Delete

**Algorithm: DELETE (Q[size], F, R, Data)**

Delete the element from circular queue using Front pointer. Where k=size

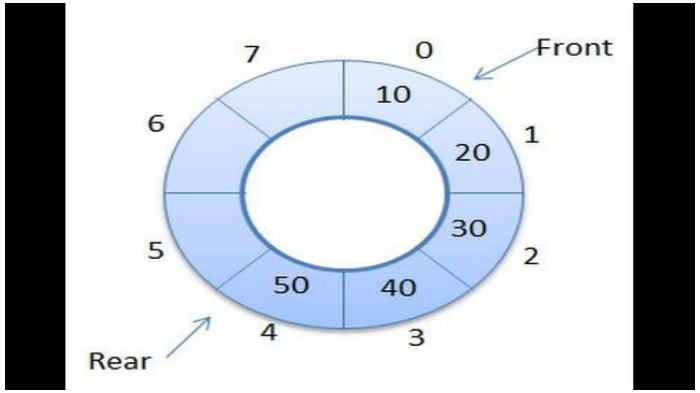
1. [Circular Queue Empty]

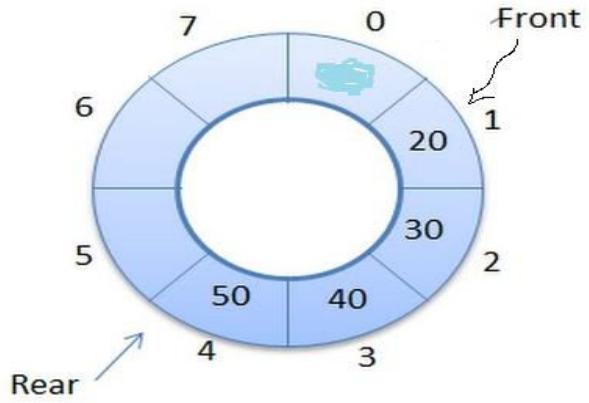
```
  if self.front==-1:
        print("CQueue is Empty!!!")
```

2.else: [Check possibility]

```
    if self.front==self.rear: [Only one element]
          self.front=-1
          self.rear=-1
     else: [Two or more element]
          self.element=self.que[self.front]
          print("element removed!!:",self.element)
          self.front=(self.front+1) % self.k
```

3. Exit



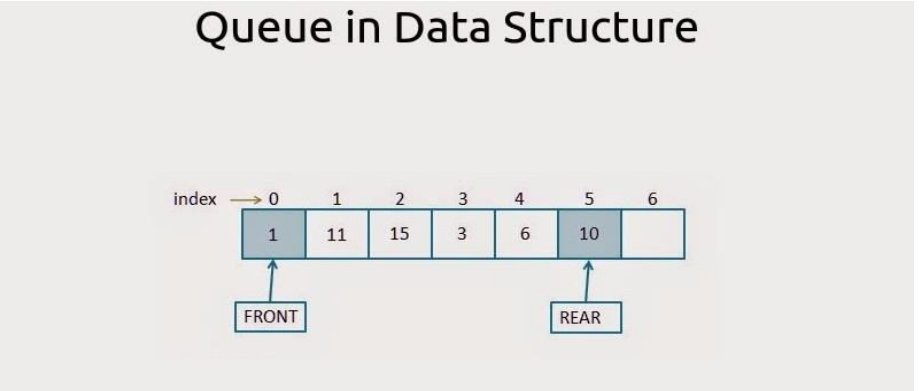



# Display

**Algorithm: Display (Q[max], Data)**

```
1. if(self.front == -1):
        print("No element in the circular queue")

2. elif (self.rear >= self.front):
        for i in range(self.front, self.rear + 1):
            print(self.queue[i], end=" ")
        print()
3.  else:
        for i in range(self.front, self.k):
            print(self.queue[i], end=" ")
        for i in range(0, self.rear + 1):
            print(self.queue[i], end=" ")
```



# Applications of Circular Queue

CPU scheduling

Memory management

Traffic Management

# Advantages

- Circular queue are used to remove the drawback of simple queue.
- Both the front and rear pointers wrap around to the beginning of the List.
- It is also called as "ring buffer".
- It takes up less memory than the linear queue.
- A new item can be inserted in the location from where a previous item is deleted.
- Infinite number of elements can be added continuously but deletion must be used.

# Linear v/s Circular Queue

| Feature | Linear queue | Circular queue |
|---|---|---|
| Data structure | Linear | Circular |
| Order of operations | FIFO (First In First Out) | FIFO (First In First Out) |
| Insertion | From the rear end | From either end From Rear End |
| Deletion | From the front end | From either end From Front End |
| Space utilization | Less efficient | More efficient |
| Applications | Simple queueing applications, such as a line of people waiting for a bus | Complex queueing applications, such as CPU scheduling and memory management |

# Any Questions ???

# Data Structure with Python

## Interconversion Expression

Mr. V. M. Vasava
GPG,IT Dept.

# Agenda

Introduction about Expression

Infix to Postfix (Using stack)

Infix to Postfix Algorithm

# Introduction

- Precedence Table (Priority)

| Operator | Name | Precedence when onstack (ISP) | Precedence when on input (ICP) |
|---|---|---|---|
| ( | Opening parentheses | 0 | 4 |
| ) | Closing parentheses | 0 | 0 |
| ^ | Exponentiation | 3 | 3 |
| * / | Multiply & divide | 2 | 2 |
| + - | Add & Subtract | 1 | 1 |

# Infix to Postfix

- A * B + C

| Steps | Symbol Scanned (ICP) | ISP | Output |
|---|---|---|---|
| 1 | A | | A |
| 2 | * | * | A |
| 3 | B | * | A B |
| 4 | + | + | A B * {pop and print the '*' before pushing the '+'} |
| 5 | C | + | A B * C |
| 6 | | | A B * C + |

# Examples

- A * (B + C)
- A * B ^ C + D becomes A B C ^ * D +

| Steps | Symbol (ICP) | Stack(ISP) | Output |
|---|---|---|---|
| 1 | A | | A |
| 2 | * | * | A |
| 3 | ( | * ( | A B |
| 4 | B | * ( | A B |
| 5 | + | * ( + | A B |
| 6 | C | * ( + | A B C |
| 7 | ) | * | A B C + |
| 8 | | | A B C + * |

# Algorithm: Infix to Postfix

**Algorithm: Infix_to_PostFix(Q, P)**

Suppose **Q** is an arithmetic expression written in infix notation. This algorithm finds the equivalent postfix expression **P**.

1. Push "(" onto STACK, and add ")" to the end of **Q**.
2. Scan Q from left to right and repeat Steps 3 to 6 for each element of Q until the STACK is empty:
3. If an operand is encountered, add it to **P**.
4. If a left parenthesis is encountered, push it onto STACK.
5. If an operator © is encountered, then:
   a) Repeatedly pop from STACK and add to **P** each operator (on the top of STACK) which has the same or higher precedence/priority than ©
   b) Add © to STACK.

   [End of If structure.]
6. If a right parenthesis is encountered, then:
   a) Repeatedly pop from STACK and add to **P** each operator (on the top of STACK) until a left parenthesis is encountered.
   b) Remove the left parenthesis. [Do not add the left parenthesis to **P**.]

   [End of If structure.]

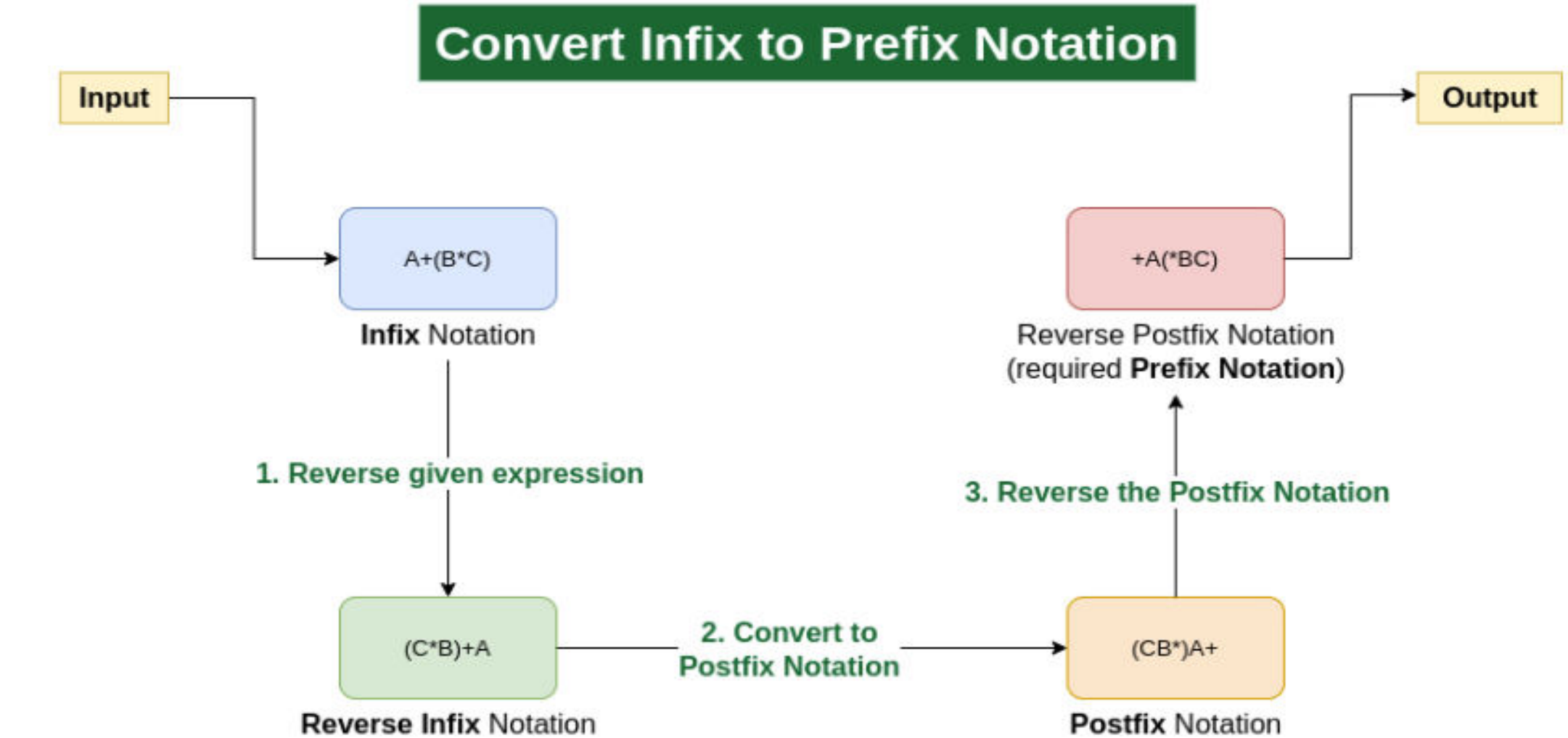
[End of Step 2 loop.]

7. Exit.

# Questions:

- (A+B)*(C-D) / E*F
- a*(c+d)+(j+k)*n+m*p
- a * b $ c / d – e + f – g
- ( ( ( A + B ) * C ) - ( ( D + E ) / F ) ) )
- (A+B)*C/D-E

# Infix to Prefix



# Infix to Prefix

- Algo: IPre(Q,P)
- 1. Reverse of String
- 2. Apply Algorithm Infix to Postfix

**Algorithm: Infix_to_PostFix(Q, P)**

Suppose **Q** is an arithmetic expression written in infix notation. This algorithm finds the equivalent postfix expression **P**.

1. Push "(" onto STACK, and add ")" to the end of **Q**.
2. Scan Q from left to right and repeat Steps 3 to 6 for each element of Q until the STACK is empty:
3. If an operand is encountered, add it to **P**.
4. If a left parenthesis is encountered, push it onto STACK.
5. If an operator © is encountered, then:
   a) Repeatedly pop from STACK and add to **P** each operator (on the top of STACK) which has the same or higher precedence/priority than ©
   b) Add © to STACK.

   [End of If structure.]
6. If a right parenthesis is encountered, then:
   a) Repeatedly pop from STACK and add to **P** each operator (on the top of STACK) until a left parenthesis is encountered.
   b) Remove the left parenthesis. [Do not add the left parenthesis to **P**.]

   [End of If structure.]

[End of Step 2 loop.]

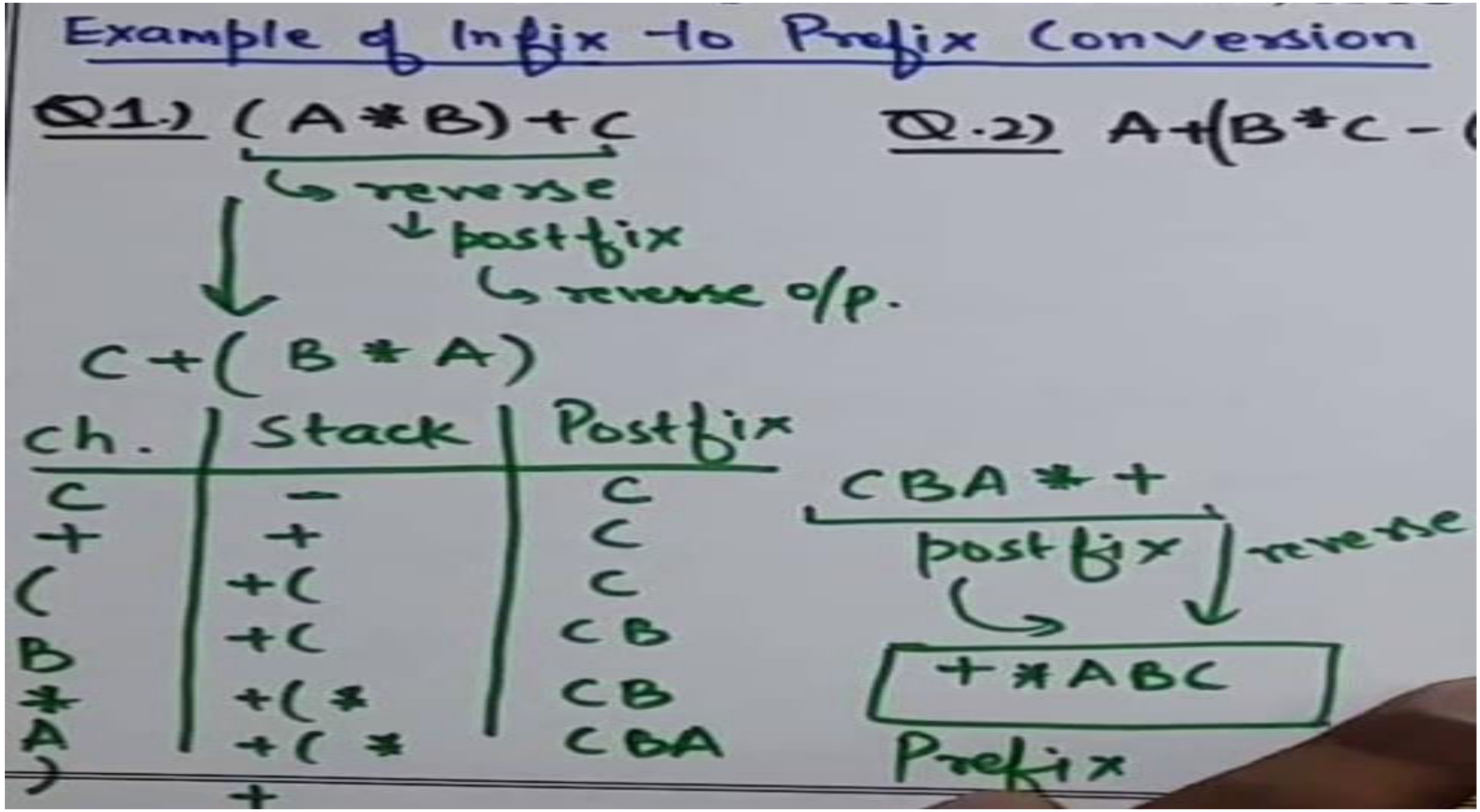
7. Exit.

- 3. The output of Reverse string

# Infix to prefix

- A * B ^ C + D
- Reverse of string :  D+C^B*A

# Infix to Prefix



# Evaluation of postfix Expression

A postfix expression is a collection of operators and operands in which the operator is placed after the operands.

Postfix Expression has following general structure.

Operand1 Operand2 Operator

For Example:

A B +

# Algorithm : Evaluation of postfix

- Algo: EvalPostfix(Q)

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
    1. A) .perform TWO pop operations and store the two popped operands in two different variables (A and B).
    2. B) Then perform operation using B  A and push result back on to the Stack.
4. Finally ! Display the value as result.
5. End

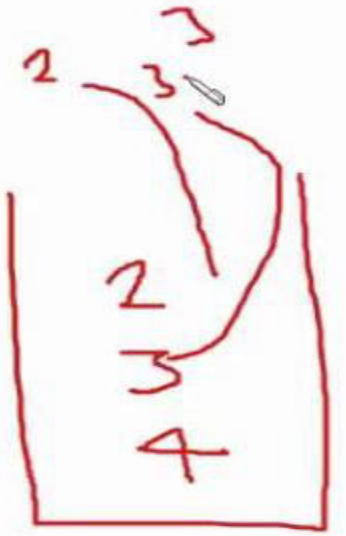
# Examples

## Evaluation of Postfix Expression

Postfix → a b c * + d - Let, a = 4, b = 3, c = 2, d = 5

Postfix → 4 3 2 * + 5 -

| Operator/Operand | Action | Stack |
|---|---|---|
| 4 | Push | 4 |
| 3 | Push | 4, 3 |
| 2 | Push | 4, 3, 2 |
| * | Pop (2, 3) and 3*2 = 6 then Push 6 | 4, 6 |
| + | Pop (6, 4) and 4+6 = 10 then Push 10 | 10 |
| 5 | Push | 10, 5 |
| - | Pop (5, 10) and 10-5 = 5 then Push 5 | 5 |



# Examples

1. Evaluate the expression 5 6 2 + * 12 4 / - using stack.

2. Evaluate the postfix expression :

   P: 12,7, 3,-,/,2,1,5,+,*,+

3. Evaluate the postfix expression :

   P: 2 3 1 * + 9 -

# Solve Example

- (A+B)*C/D-E

# Any Questions ??

# Data Structure Unit –III(stack)

Mr. V. M. Vasava

GPG,IT Dept.

# Agenda

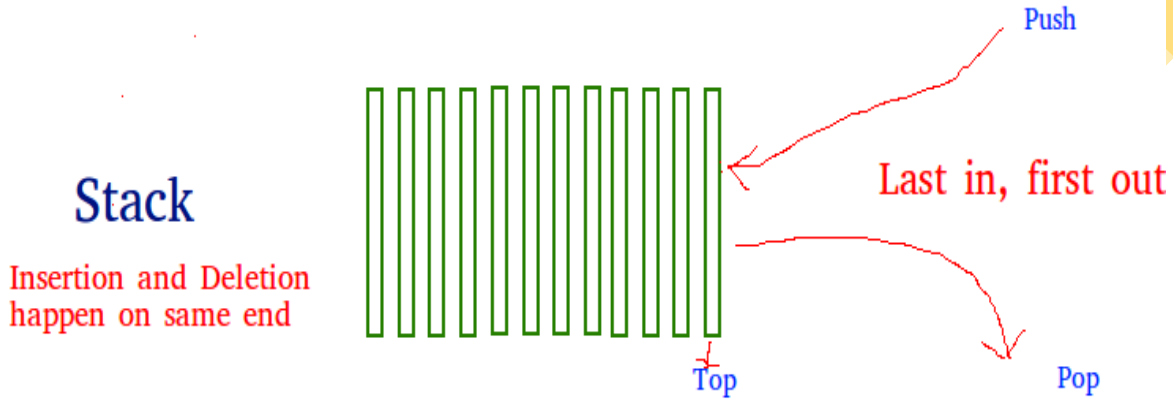
Introduction Linear

Concepts of Stack

Operations of Stack

# Stack

- Stack is a non-linear data structure in which insertion and deletion can done one end (TOP) called top of stack. Hence, this structure is also known as LIFO (Last in first out).
- The order may be LIFO(Last In First Out) or FILO(First In Last Out).




- Consider an example of plates stacked over one another in the canteen, a deck of cards etc.

# Stack Representation

- A stack may be represented in the memory in various ways.
- There are two main ways: using a one-dimensional array and a single linked list.
- Stack can either be a fixed size one or it may have a sense of dynamic resizing.

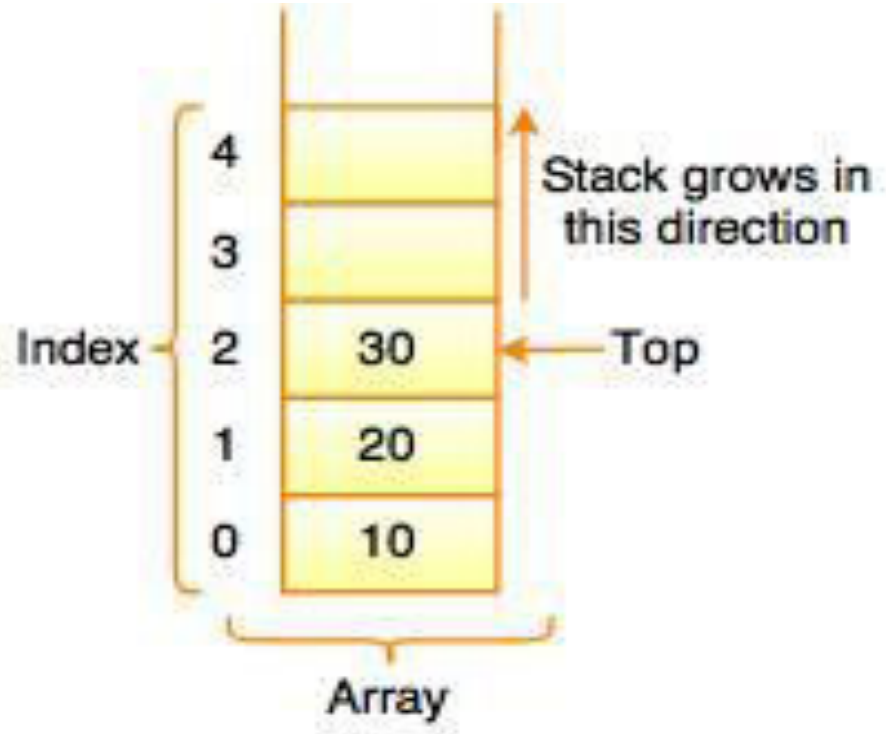



Fig. Implementation Stack using Array

# Basic Terminology of Stack

Maxsize –The term refers to maximum size of the stack.

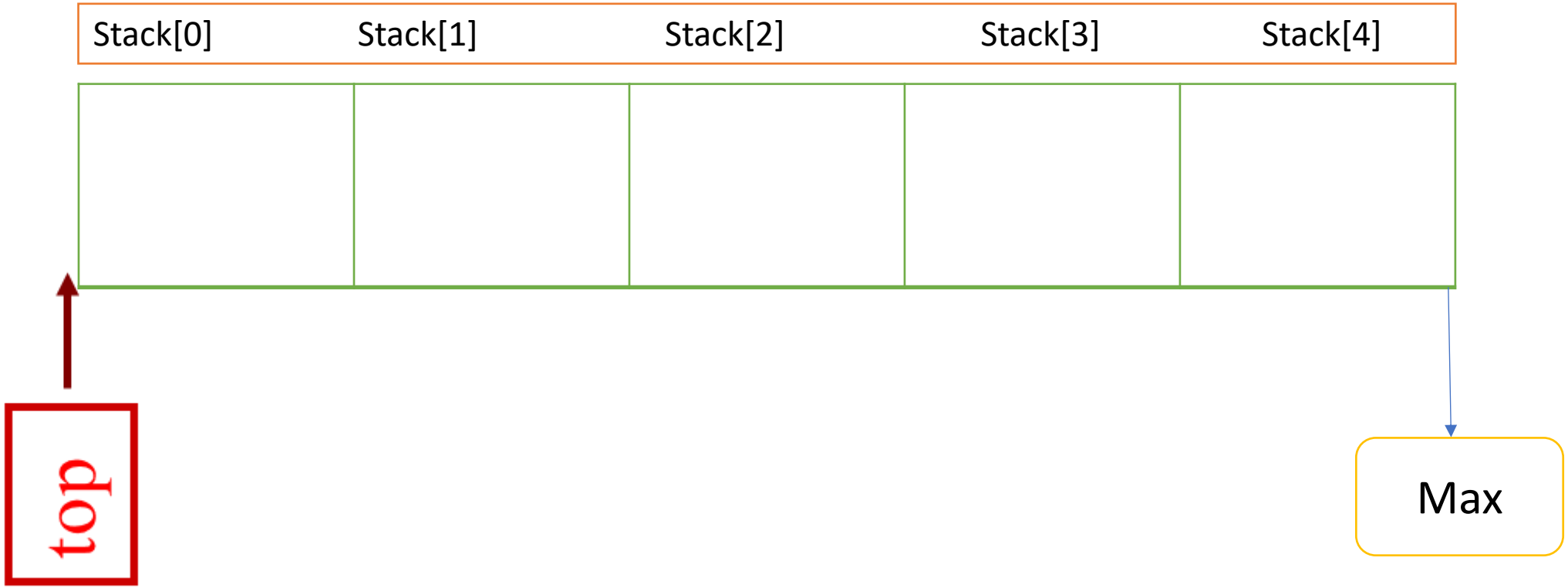
Top –The term refers to top of stack(TOS). The stack top is used to check the overflow or underflow condition. Initially Top=-1 indicate empty.

Stack underflow: This is the situation when the stack contains no element. hence the stack is initially Top =-1

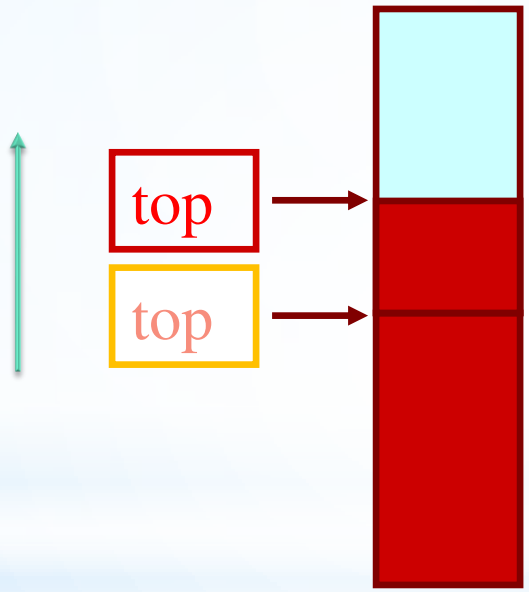
Stack Overflow: This is the situation when the stack become full and no more elements can be pushed into the stack. Hence the stack reached Top =max-1

# Array Representation

- Array Representation of stack with Python  (Initially Top=-1)

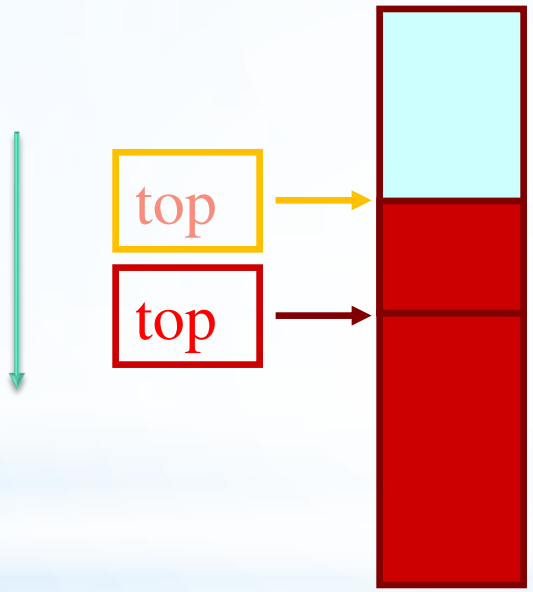


# Push using Stack




PUSH (Top =Top+1)

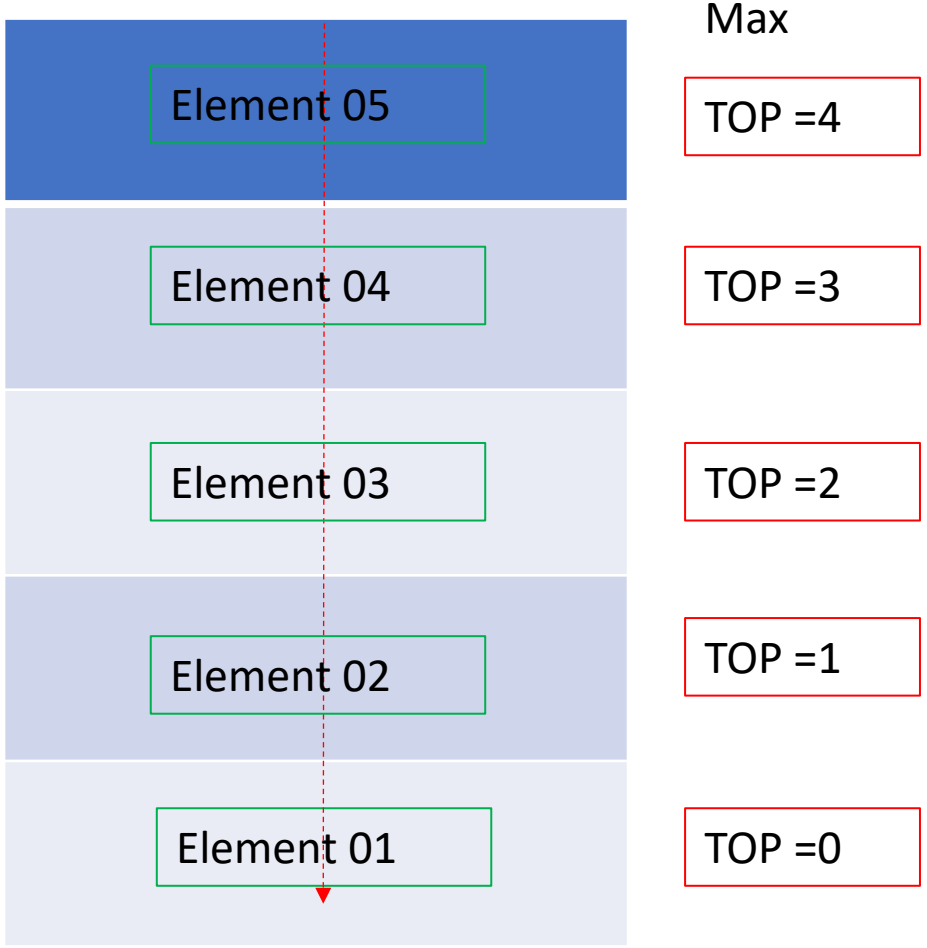
# Pop using Stack




POP (Top=Top-1)

# Operation of Stack : Push (Insert the element in stack)

Consider stack is static size 5 (Using Array)




Max

TOP =4

TOP =3

TOP =2

TOP =1

TOP =0

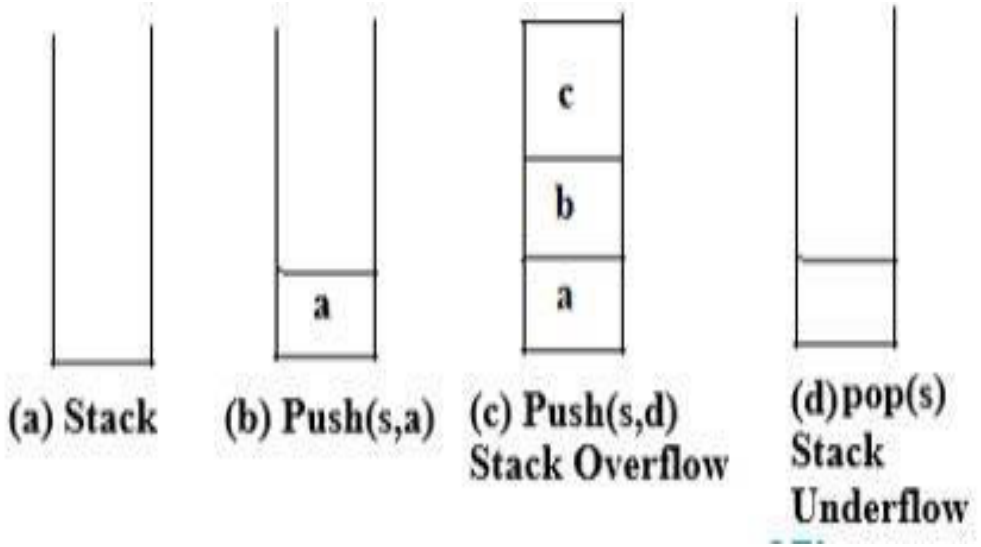

Top =Top +1

# Algorithm: Push

Push operation is adding the data in a stack. Where stack is create the stack that hold the max data. Top is a location of insertion and deletion of data item in stack.

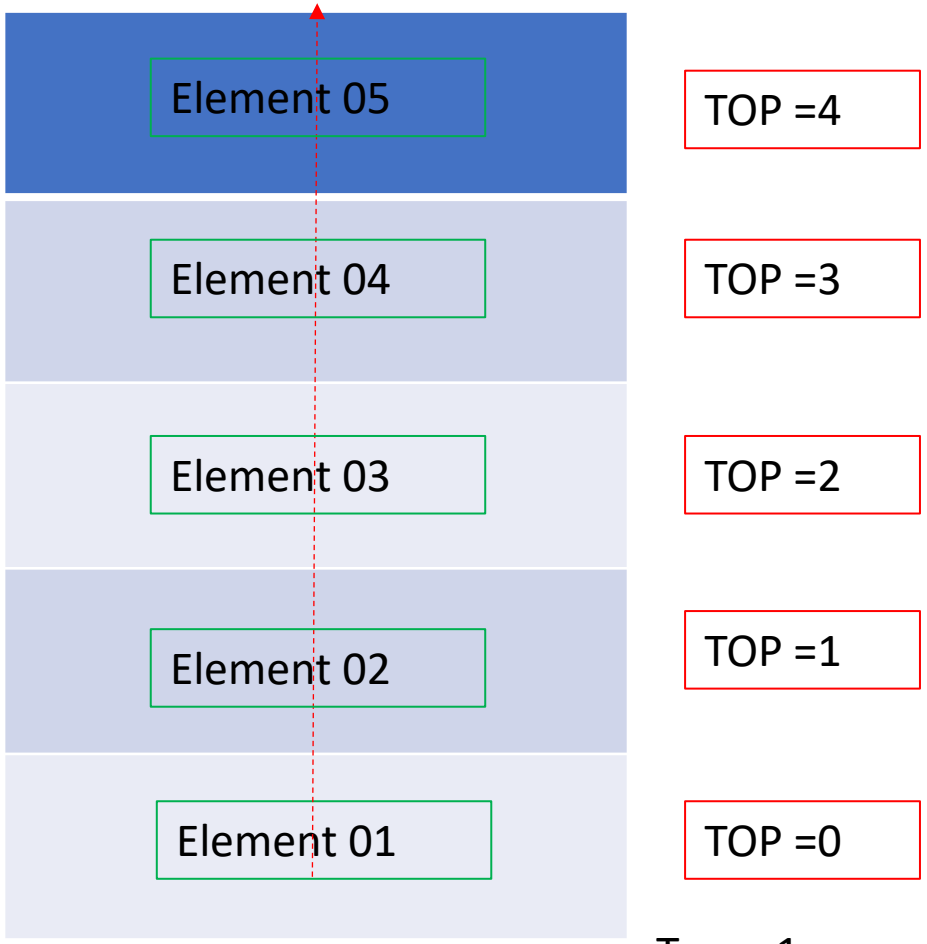
Algorithm: PUSH (Stack , Data)

1. [Stack Overflow Condition]
   if self.top == self.msize- 1:
      Print “Stack is overflow”.
   Else:
2. [Increment by top pointer for insertion]
   self.top =self.top+ 1
3. [Store data in stack array at Top position]
   self.stack[self.top] = element
4. Return



# Operation of Stack : POP (Remove element from stack )

Consider stack is static size 5 (Using Array)




# Algorithm: POP

**Algorithm: POP ()**

Pop operation is removing the data from existing stack elements.

1. **[Empty /Underflow condition]**

```
if self.top == -1:
        print("Stack underflow")
        return None
  Else:
```

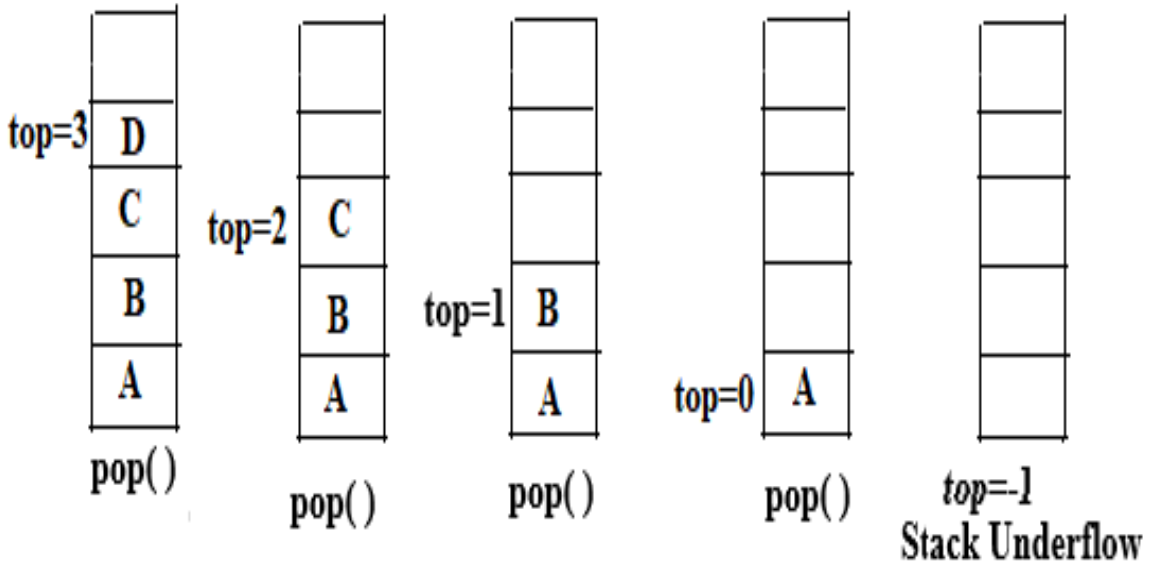
2. [Select data of Top position]

```
        element = self.stack[self.top]
```

3. [Decrement by top pointer]

```
    self.top =self.top- 1
```

4. Return



# Display()

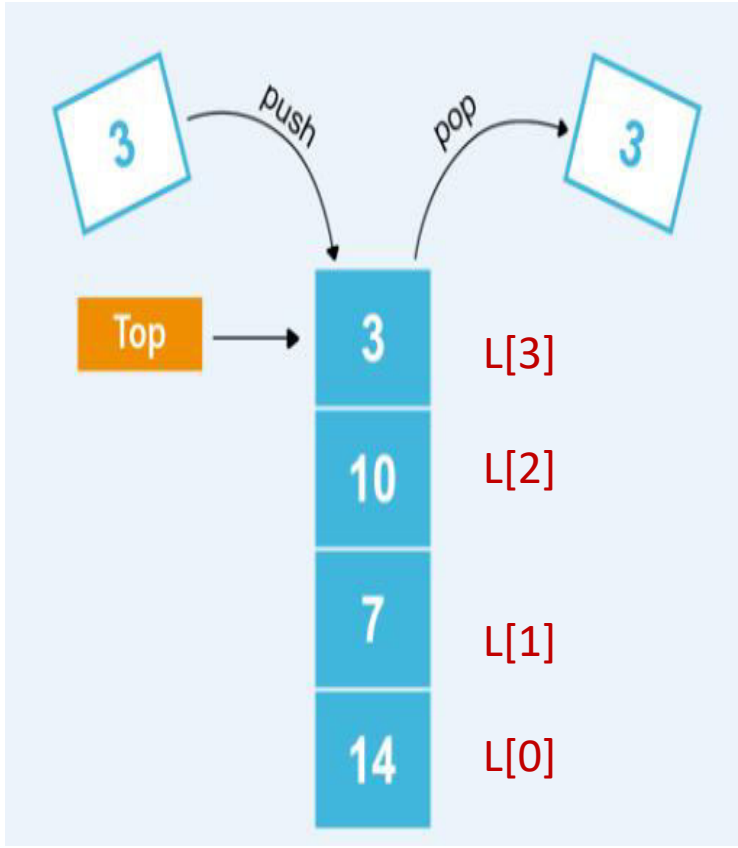
- **The display() function displays all the elements in the stack.**

**1.[Check Stack empty or not]**

```
if  self.top==-1:
        print("Stack is empty")
```

**2.else: [Display element from Top to 0th position]**

```
        # a=self.stack[::-1]
        #print(a)
        #print("Element at the top",self.stack[-1])
        for a in range(self.top,-1,-1):
            print(f'Stack{[a]}=',self.stack[a])
```



# Advantages & Disadvantages

## Advantages:

1. Easy to program
2. Easy to check overflow
3. An array allow random access
4. Insertion & Deletion can very quick.
5. Provide LIFO structure.

## Disadvantages:

1. Fixed size of stack
2. Waste Memory

# Application of stack

- Stack works on the LIFO principle i.e. they are used to reverse the string.
- Stack is used in **Expression Evaluation** and Expression Conversion.
- It is also used in web browsers for the forward and the backward features.
- It is used for syntax parsing.
- It is used for Backtracking in the searching algorithms and parenthesis matching.
- Stacks are useful for function calls, storing the activation records and deleting them after returning from the function.

# Any questions??

# Data Structure with Python Unit –III(stack)

Mr. V. M. Vasava

GPG,IT Dept.

# Agenda

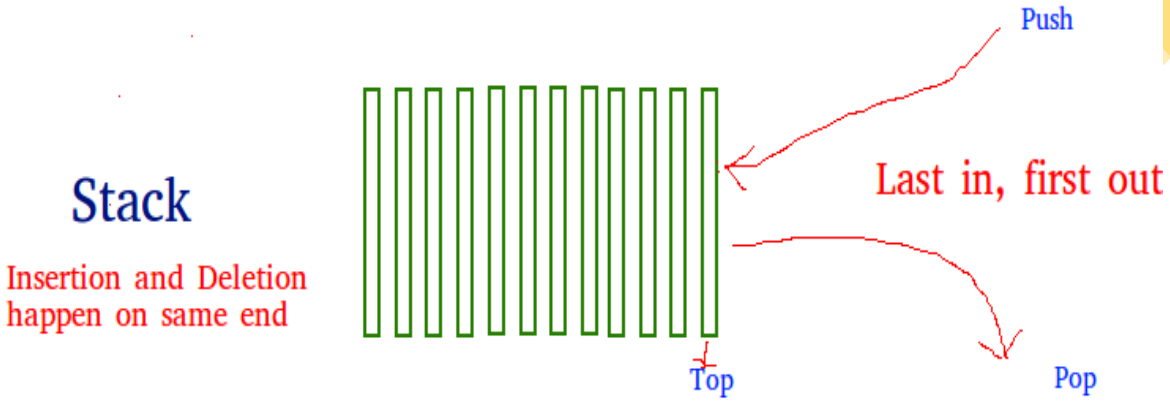
Introduction

Concepts of Stack

Operations of Stack

# Stack

- A **stack** is a linear data structure that stores items in a **Last-In/First-Out (LIFO)**or First-In/Last-Out (FILO) manner.
- In stack, a new element is added at one end and an element is removed from that end only. The insert and delete operations are often called push and pop.
- Consider an example of plates stacked over one another in the canteen, a deck of cards etc.




# Operation of stack

The basic operations required to manipulate a stack are:

1. PUSH: To insert an item into a stack

2. POP: To remove an item from a stack

3.Display :  Traverse the element till Topmost element.

4.**Peek:** Get the topmost element of the stack

# Stack implementation

- List
- Collections.deque
- queue.LifoQueue

- Push --->append()
- Pop  -->pop()

```
In [4]: stack=[]
   ...: stack.append(10)
   ...: stack.append(20)
   ...: stack.append(30)
   ...: print(stack)
   ...: stack.pop()
   ...: stack.pop()
   ...: stack.pop()
   ...: print(stack)
[10, 20, 30]
[]
```

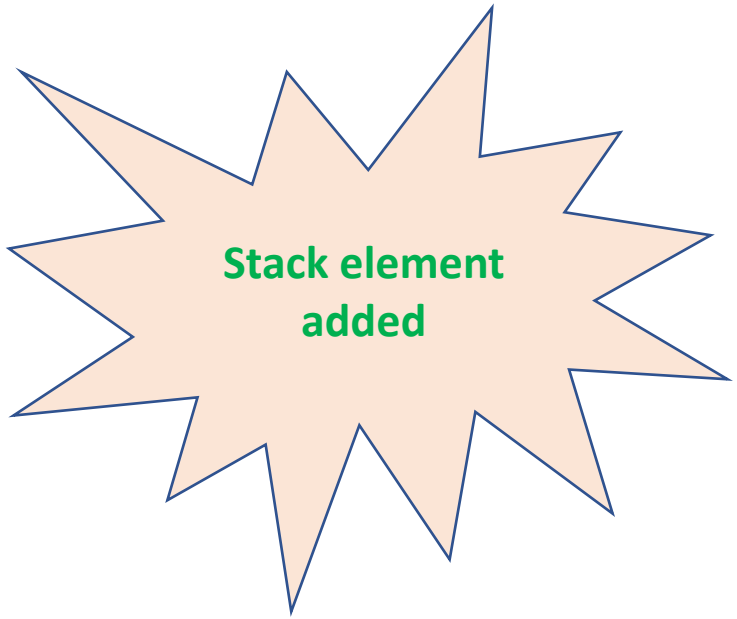
# Basic Terminology of Stack

Top –The term refers to top of stack(TOS). The stack top is used to check the underflow condition. Initially len(stack)==0 or not stack indicate empty.

Stack underflow: This is the situation when the stack contains no element. hence the stack is initially len(stack)==0  or not stack

# Operation of Stack : Push (Insert the element in stack)

Consider stack is static size 5 (Using List)


Top =Top +1

# Algorithm: Push

- To add an element to a stack in Python, add it to the end of our list.
- There is a predefined method in Python - ***append( ).***

**Algorithm: PUSH (Stack , Data)**

Push operation is adding the data in a stack. Where stack is create the stack that hold the max data. Top is a location of insertion and deletion of data item in stack.

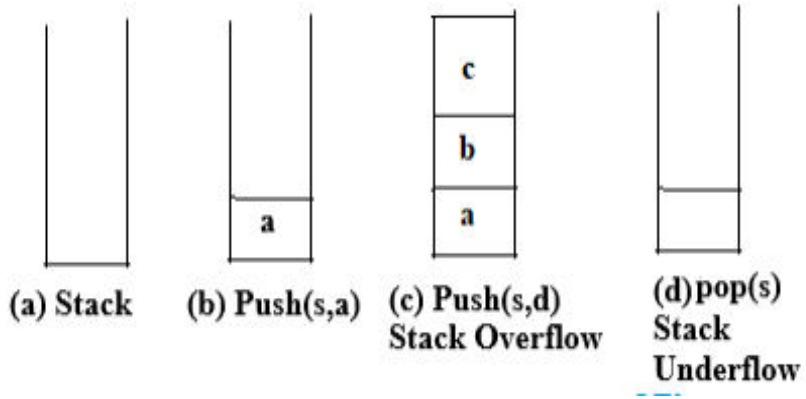
1.Create a stack as list, initialize var.

2.Read Data

3. [using append method for insertion]

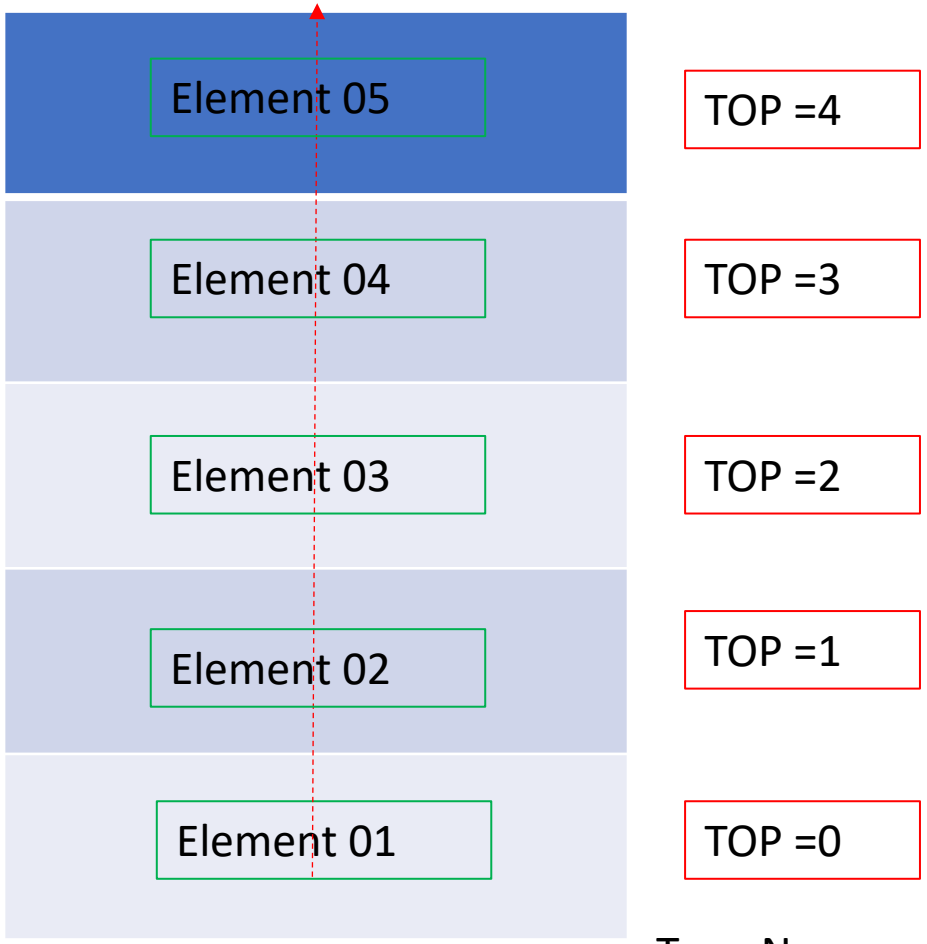
   stack.append(data)

4. Return



# Operation of Stack : POP (Remove element from stack )

Consider stack is static size 5 (Using List)




TOP =4

TOP =3

TOP =2

TOP =1

TOP =0

Top =None

Top =Top -1

# Algorithm: POP

- In Python, a *pop( )* method takes an index as a parameter and removes the element in that index from the list.
- In a stack in Python, we want to remove only the last element. So, to do that, we won't give any parameter to the function, and the last element in the list will be popped as it happens in a stack.

**Algorithm: POP (Stack )**

Pop operation is removing the data from existing stack elements.

1. [Empty /Underflow condition]

If (len(Stack)==0)

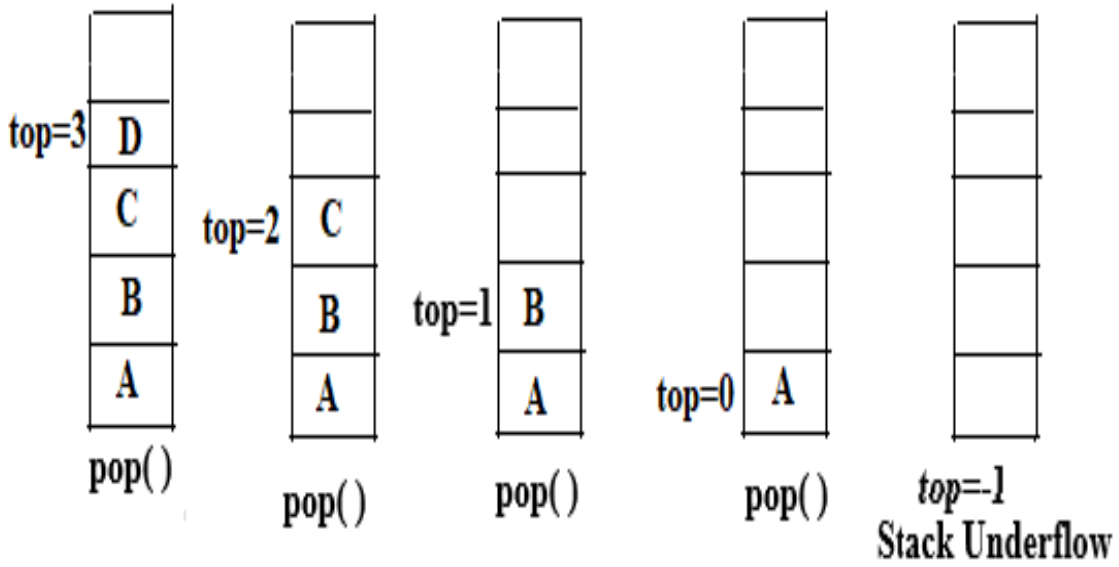
    Print "Empty Stack".

Else

2. [Use inbuilt POP method]

        Data = stack.pop()

3. Return



# Empty or Underflow

- ✓This operation is used to check if the given stack is empty.
- ✓It returns a Boolean value; that is ***true*** when the stack is empty, otherwise ***false***.
- ✓if stack == []  or len(stack)==0 or not stack:
- ✓ return True
- ✓ else:
- ✓ return False

# Display()

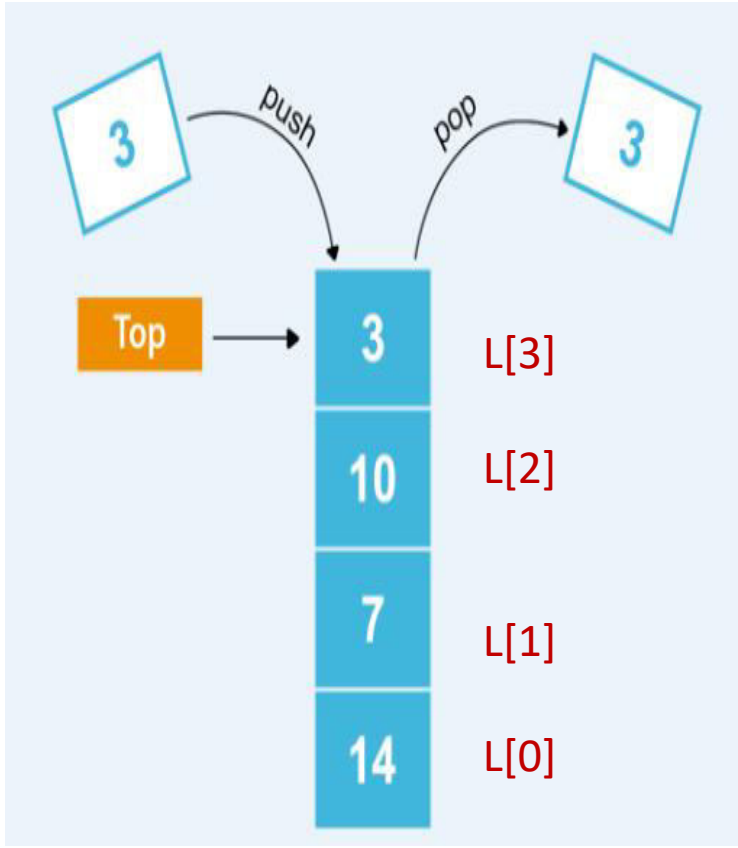
- **The display() function displays all the elements in the stack.**

1.**[Check Stack empty or not]**

```
if len(self.stack)==0:
        print("Stack is empty")
```

2.else: **[Display element from Top to 0th position]**

```
        # a=self.stack[::-1]
         #print(a)
         #print("Element at the top",self.stack[-1])
        for a in range(self.top-1,-1,-1):
            print(self.stack[a])
```



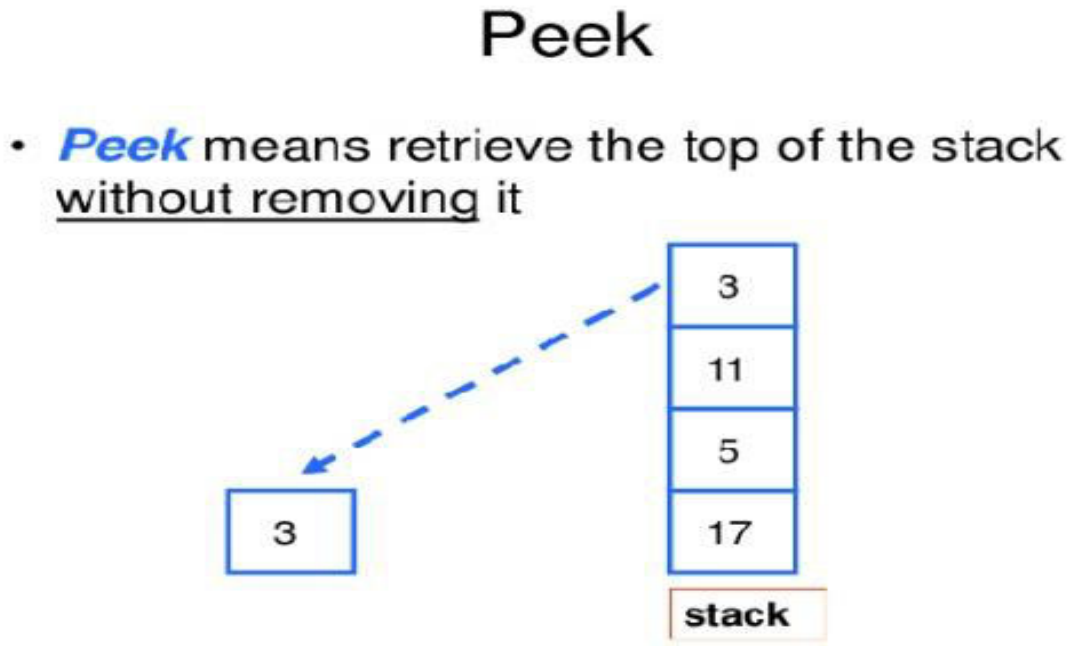
# Peek() operation

- **Peek(self)**:
- Objective:Get the value of the top element without removing it.

Step-1. [check empty or not]

```
        if len(self.stack)==0:
                print("Stack is empty")
        else:
        [find top position of element]
                self.top=len(self.stack)-
```

Step-2. return self.stack[s



# Application of stack

- Stack works on the LIFO principle i.e. they are used to reverse the string.
- Stack is used in **Expression Evaluation** and Expression Conversion.
- It is also used in web browsers for the forward and the backward features.
- It is used for syntax parsing.
- It is used for Backtracking in the searching algorithms and parenthesis matching.
- Stacks are useful for function calls, storing the activation records and deleting them after returning from the function.

# Advantages & Disadvantages

## Advantages:

1.Compiler can allocate space during compilation.

2.Easy to program

3.Insertion & Deletion can very quick.

4.Provide LIFO structure.

## Disadvantages:

1. **Limited memory size:** Stack memory is very limited.

2. **Random access is not possible:** In a stack, random accessing the data is not possible.

Any questions??

# Data Structure with Python
# Expression Conversion

Mr. V. M. Vasava

DIT ,GPG, Surat.

# Agenda

Introduction to Expression(Notation)

Infix, Prefix,Postfix

Interconversion Expression

Examples

# Expression

- The way to write arithmetic expression is known as a **notation**. There are three notation in data structures below:

1. Infix Notation
2. Prefix (Polish) Notation
3. Postfix (Reverse-Polish) Notation

# Algebraic Expression

- An algebraic expression is a legal combination of operands and the operators.
- Operand is the quantity (unit of data) on which a mathematical operation is performed.
- Operand may be a variable like x, y, z or a constant like 5, 4,0,9,1 etc.
- Operator is a symbol which signifies a mathematical or logical operation between the operands. operators include +,-,*, /, ^
- An example of expression as x+y*z.

# Infix, Prefix, Postfix

1.Infix : It is the form of an arithmetic expression in which we fix (place) the arithmetic operators in between two operands.

Example: (A + B)*C

2.Prefix

It is the form of an arithmetic expression in which we fix (place) the arithmetic operators before its two operands.

Prefix notation also Known as Polish notation.

Example: *+ABC

3.Postfix: Operators are written after their operands.

Postfix notation is called suffix notation and is refered as Reverse Polish notation.

Example: AB+C*

# Conversion Expression Infix to Postfix

- Precedence Table (Priority)

| Operator | Name | Precedence when on stack | Precedence when on input |
|---|---|---|---|
| ( | Opening parentheses | always stack<br>0 | 4 |
| ) | Closing parentheses | never stacked 0 | 0 |
| ^ | Exponentiation | 3 | 3 |
| * / | Multiply & divide | 2 | 2 |
| + - | Add & Subtract | 1 | 1 |

- Order of evaluation is determined by:
- precedence rules
- parentheses
- association (L to R or R to L)

# Example : Infix to Postfix

1. A+ B*C/D
2. (A + B) * (C + D)
3. A * ( B + C ) / D
4. (A + B) * C - (D - E) * (F + G)
5. (5 + 6) * 9 +10

# Example : Infix to Postfix

- A+ B*C/D

# Example: Infix to Postfix

Move each operator to the Right of its operands & remove the parentheses:

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


1. ( [ A B + ] * [ C D + ])
2. A B + C D + *

# Infix to Prefix

1. A+ B*C/D
2. (A + B) * (C + D)
3. A * ( B + C ) / D
4. (A + B) * C - (D - E) * (F + G)
5. A+B/C*D-E/F-G

# Example: Infix to prefix

- A+ B*C/D

# Any Questions ?????