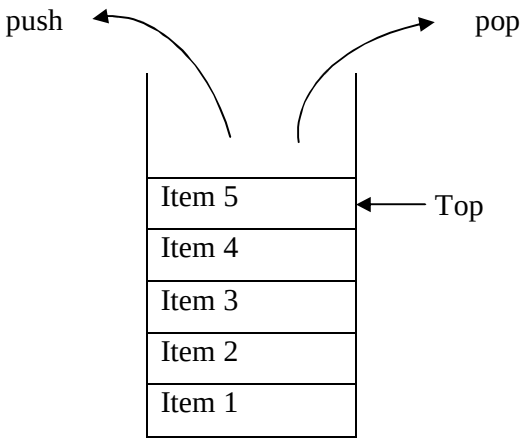
STACKS

- Stack is a linear data structure.
- Stack is an ordered collection of homogeneous data elements, where insertion and deletion operations take place at only end.
- The insertion operation is termed as PUSH and deletion operation is termed as POP operation.
- The PUSH and POP operations are performed at TOP of the stack.
- An element in a stack is termed as ITEM.
- The maximum number of elements that stack can accommodate is termed as SIZE of the stack.
- Stack follows LIFO principle. i.e. Last In First Out.



Schematic diagram of a stack

Representation of stack

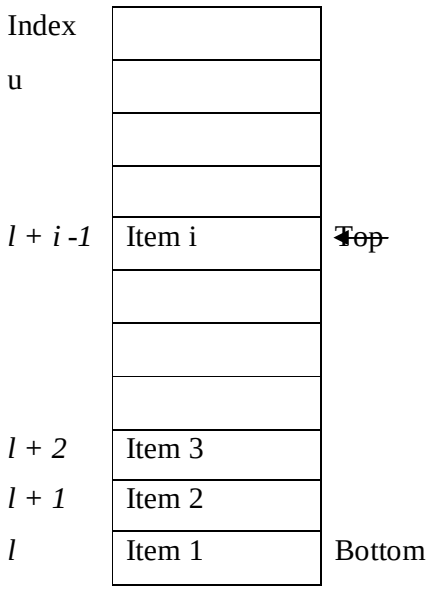
There are two ways of representation of a stack.

1. Array representation of a stack.
2. Linked List representation of a stack.

1. Array representation of a stack.

First we have to allocate memory for array.

Starting from the first location of the memory block, items of the stack can be stored in sequential fashion.



Array representation of stack

In the above figure item i denotes the ith item in stack.

l and u denotes the index ranges.

Usually *l* value is 1 and u value is size.

From the above representation tthe following two status can be stated.

Empty Stack: top < *l* i.e. top < 1

Stack is full: top > = *u*+*l*-1

i.e. top >= size + 1 -1

top >= size

Stack overflow

Trying to PUSH an item into full stack is known as stack overflow.

Stack overflow condition is top >= size

Stack underflow

Trying to POP an item from empty stack is known as Stack underflow.

Stack underflow condition is top < 1 or top = 0

Operations on Stack

PUSH : To insert element in to stack

POP : To delete element from stack

Status : To know present status of the stack

Algorithm Stack_PUSH(item)

Input: item is new item to push into stack

Output: pushing new item into stack at top whenever stack is not full.

1. if(top >= size)
    a) print(stack is full, not possible to perform push operation)
2. else
    a) top=top+1
    b) s[top]=item
3. End if

End Stack_PUSH

Algorithm Stack_POP( )

Input: Stack with some elements.

Output: item deleted at top most end.

1. if(top < 1)
    a) print(stack is empty not possible to pop)
2. else
    a) item=s[top]
    b) top=top-1
    c) print(deleted item)
3. End if

End Stack_POP

Algorithm Stack_Status( )

Input: Stack with some elements.

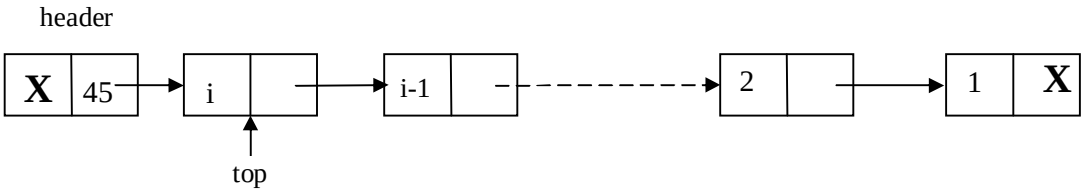
Output: Status of stack. i.e. Stack is empty or not, full or not, top most element in Stack.

1. if(top > = size)
    a) print(stack is full)
2. else if(top < 1)
    a. print(stack is empty)
3. else
    a) print(top most item in stack is s[top])
4. end if

End Stack_Status

2. Linked List representation of a stack

- The array representation of stack allows only fixed size of stack. i.e. static memory allocation only.
- To overcome the static memory allocation problem, linked list representation of stack is preferred.
- In linked list representation of stack, each node has two parts. One is data field is for the item and link field points to next node.



Linked List representation of stack

- Empty stack condition is

  top = NULL or header.link=NULL

- Full condition is not applicable for Linked List representation of stack. Because here memory is dynamically allocated**.**
- In linked List representation of stack, top pointer always points to top most node only. i.e. first node in the list.

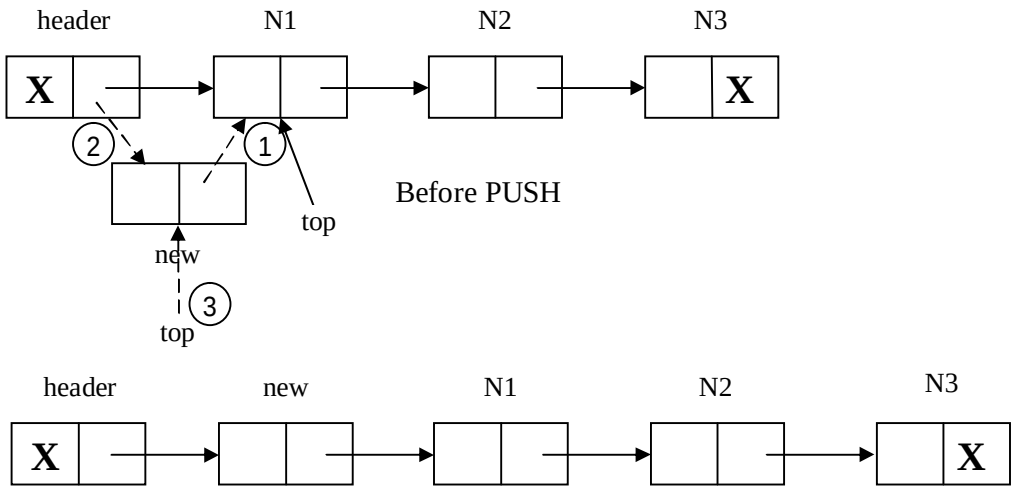
Operations on Stack with linked list representation

PUSH	:	To insert element in to stack
POP	:	To delete element from stack
Status	:	To know present status of the stack

Algorithm Stack_PUSH_LL(item)

1. new = getnewnode( )
2. if(new = = NULL)
   a) print(Required node is not available in memeory)
3. else
   a) new.link=header.link
   b) header.link=new
   c) top=new
   d) new.data=item

End Stack_PUSH_LL



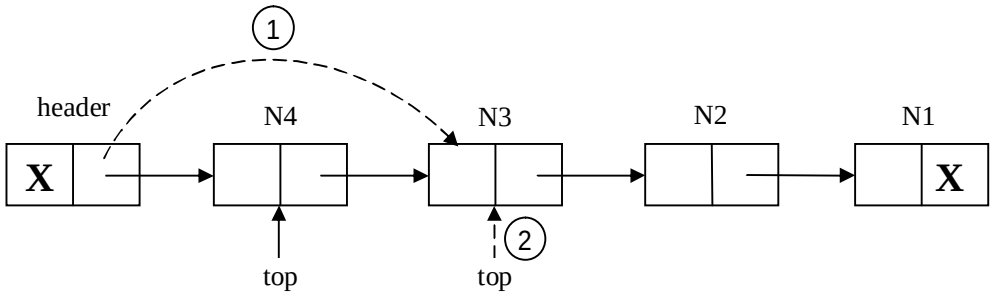
After PUSH

1. The link part of the new node is re placed with address of the previous top most node.
2. The link part of the header node is replaced with address of the new node.
3. Now the new node becomes top most node. So top is points to new node.

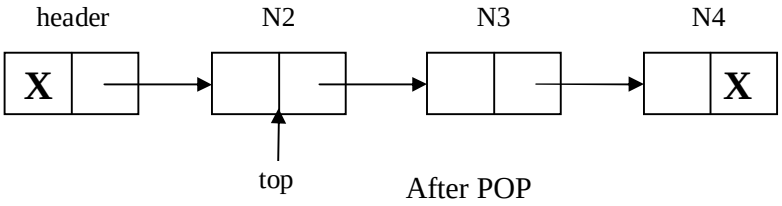
Algorithm Stack_POP_LL( )

1. if(header.link = = NULL)
    a) print(Stack is empty, unable to perform POP operartion)
2. else
    a) header.link=top.link
    b) item=top.data
    c) top=header.link

End Stack_POP_LL



Before POP



After POP

1. Link part of the header node is replaced with the address of the second node in the list.
2. After deletion of top most node from list, the second node becomes the top most node in the list. So top points to the second node.

Algorithm Stack_Status_LL( )

Input: Stack with some elements.

Output: Status of stack. i.e. Stack is empty or not, top most element in Stack.

1. if header.link = = NULL or top = = NULL)
   a) print(Stack is empty)
2. else
   a) print(Element present at top of stack is top.data)

3,end if

End Stack_Status_LL

Applications of stack

1. Factorial calculation
2. Infix to postfix conversion
3. Evaluation of postfix expression
4. Reversing list of elements

1. Factorial Calculation

- To calculate the factorial of a given number using stack, we require two stacks. One for storing the parameter n and another stack is hold the return address.
- Let us assume the two stacks, one PARAM for parameter and ADDR for return address.
- Assume PUSH(X,Y) operation for pushing the items X and Y into the stack PARAM and ADDR respectively.

Algorithm Factorial_Stack(n <integer>)

Input: An integer n

Output: Factorial of n

1. Top = 0
2. Addr = step10
3. push(n,addr)
4. n = n - 1 , addr = step8
5. if (n = = 0 OR n = = 1)
    a) fact = 1
    b) goto step9
6. else
    a) push(n,addr)
    b) goto step4
7. end if
8. fact = fact * n
9. n = pop_PARAM( ), addr = pop_ADDR( )
10. goto addr
11. return(fact)

End Factorial_Stack

2. Infix to postfix conversion

An expression is a combination of operands and operators.

Eg. c= a + b

In the above expression a, b, c are operands and +, = are called as operators.

We have 3 notations for the expressions.

i. Infix notation
ii. Prefix notation
iii. Postfix notation

Infix notation: Here operator is present between two operands.

eg. a + b

The format for Infix notation as follows

<operand> <operator> <operand>

Prefix notation: Here operator is present before two operands.

eg. + a b

The format for Prefix notation as follows

<operator> <operand> <operand>

Postfix notation: Here operator is present after two operands.

eg. a b +

The format for Prefix notation as follows

<operand> <operand> <operator>

While conversion of infix expression to postfix expression, we must follow the precedence and associativity of the operators.

Operator	**Precedence**	**Associativity**
^ or $ (exponential)	3	Right to Left
* / %	2	Left to Right
+ -	1	Left to Right

In the above table * and / have same precedence. So then go for associativity rule, i.e. from Left to Right.

Similarly + and - same precedence. So then go for associativity rule, i.e. from Left to Right.

Eg. 1

(A + B) * (C – D)

A B + * (C – D)

A B + * C D –

A B + C D - *

Eg. 2

([(A - { B + C }) * D] $ E + F)

([(A - BC+) * D] $ E + F)

([ABC+- * D] $ E + F)

(ABC+-D* $ E + F)

(ABC+-D*E$ + F)

ABC=-D*E$F+

- To convert an infix expression to postfix expression, we can use one stack.
- Within the stack, we place only operators and left parenthesis only. So stack used in conversion of infix expression to postfix expression is called as operator stack.

Algorithm Conversion of infix to postfix

Input: Infix expression.

Output: Postfix expression.

1. Perform the following steps while reading of infix expression is not over
    a) if symbol is left parenthesis then push symbol into stack.
    b) if symbol is operand then add symbol to post fix expression.
    c) if symbol is operator then check stack is empty or not.
        i) if stack is empty then push the operator into stack.
        ii) if stack is not empty then check priority of the operators.
            (I) if priority of current operator > priority of operator present at top of stack then push operator into stack.
            (II) else if priority of operator present at top of stack >= priority of current operator then pop the operator present at top of stack and add popped operator to postfix expression (go to step I)
    d) if symbol is right parenthesis then pop every element form stack up corresponding left parenthesis and add the poped elements to postfix expression.
2. After completion of reading infix expression, if stack not empty then pop all the items from stack and then add to post fix expression.

End conversion of infix to postfix

3. Evaluation of postfix expression

- To evaluate a postfix expression we use one stack.
- For Evaluation of postfix expression, in the stack we can store only operand. So stack used in Evaluation of postfix expression is called as operand stack.

Algorithm PostfixExpressionEvaluation

Input: Postfix expression

Output: Result of Expression

1. Repeat the following steps while reading the postfix expression.
    a) if the read symbol is operand, then push the symbol into stack.
    b) if the read symbol is operator then pop the top most two items of the stack and apply the operator on them, and then push back the result to the stack.
2. Finally stack has only one item, after completion of reading the postfix expression. That item is the result of expression.

End PostfixExpressionEvaluation

4. Reversing List of elements

- A list of numbers can be reversed by reading each number from an array starting from 1st index and pushing into stack.
- Once all the numbers have been push into stack, the numbers can be poped one by one from stack and store ito array from the 1st index.

Algorithm Reverse_List_Stack(a <array>, n <intetger>)

Input : Array a with n elements

Output: Reversed List of elements

1. i=1 , top =0
2. while (I <=n)
    a) top = top + 1
    b) s[top] = a[i]
    c) i = I + 1
3. end while loop
4. i = 1
5. while(i <= n)
    a) a[i] = s[top]
    b) top = top – 1

c) i = i + 1

6. end while loop

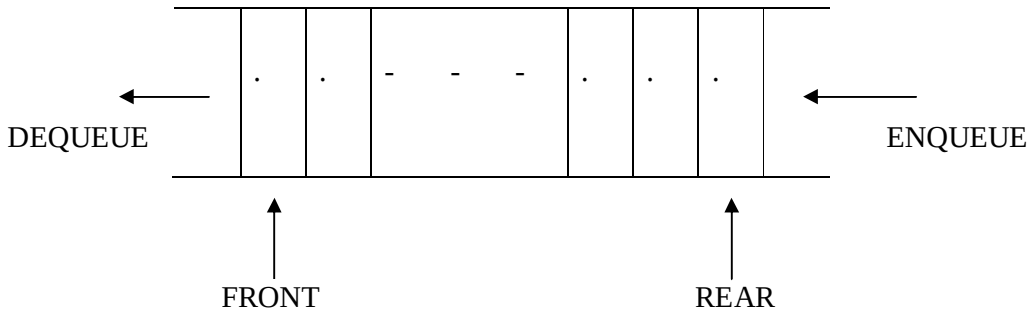
End Reverse_List_Stac

QUEUES

Queue is a linear Data structure.

Definition: Queue is a collection of homogeneous data elements, where insertion and deletion operations are performed at two extreme ends.

- The insertion operation in Queue is termed as ENQUEUE.
- The deletion operation in Queue is termed as DEQUEUE.
- An element present in queue are termed as ITEEM.
- The number of elements that a queue can accommodate is termed as LENGTH of the Queue.
- In the Queue the ENQUEUE (insertion) operation is performed at REAR end and DEQUEUE (deletion) operation is performed at FRONT end.
- Queue follows FIFO principle. i.e. First In First Out principle. i.e. a item First inserted into Queue, that item only First deleted from Queue, so queue follows FIFO principle.



Schematic Representation of Queue

Representation of Queue

A Queue can be represented in two ways

1. Using arrays
2. Using Linked List

1. Representation of Queue using arrays

A one dimensional array Q[1-N] can be used to represent a queue.

Array representation of Queue

In array representation of Queue, two pointers are used to indicate two ends of Queue.

The above representation states as follows.

1. Queue Empty condition

   Front = 0 and Rear = 0

2. Queue Full condition

   Rear = N where N is the size of the array we are taken

3. Queue conta e element

   Front = = Rear

4. Number of items in Queue is

   Rear – Front + 1

Queue overflow: Trying to perform ENQUEUE (insertion) operation in full Queue is known as Queue overflow.

Queue overflow condition is Rear > = N

Queue Underflow: Trying to perform DEQUEUE (deletion) operation on empty Queue is known as Queue Underflow.

Queue Underflow condition is Front = 0

Operation on Queue

1. ENQUEUE : To insert element in to Queue
2. DEQUEUE : To delete element from Queue
3. Status : To know present status of the Queue

Algorithm Enqueue(item)

Input: item is new item insert in to queue at rear end.

Output: Insertion of new item queue at rear end if queue is not full.

1. if(rear = = N)

   a)print(queue is full, not possible for enqueue operation)

2. else

i) if(front = = 0 and rear = = 0) /* Q is Empty */

a) rear=rear+1

b) Q[rear]=item

c) front=1

ii) else

a) rear=rear+1

b) Q[rear]=item

iii) end if

3.end if

End Enqueue

While performing ENQUEUE operation two situations are occur.

1. if queue is empty, then newly inserting element becomes first element and last element in the queue. So Front and Rear points to first element in the list.
2. If Queue is not empty, then newly inserting element is inserted at Rear end.

Algorithm Dequeue( )

Input: Queue with some elements.

Output: Element is deleted from queue at front end if queue is not empty.

1. if(front = = 0 and rear = = 0)

a) print(Q is empty, not possible for dequeue operation)

2. else

i) if(front = = rear) /* Q has only one element */

a) item=Q[front]

b) front=0

c) rear=0

ii) else

a)item=Q[front]

b)front=front+1

iii) end if

iv) print(deleted item is item)

3. end if

End Dequeue

While performing DEQUEUE operation two situations are occur.

1. if queue has only one element, then after deletion of that element Queue becomes empty. So Front and Rear becomes 0.
2. If Queue has more than one element, then first element is deleted at Front end.

Algorithm Queue_Status( )

Input: Queue with some elements.

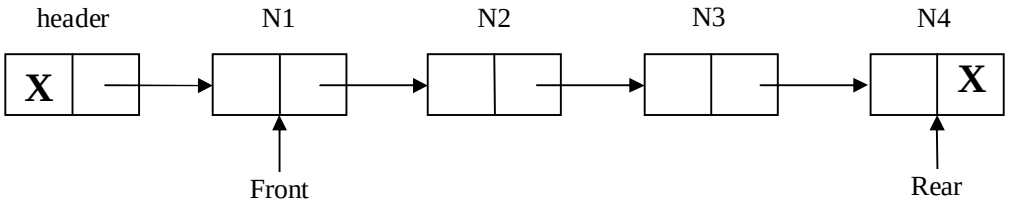
Output: Status of the queue. i.e. Q is empty or not, Q is full or not, Element at front end and rear end.

1. if(front = = 0 and rear = = 0)
    a) print(Q is empty)
2. else if (rear = = size)
    a) print(Q is full)
3. else
    i) if(front = = rear)
        a) print(Q has only one item)
    ii) else
        a) print(element at front end is Q[front])
        b) print(element at rear end is Q[rear])
    iii) end if
4. end if

End Queue_Status

2. Representation of Queue using Linked List

- Array representation of Queue has static memory allocation only.
- To overcome the static memory allocation problem, Queue can be represented using Linked List.



Linked List Representation of Queue

- In Linked List Representation of Queue, **Front** always points to **First** node in the Linked List and **Rear** always points to **Last** node in the Linked List.

The Linked List representation of Queue stated as follows.

1. Empty Queue condition is

   Front = NULL and Rear = NULL or header.link == NULL

2. Queue full condition is not available in Linked List representation of Queue, because in Linked List representation memory is allocated dynamically.

3. Queue has only one element

   Front = = Rear

Operation on Linked List Representation of Queue

1. ENQUEUE : To insert element in to Queue
2. DEQUEUE : To delete element from Queue
3. Status : To know present status of the Queue

Algorithm Enqueue _LL(item)

Input: item is new item to be insert.

Output : new item i.e new node is inserted at rear end.

```
1. new=getnewnode()
2. if(new = = NULL)
       a) print(required node is not available in memory)
3. else
       i) if(front = = NULL and rear = = NULL)     /* Q is EMPTY */
              a) header.link=new
              b) new.link=NULL
```

c) front=new

d) rear=new

e) new.data=item

ii) else /* Q is not EMPTY */

a) rear.link=new /* 1 */

b) new.link=NULL /* 2 */

c) rear=new /* 3 */

d) new.data=item

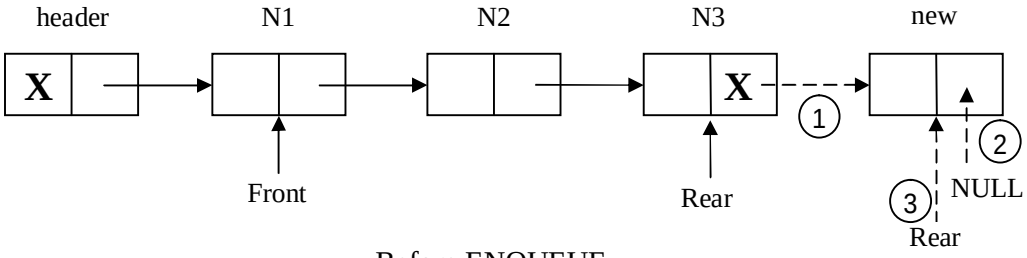
iii) end if

4. end if

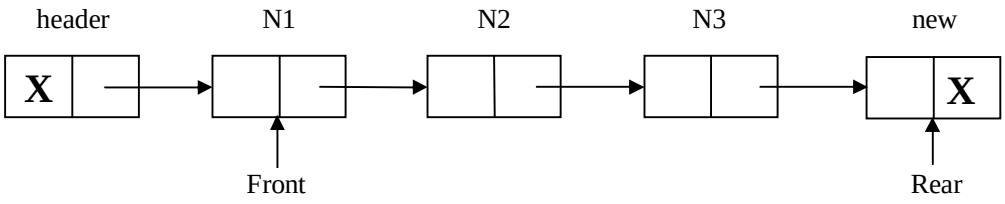
End_Enqueue_LL

While performing ENQUEUE operation two situations are occur.

1. if queue is empty, then newly inserting element becomes first node and last node in the queue. So Front and Rear points to first node in the list.
2. If Queue is not empty, then newly inserting node is inserted at last.



Before ENQUEUE



After ENQUEUE

1. Previous last node link part is replaced with address of new node.
2. Link part of new node is replaced with NULL, because new nodes becomes the last node.
3. Rear is points to last node in the list. i.e. newly inserted node in the list.

Algorithm Dequeue_LL( )

Input: Queue with some elements

Output: Element is deleted at front end if queue is not empty.

1.if(front= =NULL and rear = =NULL)

a) print(queue is empty, not possible to perform dequeue operation)

2. else

i) if(front = = rear) /* Q has only one element */

a) header.link = NULL

b) item=front.data

c) front=NULL

d) rear=NULL

b) else /* Q has more than one element */

a) header.link = front.link /* 1 */

b) item=front.data

c) free(front)

d)front=header.link /* 2 */

c) end if

d) print(deleted element is item)

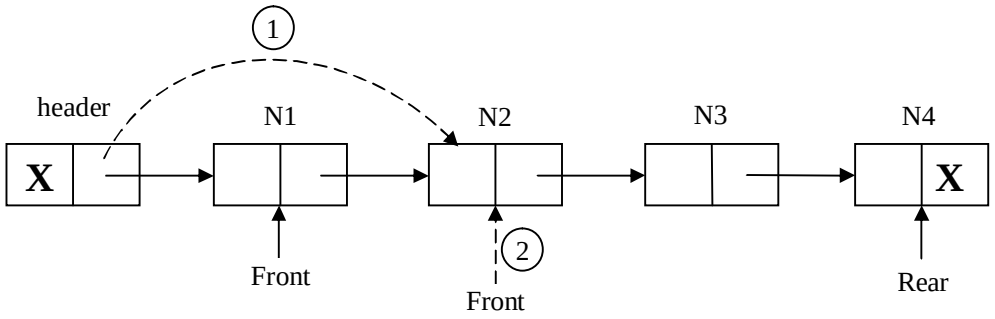
3. end if

End_Dequeue_LL

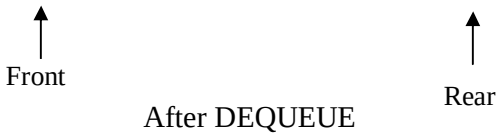
While performing DEQUEUE operation two situations are occur.

1. if queue has only one element, then after deletion of that element Queue becomes empty. So Front and Rear points to NULL.

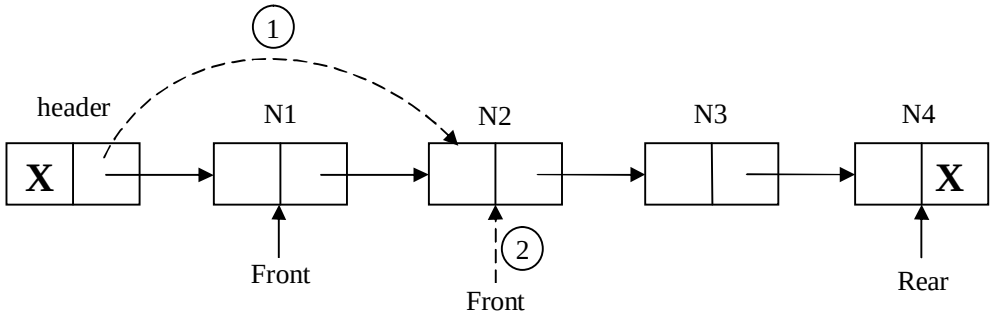
2. If Queue has more than one element, then first node is deleted at Front end.



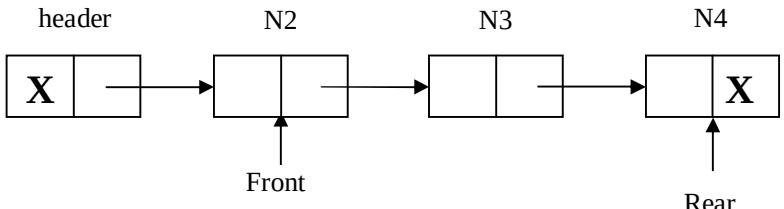
Before DEQUEUE



After DEQUEUE



Before DEQUEUE



After DEQUEUE

1. Link part of the header node is replaced with address of second node. i.e. address of second node is available in link part of first node.
2. Front is set to first node in the list.

Algorithm Queue_Status_LL

Input: Queue with some elements

Output: Status of the queue. i.e. Q is empty or not, Q is full or not, Element at front end and rear end.

1. if(front = = NULL and rear = = NULL)
    a) print(Q is empty)
2. else if(front = = rear)
    a) print(Q has only one item)
3. else
    a) print(element at front end is front.data)
    b) print(element at rear end is rear.data)
4. end if

End Queue_Status_LL

Queue operations using stack

To perform Queue operations using stack, we require two stacks named as stack1 and stack2. The Queue operations using stack can perform in two ways.

1. **Enqueue operation is cost effective**
2. **Dequeue operation is cost effective**

1. Enqueue operation is cost effective

Algorithm Enqueue_stack(item)

1. While stack1 is not empty, PUSH every element from stack1 to stack2.
2. PUSH item in to stack1.
3. While stack2 is not empty, PUSH every element from stack2 to stack1.

End Enqueue_stack

Algorithm Dequeue_stack( )

1. If stack1 is empty then error occurs. i.e. queue is empty.
2. Else POP an item from stack1.

End Dequeue_stack

2. Dequeue operation is cost effective

Algorithm Enqueue_stack(item)

1. PUSH item into stack1

End Enqueue_stack

Algorithm Dequeue_stack( )

1. If stack1 and stack2 are empty then error occur. i.e. Queue is empty.
2. Else if stack2 is empty
    a) While stack1 is not empty, PUSH ever element from stack1 to stack2.
    b) POP element from stack2.
    c) While stack2 is not empty, PUSH ever element from stack2 to stack1
3. End if

End Dequeue_stack

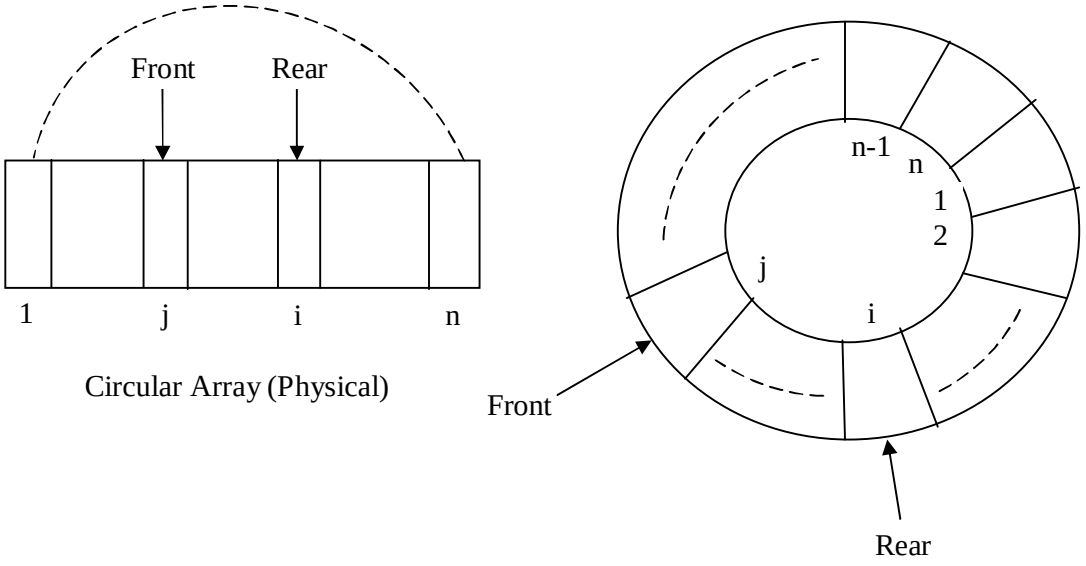
Various Queue Structures

1. **Circular Queues**
2. **Deque**
3. **Priority Queue**

1. Circular Queues

Physically a circular array is same as ordinary arra, say a[i-N], but logically it implements that a[1] comes after a[N] or a[N] comes after a[1].

The following figure shows the physical and logical representation for circular array



Logical and physical view of a Circular Queue

- Here both Front and Rear pointers are move in clockwise direction. This is controlled by the MOD operation.
- For e.g. if the current pointer is at i, then shift next location will be
  (i mod LENTH) +1, 1<= i <= Length

Circular Queue empty condition is

Front = 0 and Rear = 0

Circular Queue is full

Front = = (Rear Mod Length) + 1

Algorithm CQ_Enqueue(item)

Input: item is new item insert in to Circular queue at rear end.

Output: Insertion of new item Circular queue at rear end if vqueue is not full.

1. next = (rear % N) = 1
2. if(front == next)
   a)print(Circular queue is full, not possible for enqueue operation)

3. else

i) if(front = = 0 and rear = = 0) /* CQ is Empty */

a) rear=(rear % N) = 1

b) CQ[rear]=item

c) front=1

ii) else

a) rear=(rear % N) = 1

b) CQ[rear]=item

iii) end if

4.end if

End CQ_Enqueue

Algorithm CQ_Dequeue( )

Input: Circular Queue with some elements.

Output: Element is deleted from circular queue at front end if circular queue is not empty.

1. if(front = = 0 and rear = = 0)

a) print(CQ is empty, not possible for dequeue operation)

2. else

i) if(front = = rear) /* Q has only one element */

a) item=CQ[front]

b) front=0

c) rear=0

ii) else

a)item=CQ[front]

b)front=(front % N)+1

iii) end if

iv) print(deleted item is item)

3. end if

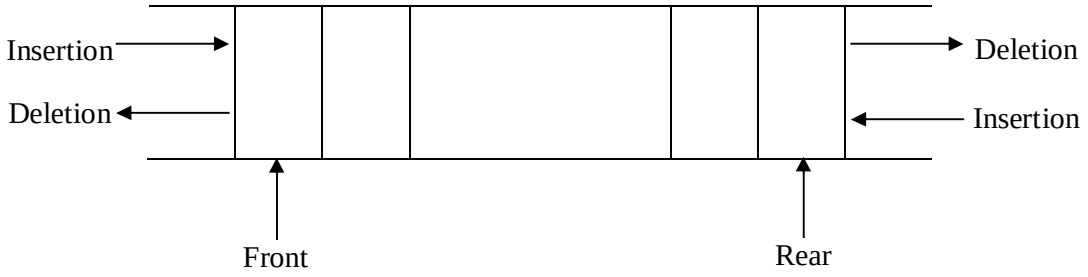
End CQ_Dequeue

2. Deque

Another variation of queue is known as DEQue.

In DEQue, both ENQUEUE (insertion) and DEQUEUE (deletion) operations can be made either of the ends.

DEQue is organized from Double Ended Queue.



A DEQue structure

Here DEQue structure is general representation of stack and Queue. In other words, a DEQue can be used as stack and Queue.

DeQue can be represented in two ways.

1. Using Double Linked List
2. Using a Circular Queue

Here Circular array is popular representation of DEQue.

On DEQue, the following four operations can be performed.

1. PUSHDQ(item) : To insert item at FRONT end of DEQue.
2. POPDQ( ) : To delete the FRONT end item from DEQue.
3. INJECT(item) : To insert item at REAR end of DEQue.
4. EJECT( ) : To delete the REAR end item from DEQue.

PUSHDQ(item)

If FRONT = 1, then next position of FRONT = Length.

(Here FRONT = 1 means FRONT points to extreme Left)

If FRONT = Length, then next position of FRONT = 1.

(Here FRONT = Length means FRONT points to extreme Right)

Otherwise, i.e. FRONT is at intermediate position. Then next position of FRONT = FRONT – 1.

Algorithm PUSHDQ(item)

Input: item is new item is inserted in to Queue at FRONT end.

Output: New item is inserted in to Queue at FRONT end.

If (front = = 1)

(i) next = Length

else if (front = = Length OR front = = 0)

(i) next = 1

else

(i) next = front – 1

end if

if(next = rear)

(i) Print(Queue is full)

else

(i) Front = next

(ii) DQ[front] = item

end if

End PUSHDQ

POPDQ( ) : This algorithm is same as CQDeque( ) algorithm.

INJECT(item) : This algorithm is same as CQEnque( ) algorithm.

EJECT( )

If REAR = 1, then next position of REAR = Length.

(Here REAR = 1 means REAR points to extreme Left)

If REAR = Length, then next position of REAR = 1.

(Here REAR = Length means REAR points to extreme Right)

Otherwise, i.e. REAR is at intermediate position. Then next position of REAR = REAR – 1.

Algorithm EJECT( )

Input : A DEQUE with item.

Output: An item is deleted from REAR end.

if(front == 0 and rear = = 0)

(i) print(Queue is empty, not possible to delete)

else

(i) if(front = = rear)

(a) item = DQ[item]

Front = 0

(c) Rear = 0

else /* DEQue has more than one element */

(a) if(rear = 1)

(A) item = DQ[rear]

(B) rear = Length

(b) else if(rear = = Length)

(A) item = DQ[rear]

(B) rear = 1

(c) else

(A) item = DQ[rear]

(B) rear = rear -1

(d) End if

(iii) End if

End if

End EJECT

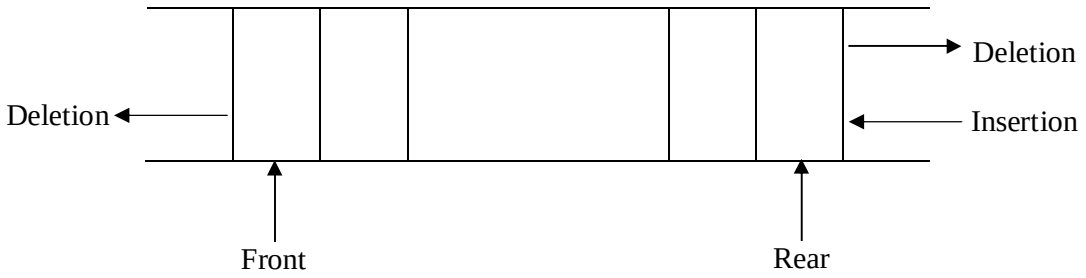
There are two variations of DEQue known as

Input restricted DEQue

Output restricted DEQue

Input restricted DEQue

Here DEQue allows insertion at one end (say REAR end) only, but allows deletion at both ends.

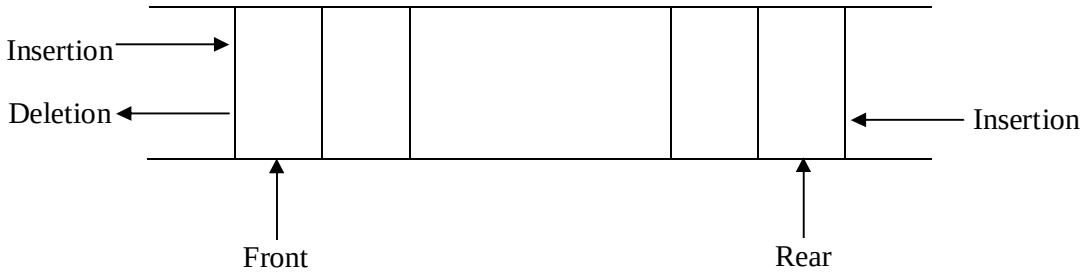


Input restricted DEQue

Output restricted DEQue

Here DEQue allows deletion at one end (say FRONT end) only, but allows insertion at both ends.

DEQue is organized from Double Ended Queue.



Output restricted DEQue

\

UNIT-III
Assignment-Cum-Tutorial Questions
SECTION-A

Objective Questions

1) To add and remove nodes from a queue _____ access is used. [ ]

a.) LIFO, Last In First Out b). FIFO, First In First Out
c). Both a and b d) . None

2.)Which one of the following is an application of Queue Data Structure?
a) When a resource is shared among multiple consumers. [ ]
b) When data is transferred asynchronously
c) Load Balancing
d) All of the above

3.)Which of the following is not the type of queue? [ ]
a) Ordinary queue b) Single ended queue c) Circular queue d) Priority queue

4.) suppose a circular queue of capacity (n – 1) elements is implemented with an array of n elements. Assume that the insertion and deletion operation are carried out using REAR and FRONT as array index variables, respectively. Initially, REAR = FRONT = 0. The conditions to detect queue full and queue empty are
a) Full: (REAR+1) mod n == FRONT, empty: REAR == FRONT [ ]
b) Full: (REAR+1) mod n == FRONT, empty: (FRONT+1) mod n == REAR
C) Full: REAR == FRONT, empty: (REAR+1) mod n == FRONT
d) Full: (FRONT+1) mod n == REAR, empty: REAR == FRONT

5.)What is the need for a circular queue? [ ]
a) effective usage of memory b) easier computations
c) all of the mentioned d.)none

6.What is the space complexity of a linear queue having n elements? [ ]
a) O(n) b) O(nlogn) c) O(logn) d) O(1)

7).In linked list implementation of a queue, where does a new element be deleted? [ ]

a) At the head of linked list b) At the tail of the linked list

c) At the centre position in the linked list d) None of the above

8.)In a circular queue, how do you increment the rear end of the queue?

a) rear++ b) (rear+1) % CAPACITY [ ]

c) (rear % CAPACITY)+1 d) rear–

9. In linked list implementation of a queue, front and rear pointers are tracked. Which of these pointers will change during an insertion into a NONEMPTY queue? [ ]

a) Only front pointer b) Only rear pointer c) Both front and rear pointer

d) None of the mentioned

10.The value of REAR is increased by 1 when ……. [ ]

a. An element is deleted in a queue b. An element is traversed in a queue

c. An element is added in a queue d.None

11.)What is the time complexity of pop() operation when the stack is implemented using an array? [ ]

a) O(1) b) O(n) c) O(logn) d) O(nlogn)

12.)Which of the following is true about linked list implementation of stack?

a) In push operation, if new nodes are inserted at the beginning of linked list, then in pop operation, nodes must be removed from end. [ ]

b) In push operation, if new nodes are inserted at the end, then in pop operation, nodes must be removed from the beginning.

c) Both of the above

d**)** None of the above

13.)Suppose a stack is to be implemented with a linked list instead of an array. What would be the effect on the time complexity of the push and pop operations of the stack implemented using linked list (Assuming stack is implemented efficiently)?

a) O(1) for insertion and O(n) for deletion [ ]
b) O(1) for insertion and O(1) for deletion
c) O(n) for insertion and O(1) for deletion
d) O(n) for insertion and O(n) for deletion

14.)Which of the following permutation can be obtained in the same order using a stack assuming that input is the sequence 5, 6, 7, 8, 9 in that order? [ ]
a) 7, 8, 9, 5, 6 b) 5, 9, 6, 7, 8 c) 7, 8, 9, 6, 5 d) 9, 8, 7, 5, 6

15.)If the sequence of operations – push (1), push (2), pop, push (1), push (2), pop, pop, pop, push (2), pop are performed on a stack, the sequence of popped out values [ ]
a) 2,2,1,1,2 b) 2,2,1,2,2 c) 2,1,2,2,1 d) 2,1,2,2,2

16.)The postfix form of the expression (A+ B)*(C*D- E)*F / G is? [ ]
a) AB+ CD*E – FG /** b) AB + CD* E – F **G /
c) AB + CD* E – *F *G / d) AB + CDE * – * F *G /

17.)The postfix form of A*B+C/D is? [ ]
a) *AB/CD+ b) AB*CD/+ c) A*BC+/D d) ABCD+/*

18.The prefix form of A-B/ (C * D ^ E) is? [ ]
a) -/*^ACBDE b) -ABCD*^DE c) -A/B*C^DE d) -A/BC*^DE

19.) The result of evaluating the postfix expression 5, 4, 6, +, *, 4, 9, 3, /, +, * is? [ ]
a) 600 b) 350 c) 650 d) 588

20.)Which of the following data structures can be used for parentheses matching? [ ]
a) n-ary tree b) queue c) priority queue d) stack

SECTION-B

SUBJECTIVE QUESTIONS

1 .Explain the prefix and post fix notation of (a + b) * (c + d) ?

2 . Define what is stack? Why do we use stack ? And what are the operations performed on stacks?

3. Convert the expression **(a+b)/d-((e-f)%g)** into reverse polish notation using stack and show the contents of stack for every operation.

4. Evaluate the expression **12/3*6+6-6+8%2** using stack.

5. Convert the expression **a+b*c/d%e-f** into postfix expression using stack.

6. Implement queue using arrays?

7 .Implement queue using Linked List?

8. What is Queue? discuss the types of Queues ?And explain why we we are going for circular queue?

9.Lsit out Applications of Stacks?

10.List out applications of queues?

SECTION-C

QUESTIONS AT THE LEVEL OF GATE

1.Consider the following pseudocode that uses a stack

```
declare a stack of characters
while ( there are more characters in the word to read )
{
  read a character
  push the character on the stack
}
while ( the stack is not empty )
{
  pop a character off the stack
  write the character to the screen
}
```

What is output for input "geeksquiz"?

What is output for input “geeksquiz”? [ ]

(A) geeksquizgeeksquiz
(B) ziuqskeeg
(C) geeksquiz
(D) ziuqskeegziuqskeeg

2. Assume that the operators +, -, × are left associative and ^ is right associative. The order of precedence (from highest to lowest) is ^, x , +, -. The postfix expression corresponding to the infix expression a + b × c - d ^ e ^ f is

abc × + def ^ ^ - [ ]

abc × + de ^ f ^ -

ab + c × d - e ^ f ^

- + a × bc ^ ^ def

3.The following postfix expression with single digit operands is evaluated using a stack: [ ]

```
8 2 3 ^ / 2 3 * + 5 1 * -
```

Note that ^ is the exponentiation operator. The top two elements of the stack after the first * is evaluated are:

(A) 6, 1 (B) 5, 7 (C) 3, 2 (D) 1, 5