UNIT-3 STACK AND QUEUE

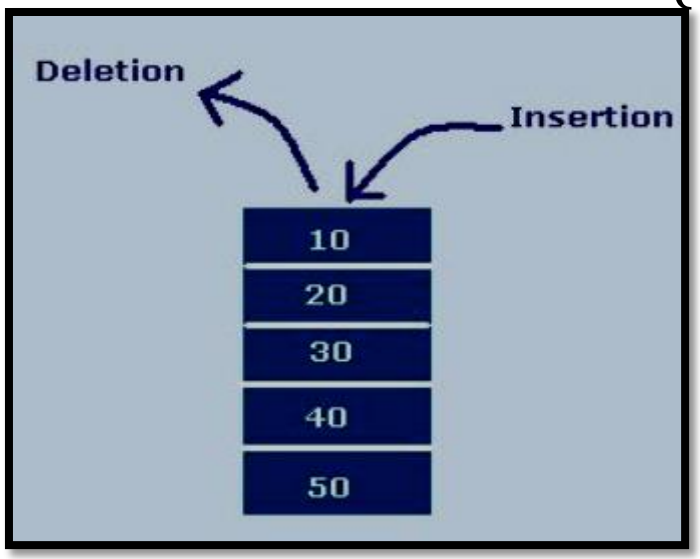
Prepared By:
Dhaval R. Gandhi
Lecturer in IT

Learning Outcomes

- Overview of Stack
- Operations on Stack - Push, Pop
  - Implementation of Stack using List
  - Application of Stack - Infix, Prefix and Postfix Forms of Expressions, Evaluations of postfix expression, Recursive
  - Functions (factorial, Fibonacci series)
- Overview of Queue
  - Operations on Queue - Enqueue and Dequeue
  - Implementation of Queue using List
  - Limitation of Single Queue
  - Concepts of Circular Queue
  - Application of queue
  - Differentiate circular queue and simple queue

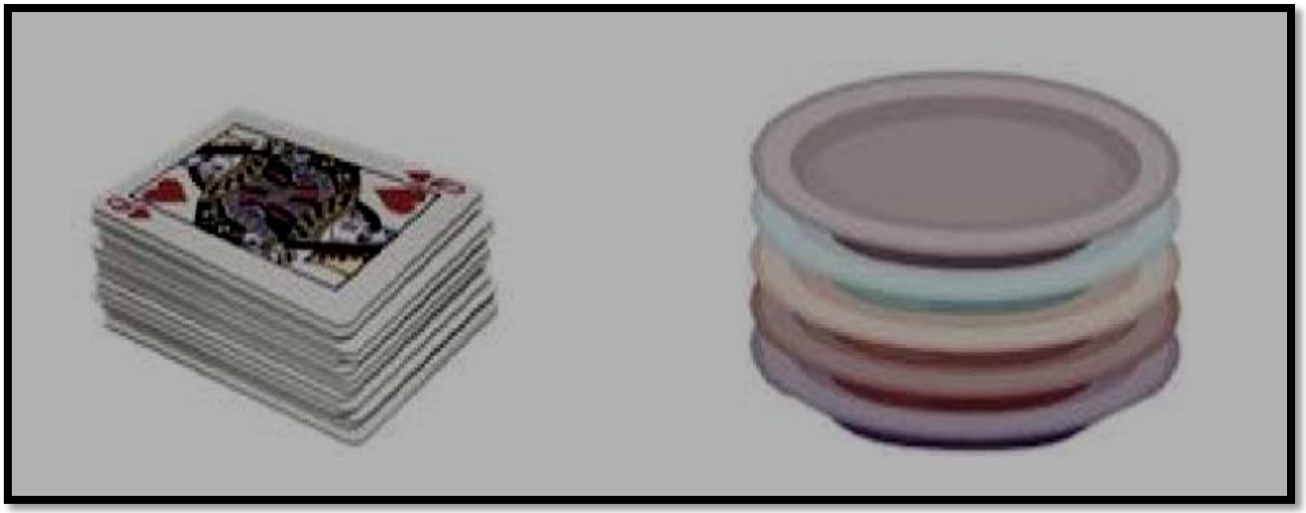
Stack

- A stack is a linear list in which insertion and deletion operations are performed at only one end of the list. Thus you are able to insert as well as delete the element from only one end of the stack.
- Since insertion and deletion operations are performed at same end, the element which is inserted Last is first to delete.
- Stack is also known as Last In First Out (LIFO).



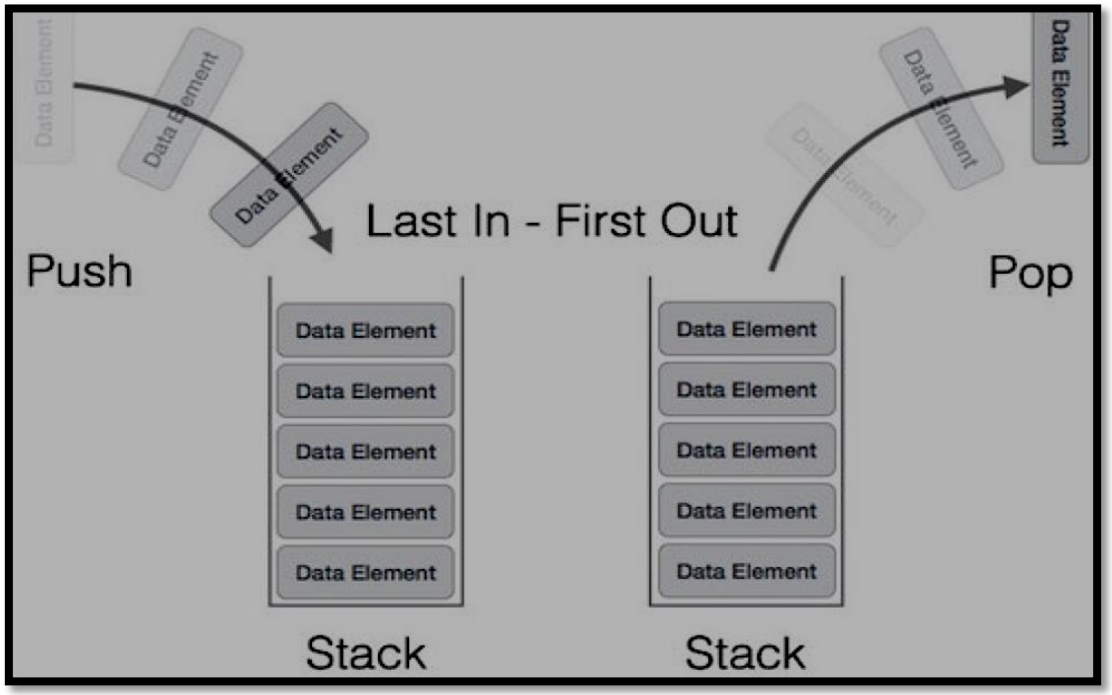
Stack

- 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.



Stack Representation

- A stack can be implemented by means of Array, List, dequeue and Life Queue.
- Here, we are going to implement stack using List.



Methods of Python Stack

- **isempty() –** Returns a Boolean (True/False) value as True if the stack is empty
- **size() –** Returns size of the stack or number of items/elements stored in the stack
- **top()/peek() –** Returns a reference to the topmost available element stored in the stack
- **push(a) –** Inserts an element at the top of the stack
- **pop() –** Deletes the topmost element from the last occupied memory location of the stack

Working of Stack Data Structure

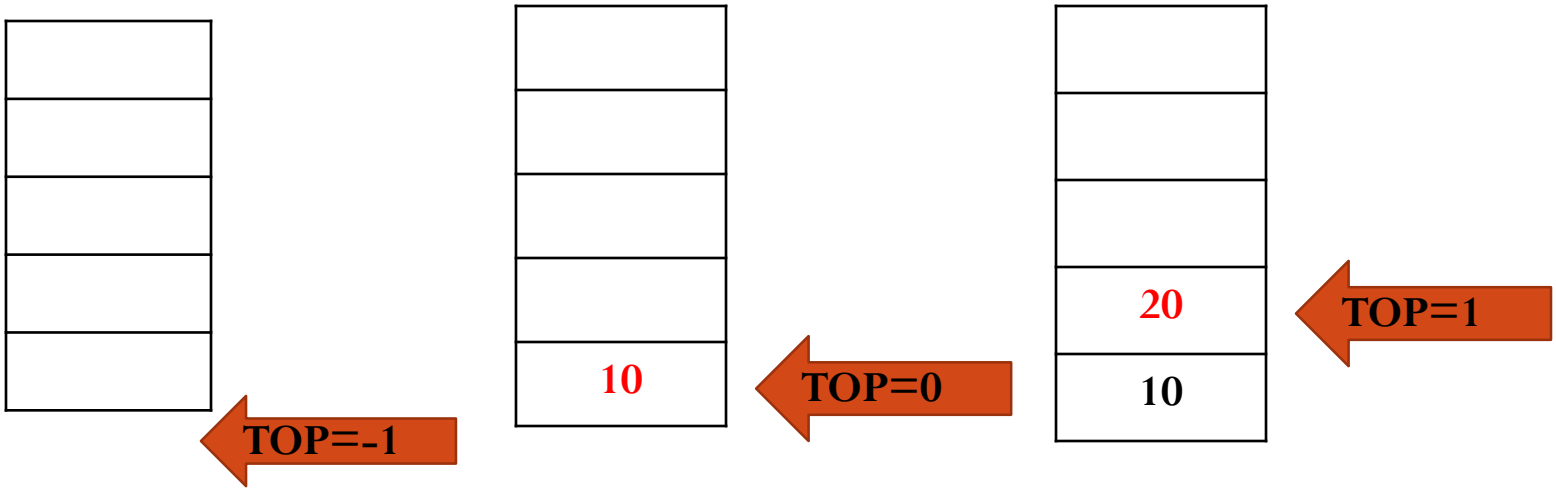
- A pointer called TOP is used to keep track of the top element in the stack.
- When initializing the stack, we set its value to -1 so that we can check if the stack is empty by comparing TOP == -1.
- On pushing an element, we increase the value of TOP and place the new element in the position pointed to by TOP.
- On popping an element, we return the element pointed to by TOP and reduce its value.
- Before pushing, we check if the stack is already full.
- Before popping, we check if the stack is already empty.

Operations of Stack

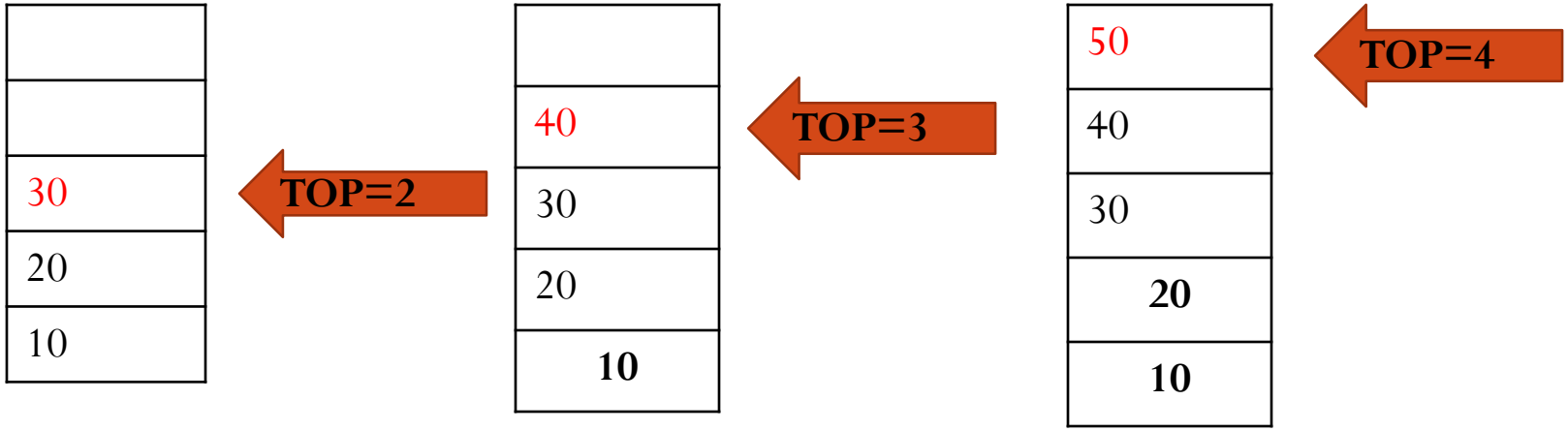
- **PUSH : It is used to insert items into the stack.**
- **POP: It is used to delete items from stack.**

TOP:

- It represents the current location of data in stack.
- It is used to keep track of the topmost element in the stack.
- A variable TOP contains an index(Position) of the topmost element in the stack.



Operations of Stack



PUSH Operations on stack

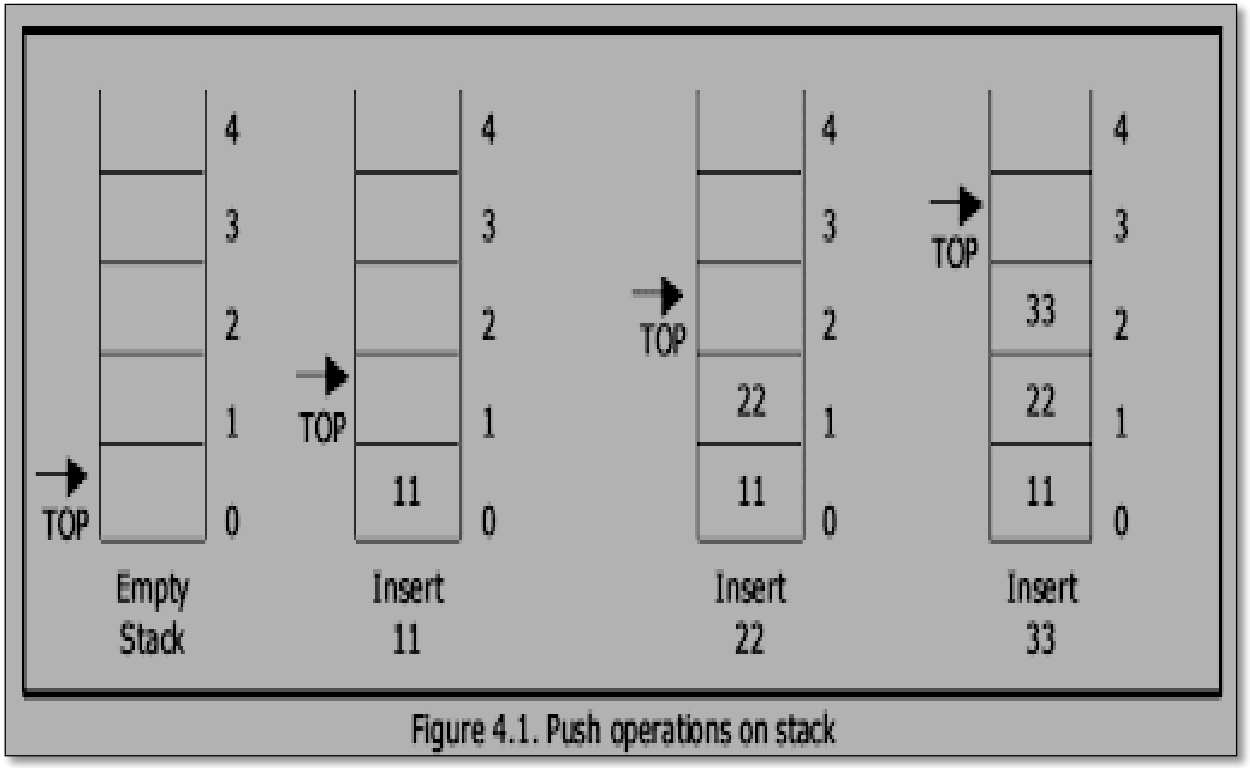


Figure 4.1. Push operations on stack

POP Operations on stack

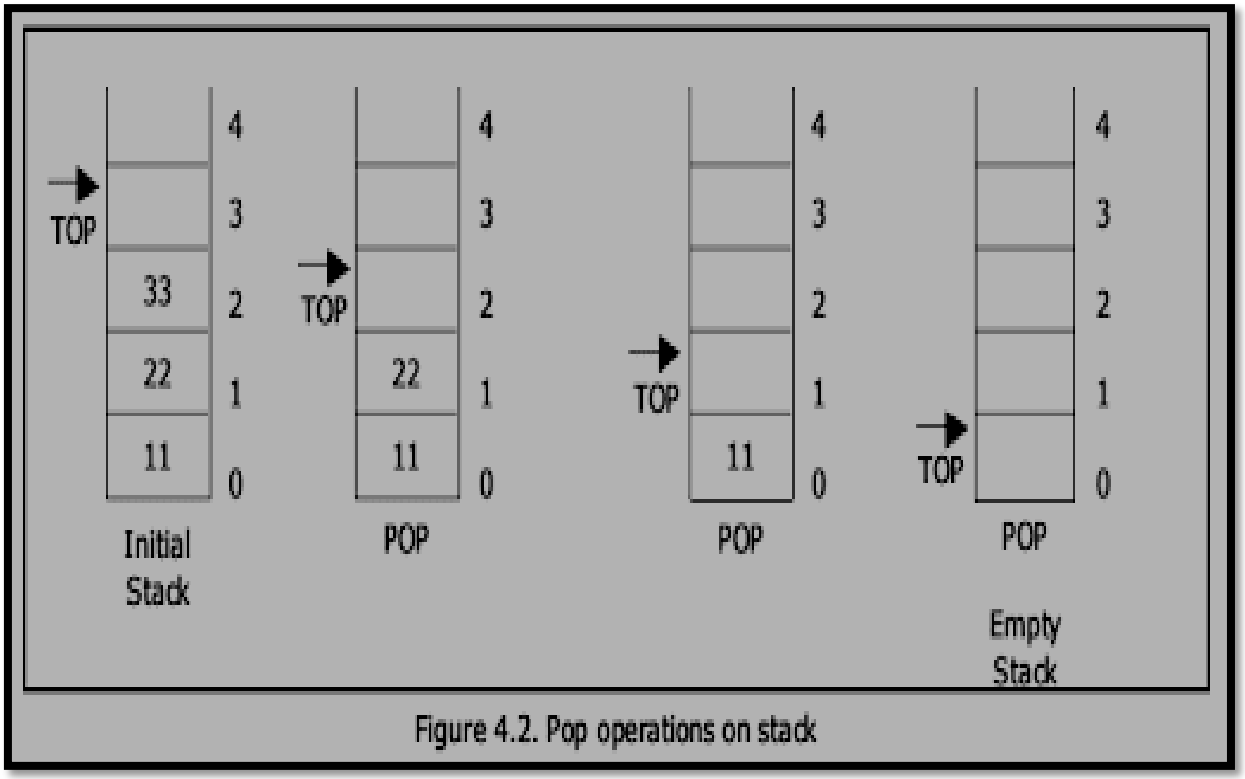
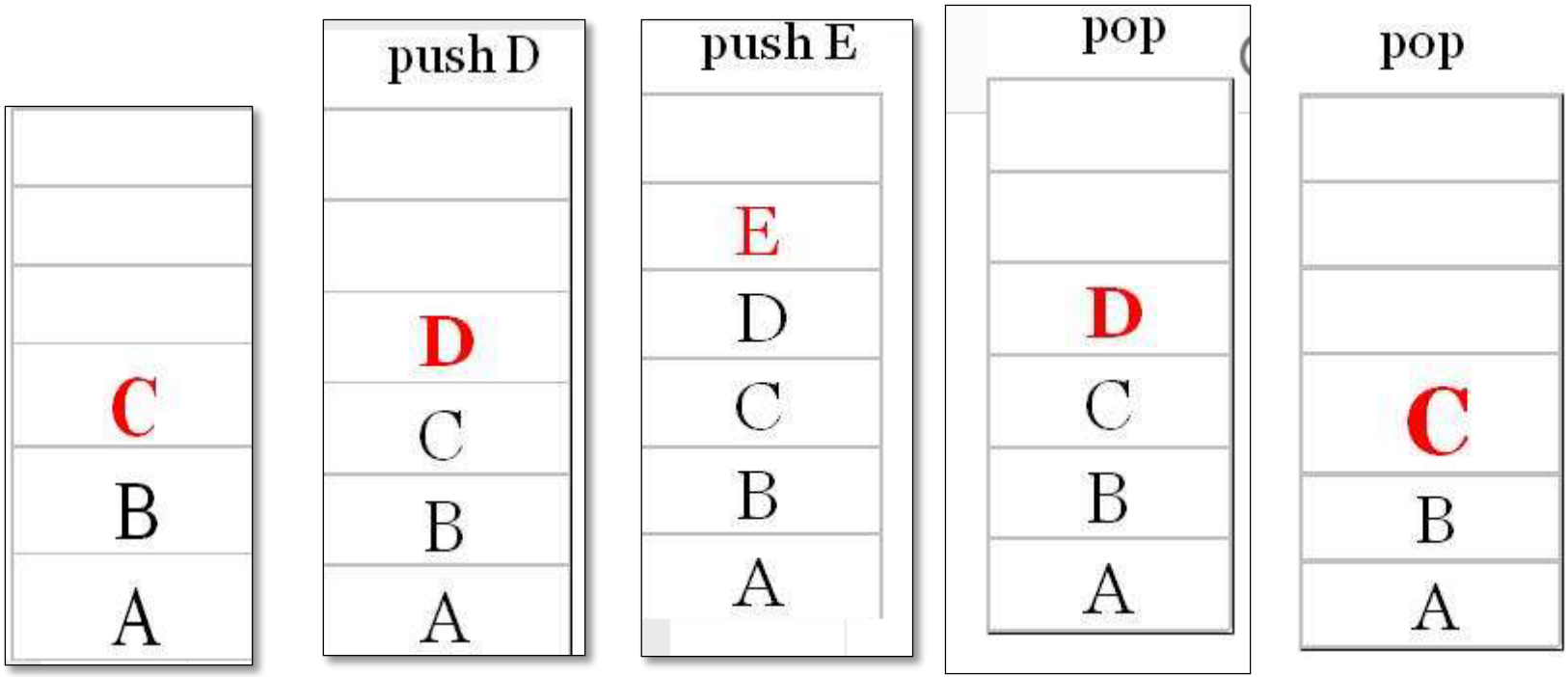
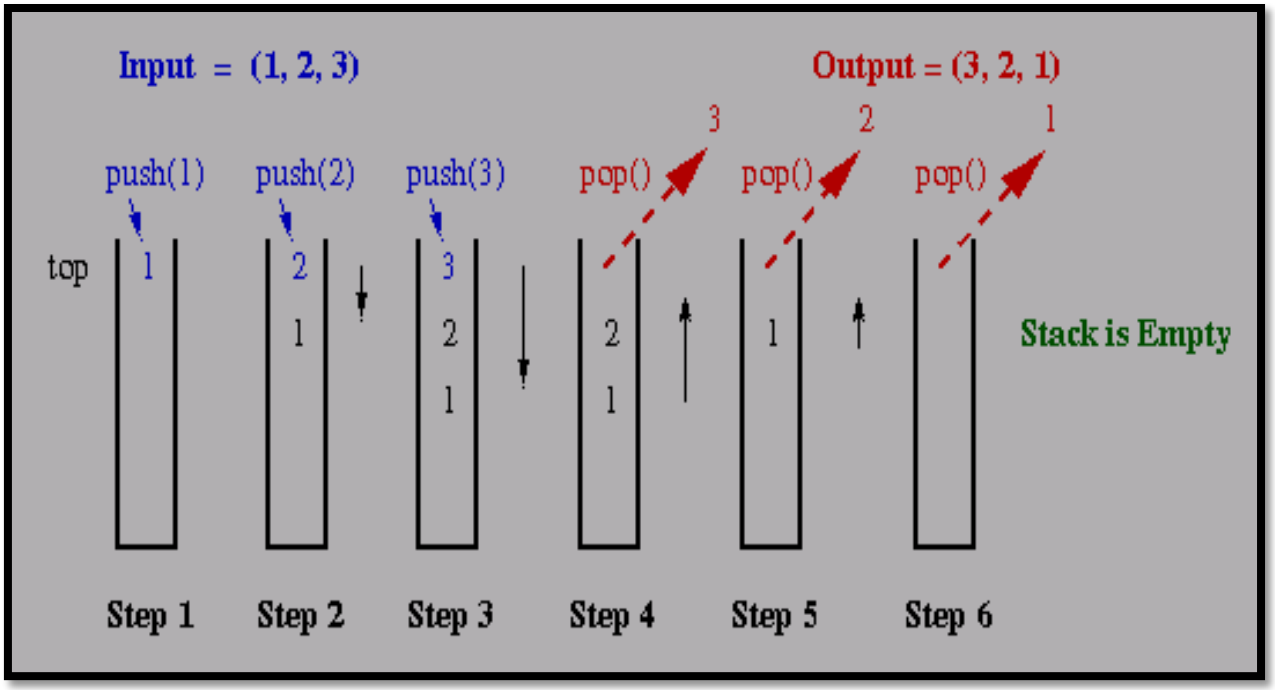


Figure 4.2. Pop operations on stack

Operations of Stack



Operations of Stack



Operations of Stack

Operation	Return Value	Stack Contents
S.push(5)	–	[5]
S.push(3)	–	[5, 3]
len(S)	2	[5, 3]
S.pop( )	3	[5]
S.is_empty( )	False	[5]
S.pop( )	5	[ ]
S.is_empty( )	True	[ ]
S.pop( )	"error"	[ ]
S.push(7)	–	[7]
S.push(9)	–	[7, 9]
S.top( )	9	[7, 9]
S.push(4)	–	[7, 9, 4]
len(S)	3	[7, 9, 4]
S.pop( )	4	[7, 9]
S.push(6)	–	[7, 9, 6]
S.push(8)	–	[7, 9, 6, 8]
S.pop( )	8	[7, 9, 6]

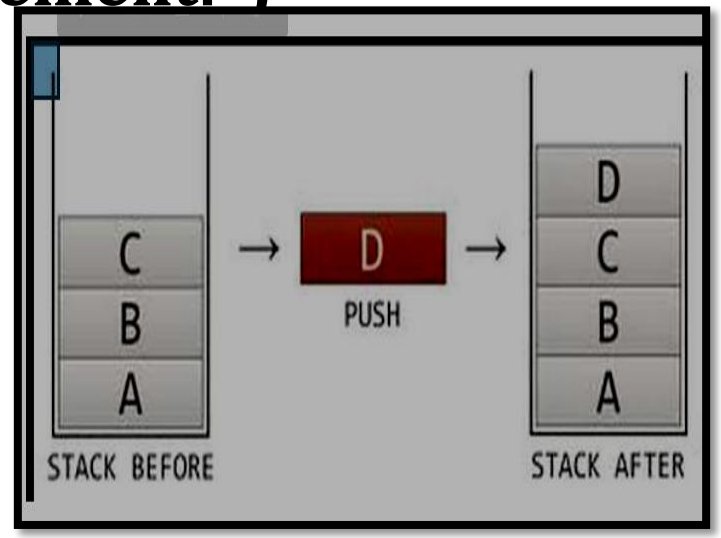
Implementation Using List

- Python list can be used as the stack.
- It uses the append() method to insert elements to the list where stack uses the push() method.
- The list also provides the pop() method to remove the last element, but there are shortcomings in the list.
- The list becomes slow as it grows.

Operations of Stack: Push

- **Push:-**

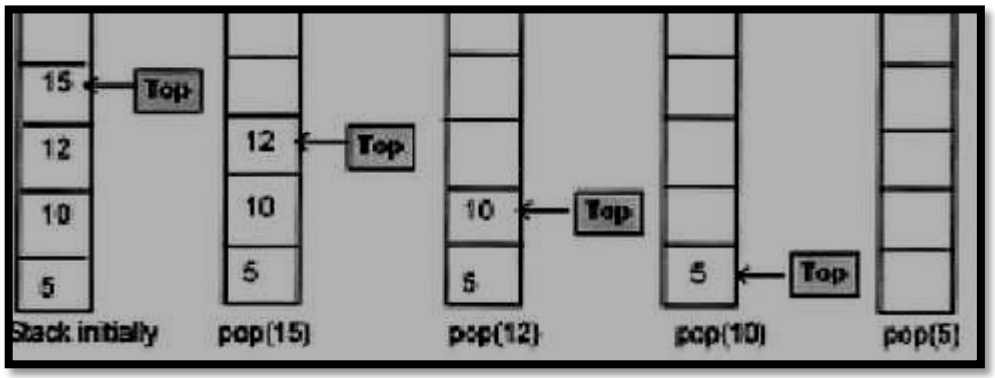
```
def push(stack, item):
if len(s)==size: # check wether the stack is full or not
print("Stack is Full!!!!")
else:
element=input("Enter the element:")
s.append(element)
```



Operations of Stack: Push

- **POP:-**

```
def pop(stack):
if not s:# or if len(stack)==0
print("Stack is Empty!!!")
else:
e=s.pop()
```



```python
# Stack implementation in python
# Creating a stack
def create_stack():
    stack = []
    return stack

# Creating an empty stack
def check_empty(stack):
    return len(stack) == 0
# Adding items into the stack
def push(stack, item):
    stack.append(item)
    print("pushed item: " + item)
# Removing an element from the stack
def pop(stack):
    if (check_empty(stack)):
        return "stack is empty"
    return stack.pop()

stack = create_stack()
push(stack, str(1))
push(stack, str(2))
push(stack, str(3))
push(stack, str(4))
print("popped item: " + pop(stack))
print("popped item: " + pop(stack))
print("stack after popping an element: " + str(stack))
```

```
pushed item: 1
pushed item: 2
pushed item: 3
pushed item: 4
popped item: 4
popped item: 3
stack after popping an element: ['1', '2']
```

Implementation of Stack using class

```
class Stack:
    def __init__(self):
        self.items = []

    def isEmpty(self):
        return self.items == []

    def push(self, item):
        self.items.append(item)

    def pop(self):
        if self.isEmpty():
            return -1
        else:
            return self.items.pop()

    def peek(self):
        return self.items[len(self.items)-1]

    def size(self):
        return len(self.items)
    def display(self):
        return self.items
```

lass

```
from stack import Stack
s=Stack()
choice=0
while choice<6:
    print('STACK OPERATIONS')
    print('1 Push Element')
    print('2 Pop Element')
    print('3 Peep Element')
    print('4 Display Element')
    print('5 size of stack')
    print('6 Exit')
    choice=int(input('Enter Your Choice:'))
    if choice==1:
        item=int(input('Enter Element you want to insert:'))
        s.push(item)
    elif choice==2:
        item=s.pop()
        if item==-1:
            print('The Stack is Empty')
        else:
            print('Popped Element =',item)
    elif choice==3:
        item=s.peek()
        print('Topmost Element =',item)
    elif choice==4:
        item=s.display()
        print('Element in stacks are =',item)
    elif choice==5:
        item=s.size()
        print('Size of stack is  =',item)
    else:
```

Application of stack

Expression Evaluation:-

- Stack is used to evaluate prefix, postfix and infix expressions.

Expression Conversion:-

- An expression can be represented in prefix, postfix or infix notation. Stack can be used to convert one form of expression to another.

Recursion:-

- It is widely used in concept of recursion because of its LIFO characteristics.

Syntax Parsing :-

- Many compilers use a stack for parsing the syntax of expressions, program blocks etc. before translating into low level code.

Application of stack

Parenthesis Checking:-

- Stack is used to check the proper opening and closing of parenthesis.

Function Call:-

- Stack is used to keep information about the active functions or subroutines.

String Reversal:-

- Stack is used to reverse a string. We push the characters of string one by one into stack and then pop character from stack.

Backtracking:-

- Suppose we are finding a path for solving maze problem. We choose a path and after following it we realize that it is wrong. Now we need to go back to the beginning of the path to start with new path. This can be done with the help of stack.

Polish Notations

There are basically three types of polish notation:

Infix:

- When the operator is written between two operands then it is known as Infix notation. For example A+B

Prefix:

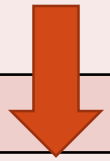
- When the operator is written before their operands then it is known as Prefix notation. For example +AB

Postfix:

- When the operator is written after their operands then it is known as Postfix notation. For example AB+

Conversion of Infix Expression into Postfix Expression

- Before understanding the conversion process of Infix expression into Postfix expression we must know the precedence of each operator.
- Following are the precedence of each operator from highest to lowest.

Operator	Priority	Precedence
( ), [ ]	Highest	Left-> Right
^		Right->Left
*,/		Left-> Right
+,-	Lowest	Left-> Right

Conversion of Infix Expression into Postfix Expression

For example, consider the following expression

5 * (6 + 2) - 12 / 4

- Since parenthesis has the highest precedence among the arithmetic operators, (6 +2) = 8 will be evaluated first.
- Now, the expression becomes

5 * 8 - 12 / 4

- * and / have equal precedence and their associatively is from left-to-right. So, start evaluating the expression from left-to-right.

5 * 8 = 40 and 12 / 4 = 3

- Now, the expression becomes

40 - 3

- And the value returned after the subtraction operation is 37.

Infix to postfix algorithm

1) Initialize an empty stack.
2) Scan the Infix string from left to right.
3) If the scanned character is an **operand**, add it to the Postfix string.
4) If the scanned character is left parenthesis **"("**, PUSH it into stack.
5) If the scanned character is an **operator** and if the stack is empty or contains **"("** at topmost element then push the character to stack.
6) If scanned character is right parenthesis **")"** then POP all the operators from stack until **"("** is encountered in the stack.
7) If the scanned character is an **Operator** and the stack is not empty, compare the precedence of the character with the element on top of the stack as follow:

a) If precedence of scanned operator is **less then or equal** to the precedence of topmost operator in the stack then **POP** the operator from the stack and append it into postfix string and push scanned operator into stack.

b) If precedence of scanned operator is greater then the precedence of topmost operator in the stack then **PUSH** scanned operator into stack.

Infix to postfix algorithm

8) Repeat step 7 until “(“ is encountered into stack or stack becomes empty.
9) When all the characters are scanned from infix string, POP remaining characters from stack and append it into postfix string.

Infix to postfix Example (A+B) * C

Scanned Char	Stack	postfix
	Empty	Empty
(	(	Empty
A	(	A
+	(+	A
B	(+	AB
)	Empty	AB+
*	*	AB+
C	*	AB+C
		AB+C*

Infix to postfix Example (A+B) * (C+D)

Scanned Char	Stack	postfix
	Empty	Empty
(	(	Empty
A	(	A
+	(+	A
B	(+	AB
)	Empty	AB+
*	*	AB+
(	*(	AB+
C	*(	AB+C
+	*(+	AB+C
D	*(+	AB+CD
)	*	AB+CD+
		AB+CD+*

a+b*c/d*e-f+g*h/i

Scanned Char	Stack	postfix
	Empty	Empty
a	Empty	a
+	+	a
b	+	ab
*	+*	ab
c	+*	abc
/	+/	abc*
d	+/	abc*d
*	+*	abc*d/
e	+*	abc*d/e
-	-	abc*d/e*+
f	-	abc*d/e*+f

a+b*c/d*e-f+g*h/i

Scanned Char	Stack	postfix
f	-	abc*d/e*+f
+	+	abc*d/e*+f-
g	+	abc*d/e*+f-g
*	+*	abc*d/e*+f-g
h	+*	abc*d/e*+f-gh
/	+/	abc*d/e*+f-gh*
i	+/	abc*d/e*+f-gh*i
		abc*d/e*+f-gh*i/+

a/b^c+d*e-A*C

Scanned Char	Stack	postfix
	Empty	Empty
a	Empty	a
/	/	a
b	/	ab
^	/^	ab
c	/^	abc
+	+	abc^/
d	+	abc^/d
*	+*	abc^/d
e	+*	abc^/de
-	-	abc^/de*+
A	-	abc^/de*+A
*	-*	abc^/de*+A

a/b^c+d*e-A*C

Scanned Char	Stack	postfix
*	-*	abc^/de*+A
C	-*	abc^/de*+AC
		abc^/de*+AC*-

a^b*c-d+e/f/(g+h)

Scanned Char	Stack	postfix
	Empty	Empty
a	Empty	a
^	^	a
b	^	ab
*	*	ab^
c	*	ab^c
-	-	ab^c*
d	-	ab^c*d
+	+	ab^c*d-
e	+	ab^c*d-e
/	+/	ab^c*d-e
f	+/	ab^c*d-ef
/	+/	ab^c*d-ef/

a^b*c-d+e/f/(g+h)

Scanned Char	Stack	postfix
/	+/	ab^c*d-ef/
(	+/(	ab^c*d-ef/
g	+/(	ab^c*d-ef/g
+	+/(+	ab^c*d-ef/g
h	+/(+	ab^c*d-ef/gh
)	+/	ab^c*d-ef/gh+
		ab^c*d-ef/gh+/+

22-2^4+10/5+9-8

Scanned Char	Stack	postfix
	Empty	Empty
22	Empty	22
-	-	22
2	-	22 2
^	-^	22 2
4	-^	22 2 4
+	+	22 2 4^-
10	+	22 2 4^-10
/	+/	22 2 4^-10
5	+/	22 2 4^-10 5
+	+	22 2 4^-10 5/+
9	+	22 2 4^-10 5/+9
-	-	22 2 4^-10 5/+9+

22-2^4+10/5+9-8

Scanned Char	Stack	postfix
-	-	22 2 4^-10 5/+9+
8	-	22 2 4^-10 5/+9+8
		22 2 4^-10 5/+9+8-

Home Work

1. a*(b+c)*d
2. a+(b+c)*d
3. x *(c + d) +(j+ k) * n+ m* p
4. A + B * C /D –E +F * G+H
5. (A-B^C^D) * (E-F/D)
6. (A + B) * D + E / (F + A * D * E) + C

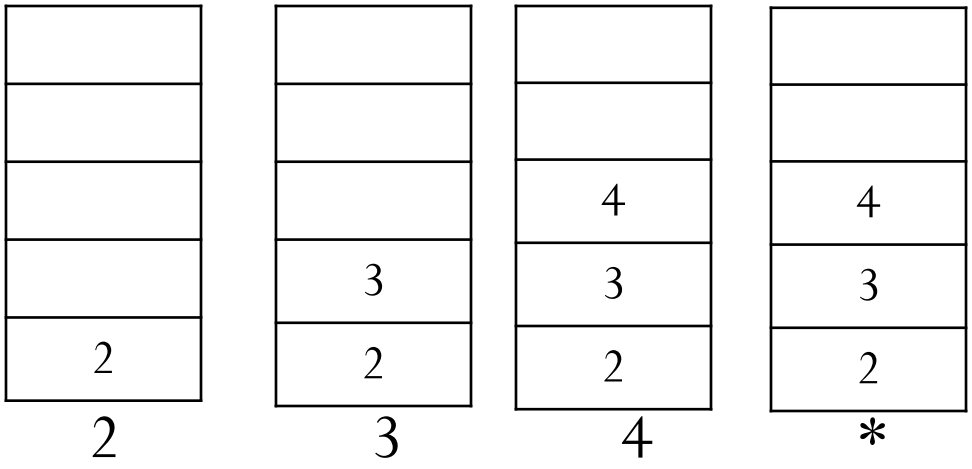
Home Work (answers)

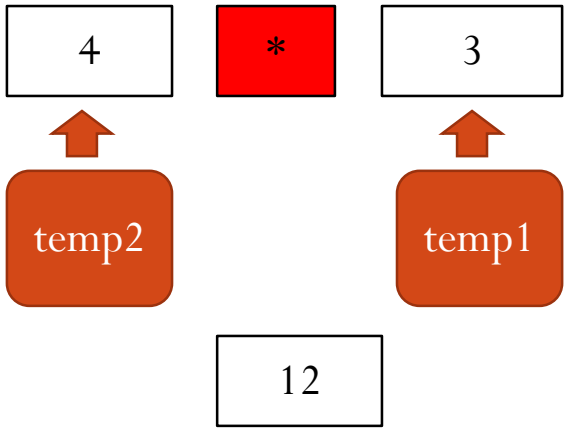
1. abc+*d*
2. abc+d*+
3. x c d+ * j k+ n*+ m p*+
4. A B C *D –E /+F G*+H+
5. ABC^D^- EFD/-*

postfix to infix algorithm

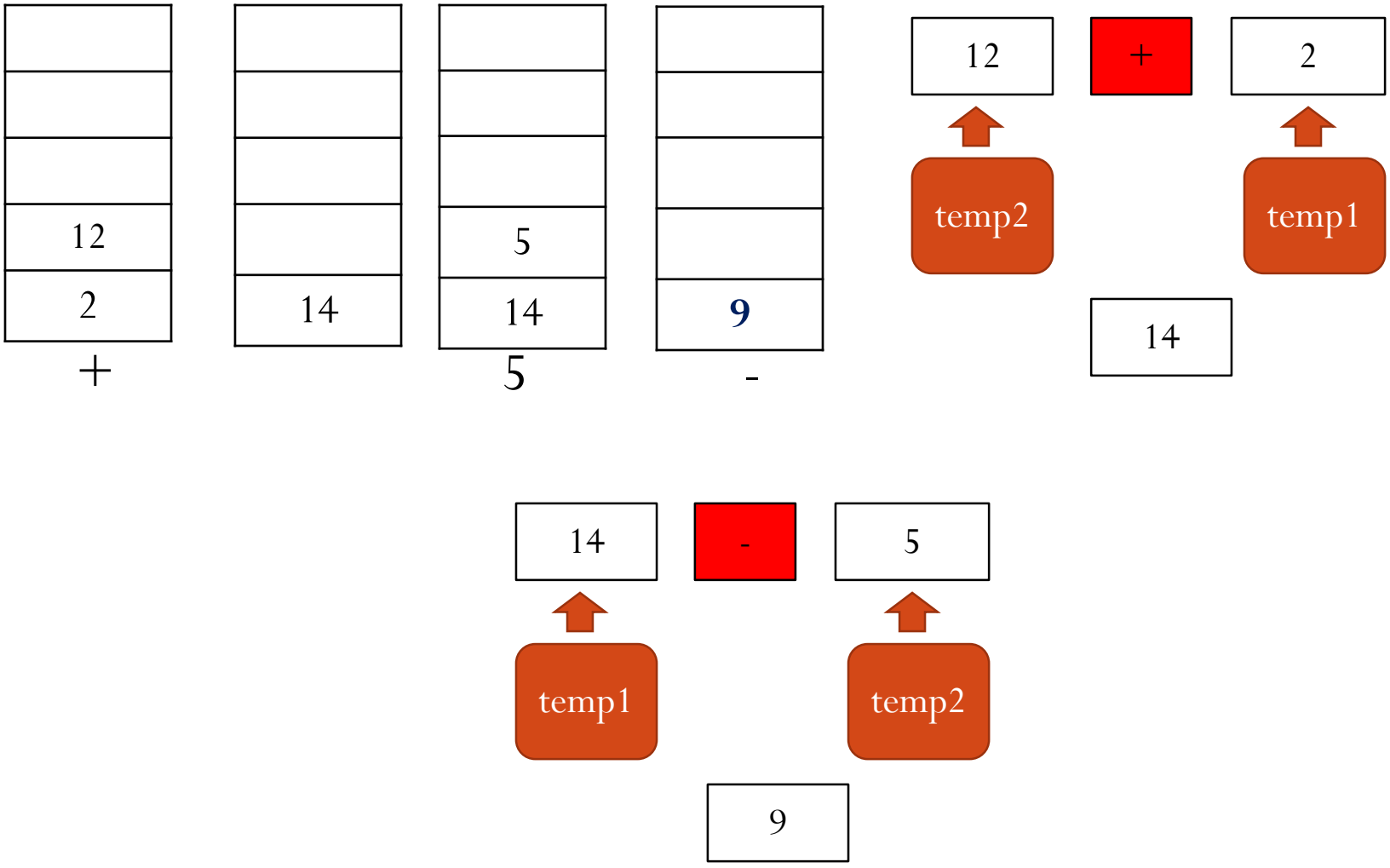
1) Initialize an empty stack.
2) Scan the postfix string from left to right.
3) If the scanned character is an **operand**, then PUSH it into stack.
4) If the scanned character is left parenthesis **"("**, PUSH it into stack.
5) If the scanned character is an **operator** then POP 2 values from stack. POP topmost element from stack and store its value in temp2.again POP one element from stack and store its value in temp1.
6) Evaluate ***temp1 operator temp2*** and PUSH result string back to stack.
7) Repeat above steps until last character. If there is only one value in the stack That value in the stack is the desired infix string.
8) If there are more values in the stack (Error) The user input has too many values

postfix to infix Example 234*+5-

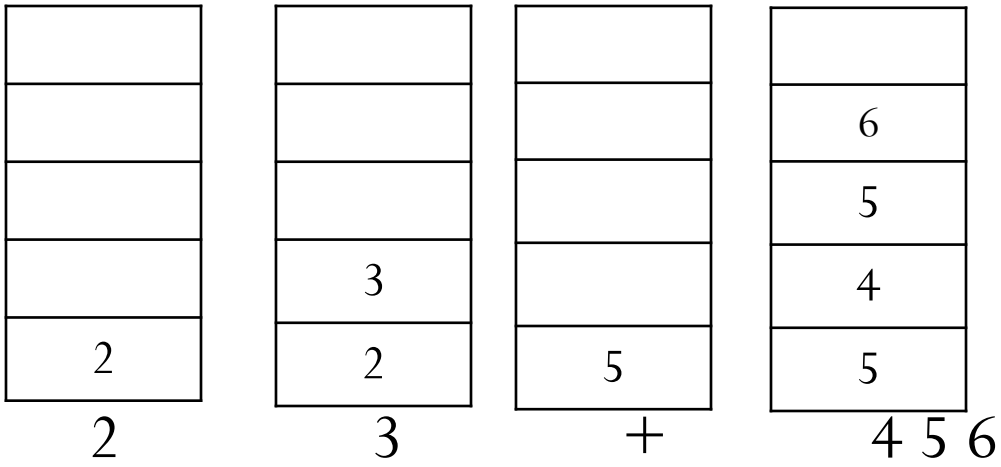


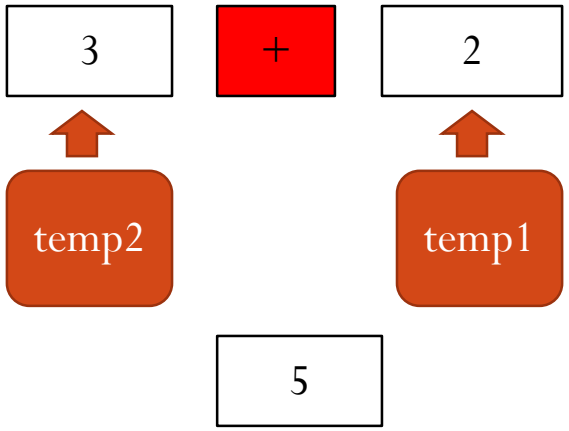


postfix to infix Example 234*+5-

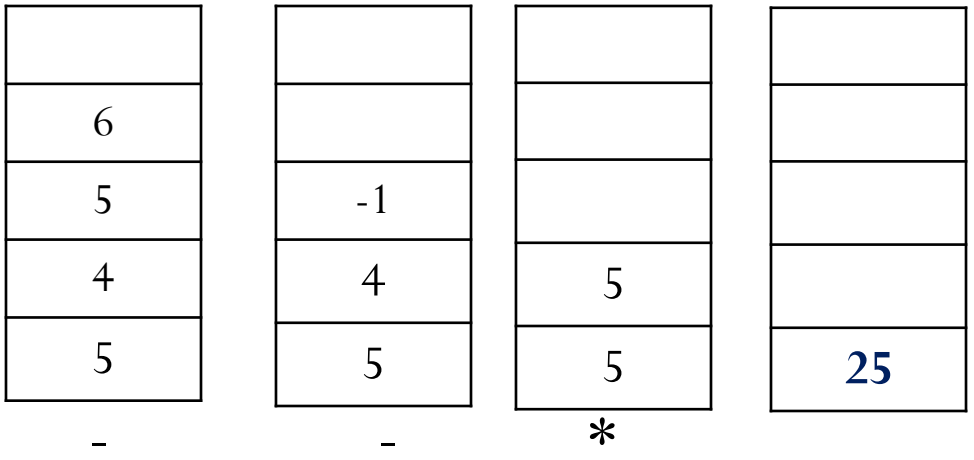


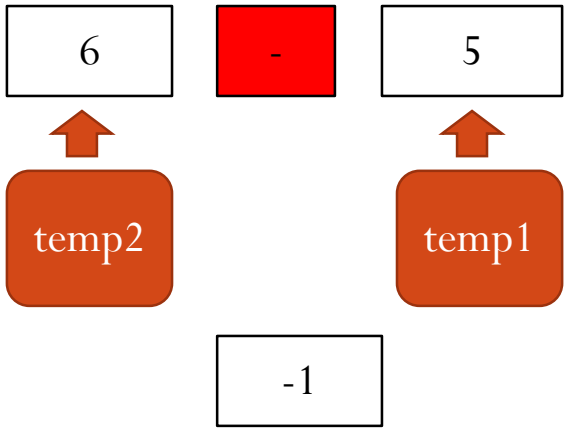
postfix to infix Example 2 3 + 4 5 6 – – *

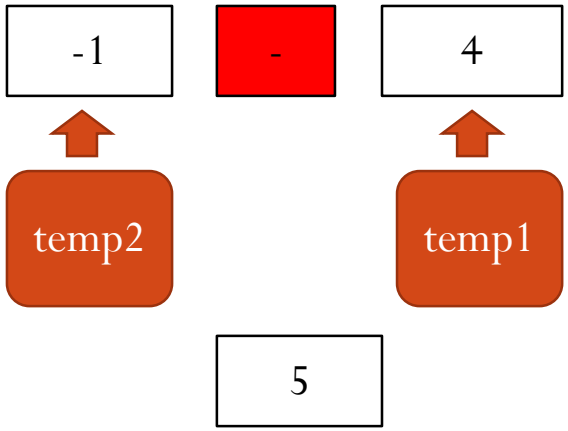




postfix to infix Example 2 3 + 4 5 6 – – *







Home work

-+3/841 top1+ top2

3+8/4-1

384/+1-

2 3 * 2 1 – / 5 4 1 – * +

Recursive Function

- Recursive function can be defined as a function calling itself.
- When a function is called from the same function then it is known as recursion.
- There are lots of problem that can be solved using concept of recursive function.
- Following are some of the problems that can be solved using recursive function:

  (1) Factorial Number
  (2) Greatest Common Divisor
  (3) Fibonacci Series

Factorial Number

- Factorial of the Number can be calculated using following equation:

n! = n * (n-1)!

FACT (n) =	1, if n = 1
	n * FACT (n-1), otherwise

Example:

5! = 5 * (5-1)!

= 5 * 4!

= 5 * 4 * (4-1)!

= 5 * 4 * 3!

= 5 * 4 * 3 * (3-1)!

= 5 * 4 * 3 * 2!

= 5 * 4 * 3 * 2 * (2-1)!

= 5 * 4 * 3 * 2 * 1!

= 5 * 4 * 3 * 2 * 1

= 120

Recursive Function

- Algorithm:

  If N=1 then

  Return 1

  Else

  Return (N * FACT (N-1))

```
def factorial(n):
    if(n <= 1):
        return 1
    else:
        return(n*factorial(n-1))
n = int(input("Enter number:"))
print("Factorial:")
print(factorial(n))
```

Fibonacci Series

- **Fibonacci Series can be given as:**

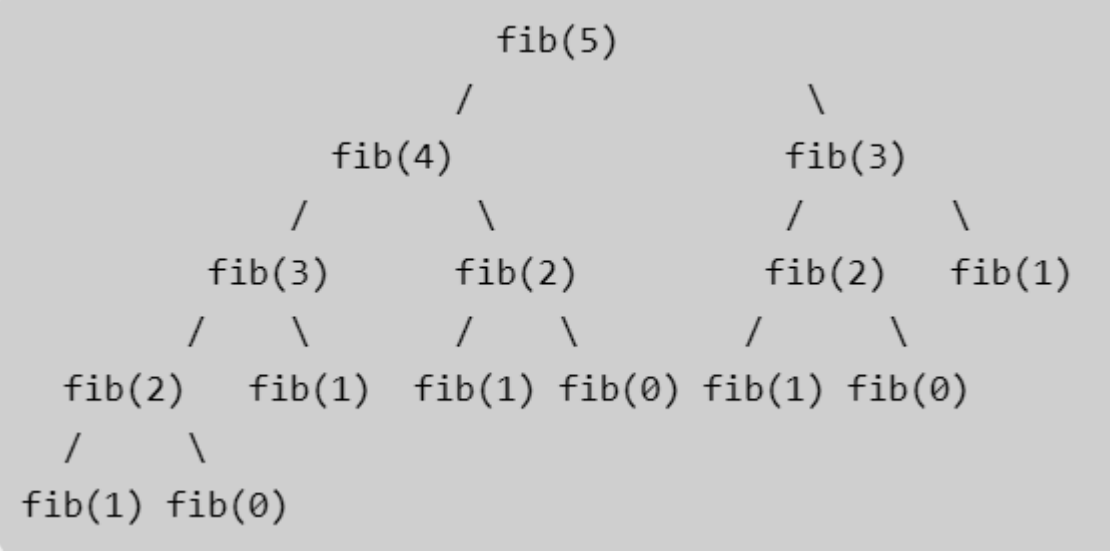
  0 1 1 2 3 5 8 13 21 34 ….

- In above Fibonacci Series first two terms are fixed 0 and 1. Third term of series can be calculated as a sum of first and second terms.
- Similarly fourth term can be calculated as a sum of second and third term and so on.
- We can find nth term of Fibonacci series using following recursive function:

FIBO (n) =	0, if n = 0 or 1
	1, if n = 2
	FIBO (n-1) + FIBO (n-2), if n > 2

Fibonacci Series

- **Example:**
  FIBO (5) = FIBO (4) + FIBO (3)
  = FIBO (3) + FIBO (2) + FIBO (2) + FIBO (1)
  = FIBO (2) + FIBO (1) + 1 + 1 + 0
  = 1 + 0 + 1 + 1 + 0
  = 2 + 1
  = 3



Fibonacci Series

- **Algorithm:**

  Step 1:If N=0 or N=1 then

  Return 0

  step 2:Else if N=2 then

  Return 1

  Else

  Return (FIBO (N-1) + FIBO (N-2))

```
def fibonacci(n):
    if(n <= 1):
        return n
    else:
        return(fibonacci(n-1) + fibonacci(n-2))
n = int(input("Enter number of terms:"))
print("Fibonacci sequence:")
for i in range(n):
    print(fibonacci(i))
```

Greatest Common Divisor

- **Greatest Common Divisor** of two integer number is the largest integer number that divides both the numbers.
- Greatest Common Divisor can be calculated using following recursive equation:

GCD (m, n)=	n, if n divides m
	GCD (n, m mod n), otherwise

- **Example:**

GCD (62, 8) = 62 % 8 = 6
= GCD (8, 6)
= 8 % 6 = 2
= GCD (6, 2)
= 6 % 2 = 0
Thus, GCD (62, 8) = 2

Greatest Common Divisor

- **Algorithm:**

  Step 1: REMINDER = (M mod N)

  Step 2: If REMINDER = 0 then

  Return N

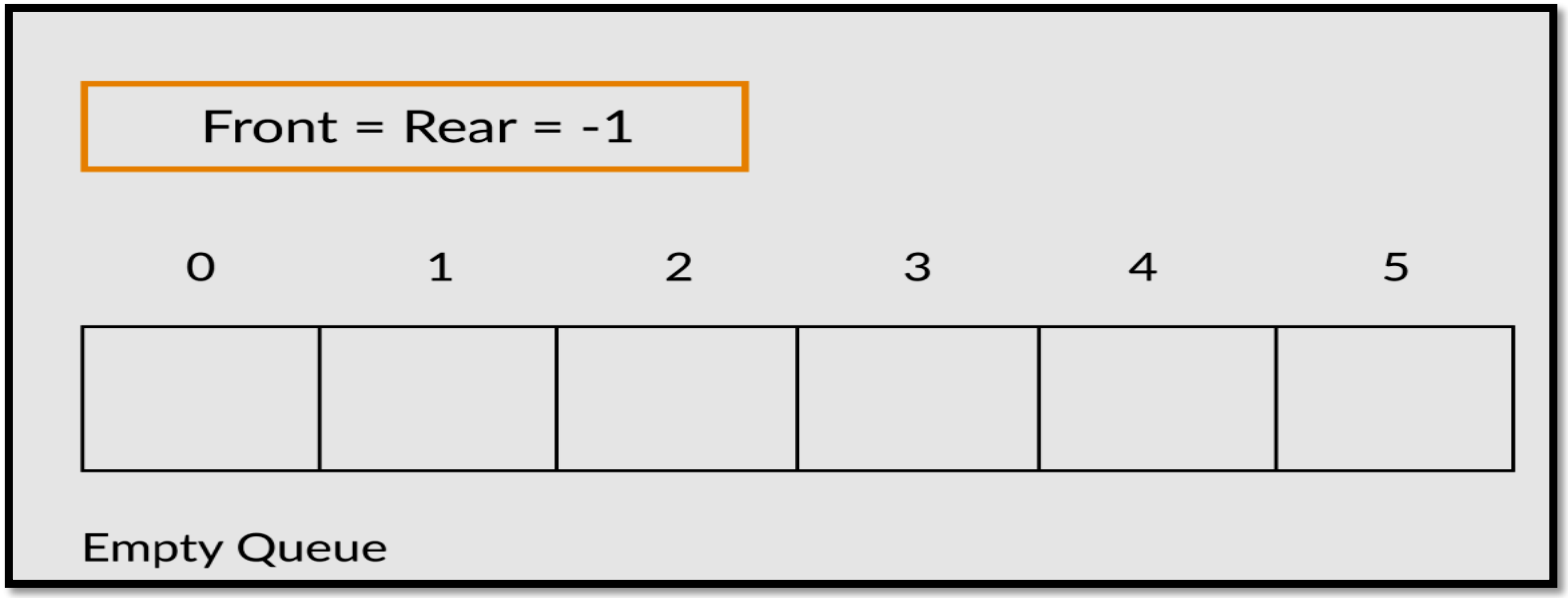
  Else

  Return (GCD (N, REMINDER))

```
def gcd(a,b):
    if(b==0):
        return a
    else:
        return gcd(b,a%b)
a=int(input("Enter first number:"))
b=int(input("Enter second number:"))
GCD=gcd(a,b)
print("GCD is: ")
print(GCD)
```

Queue

- Queue is a linear Data Structure in which insertion operation is performed at one end called Rear and deletion operation is performed at another end called front.



Queue

- Since insertion operation is performed at rear end and deletion operation is performed at front end the element which is inserted first is first to delete. So Queue is also known as First in First out (FIFO).

Example:

- Consider a Queue of Students at Fee Counter waiting to pay fee.

Various operations implemented on a queue are:

- Insert
- Delete

Implementation of Queue

- A variable front and rear is always used with queue.
- Front variable contains an index of first element in the Queue.
- Rear Variable contains an index of last element in queue.
- Initially when queue is empty, front and rear=-1.
- If Queue contains only one element then value of front and rear=0.
- Each time an element is inserted into queue rear variable is increment by one, so it points last element in queue.
- Each time an element is deleted from queue front variable is increment by one, so it points first element in queue.
- **Queue overflow condition:-**

Rear=size-1

- **Queue underflow condition:-**

Front=-1.

Basic operation Queue:-

- A queue is an object (an abstract data structure - ADT) that allows the following operations:
- Enqueue - The enqueue is an operation where we add items to the queue. If the queue is full, it is a condition of the Queue
- Dequeue - The dequeue is an operation where we remove an element from the queue. An element is removed in the same order as it is inserted. If the queue is empty, it is a condition of the Queue Underflow.
- IsEmpty: Check if the queue is empty
- IsFull: Check if the queue is full
- Peek: Get the value of the front of the queue without removing it

Implementation of Queue

- Queue in Python can be implemented in the following ways:
- list
- collections.deque
- queue.Queue

Working of Queue:-

- Queue operations work as follows:
- two pointers FRONT and REAR
- FRONT track the first element of the queue
- REAR track the last element of the queue
- initially, set value of FRONT and REAR to -1

Implementation of Queue

Enqueue Operation:-

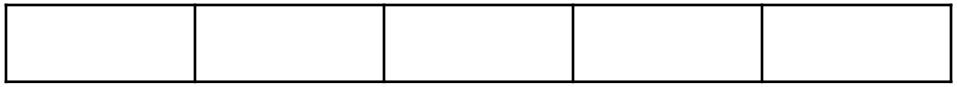
- check if the queue is full
- for the first element, set the value of FRONT to 0
- increase the REAR index by 1
- add the new element in the position pointed to by REAR

Dequeue Operation:-

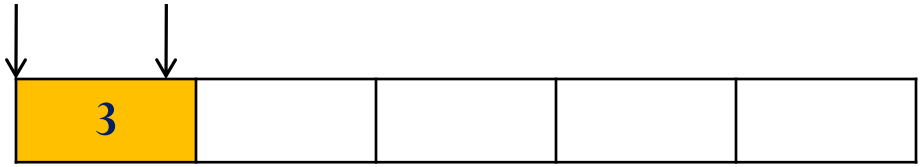
- check if the queue is empty
- return the value pointed by FRONT
- increase the FRONT index by 1
- for the last element, reset the values of FRONT and REAR to -1

Implementation of Queue (insertion)

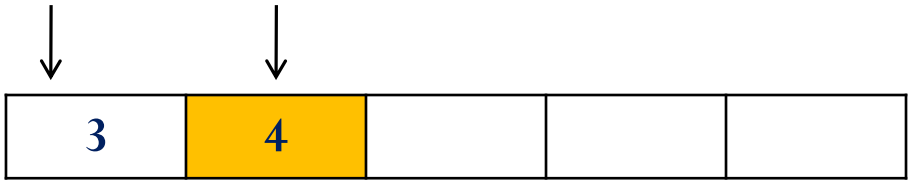
Front=-1
Rear=-1



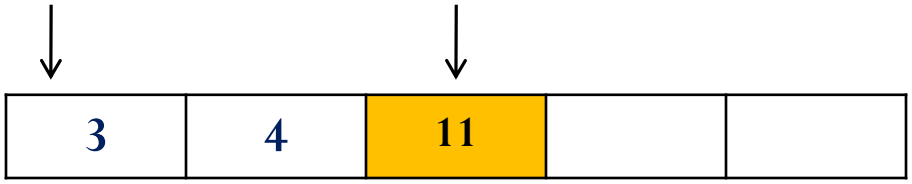
Front=0 Rear=0



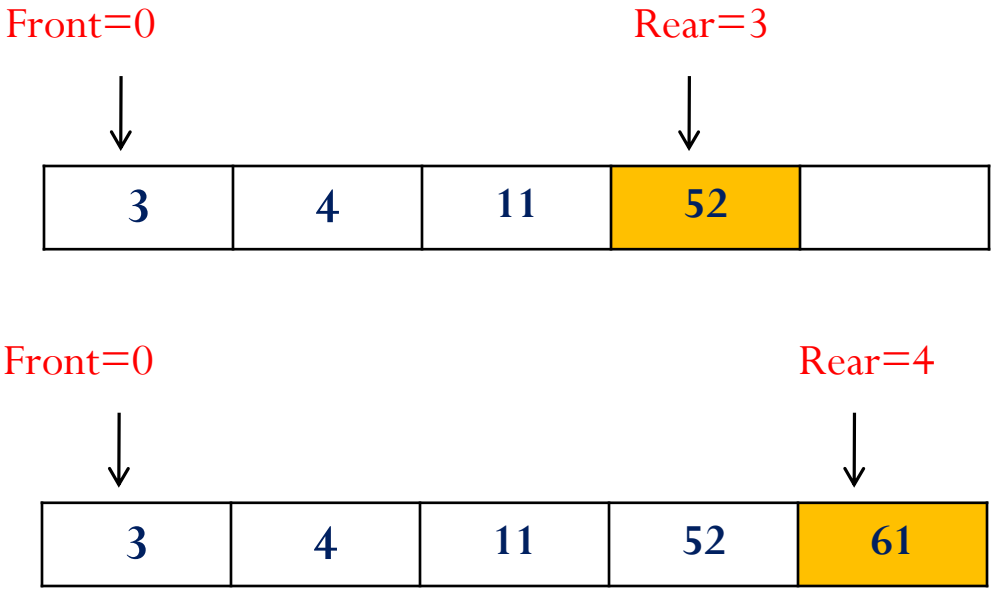
Front=0 Rear=1



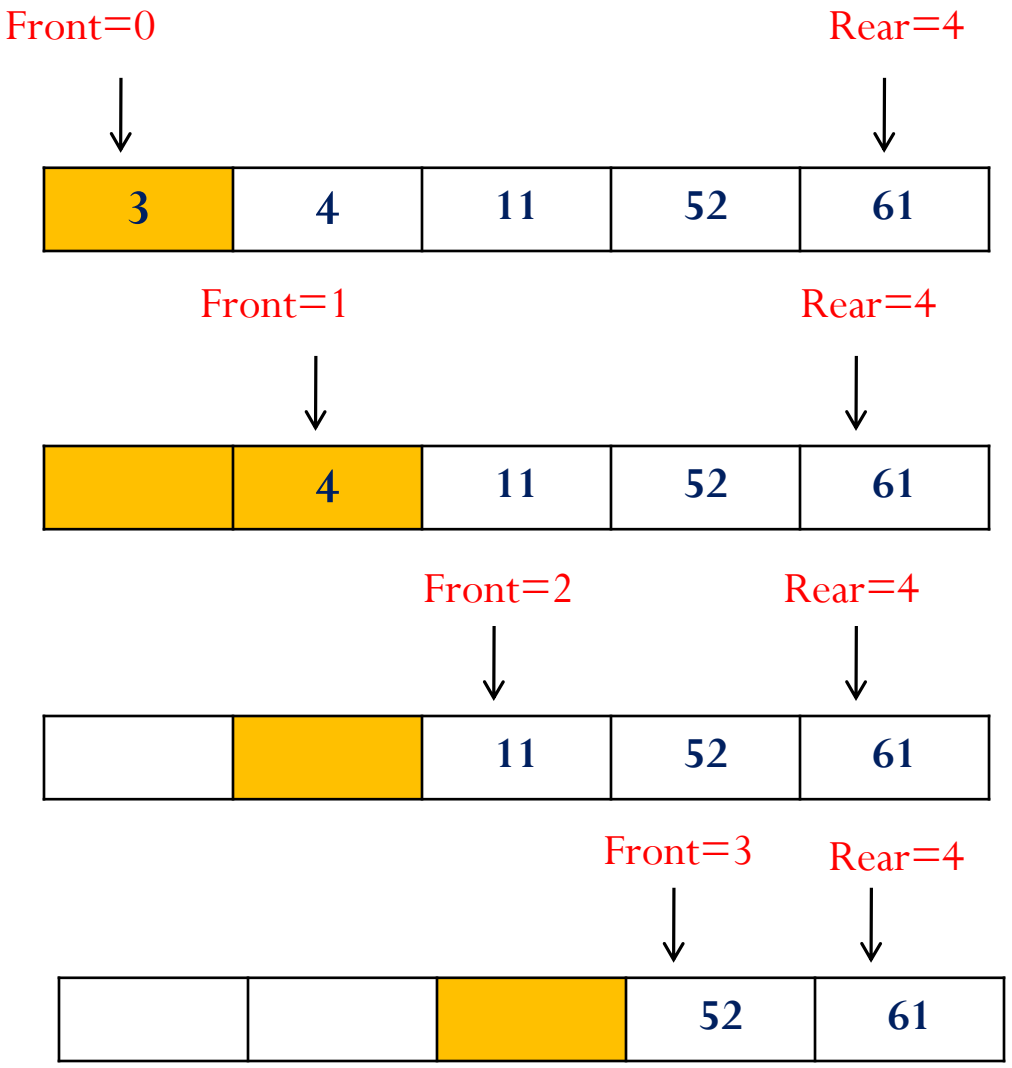
Front=0 Rear=2



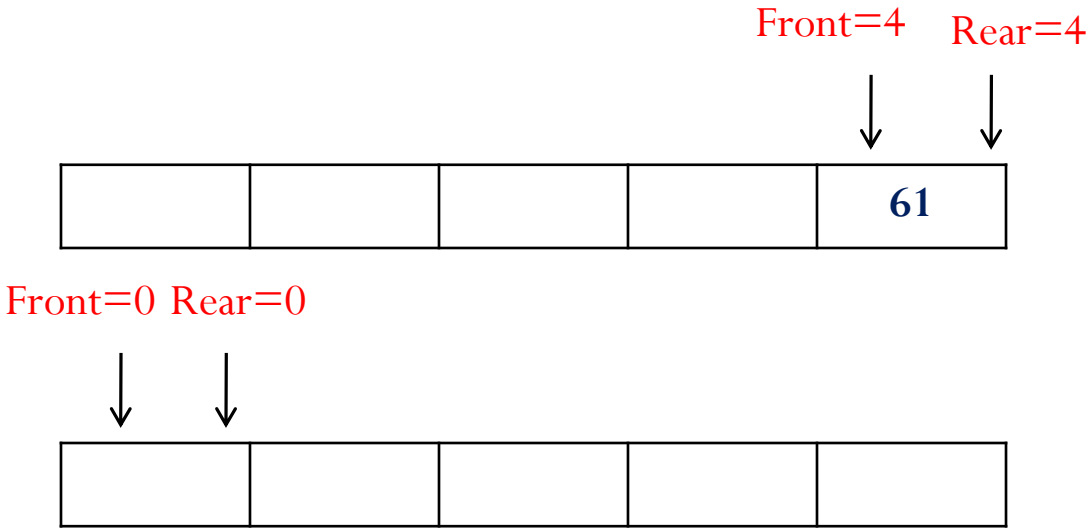
Implementation of Queue (Insertion)



Implementation of Queue (Deletion)



Implementation of Queue (Deletion)



Insertion operation Queue(Enqueue)

- Insert operation is used to insert an element into Queue.
- In order to insert an element into Queue first we have to check weather space is available in the Queue or not.
- If Queue is full then we can not insert an element into Queue.
- If value of ***REAR >=SIZE – 1*** then we can not insert an element into Queue. This condition is known as "**Overflow".**
- If Queue is not overflow then we can insert an element into Queue.
- First we have to **increment the value of REAR variable by one** and then insert an element into Queue.
- If the element which is inserted in Queue is **first element then we have to set the value of FRONT variable to 0.**

Insert Operation in simple Queue

Insert (Q, Front, Rear, X)

Step 1:[Check of Queue Overflow]

If REAR >= SIZE – 1 then

Write "Queue is Overflow"

Exit

Step 2: [Increment rear pointer by one]

Rear= Rear + 1

Step 3: [Perform Insertion]

Q[Rear] = X

Step 4 :[If first element is inserted]

If Front = -1 then

Front = 0

Step5 :[finished]

Exit

Deletion operation Queue(Dequeue)

- Delete operation is used to delete an element from Queue.
- In order to delete an element from Queue first we have to check weather Queue is empty or not.
- If Queue is empty then we can not delete an element from Queue. IF ***Front =-1*** ,This condition is known as "**Underflow".**
- If Queue is not underflow then we can delete element from Queue.
- After deleting element from Queue we have to set the value of FRONT and REAR variables according to the elements in the Queue.

Delete Operation in simple queue

Delete (Q, Front, Rear, X)

Step 1:[Check of Queue underflow]

If FRONT = -1then

Write "Queue is Underflow"

Exit

Step 2: [Remove an element from Queue]

X=Q[Front]

Step 3 :[Check for Queue condition]

If FRONT = REAR then

FRONT = -1

REAR = -1

Else

FRONT = FRONT + 1

Step 4 :[finished]

Exit

```
# Queue implementation in Python
class Queue:
    def __init__(self):
        self.queue = []
    # Add an element
    def enqueue(self, item):
        self.queue.append(item)
    # Remove an element
    def dequeue(self):
        if len(self.queue) < 1:
            return None
        return self.queue.pop(0)
    # Display  the queue
    def display(self):
        print(self.queue)

    def size(self):
        return len(self.queue)
q = Queue()
q.enqueue(1)
q.enqueue(2)
q.enqueue(3)
q.enqueue(4)
q.enqueue(5)
q.display()
q.dequeue()
print("After removing an element")
q.display()
```

[1, 2, 3, 4, 5]
After removing an element
[2, 3, 4, 5]

Implementation of Queue using list

```
# Implement Queue using List(Functions)
q=[]
def Enqueue():
    if len(q)==size: # check wether the stack is full or not
        print("Queue is Full!!!!")
    else:
        element=input("Enter the element:")
        q.append(element)
        print(element,"is added to the Queue!")
def dequeue():
    if not q:# or if len(stack)==0
        print("Queue is Empty!!!")
    else:
        e=q.pop(0)
        print("element removed!!:",e)
def display():
    print(q)
size=int(input("Enter the size of Queue:"))
while True:
        print("Select the Operation:1.Add 2.Delete 3. Display 4. Quit")
        choice=int(input())
        if choice==1:
            Enqueue()
        elif choice==2:
            dequeue()
        elif choice==3:
            display()
        elif choice==4:
            break
        else:
            print("Invalid Option!!!")
```

Implementation of Queue class

```
class Queue:
    def __init__(self):
        self.qu = []

    def isEmpty(self):
        return self.qu == []

    def add(self, element):
        self.qu.append(element)

    def delete(self):
        if self.isEmpty():
            return -1
        else:
            return self.qu.pop(0)

    def search(self,element):
        if self.isEmpty():
            return -1
        else:
            n=self.qu.index(element)
            return n+1

    def size(self):
        return len(self.element)
    def display(self):
        return self.qu
```

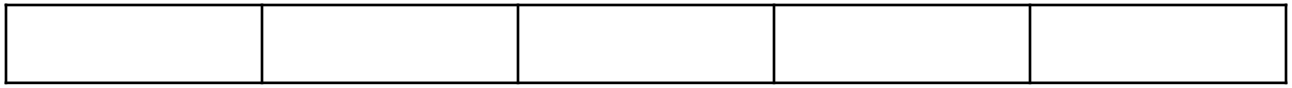
Implementation of Queue class

```
from que1 import Queue
q=Queue()
choice=0
while choice<5:
    print('QUEUE OPERATIONS')
    print('1 Add Element')
    print('2 Delete Element')
    print('3 Search Element')
    print('4 Exit Element')

    choice=int(input('Enter Your Choice:'))
    if choice==1:
        element=int(input('Enter Element you want to insert:'))
        q.add(element)
    elif choice==2:
        element=q.delete()
        if element==-1:
            print('The Queue is Empty')
        else:
            print('Removed  Element =',element)
    elif choice==3:
       element=int(input('Enter Element you want to search:'))
       pos=q.search(element)
       if pos==-1:
          print('The Queue is empty')
       else:
           print('Element found at ppostion:',pos)
    else:
        break
    print('Queue=',q.display())
```

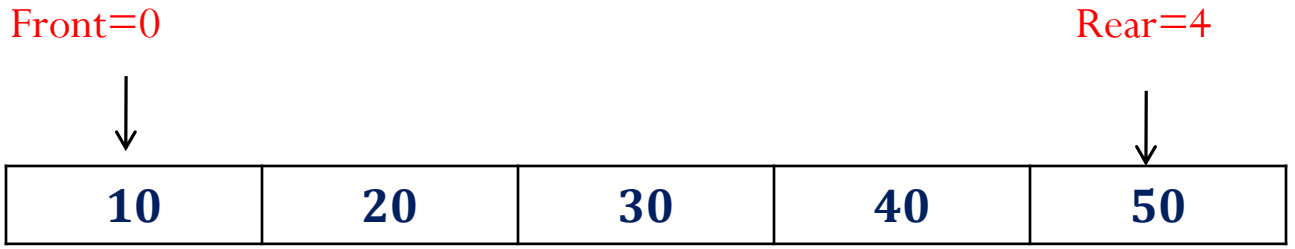
Limitation of simple Queue

- By the definition of a queue, when we add an element in Queue, rear pointer is increased by 1 whereas, when we remove an element front pointer is increased by 1.
- But in array implementation of queue this may cause problem as follows:
- Consider operations performed on a Queue (with SIZE = 5) as follows:
- Initially empty Queue is there so, **front = -1 and rear = -1**

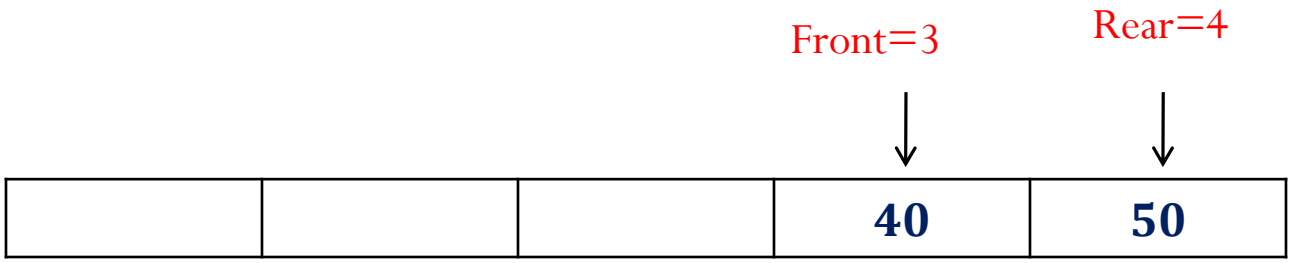


Limitation of simple Queue

- When we add 5 elements to queue,10,20,30,40,50 the state of the queue becomes as follows with



- Now suppose we delete 3 elements from Queue (10, 20,30) then, the state of the Queue becomes as follows:

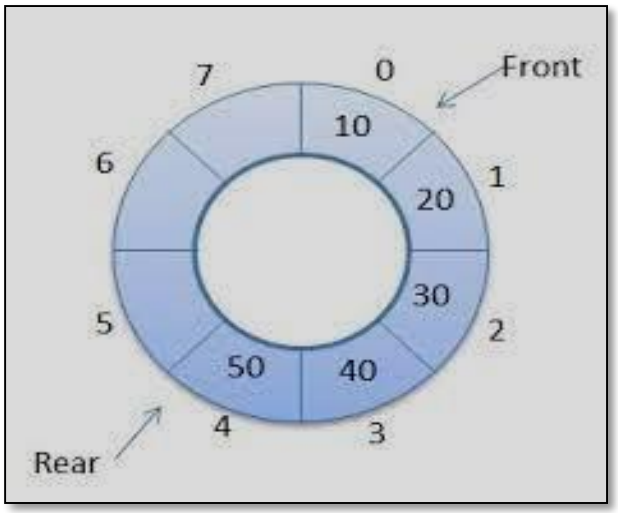


Limitation of simple Queue

- Now, actually we have deleted 3 elements from queue.
- There should be space for another 3 elements in the queue.
- But as rear pointer is pointing at last position and Queue overflow condition **(Rear == SIZE-1)** is true, we can't insert new element in the queue even if it has an empty space.
- Even if we have a free memory space we can not use that free memory space in simple queue.
- To overcome this problem there is another variation of queue called ***CIRCULAR QUEUE.***

Circular Queue

- Circular Queue is a linear Data Structure in which elements are arranged such that first element in the queue follows the last element.
- 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.
- Circular Queue is ***designed to overcome the limitation of Simple Queue.***



Circular Queue

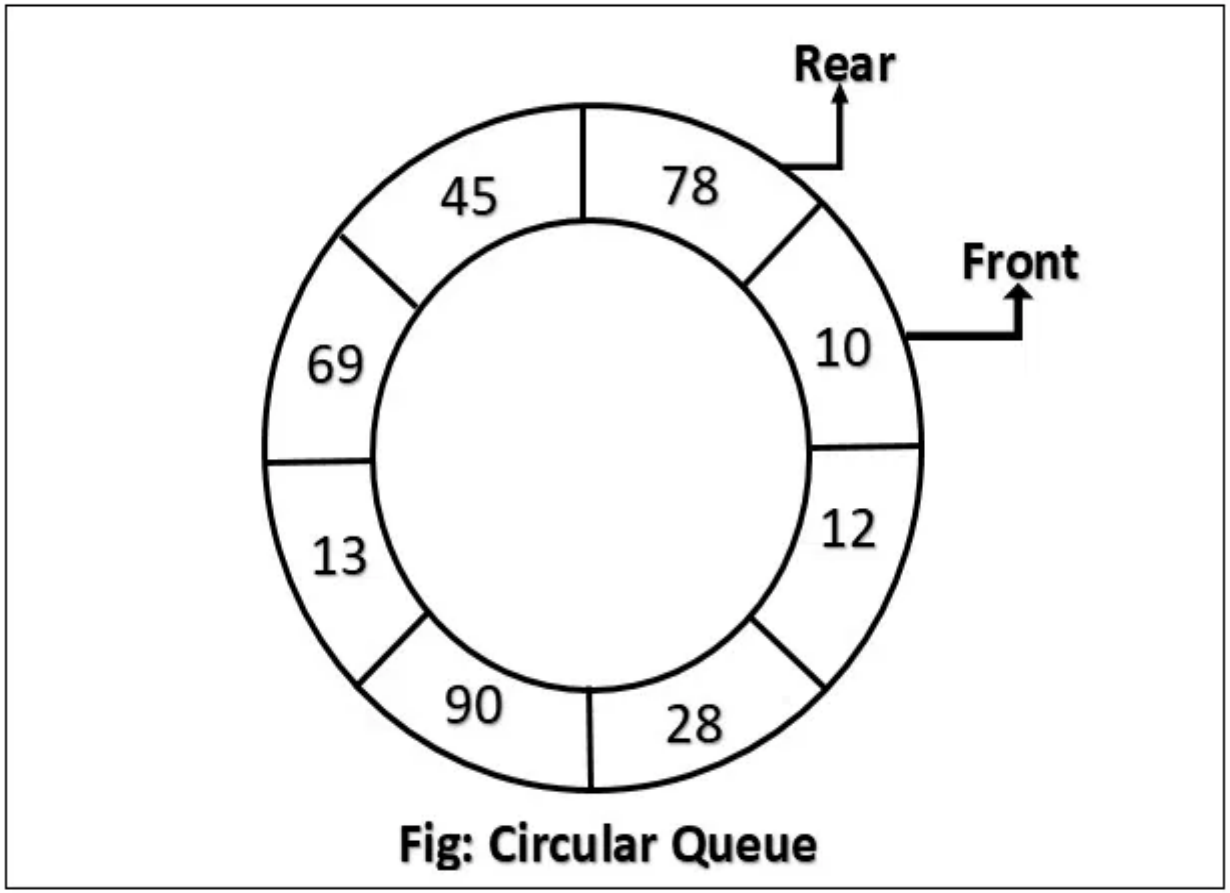
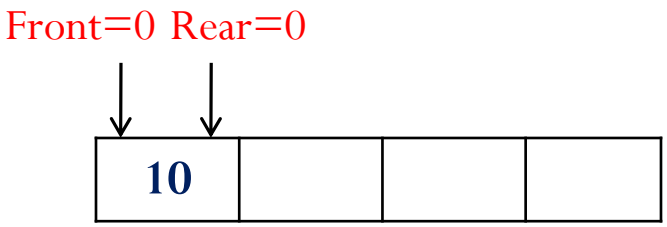


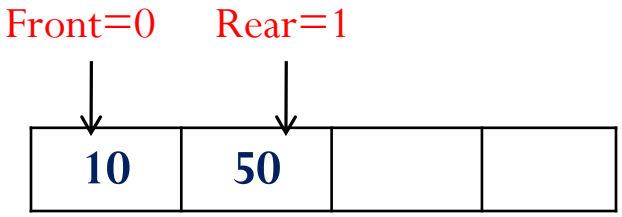
Fig: Circular Queue

Implementation of Circular Queue

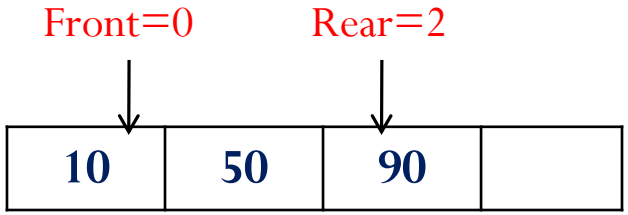
- Insert 10



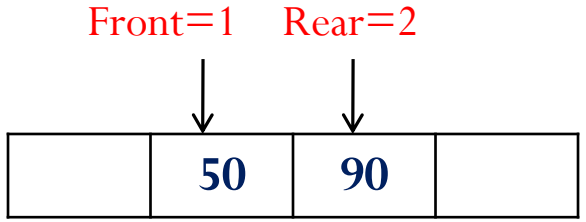
- Insert 50



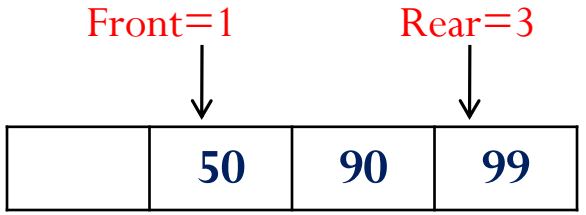
- Insert 90



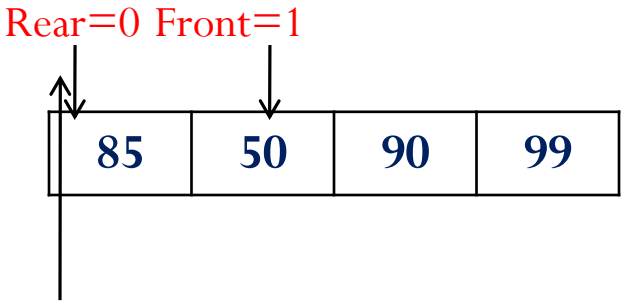
- Remove 10



- Insert 99



- Insert 85

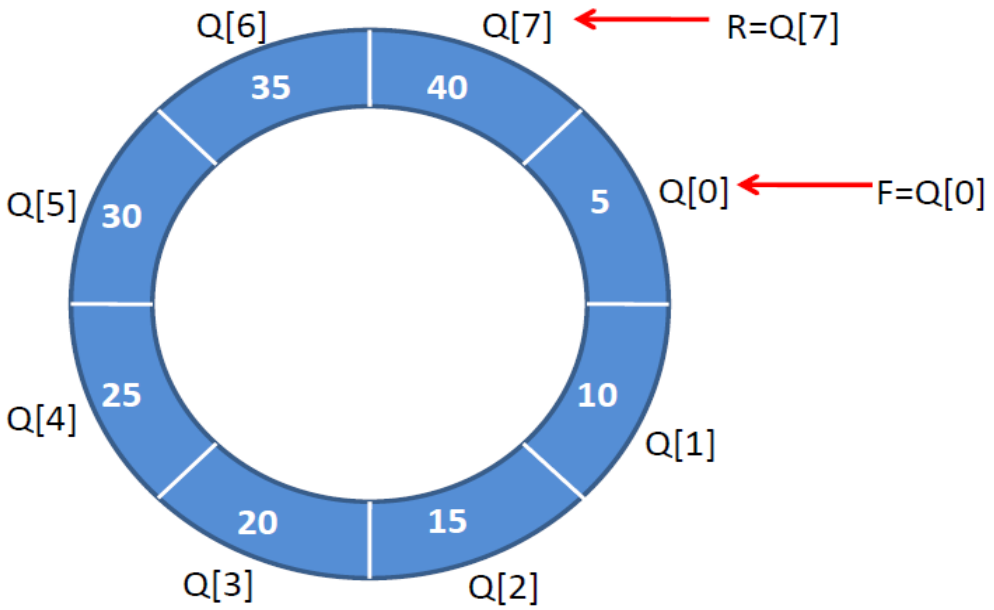


Algorithm for Circular Queue

- Initialize the queue, with size of the queue defined (maxSize), and head and tail pointers.
- **enqueue: Check if the number of elements is equal to maxSize - 1:**
- **If Yes, then return Queue is full**
- **If No, then add the new data element to the location of tail pointer and increment the tail pointer.**
- **dequeue: Check if the number of elements in the queue is zero:**
- **If Yes, then return Queue is empty**
- **If No, then increment the head pointer.**
- **size:**
- **If, tail >= head, size = tail - head**
- **But if, head > tail, then size = maxSize - (head - tail)**

Insert Operation of circular Queue

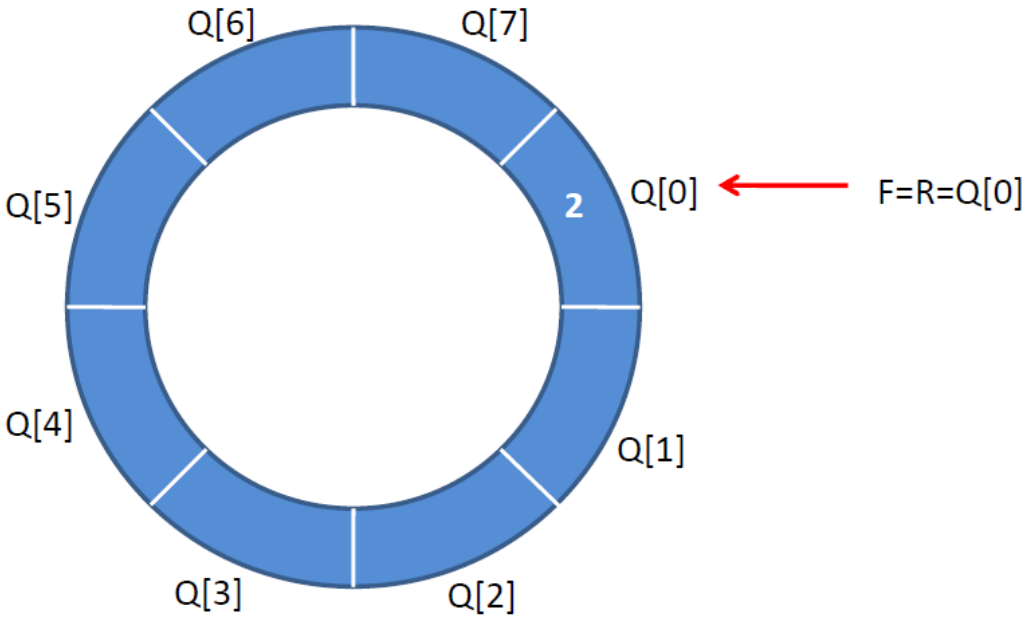
e.g.:



Queue is full(Overflow)

Insert Operation of circular Queue

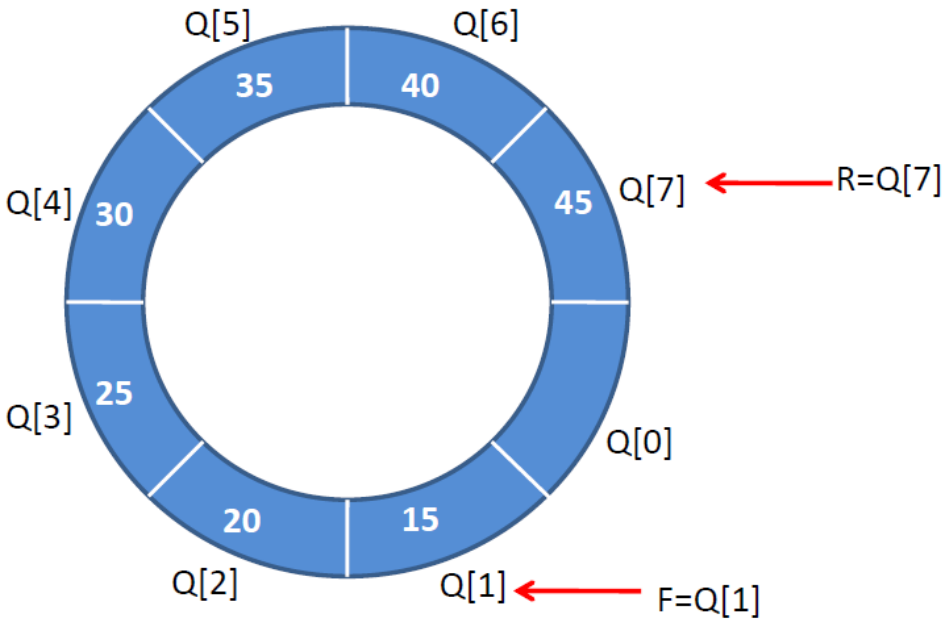
e.g.: (cond..)



If F=R=-1 then F=R=0

Insert Operation of circular Queue

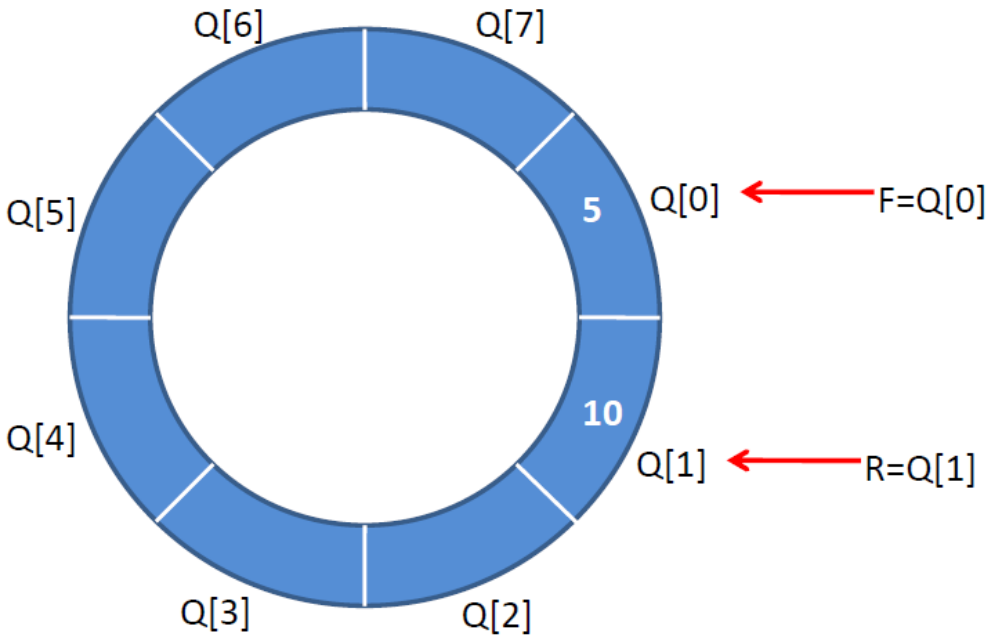
e.g.: (cond..)



If REAR=MAX-1 and FRONT!=0

Insert Operation of circular Queue

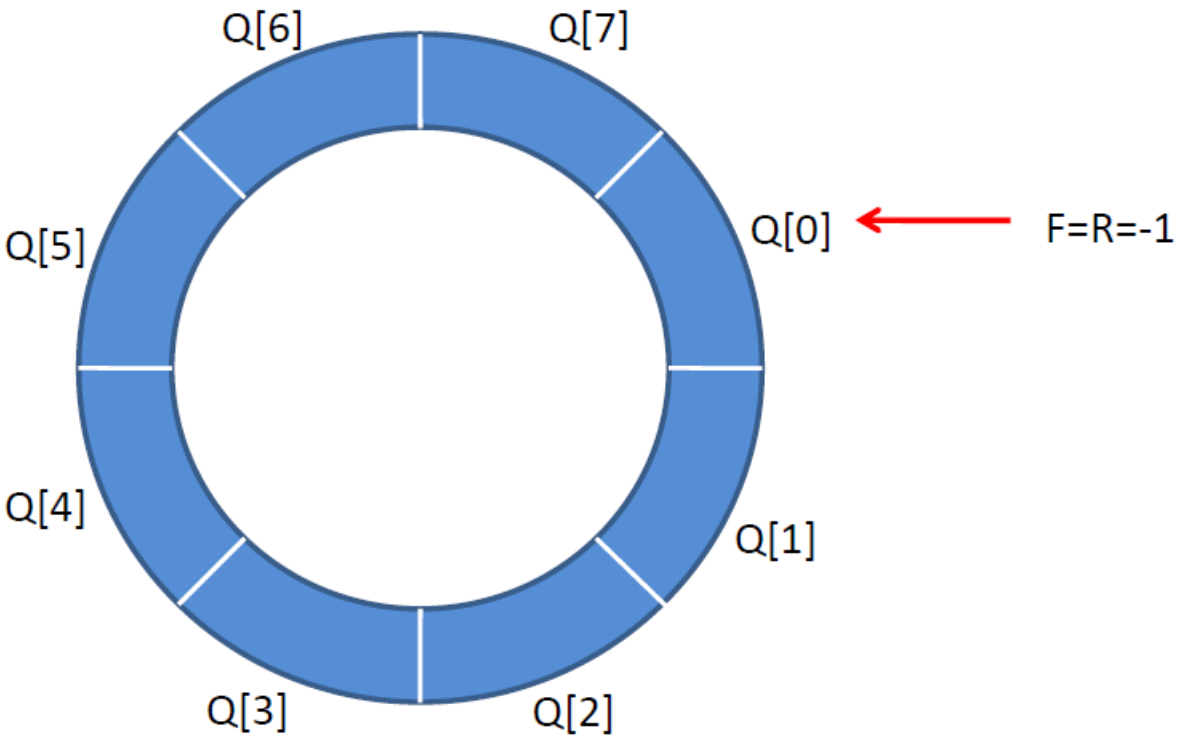
e.g.: (cond..)



Rear=Rear+1

Delete Operation in Circular queue

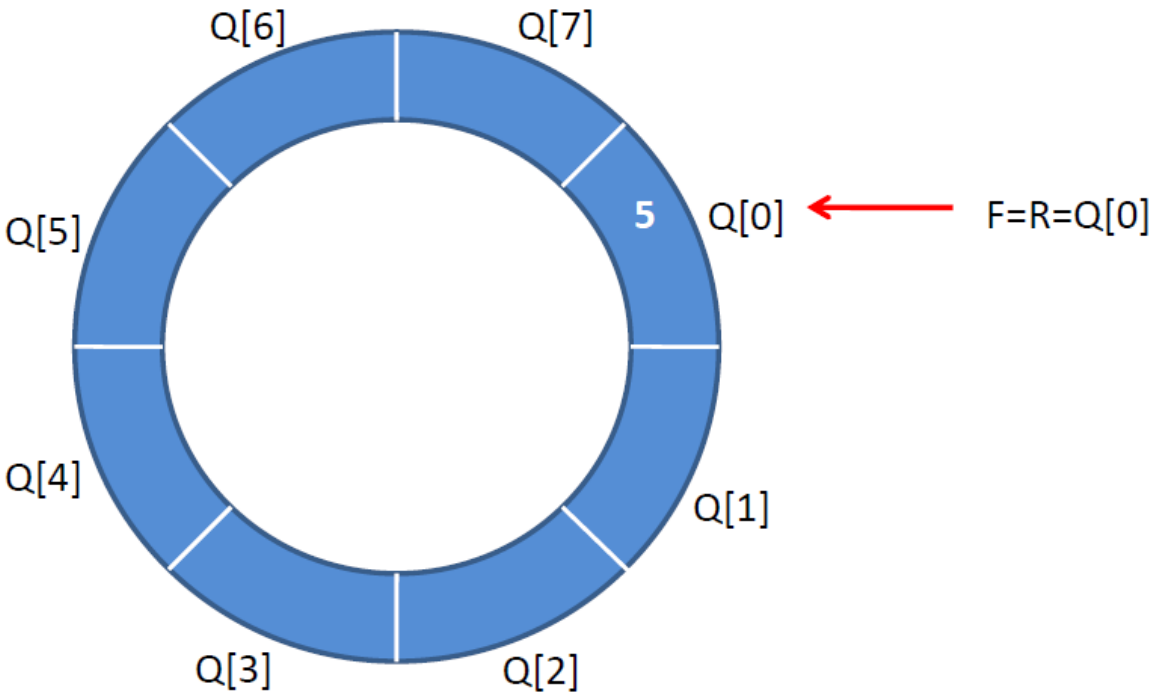
e.g.:



Queue is Empty(Underflow)

Delete Operation in Circular queue

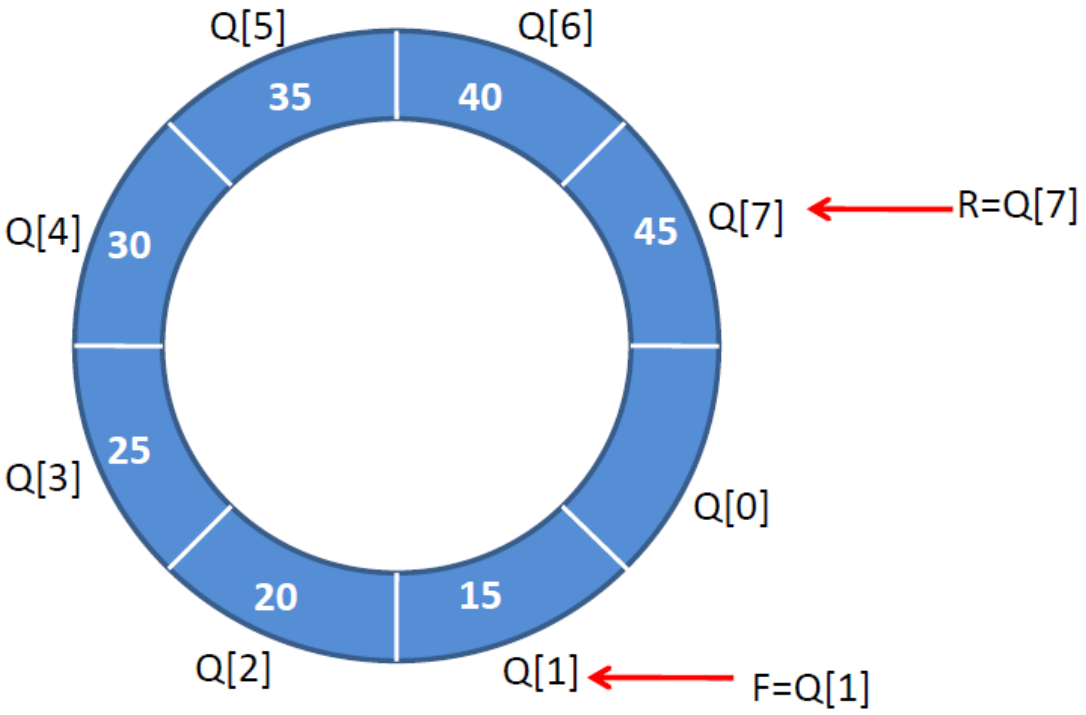
e.g.:



If F=R=0 then F=R=-1

Delete Operation in Circular queue

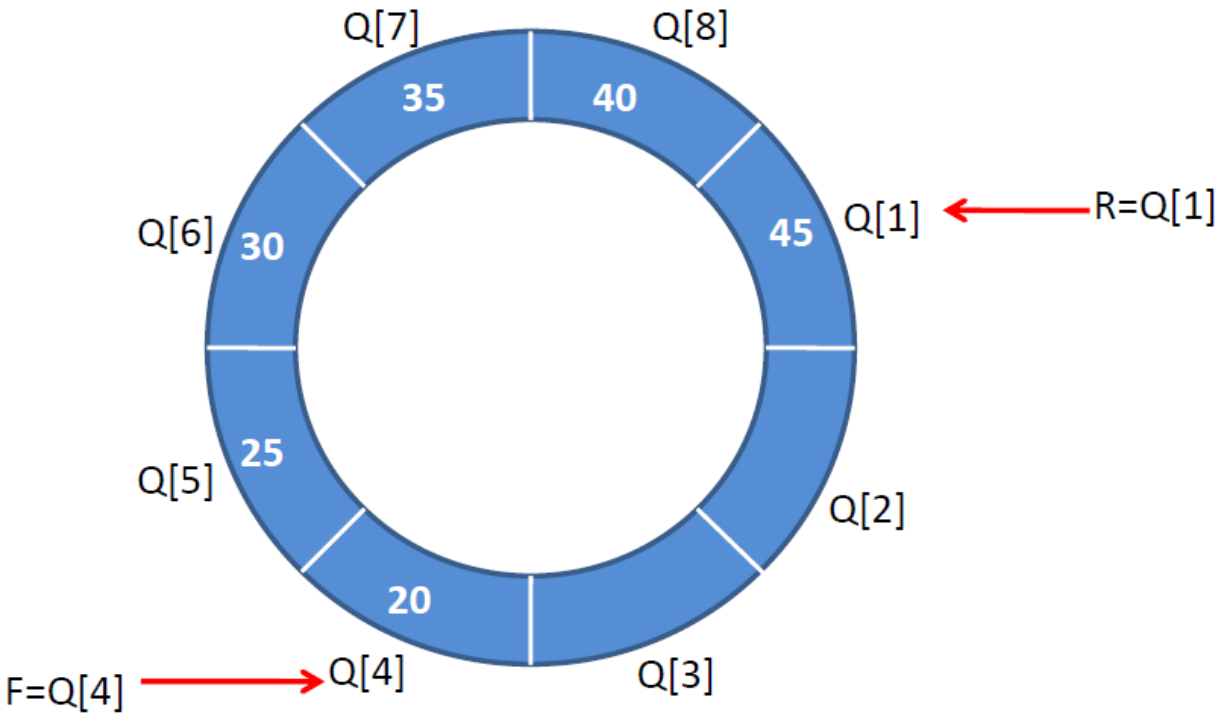
e.g.: (cond..)



If Front=MAX-1 then Front=0

Delete Operation in Circular queue

e.g.: (cond..)



Front=Front+1

Application of Queue

- Printer Spooling
- CPU Scheduling
- Mail Service
- Keyboard Buffering
- Elevator
- Simulation
- Asynchronous transmission

Application of Queue

PRINTER SPOOLING

- A printer may receive multiple print requests in a short span of time. The rate at which these requests are received is much faster than the rate at which they are processed.
- Therefore, a temporary storage mechanism is required to store these requests in the order of their arrival.
- A queue is the best choice in this case, which stores the print requests in such a manner so that they are processed on a first-come-first-served basis.

Application of Queue

CPU Scheduling

- A CPU can process one request at a time. The rate at which the CPU receives requests is usually much greater than the rate at which the CPU processes the requests.
- Therefore, the requests are temporarily stored in a queue in the order of their arrival.
- Whenever CPU becomes free, it obtains the requests from the queue.
- Once a request is processed, its reference is deleted from the queue. The CPU then obtains the next request in sequence and the process continues.

Application of Queue

Mail Service

- In various organizations, many transactions are conducted through mails.
- If the mail server goes down, and someone sends you a mail, the mail is bounced back to the sender.
- To avoid any such situation, many organizations implement a mail backup service. Whenever there is some problem with the mail server because of which the messages are not delivered, the mail is routed to the mail's backup server.
- The backup server stores the mails temporarily in a queue. Whenever the mail server is up, all the mails are transferred to the recipient in the order in which they arrived.

Application of Queue

Keyboard Buffering

- Queues are used for storing the keystrokes as you type through the keyboard.
- Sometimes the data, which you type through the keyboard is not immediately displayed on the screen.
- This is because during that time, the processor might be busy doing some other task.
- In this situation, the data is temporarily stored in a queue, till the processor reads it.
- Once the processor is free, all the keystrokes are read in the sequence of their arrival and displayed on the screen.

Application of Queue

Elevator

- An elevator makes use of a queue to store the requests placed by users.
- Suppose the elevator is currently on the first floor. A user on the ground floor presses the elevator button to request for the elevator. At almost the same time a user on the second floor also presses the elevator button.
- In that case, the elevator would go to the floor on which the button was pressed earlier, that is, the requests will be processed on a FCFS basis.

Simple Queue V/S Circular Queue

Simple Queue	Circular Queue
A simple queue is a linear data structure that stores data as a sequence of elements similar to a real-world queue	A circular queue is a linear data structure in which the last item connects back to the first item forming a circle.
The insertion and deletion of the elements is fixed in simple queue i.e, addition from the rear end and deletion from the front end.	The circular queue is capable of inserting and deleting the element from any point until it is unoccupied.
Simple queue wastes the memory space.	circular queue makes the efficient use of space.
A simple queue is less efficient then a circular queue.	A circular queue is more efficient than a simple queue.
Problem of overflow in a queue frequently.	It is overflow only when sufficient elements are there.

Stack V/S Queue

Stack	Queue
Stack is internally implemented in such a way that the element inserted at the last in stack would be the first element to come out of it. It follows **LIFO (Last in and First out).**	Queue is implemented in such manner that the element inserted at the first in queue would be the first element to come out of it. It follows **FIFO (First in and First out).**
Stack operations on element take place only from one end of the list called the top.	Queue operations on element i.e insertion takes place at the rear end and the deletion takes place from the front end.
In stacks we maintain only one pointer to access the list, called the top, which always points to the last element present in the list.	In queues we maintain two pointers to access the list. The front pointer always points to the first element inserted in the list and the rear pointer always points to the last inserted element.
In Stack operations termed as Push and Pop.	Queue operations termed as Enqueue and Dequeue.
Stack does not have any variant and thus do not implemented further.	Queue has variants like circular queue, priority queue, doubly ended queue.
Stack is more simpler than Queue.	Queue is more complex as compare to Stack.

Stack V/S Queue

Stack	Queue
Stack is used in infix to postfix conversion, scheduling algorithms, depth first search and evaluation of an expression and recursion.	Queue is used in solving problems having sequential processing, Mail services, CPU Scheduling, Elevators and simulation.
To check if a stack is empty, the following condition is used: ***TOP==-1.***	To check if a queue is empty, the following condition is used: ***FRONT==-1***.
To check if a stack is full, the following condition is used: ***TOP==MAX-1***.	To check if a queue is full, the following condition is used: ***REAR==MAX-1***.