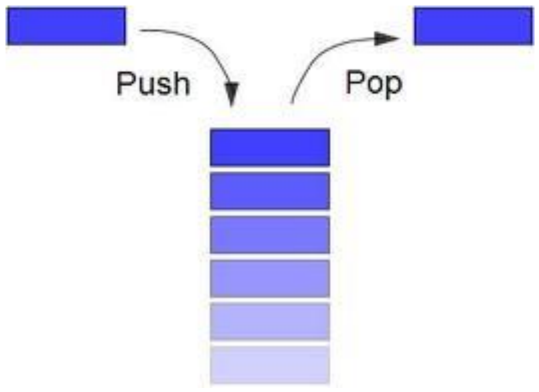
UNIT – II LINEAR DATA STRUCTURES

Stacks Primitive Operations:

A stack is a container of objects that are inserted and removed according to the last-in first-out (LIFO) principle. In the pushdown stacks only two operations are allowed: push the item into the stack, and pop the item out of the stack. A stack is a limited access data structure - elements can be added and removed from the stack only at the top. Push adds an item to the top of the stack, pop removes the item from the top. A helpful analogy is to think of a stack of books; you can remove only the top book, also you can add a new book on the top.

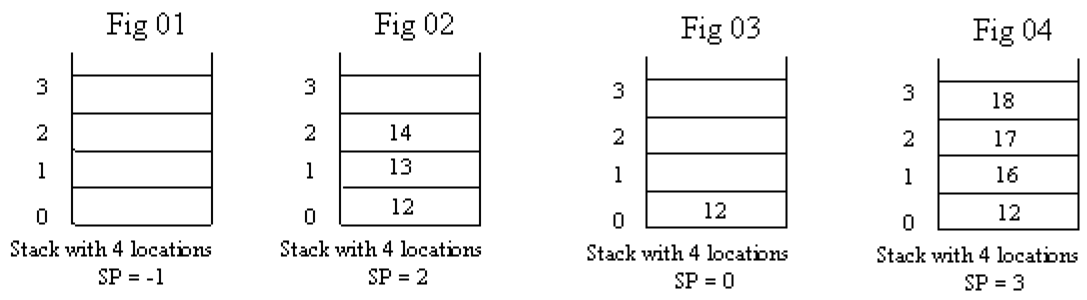
A stack may be implemented to have a bounded capacity. If the stack is full and does not contain enough space to accept an entity to be pushed, the stack is then considered to be in an overflow state. The pop operation removes an item from the top of the stack. A pop either reveals previously concealed items or results in an empty stack, but, if the stack is empty, it goes into underflow state, which means no items are present in stack to be removed.



Stack (ADT) Data Structure:

Stack is an Abstract data structure (ADT) works on the principle Last In First Out (LIFO). The last element add to the stack is the first element to be delete. Insertion and deletion can be takes place at one end called TOP. It looks like one side closed tube.

- The add operation of the stack is called push operation
- The delete operation is called as pop operation.
- Push operation on a full stack causes stack overflow.
- Pop operation on an empty stack causes stack underflow.
- SP is a pointer, which is used to access the top element of the stack.
- If you push elements that are added at the top of the stack;
- In the same way when we pop the elements, the element at the top of the stack is deleted.



Operations of stack:

There are two operations applied on stack they are

1. push
2. pop.

While performing push & pop operations the following test must be conducted on the stack.

1) Stack is empty or not

2) Stack is full or not

Push:

Push operation is used to add new elements in to the stack. At the time of addition first check the stack is full or not. If the stack is full it generates an error message "stack overflow".

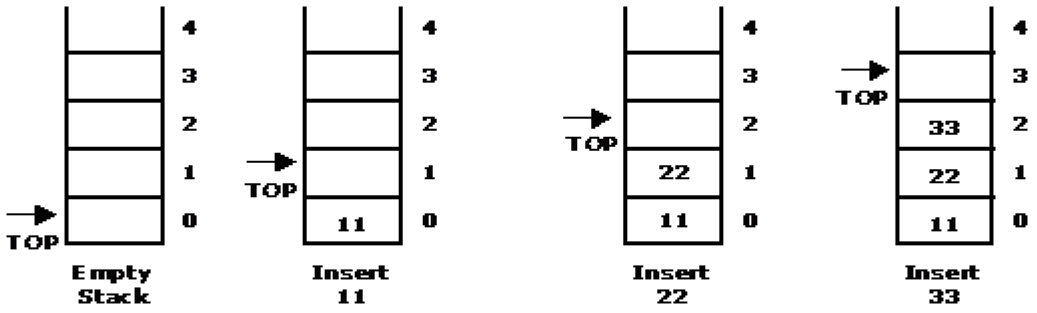
Pop:

Pop operation is used to delete elements from the stack. At the time of deletion first check the stack is empty or not. If the stack is empty it generates an error message "stack underflow".

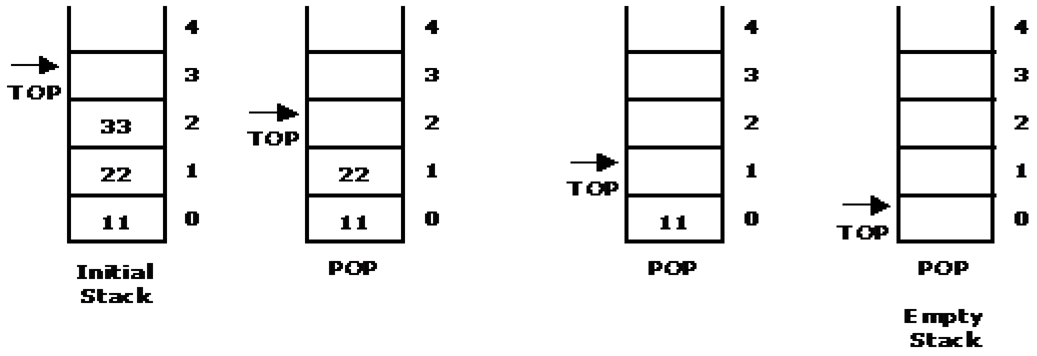
Representation of a Stack using Arrays:

Let us consider a stack with 6 elements capacity. This is called as the size of the stack. The number of elements to be added should not exceed the maximum size of the stack. If we attempt to add new element beyond the maximum size, we will encounter a stack overflow condition. Similarly, you cannot remove elements beyond the base of the stack. If such is the case, we will reach a stack underflow condition.

When a element is added to a stack, the operation is performed by push().



When an element is taken off from the stack, the operation is performed by pop().



Source code for stack operations, using array:

STACK: Stack is a linear data structure which works under the principle of last in first out. Basic operations: push, pop, display.

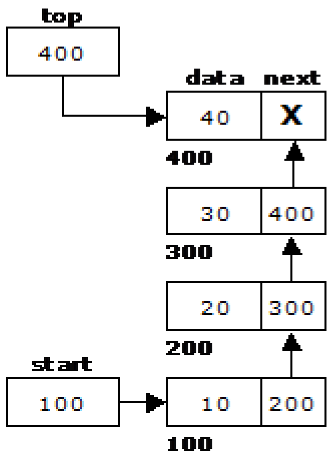
1. **PUSH**: if (top==MAX), display **Stack overflow** else reading the data and making stack [top] =data and incrementing the top value by doing top++.
2. **POP:** if (top==0), display **Stack underflow** else printing the element at the top of the stack and decrementing the top value by doing the top.
3. **DISPLAY**: IF (TOP==0), display **Stack is empty** else printing the elements in the stack from stack [0] to stack [top].

```
# Python program for implementation of stack

# import maxsize from sys module
# Used to return -infinite when stack is empty from sys import maxsize

# Function to create a stack. It initializes size of stack as 0
def createStack():
   stack = []
   return stack

# Stack is empty when stack size is 0
def isEmpty(stack):
   return len(stack) == 0

# Function to add an item to stack. It increases size by 1
def push(stack, item):
   stack.append(item)
   print("pushed to stack " + item)

# Function to remove an item from stack. It decreases size by 1
def pop(stack):
   if (isEmpty(stack)):
      print("stack empty")
```

```
        return str(-maxsize -1) #return minus infinite

    return stack.pop()

# Driver program to test above functions
stack = createStack()
print("maximum size of array is",maxsize)
push(stack, str(10))
push(stack, str(20))
push(stack, str(30))
print(pop(stack) + " popped from stack")
print(pop(stack) + " popped from stack")
print(pop(stack) + " popped from stack")
print(pop(stack) + " popped from stack")
push(stack, str(10))
push(stack, str(20))
push(stack, str(30))
print(pop(stack) + " popped from stack")
```

Linked List Implementation of Stack:

We can represent a stack as a linked list. In a stack push and pop operations are performed at one end called top. We can perform similar operations at one end of list using top pointer.



Source code for stack operations, using linked list:

```
# Python program for linked list implementation of stack

# Class to represent a node
class StackNode:

    # Constructor to initialize a node
    def __init__(self, data):
        self.data = data
        self.next = None

class Stack:

    # Constructor to initialize the root of linked list
    def __init__(self):
        self.root = None

    def isEmpty(self):
        return True if self.root is None else  False

    def push(self, data):
        newNode = StackNode(data)
        newNode.next = self.root
        self.root = newNode
        print ("%d pushed to stack" %(data))

    def pop(self):
        if (self.isEmpty()):
            return float("-inf")
        temp = self.root
        self.root = self.root.next
        popped = temp.data
        return popped

    def peek(self):
        if self.isEmpty():
            return float("-inf")
        return self.root.data

# Driver program to test above class
stack = Stack()
stack.push(10)
stack.push(20)
stack.push(30)

print ("%d popped from stack" %(stack.pop()))
print ("Top element is %d " %(stack.peek()))
```

Stack Applications:

1. Stack is used by compilers to check for balancing of parentheses, brackets and braces.
2. Stack is used to evaluate a postfix expression.
3. Stack is used to convert an infix expression into postfix/prefix form.

4. In recursion, all intermediate arguments and return values are stored on the processor's stack.
5. During a function call the return address and arguments are pushed onto a stack and on return they are popped off.
6. Depth first search uses a stack data structure to find an element from a graph.

In-fix- to Postfix Transformation:

Procedure:

Procedure to convert from infix expression to postfix expression is as follows:

1. Scan the infix expression from left to right.
2. a) If the scanned symbol is left parenthesis, push it onto the stack.

   b) If the scanned symbol is an operand, then place directly in the postfix expression (output).

   c) If the symbol scanned is a right parenthesis, then go on popping all the items from the stack and place them in the postfix expression till we get the matching left parenthesis.

   d) If the scanned symbol is an operator, then go on removing all the operators from the stack and place them in the postfix expression, if and only if the precedence of the operator which is on the top of the stack is greater than (or equal) to the precedence of the scanned operator and push the scanned operator onto the stack otherwise, push the scanned operator onto the stack.

Convert the following infix expression A + B * C – D / E * H into its equivalent postfix expression.

Symbol	Postfix string	Stack	Remarks
A	A		
+	A	+	
B	A B	+	
*	A B	+ *	
C	A B C	-	
-	A B C * +	-	
D	A B C * + D	-	
/	A B C * + D	- /	
E	A B C * + D E	- /	
*	A B C * + D E /	- *	

H	A B C * + D E / H	- *	
End of string	A B C * + D E / H * -	The input is now empty. Pop the output symbols from the stack until it is empty.	

Source Code:

```
# Python program to convert infix expression to postfix

# Class to convert the expression
import string
class Conversion:

    # Constructor to initialize the class variables
    def __init__(self, capacity):
        self.top = -1
        self.capacity = capacity
        # This array is used a stack
        self.array = []
        # Precedence setting
        self.output = []
        self.precedence = {'+':1, '-':1, '*':2, '/':2, '^':3}

    # check if the stack is empty
    def isEmpty(self):
        return True if self.top == -1 else False

    # Return the value of the top of the stack
    def peek(self):
        return self.array[-1]

    # Pop the element from the stack
    def pop(self):
        if not self.isEmpty():
            self.top -= 1
            return self.array.pop()
        else:
            return "$"

    # Push the element to the stack
    def push(self, op):
        self.top += 1
        self.array.append(op)

    # A utility function to check is the given character
    # is operand
    def isOperand(self, ch):
        return ch.isalpha()

    # Check if the precedence of operator is strictly
    # less than top of stack or not
    def notGreater(self, i):
        try:
            a = self.precedence[i]
            b = self.precedence[self.peek()]
            return True if a  <= b else False
        except KeyError:
            return False

    # The main function that converts given infix expression
    # to postfix expression
    def infixToPostfix(self, exp):

        # Iterate over the expression for conversion
        for i in exp:
            # If the character is an operand,
            # add it to output
            if self.isOperand(i):
                self.output.append(i)

            # If the character is an '(', push it to stack
            elif i  == '(':
                self.push(i)

            # If the scanned character is an ')', pop and
            # output from the stack until and '(' is found
            elif i == ')':
                while( (not self.isEmpty()) and self.peek() != '('):
                    a = self.pop()
                    self.output.append(a)
                if (not self.isEmpty() and self.peek() != '('):
                    return -1
                else:
                    self.pop()

            # An operator is encountered
            else:
                while(not self.isEmpty() and self.notGreater(i)):
                    self.output.append(self.pop())
                self.push(i)

        # pop all the operator from the stack
        while not self.isEmpty():
            self.output.append(self.pop())

        result= "".join(self.output)
        print(result)
# Driver program to test above function
exp = "a+b*(c^d-e)^(f+g*h)-i"
obj = Conversion(len(exp))
obj.infixToPostfix(exp)
```

Evaluating Arithmetic Expressions:

Procedure:

The postfix expression is evaluated easily by the use of a stack. When a number is seen, it is pushed onto the stack; when an operator is seen, the operator is applied to the two numbers that are popped from the stack and the result is pushed onto the stack.

Evaluate the postfix expression: 6 5 2 3 + 8 * + 3 + *

Symbol	Operand 1	Operand 2	Value	Stack	Remarks
6				6	
5				6, 5	
2				6, 5, 2	
3				6, 5, 2, 3	The first four symbols are placed on the stack.
+	2	3	5	6, 5, 5	Next a '+' is read, so 3 and 2 are popped from the stack and their sum 5, is pushed
8	2	3	5	6, 5, 5, 8	Next 8 is pushed
*	5	8	40	6, 5, 40	Now a '*' is seen, so 8 and 5 are popped as 8 * 5 = 40 is pushed
+	5	40	45	6, 45	Next, a '+' is seen, so 40 and 5 are popped and 40 + 5 = 45 is pushed
3	5	40	45	6, 45, 3	Now, 3 is pushed
+	45	3	48	6, 48	Next, '+' pops 3 and 45 and pushes 45 + 3 = 48 is pushed
*	6	48	288	**288**	Finally, a '*' is seen and 48 and 6 are popped, the result 6 * 48 = 288 is pushed

Source Code:

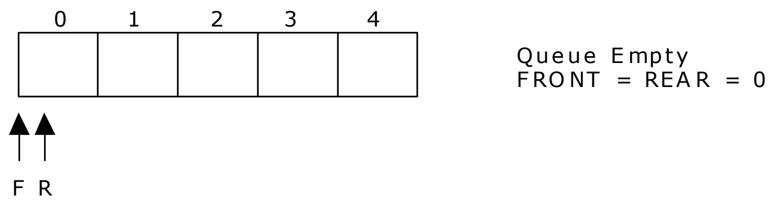
```
# Python program to evaluate value of a postfix expression

# Class to convert the expression
class Evaluate:

    # Constructor to initialize the class variables
    def __init__(self, capacity):
        self.top = -1
        self.capacity = capacity
        # This array is used a stack
        self.array = []

    # check if the stack is empty
    def isEmpty(self):
        return True if self.top == -1 else False

    # Return the value of the top of the stack
    def peek(self):
        return self.array[-1]

    # Pop the element from the stack
    def pop(self):
        if not self.isEmpty():
            self.top -= 1
            return self.array.pop()
        else:
            return "$"

    # Push the element to the stack
    def push(self, op):
        self.top += 1
        self.array.append(op)

    # The main function that converts given infix expression
    # to postfix expression
    def evaluatePostfix(self, exp):

        # Iterate over the expression for conversion
        for i in exp:

            # If the scanned character is an operand
            # (number here) push it to the stack
            if i.isdigit():
                self.push(i)

            # If the scanned character is an operator,
            # pop two elements from stack and apply it.
            else:
                val1 = self.pop()
                val2 = self.pop()
                self.push(str(eval(val2 + i + val1)))

        return int(self.pop())

# Driver program to test above function
exp = "231*+9-"
obj = Evaluate(len(exp))
print ("Value of {0} is {1}".format(exp, obj.evaluatePostfix(exp)))
```

Basic Queue Operations:

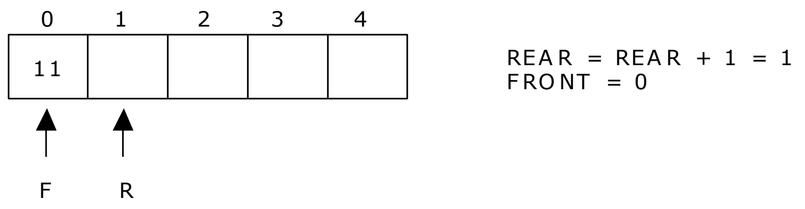
A queue is a data structure that is best described as "first in, first out". A queue is another special kind of list, where items are inserted at one end called the rear and deleted at the other end called the front. A real world example of a queue is people waiting in line at the bank. As each person enters the bank, he or she is "enqueued" at the back of the line. When a teller becomes available, they are "dequeued" at the front of the line.

Representation of a Queue using Array:

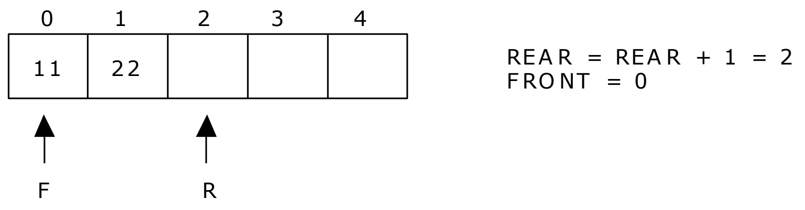
Let us consider a queue, which can hold maximum of five elements. Initially the queue is empty.



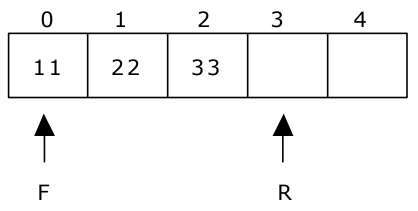
Now, insert 11 to the queue. Then queue status will be:



Next, insert 22 to the queue. Then the queue status is:

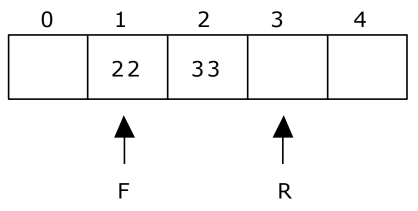


Again insert another element 33 to the queue. The status of the queue is:



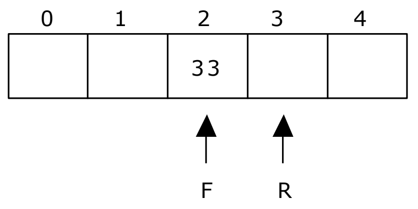
REAR = REAR + 1 = 3
FRONT = 0

Now, delete an element. The element deleted is the element at the front of the queue. So the status of the queue is:



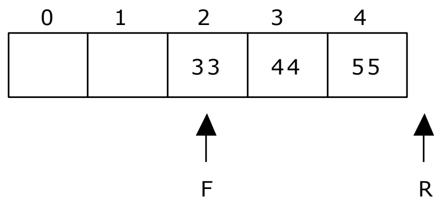
REAR = 3
FRONT = FRONT + 1 = 1

Again, delete an element. The element to be deleted is always pointed to by the FRONT pointer. So, 22 is deleted. The queue status is as follows:



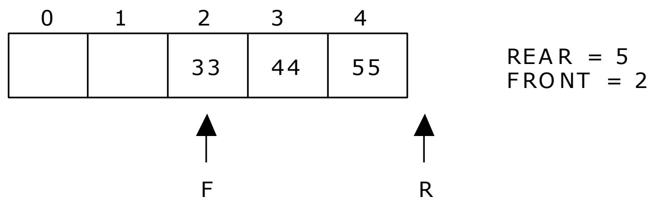
REAR = 3
FRONT = FRONT + 1 = 2

Now, insert new elements 44 and 55 into the queue. The queue status is:

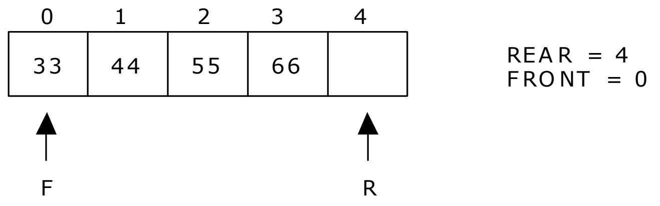


REAR = 5
FRONT = 2

Next insert another element, say 66 to the queue. We cannot insert 66 to the queue as the rear crossed the maximum size of the queue (i.e., 5). There will be queue full signal. The queue status is as follows:



Now it is not possible to insert an element 66 even though there are two vacant positions in the linear queue. To over come this problem the elements of the queue are to be shifted towards the beginning of the queue so that it creates vacant position at the rear end. Then the FRONT and REAR are to be adjusted properly. The element 66 can be inserted at the rear end. After this operation, the queue status is as follows:



This difficulty can overcome if we treat queue position with index 0 as a position that comes after position with index 4 i.e., we treat the queue as a **circular queue.**

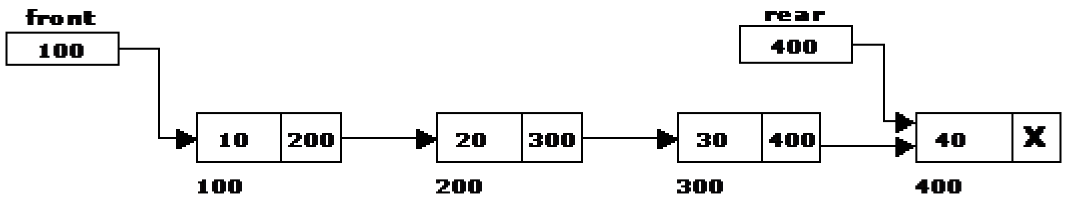
Procedure for Queue operations using array:

In order to create a queue we require a one dimensional array Q(1:n) and two variables *front* and *rear*. The conventions we shall adopt for these two variables are that *front* is always 1 less than the actual front of the queue and rear always points to the last element in the queue. Thus, front = rear if and only if there are no elements in the queue. The initial condition then is front = rear = 0.

The various queue operations to perform creation, deletion and display the elements in a queue are as follows:

1. insertQ(): inserts an element at the end of queue Q.
2. deleteQ(): deletes the first element of Q.
3. displayQ(): displays the elements in the queue.

Linked List Implementation of Queue: We can represent a queue as a linked list. In a queue data is deleted from the front end and inserted at the rear end. We can perform similar operations on the two ends of a list. We use two pointers front and rear for our linked queue implementation.



Source Code:

```
front = 0
rear = 0
mymax = 3
# Function to create a stack. It initializes size of stack as 0
def createQueue():
    queue = []
    return queue
 # Stack is empty when stack size is 0
def isEmpty(queue):
    return len(queue) == 0
# Function to add an item to stack. It increases size by 1
def enqueue(queue,item):
    queue.append(item)
# Function to remove an item from stack. It decreases size by 1
def dequeue(queue):
    if (isEmpty(queue)):
        return "Queue is empty"
    item=queue[0]
    del queue[0]
    return item
  # Driver program to test above functions
queue = createQueue()
while True:
    print("1 Enqueue")
    print("2 Dequeue")
    print("3 Display")
    print("4 Quit")
    ch=int(input("Enter choice"))
    if(ch==1):
        if(rear < mymax):
            item=input("enter item")
            enqueue(queue, item)
            rear = rear + 1
        else:
            print("Queue is full")
    elif(ch==2):
        print(dequeue(queue))
    elif(ch==3):
        print(queue)
    else:
        break
```

Applications of Queues:

1. It is used to schedule the jobs to be processed by the CPU.
2. When multiple users send print jobs to a printer, each printing job is kept in the printing queue. Then the printer prints those jobs according to first in first out (FIFO) basis.
3. Breadth first search uses a queue data structure to find an element from a graph.

Disadvantages of Linear Queue:

There are two problems associated with linear queue. They are:

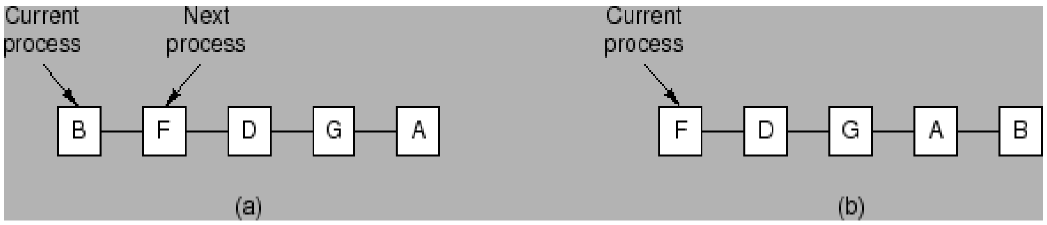
- Time consuming: linear time to be spent in shifting the elements to the beginning of the queue.
- Signaling queue full: even if the queue is having vacant position.

Round Robin Algorithm:

The round-robin (RR) scheduling algorithm is designed especially for time-sharing systems. It is similar to FCFS scheduling, but pre-emption is added to switch between processes. A small unit of time, called a time quantum or time slices, is defined. A time quantum is generally from 10 to 100 milliseconds. The ready queue is treated as a circular queue. To implement RR scheduling

- We keep the ready queue as a FIFO queue of processes.
- New processes are added to the tail of the ready queue.

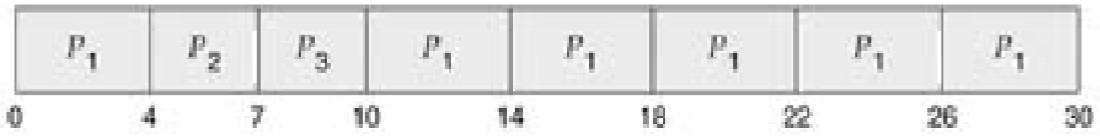
- The CPU scheduler picks the first process from the ready queue, sets a timer to interrupt after 1 time quantum, and dispatches the process.
- The process may have a CPU burst of less than 1 time quantum.
    - In this case, the process itself will release the CPU voluntarily.
    - The scheduler will then proceed to the next process in the ready queue.
- Otherwise, if the CPU burst of the currently running process is longer than 1 time quantum,
    - The timer will go off and will cause an interrupt to the OS.
    - A context switch will be executed, and the process will be put at the tail of the ready queue.
    - The CPU scheduler will then select the next process in the ready queue.



The average waiting time under the RR policy is often long. Consider the following set of processes that arrive at time 0, with the length of the CPU burst given in milliseconds: (a time quantum of 4 milliseconds)

	Burst	Waiting	Turnaround
Process	Time	Time	Time
	24	6	30
	3	4	7
	3	7	10
Average	-	5.66	15.66

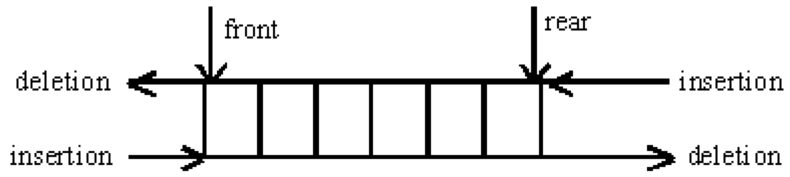
Using round-robin scheduling, we would schedule these processes according to the following chart:



DEQUE(Double Ended Queue):

A **double-ended queue** (**dequeue**, often abbreviated to **deque**, pronounced *deck*) generalizes a queue, for which elements can be added to or removed from either the front (head) or back (tail).It is also often called a **head-tail linked list.** Like an ordinary queue, a double-ended queue is a data structure it supports the following operations: enq_front, enq_back, deq_front, deq_back, and empty. Dequeue can be behave like a queue by using only enq_front and deq_front , and behaves like a stack by using only enq_front and deq_rear.

The DeQueue is represented as follows.



DEQUE can be represented in two ways they are

1) Input restricted DEQUE(IRD)

2) output restricted DEQUE(ORD)

The output restricted DEQUE allows deletions from only one end and input restricted DEQUE allow insertions at only one end. The DEQUE can be constructed in two ways they are

1) Using array

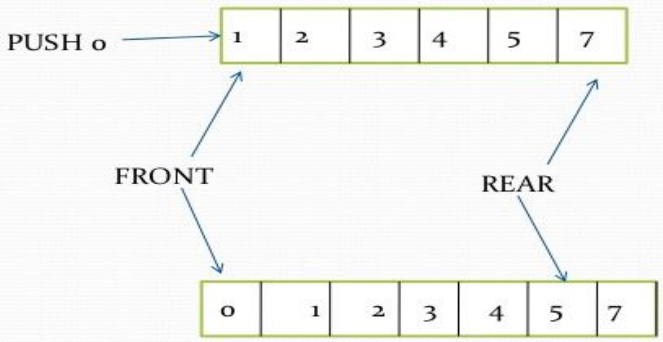
2)Using linked list

Operations in DEQUE

1. Insert element at back
2. Insert element at front
3. Remove element at front
4. Remove element at back

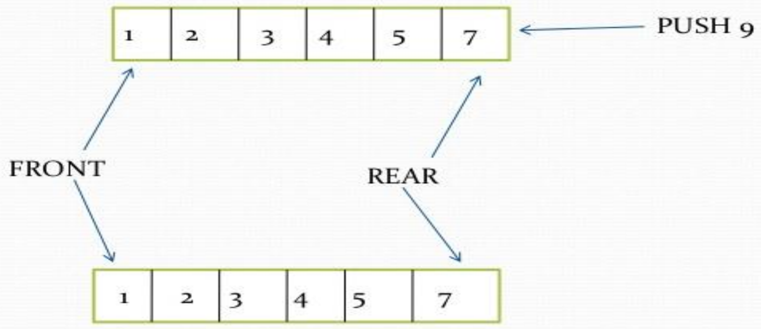
Insert_front

- insert_front() is a operation used to push an element into the front of the *Deque*.



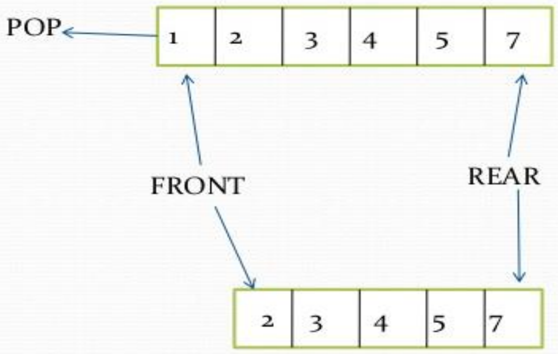
Insert_back

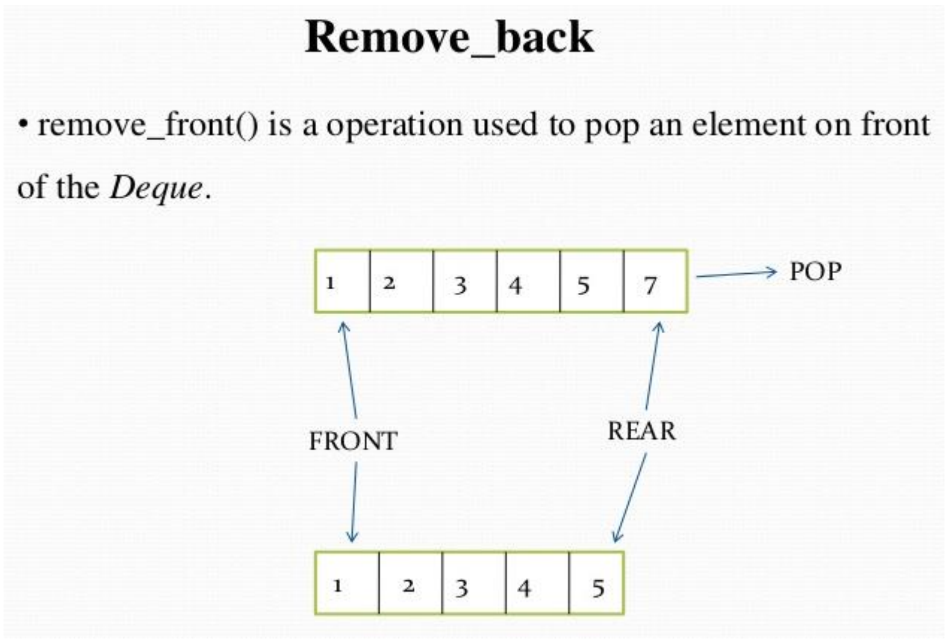
- insert_back() is a operation used to push an element at the back of a *Deque*.



Remove_front

- remove_front() is a operation used to pop an element on front of the *Deque*.





Remove_back

• remove_front() is a operation used to pop an element on front of the *Deque*.

Applications of DEQUE:

1. The A-Steal algorithm implements task scheduling for several processors (multiprocessor scheduling).
2. The processor gets the first element from the deque.
3. When one of the processor completes execution of its own threads it can steal a thread from another processor.
4. It gets the last element from the deque of another processor and executes it.

Circular Queue:

Circular queue is a linear data structure. It follows FIFO principle. In circular queue the last node is connected back to the first node to make a circle.

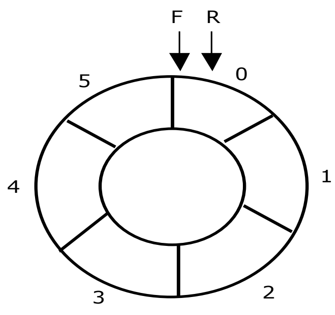
- Circular linked list fallow the First In First Out principle
- Elements are added at the rear end and the elements are deleted at front end of the queue
- Both the front and the rear pointers points to the beginning of the array.
- It is also called as “Ring buffer”.
- Items can inserted and deleted from a queue in O(1) time.

Circular Queue can be created in three ways they are

1. Using single linked list
2. Using double linked list
3. Using arrays

Representation of Circular Queue:

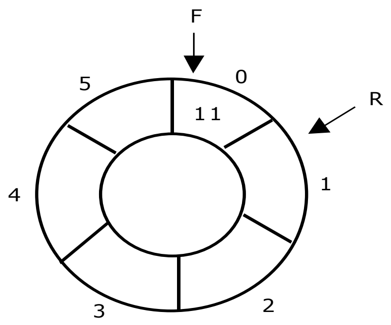
Let us consider a circular queue, which can hold maximum (MAX) of six elements. Initially the queue is empty.



Queue Empty
MAX = 6
FRONT = REAR = 0
COUNT = 0

Circular Queue

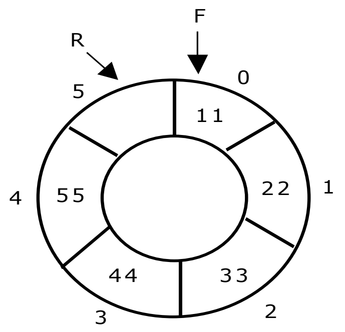
Now, insert 11 to the circular queue. Then circular queue status will be:



FRONT = 0
REAR = (REAR + 1) % 6 = 1
COUNT = 1

Circular Queue

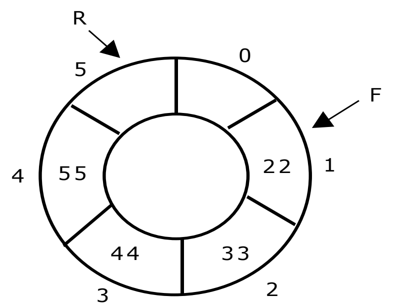
Insert new elements 22, 33, 44 and 55 into the circular queue. The circular queue status is:



FRONT = 0, REAR = 5
REAR = REAR % 6 = 5
COUNT = 5

Circular Queue

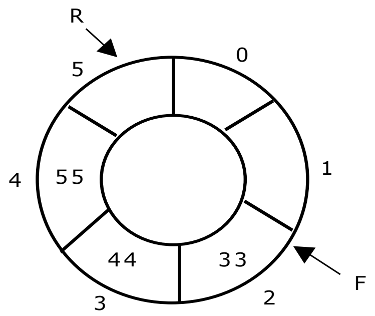
Now, delete an element. The element deleted is the element at the front of the circular queue. So, 11 is deleted. The circular queue status is as follows:



Circular Queue

```
FRONT = (FRONT + 1) % 6 = 1
REAR = 5
COUNT = COUNT - 1 = 4
```

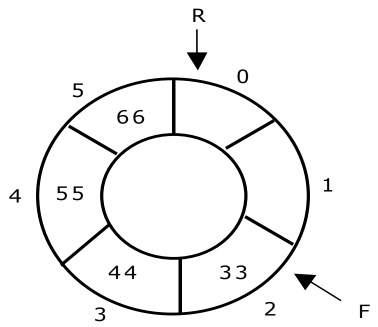
Again, delete an element. The element to be deleted is always pointed to by the FRONT pointer. So, 22 is deleted. The circular queue status is as follows:



Circular Queue

```
FRONT = (FRONT + 1) % 6 = 2
REAR = 5
COUNT = COUNT - 1 = 3
```

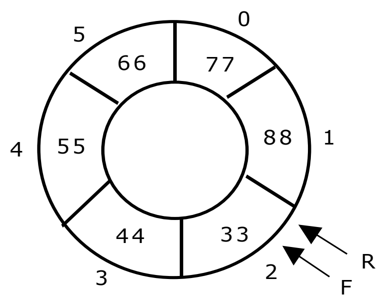
Again, insert another element 66 to the circular queue. The status of the circular queue is:



Circular Queue

```
FRONT = 2
REAR = (REAR + 1) % 6 = 0
COUNT = COUNT + 1 = 4
```

Now, insert new elements 77 and 88 into the circular queue. The circular queue status is:



FRONT = 2, REAR = 2
REAR = REAR % 6 = 2
COUNT = 6

Circular Queue

Now, if we insert an element to the circular queue, as COUNT = MAX we cannot add the element to circular queue. So, the circular queue is *full*.