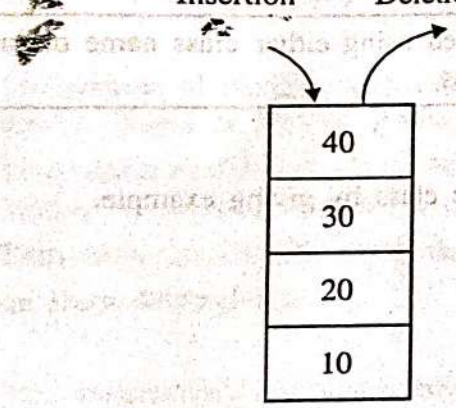
UNIT 3

STACK AND QUEUE

3.1 Overview of Stack :

- Stack is a linear Data Structure in which insertion and deletion operation are performed at same e
- Following figure represents Stack :



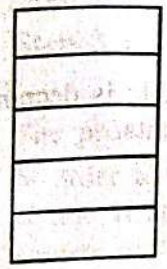
- Since insertion and deletion operations are performed at same end, the element which is inserte Last is first to delete. So Stack is also known as **Last In First Out (LIFO).**
- Consider a stack of plates at dinner counter. The person who comes for dinner takes off the pla which is at top of the stack. After washing the plates the waiter places the washed plates on t top of the stack. So the plate that is placed last is first take by person.

3.2 Implementation of Stack using list :

- A Stack can be implemented using list data structure in python. Python does not restrict us to specify size of the list But, if we wish to specify size of the list than we can use separate variable named **maxsize** which holds size of the list.
- A variable **top** is always used with 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.
- Suppose we define a class named Stack which contains three members(element, maxsize and top) as shown below :

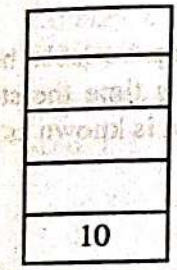
```
class Stack:
    def __init__(self):
        self.element = []
        self.maxsize = 5
        self.top=-1
```

- Initially Stack is empty so value of top is –1 as shown below :



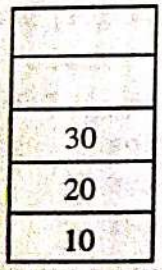
top = –1

- In order to insert an element into stack we have to increment the value of top variable by one. Consider Following figure which represents a stack with one element.



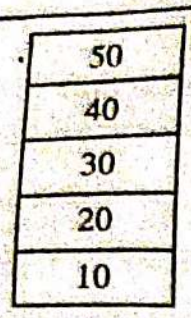
top=0

- Each time an element is inserted into stack the value of top variable is incremented by one. Consider following figure which represents a stack with three elements.

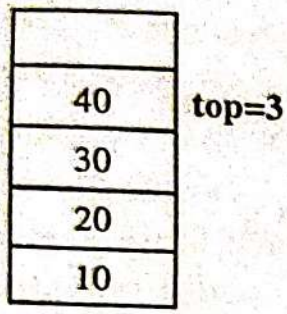


top=2

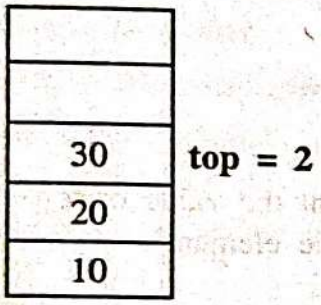
- After inserting elements one by one into stack, when the value of top variable becomes equal to **maxsize–1** we cannot insert new element into stack. Because at this time the stack is full. Consider following figure which represents a stack is full. This condition is known as **"Overflow" condition in stack.**



❖ In order to delete an element from stack we have to decrement the value of **top** variable by one. Consider Following figure which represents a stack after deleting topmost element (50) from the stack.



❖ Each time an element is deleted from the stack the value of top variable is decremented by one. Consider following figure which represents a stack with three elements.



❖ After deleting elements one by one from stack, when the value of top variable becomes equal to –1 then we cannot delete an element from the stack. Because at this time the stack is empty. Consider following figure which represents an empty stack. This condition is known as **"Underflow"** condition in stack.

IMP

3.3 Operations on Stack – Push, Pop :

❖ There are two operations that can be performed on stack.

(1) PUSH :

- The process of inserting an element into stack is known as **PUSH** operation.
- In order to insert an element into stack first we have to check weather free space is available in the stack or not.

- If stack is full then we cannot insert an element into stack. This condition is known as **"Overflow"**. Here we will implement Stack using list and python does not restrict us to specify size of the list. So Overflow condition will never encounter while implementing Stack using List. But, we can specify maximum size of the list using separate variable named **maxsize**. In this case we use variable called **top** which holds index of the topmost element of the stack.
- If value of top variable is equal to maxsize–1 than stack is full. This condition is known as "Overflow".
- If stack is not overflow then we can insert an element into stack. First we have to increment the value of variable top by one and then insert an element into stack.
- **Algorithm for PUSH operation**

Step 1 : if self.top>=self.maxsize–1 then
write "Stack Is Overflow"
return

Step 2 : self.top ← self.top + 1

Step 3 : self.element.append(Value)

Imp

(2) POP

- The process of deleting an element from stack is known as **POP** operation.
- In order to delete an element from stack first we have to check weather stack is empty or not. If stack is empty then we cannot delete an element from stack. This condition is known as **"Underflow"**.
- If value of top variable is –1 then stack is empty. So we cannot delete an element from stack.
- If stack is not underflow then we can delete topmost element from stack. After deleting topmost element from stack, we have to decrement the value of TOP by one, so that it can point to the next topmost element in the stack.
- **Algorithm for POP operation**

Step 1 : if self.top=–1 then
write "Stack Is Underflow"
return

Step 2 : write (self.element.pop())

Step 3 : self.top ← self.top – 1

❖ Following program performs various operations on Stack using list data structure :

```
class Stack:
    def __init__(self):
        self.element = []
        self.maxsize = 10
        self.top=-1
    def Push(self,Value):
        if(self.top>=self.maxsize-1):
            print("Stack Is Overflow")
            return
```

```
            self.top=self.top+1
            self.element.append(Value)
        def size(self):
            return len(self.element)
        def topofstack(self):
            if(self.top==-1):
                print('Stack is Underflow')
            else:
                print("TOP Element
Is:{}".format(self.element[self.top]))
        def Pop(self):
            if(self.top==-1):
                print('Stack Is Underflow')
            else:
                print("Deleted Element
Is:{}".format(self.element.pop()))
                self.top=self.top-1
        def display(self):
            if(self.top==-1):
                print("Stack Is Underflow")
            else:
                print("Elements in the stack are:")
                for i in range(self.top,-1,-1):
                    print(self.element[
s = Stack()
while(True):
    print("Select Your Option")
    print("(1) PUSH Element")
    print("(2) POP Element")
    print("(3) Display Element of Stack")
    print("(4) Display TOPMOST Element of Stack")
    print("(5) Size of Stack")
    print("(6) EXIT")
    choice = input("Enter Your Choice:")
    if(choice.isnumeric() is False):
        print("Wrong Choice Entered")
    elif(int(choice)==1):
        Value = input("Enter Element:")
        s.Push(Value)
    elif(int(choice)==2):
        s.Pop()
    elif(int(choice)==3):
        s.display()
```

```
elif(int(choice)==4):
        s.topofstack()
elif(int(choice)==5):
        print("Size of Stack Is:{}".format(s.size()))
elif(int(choice)==6):
        break
else:
        print("Wrong Choice Entered")
```

3.4 Application of Stack :

❖ Stack is widely used in many application of computer. Following are some of the major applications of stack :

- Stack is widely used in the concept of recursion because of its Last In First Out characteristics.
- Programming Languages uses stack at the time of calling the function to store value of formal parameters and return address. At each function call the stack is pushed to save the necessary values, such as formal parameters, local variables and return address of the function call. After completion of the execution of the function the stack values are popped to restore the saved values of the function call.
- Stack is widely used in the evaluation of polish notation. Stack is used to convert polish notation from one form to another form. For Example stack is used to convert infix expression into post fix notation.
- Stack is also used by compiler for parsing the syntax of expression.
- Stack is also used to reverse the string. In which first all the characters of the string are pushed into stack and then they are popped from stack.

3.4.1 Infix, Prefix and Post fix Forms of Expressions :

❖ There are basically three types of polish notation :

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

(B) **Prefix :** When the operator is written before their operands then it is known as Prefix notation. For example +AB

(C) **Post fix :** When the operator is written after their operands then it is known as Post fix notation. For example AB+

❖ **Conversion of Infix Expression into Post fix expression**

❖ Before understanding the conversion process of Infix expression into Post fix expression we must know the precedence of each operator.

❖ Following are the precedence of each operator from highest to lowest.

Operator	Priority
(,),[,]	Highest
^	↓
*,/	↓
+,-	Lowest

❖ Generally we use infix notation in arithmetic expression. In order to convert an expression infix notation to postfix notation follows the steps given below :

(1) Initialize an empty stack.

(2) Scan infix string from left to right.

(3) If scanned character is an **operand** then append it into postfix string.

(4) If scanned character is left parenthesis "(" then PUSH it into stack.

(5) If scanned character is an **operator** and stack is either empty or contains "(" at top element then PUSH it into stack.

(6) If scanned character is right parenthesis ")" then POP all the operators from stack until "(" is encountered in the stack. Do not append parenthesis into postfix string.

(7) If scanned character is an **operator** and top of the stack also contains an operator then we have to compare the precedence of operators as follow :

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

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

(8) Repeat step7 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.

❖ **Consider Following Example :**

Infix String: (a + b) * (c + d)

Scanned Character	Content of Stack	Postfix String
	Empty	Empty
(	(	Empty
A	(	a
+	(+	a
B	(+	ab
)	Empty	ab+
*	*	ab+
(	*(	ab+
C	*(	ab+c
+	*(+	ab+c
D	*(+	ab+cd
)	*	ab+cd+
		ab+cd+*

❖ **Exercise :**

Convert Following Infix Expression into Postfix Expression

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

= A * (BC+) * D

= ABC+* * D

= ABC+*D*

(2) X * (C + D) + (J + K) * N + M * P

= X * CD+ + JK+ * N + M * P

= XCD+* + JK+ * N + M * P

= XCD+* + JK+N* + M * P

= XCD+* + JK+N* + MP*

= XCD+*JK+N*+ + MP*

= XCD+*JK+N*+ MP*+

(3) A/B^C+D*E–A*C

= A/BC^+D*E–A*C

= ABC^/+D*E–A*C

= ABC^/+DE*–AC*

= ABC^/DE*+ – AC*

= ABC^/DE*+AC*–

3.4.2 Evaluations of post fix expression :

❖ In order to evaluate Post fix Expression follows the step given below :

(1) Initialize an empty stack.

(2) Scan post fix string from left to right.

(3) If scanned character is an operand then PUSH it into the stack.

(4) If scanned character is an operator then POP topmost element from stack and store its value in variable temp2. Now again POP topmost element from stack and storeits value in variable temp1.

(5) Evaluate an expression temp1 **Operator** temp2. PUSH result of an expression into stack.

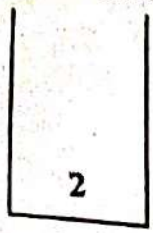
(6) Repeat above procedure until all the characters from postfix string are scanned.

❖ **Consider Following Example :**

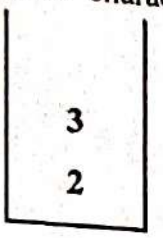
Evaluate Postfix String 234*+5–

(1) Initially Stack is Empty

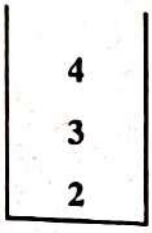
(2) Scanned character is 2 which is operand so push it into stack.



(3) Scanned character is 3 which is operand so push it into stack.



(4) Scanned character is 4 which is operand so push it into stack.

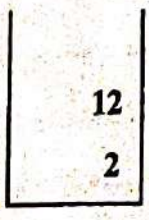


(5) Scanned character is * which is an operator so pop two elements from stack. Perform operation on them and push result back to stack.

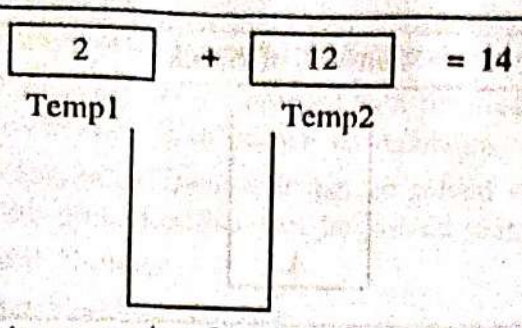
Temp1 Temp2

Evaluate expression 3 * 4 = 12. PUSH result back into stack.

So Now content of stack is



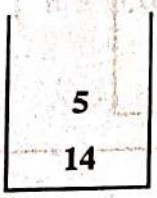
(6) Scanned character is + which is operator so pop two elements from stack. Perform operation on them and push result back to stack.



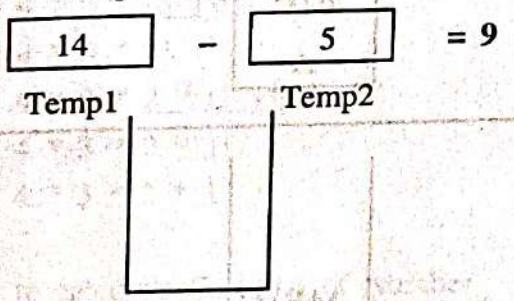
Evaluate expression 2 + 12 = 14. PUSH result back into stack.

So now content of stack is

(7) Scanned character is 5 which is operand so push it into stack.



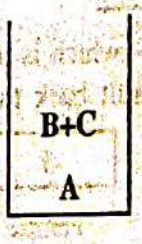
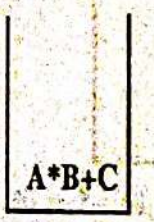
(8) Scanned character is – which is operator so pop two elements from stack. Perform operation on them and push result back to stack.



Evaluate expression 14 –5 = 09. PUSH result back into stack. So now content of stack is

Exercise :

(1) Convert ABC+*D* into infix notation

Scanned Character	Content of Stack
A	
B	
C	
+	
*	
D	
*	

3.4.3 Recursive Functions (factorial, Fibonacci series) :

- 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

(1) Factorial Number :

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

$$n! = n * (n-1)!$$

$$FACT(n) = \begin{cases} 1, \text{ if } n = 1 \\ n * FACT(n-1), \text{ otherwise} \end{cases}$$

- **For Example :**

$5! = 5 * (5-1)!$
$= 5 * 4!$
$= 5 * \underline{4 * (4-1)!}$
$= 5 * 4 * 3!$
$= 5 * 4 * \underline{3 * (3-1)!}$
$= 5 * 4 * 3 * 2!$
$= 5 * 4 * 3 * \underline{2 * (2-1)!}$
$= 5 * 4 * 3 * 2 * 1!$
$= 5 * 4 * 3 * 2 * 1$
$= 120$

- Algorithm to calculate Factorial of Number

Step 1 : if Number=1 then
return 1
else
return (Number * Factorial(Number–1))

- Program to find factorial of given number

```
def Factorial(Number):
    if(Number==1):
        return 1
    else:
        return (Number * Factorial(Number-1))
Number = int(input("Enter Number:"))
print("Factorial of {} is
{}".format(Number,Factorial(Number)))
```

(2) Greatest Common Divisor :

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

$$GCD(m, n) = \begin{cases} n, \text{ if } n \text{ divides } m \\ GCD(n, m \bmod n), \text{ otherwise} \end{cases}$$

- **For Example :**

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

Thus, GCD (62, 8) = 2

- Algorithm to calculate Greatest Common Divisor

Step 1 : R ← N1 Mod N2

Step 2 : if R = 0 then
Return N2
else
return (GCD(N2 , R))

- Program to calculate Greatest Common Divisor

```
def GCD(N1,N2):
    R=N1%N2
    if(R==0):
        return N2
    else:
        return(GCD(N2,R))
N1 = int(input("Enter Number1:"))
N2 = int(input("Enter Number2:"))
print("GCD({},{})={}".format(N1,N2,GCD(N1,N2)))
```

(3) 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 calc as a sum of first and second terms. Similarly fourth term can be calculated as a sum of s and third term and so on.
- We can find n^{th} term of Fibonacci series using following recursive function :

$$FIBO(n) = \begin{cases} 0, \text{ if } n = 0 \text{ or } 1 \\ 1, \text{ if } n = 2 \\ FIBO(n-1) + FIBO(n-2), \text{ if } n > 2 \end{cases}$$

❖ **For 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

❖ Algorithm to calculate Nth term of Fibonacci Series

Step 1 : If Number = 0 or Number = 1 then
return 0
else if Number = 2 then
return 1
else
retur (FIBO(N–1)+ FIBO(N–2))

❖ Program to calculate Nth term of Fibonacci Series

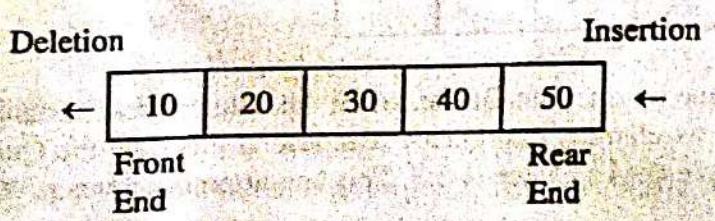
```
def FIBO(Number):
    if(Number==0 or Number==1):
        return 0
    elif(Number==2):
        return 1
    else:
        return(FIBO(Number-1)+FIBO(Number-2))
Number = int(input("Enter Number:"))
print("FIBO={}".format(FIBO(Number)))
```

3.5 Overview of Queue :

Imp

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

❖ Following figure represents Queue :



- Since insertion operation is performed at rear end and deletion operation is performed end the element which is inserted first is first to delete. So Queue is also known as First out (FIFO).
- Consider a Queue of Students at Fee Counter waiting to pay fee. The students pay fees in same order as they are standing in the queue. The students who stand first in the Queue pay fees first and come out from the queue.

3.6 Implementation of Queue using List :

- A Queue can be implemented using list data structure in python. Python does not require specify size of the list But, if we wish to specify size of the list than we can use separate named maxsize which holds size of the list.
- Variables front and rear are always used with Queue. They are used to keep track of first element and last element of the Queue. front variable holds index position of the first element of Queue and rear variable holds index position of the last element of Queue.
- Suppose we define a class named Queue which contains four members(element, maxsize and rear) as shown below :

```
class Queue:
    def __init__(self):
        self.element = []
        self.maxsize = 5
        self.front=-1
        self.rear=-1
```

- It can be represented in memory as shown below :

front=-1

rear=-1

- In above figure initially Queue is empty so front and rear is initialized with –1.
- In order to insert an element into Queue we have to increment the value of rear variable by If Queue is empty and we are inserting first element in the Queue then we have to set value front variable to 0. Consider Following figure which represents a Queue with one element

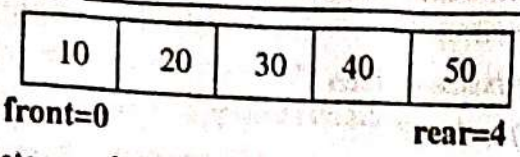
front=0

rear=0

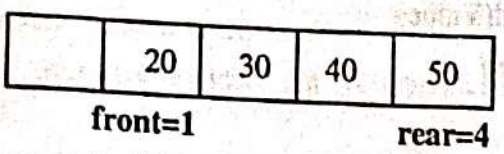
- Each time an element is inserted into Queue the value of rear variable is incremented by Consider following figure which represents a Queue with three elements.

front=0 rear=2

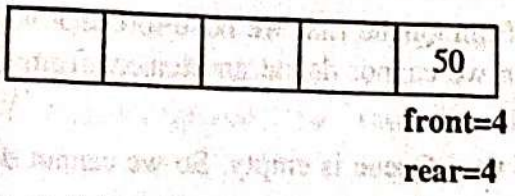
- After inserting elements one by one into Queue, when the value of rear variable equal to max 1 then we cannot insert new element into Queue. Because at this time the Queue is full. Consider following figure which represents a Queue is full. This condition is known as "Overflow" condition in Queue.



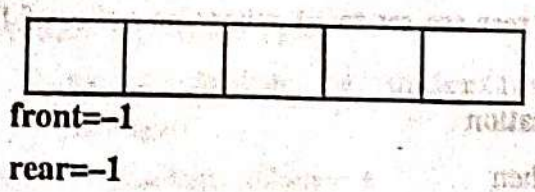
- In order to delete an element from Queue we have to increment the value of front variable by one. Consider Following figure which represents a Queue after deleting first element (10) from the Queue.



- Each time an element is deleted from Queue the value of front variable is incremented by one. Consider following figure which represents a Queue with one element.



- Suppose a queue contains only one element. After deletion of that element from queue we have to set the value of front and rear variable to –1. Because after the deletion of that element queue becomes empty.



- After deleting elements one by one from Queue, when the value of front variable becomes equal to –1 then we cannot delete an element from Queue. Because at this time the Queue is empty. Consider above figure which represents an empty Queue. This condition is known as **"Underflow"** condition in Queue.

3.7 Operations on Queue – Enqueue and Dequeue :

- There are two operations that can be performed on Queue.

(1) Enqueue :

- The process of inserting an element into Queue is known as **Enqueue** operation.
- In order to insert an element into Queue first we have to check weather free space is available in the Queue or not.
- If Queue is full then we cannot insert an element into Queue. This condition is known as **"Overflow".** Here we will implement Queue using list and python does not restrict us to specify size of the list. So Overflow condition will never encounter while implementing Queue using List. But, we can specify maximum size of the list using separate variable named **maxsize.** In this case we use variable named **front** which holds index of the first element of the Queue and **rear** which holds index of the last element of the Queue.
- If value of **rear** variable is equal to maxsize–1 than Queue is full. 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 variable rear by one and then insert an element into Queue. If value of front is –1 then we have to increment its value by one.

- **Algorithm for Enqueue operation**

Step 1 : if self.rear>=self.maxsize–1 then
write "Queue Is Overflow"
return

Step 2 : self.rear ← self.rear + 1

Step 3 : self.element.append(Value)

Step 4 : if self.front=–1 then
self.front=0

(2) Dequeue :

- The process of deleting an element from Queue is known as **Dequeue** operation.
- In order to delete an element from Queue first we have to check weather Queue is emp or not. If Queue is empty then we cannot delete an element from Queue. This condit is known as **"Underflow"**.
- If value of front is equal to –1 than Queue is empty. So we cannot delete an element fr Queue.
- If Queue is not underflow then we can delete Front most element from the Queue. Af deleting element we have to set front and rear variable. If there is only one element the queue then both front and rear are set to –1 otherwise value of front is incremen by one.
- **Algorithm for Dequeue operation**

Step 1 : if self.front = –1 then
write "Queue Is Underflow"
return

Step 2 : write self.element[self.front]

Step 3 : if self.front=self.rear then
self.front ← –1
self.rear ← –1
else
self.front ←self.front + 1

❖ **Following program performs various operations on Queue using list data structure :**

```
class Queue:
    def __init__(self):
        self.element = []
        self.maxsize = 10
        self.front=-1
        self.rear=-1
    def Enqueue(self,Value):
        if(self.rear>=self.maxsize-1):
            print("Queue Is Overflow")
```

```
                return
            self.rear=self.rear+1
            self.element.append(Value)
            if(self.front==-1):
                self.front=0
        def FirstofQueue(self):
            if(self.front==-1):
                print('Queue is Underflow')
            else:
                print("First Element
Is:{}".format(self.element[self.front]))
        def LastofQueue(self):
            if(self.front==-1):
                print('Queue is Underflow')
            el
                print("Last Element
Is:{}".format(self.element[self.rear]))
        def Dequeue(self):
            if(self.front==-1):
                print('Queue Is Underflow')
                return
            print("Deleted Element
Is:{}".format(self.element[self.front]))
            if(self.front==self.rear):
                self.front=-1
                self.rear=-1
            else:
                self.front=self.front+1
        def display(self):
            if(self.front==-1):
                print("Queue Is Underflow")
            else:
                print("Elements in the Queue are:")
                for i in range(self.front,self.rear+1,1):
                    print(self.element[i],end=" ")
                print()
q = Queue()
while(True):
    print("Select Your Option")
    print("(1) Insert Element")
    print("(2) Delete Element")
```

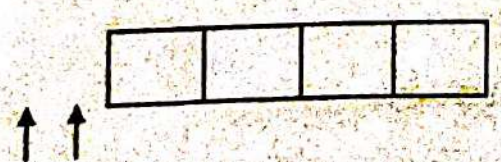
```
print("(3)  Display Element of Queue")
print("(4)  Display First Element of Queue")
print("(5)  Display Last Element of Queue")
print("(6)  EXIT")
choice = input("Enter Your Choice:")
if(choice.isnumeric() is False):
      print("Wrong Choice Entered")
elif(int(choice)==1):
      Value = input("Enter Element:")
      q.Enqueue(Value)
elif(int(choice)==2):
      q.Dequeue()
elif(int(choice)==3):
      q.display
elif(int(choice)==4):
      q.FirstofQueue()
elif(int(choice)==5):
      q.LastofQueue()
elif(int(choice)==6):
      break
else:
    print("Wrong Choice Entered")
```

3.8 Limitation of Single Queue :

- Limitation of simple queue is that even if we have a free memory space, we cannot use that memory space to insert an element. Thus there is a wastage of memory space in simple queue. This is the limitation of simple queue.
- Consider Following Example :

Here, SIZE of Simple Queue is 4.

(1) Initially Queue is Empty.



FRONT=-1

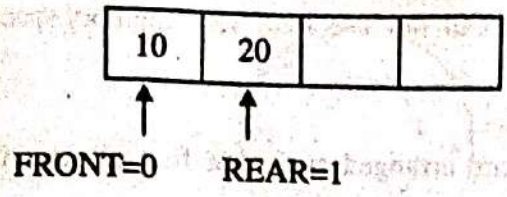
REAR=-1

(2) Insert an element 10.

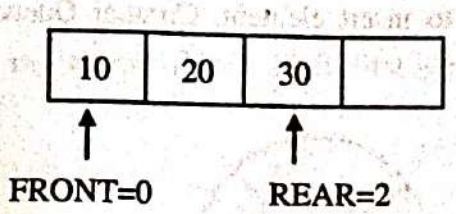
FRONT=0

REAR=0

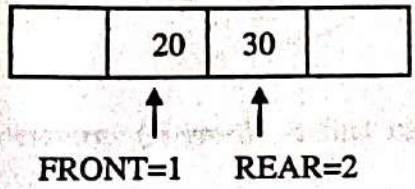
(3) Insert an element 20.



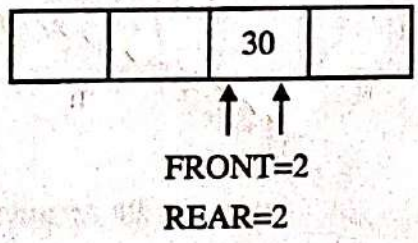
(4) Insert an element 30.



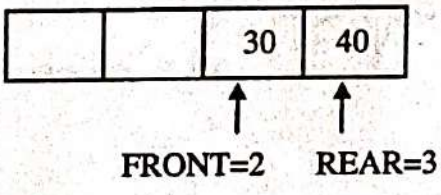
(5) Delete an element (Here, 10 is first element so it is deleted).



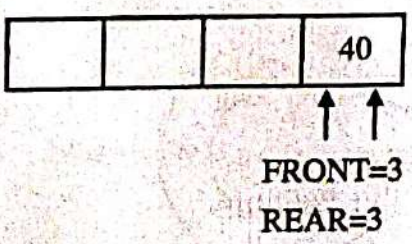
(6) Delete an element (Here, 20 is first element so it is deleted).



(7) Insert an element 40.



(8) Delete an element (Here, 30 is first element so it is deleted).

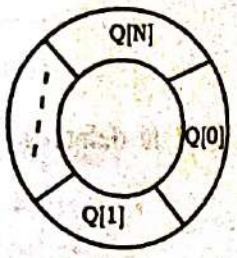


(9) Insert an element 50. Since REAR ≥ SIZE -1 an overflow occurs. We can see that the SIZE of simple queue is 4 and there is only one element in the queue. There is a space for three elements in the queue. Even then we can not use that free memory space to insert an element 50.

Imp

3.9 Concepts of Circular Queue :

- Circular Queue is a Queue in which elements are arranged such that first element in the queue follows the last element.
- The limitation of simple queue is that even if there is a free memory space available in the simple queue we cannot use that free memory space to insert element. Circular Queue is designed to overcome the limitation of Simple Queue.
- Following figure represents Circular Queue :

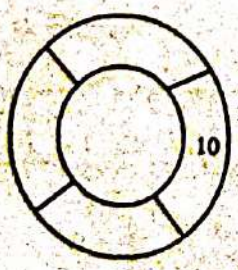


- Consider Following Example :

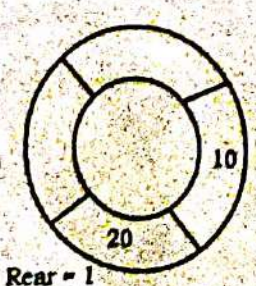
Here, SIZE of Circular Queue is 4.

(1) Initially Queue is Empty.

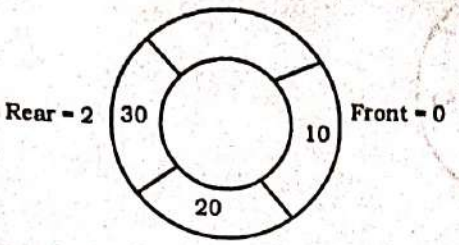
(2) Insert an element 10.



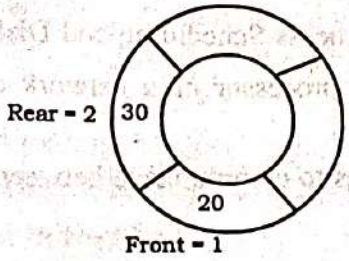
(3) Insert an element 20.



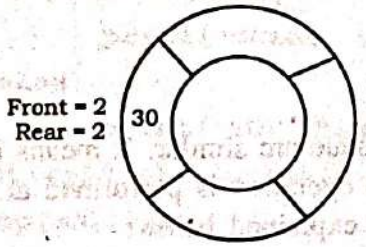
(4) Insert an element 30.



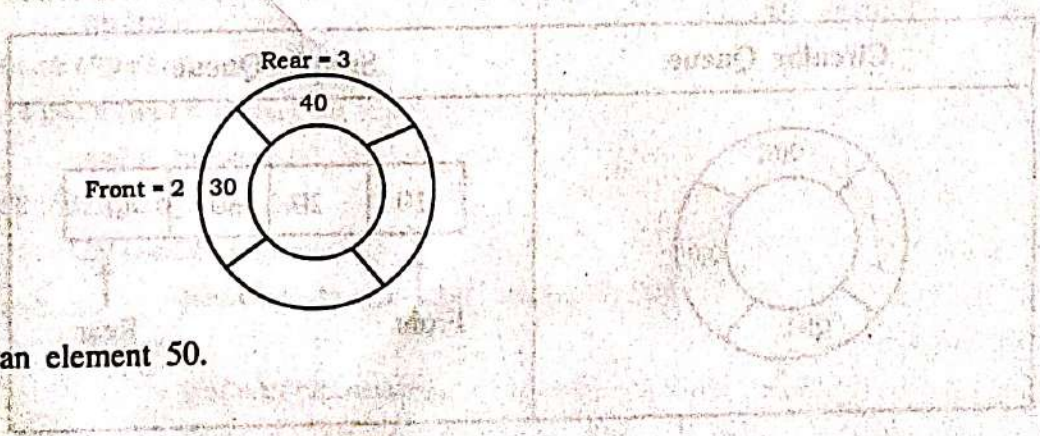
(5) Delete an element (Here, 10 is first element so it is deleted).



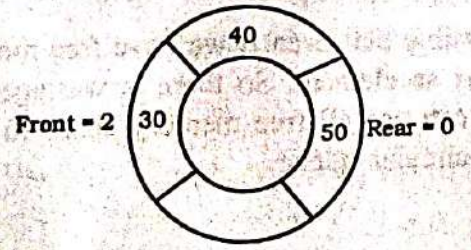
(6) Delete an element (Here, 20 is first element so it is deleted).



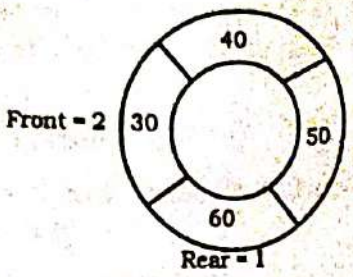
(7) Insert an element 40.



(8) Insert an element 50.



(9) Insert an element 60.



(10) Insert an element 70. Here, overflow occurs. We can see that the SIZE of circular queue i 4 and there is no space in the circular queue to insert an element 70.

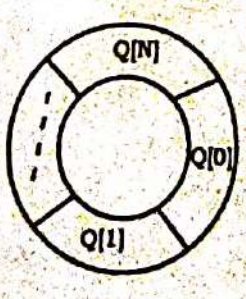
3.10 Application of queue :

- Queue is widely used by Operating System in Process Scheduling and Disk Scheduling algorithm
- Queue is used by Computer hardware such as processor or a network card to serve incomin resource requests in **First in First Out** manner.
- A mailbox or port which is used to save messages to communicate between two users or processe in a system uses queue.
- Queue is widely used in Simulation.
- Queue is used during asynchronous transmission of data where data are not necessarily receive at same rate as sent between two processes.
- Queue is also used in Printer Spooling.

3.11 Differentiate circular queue and simple queue :

- Conceptually both simple queue and circular queue are similar. It means in both queue insertion operation is performed at one end and deletion operation is performed at another end. But ther are some differences between them which are explained below:

(1) In circular queue elements are arranged such that first element in the queue follows the las element in the queue. While in simple queue first element does not follows last element.

Circular Queue	Simple Queue
	

(2) In Simple queue sometimes it is possible that even if we have free memory space we cannot use that free memory space to insert an element. So there is wastage of memory in simple queue. While in circular queue we can use all free memory space to insert an element. So there is no wastage of memory in circular queue.