Stacks and Queues

Stack: - Stack is a linear data structure in which a data item is inserted and deleted at one end. This end is often known as **top** of stack. A stack is called a Last in First out (**LIFO**) structure because the data item that is last inserted into the stack is the first data item to be deleted from the stack.

In general, **writing** a value to the stack is called as a **push** operation whereas **reading** a value from it is called as a **pop** operation.

Operations on stack:

A stack is generally implemented with two basic operations-push and pop

Operation	**description**
Push	Adding an element at the top of the stack
Pop	Removing an element from the top of the stack
peek	Returns the value of the topmost element of the stack

EXAMPLE:-

Operation	**contents of the stack**	**Action**
Push A	A	:A is pushed
Push B	A B	:B is pushed
Pop	A	:B is popped(deleted)

Push C	A C	:C is inserted
Push D	A C D	:D is inserted
Push E	A C D E	:E is inserted
Pop	A C D	:E is deleted
Pop	A C	:D is deleted
Pop	A	:C is deleted
Pop	empty	:A is deleted

The above example shows the working of stacks for pushing an element into stack and popping an element from stack.

Here in stack, a variable top is used to represent the top of the stack. Initially top=-1(NULL). For every insertion, i.e. push operation the top is increased by one and element is stored in stack and for every deletion, i.e. pop operation the top element is deleted and decremented by one.

Implementation of Stacks:

Stacks can be implemented using either arrays or linked lists.

Representation of Stacks using arrays:

The stacks can be represented using array .Stack contains an ordered collection of elements.an array is used to store ordered list of elements .hence it would be very easy to manage a stack if we represent it using array..

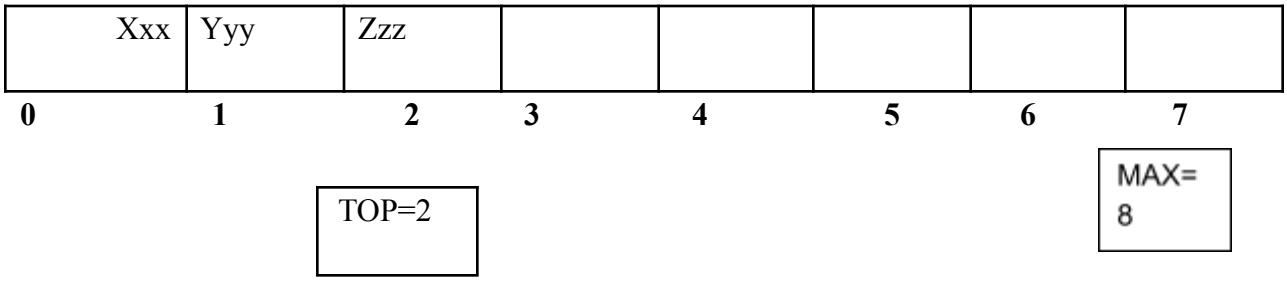
Every stack has a variable called TOP associated with it, which is used to store the address of the topmost element of the stack. It is this position where the element will be added to or deleted from.

There is another variable called MAX, which is used to store the maximum number of elements that the stack can hold.

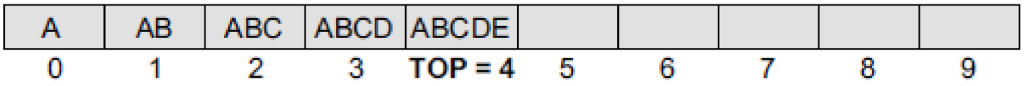
The drawback of this representation is we are required to declare the size of the array before using it in a program, means the size of the array is fixed

The variable top indicates the current top of the stack and it moves up and down depending on whether new items are added to the stack (or) existing item is removed.

Example1:



Example2:



OPERATIONS ON A STACK

A stack supports three basic operations: push, pop, and peek.

Push Operation

The push operation is used to insert an element into the stack. The new element is added at the topmost position of the stack. However, before inserting the value, we must first check if

TOP=MAX–1,because if that is the case, then the stack is full and no more insertions can be done. If an attempt is made to insert a value in a stack that is already full, an OVERFLOW message is printed. Consider the stack given in Fig. 7.5.

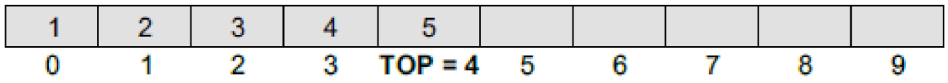


Figure 7.5 Stack

To insert an element with value 6, we first check if TOP=MAX–1. If the condition is false, then we increment the value of TOP and store the new element at the position given by stack[TOP]. Thus,the updated stack becomes as shown in Fig. 7.6.

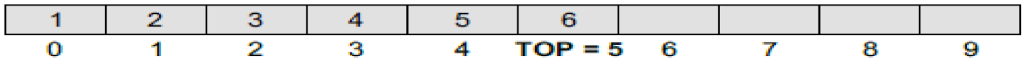


Figure 7.6 Stack after insertion

Algorithm for PUSH:

PUSH (STACK, ITEM)

This procedure inserts (pushes) an ITEM onto a stack

1. [Stack already filled?]
   if TOP= (MAX-1) then Print: OVERFLOW and Return

2 Set TOP: =TOP+1[increases TOP by 1]

3. Set STACK [TOP]:=ITEM [inserts ITEM in new TOP position]

4. Return

Pop Operation

The pop operation is used to delete the topmost element from the stack. However, before deleting the value, we must first check if TOP=NULL because if that is the case, then it means

the stack is empty and no more deletions can be done. If an attempt is made to delete a value from a stack that is already empty, an UNDERFLOW message is printed. Consider the stack given in Fig. 7.8.

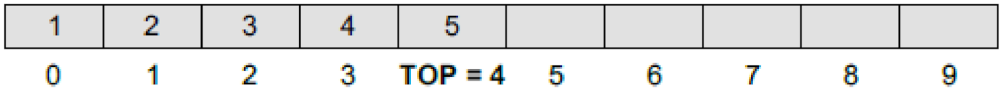


Figure 7.8 Stack

To delete the topmost element, we first check if TOP=NULL. If the condition is false, then we decrement the value pointed by TOP. Thus, the updated stack becomes as shown in Fig. 7.9.

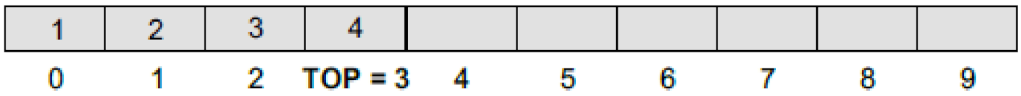


Figure 7.9 Stack after deletion

Algorithm for pop:

POP (ITEM)

This procedure deletes (pop) an element of STACK and assigns it to the variable ITEM

1. [Stack is empty?]
   if TOP=NULL then Print: UNDERFLOW, and Return
2. Set ITEM:=STACK [TOP]. [Assigns TOP element to ITEM]
3. Set TOP: =TOP-1 [decreases TOP by 1
4. Return

Peek operation:

Peek is an operation that returns the value of the topmost element of the stack without deleting it from the stack. The algorithm for Peek operation is given in Fig. 7.11.

However, the Peek operation first checks if the stack is empty,i.e., if TOP = NULL, then an appropriate message is printed, else the value is returned. Consider the stack given in Fig. 7.12.

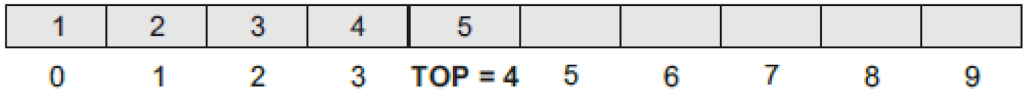


Figure 7.12 Stack

Here, the Peek operation will return 5, as it is the value of the topmost element of the stack.

Algorithm for peek:

```
1.  if top=NULL
        print "STACK IS EMPTY" and Return
2.  else
        return stack[top]

3   return.
```

Display

It is used to display all the elements from the stack

Algorithm for stack display()

```
begin
        initialize i
        if top=-1
             write "stack is empty"
       otherwise
          for i=top to 0          (or) for i=top to 0
               print stack[i]
        end of if
end of display
```

LINKED REPRESENTATION OF STACKS

We have seen how a stack is created using an array. This technique of creating a stack is easy, but the drawback is that the array must be declared to have some fixed size. But if the array size cannot be determined in advance, then the other alternative, i.e., linked representation, is used.

In a linked stack, every node has two parts—one that stores data and another that stores the address of the next node. The START pointer of the linked list is used as TOP. All insertions and deletions are done at the node pointed by TOP. If TOP = NULL, then it indicates that the stack is empty. The linked representation of a stack is shown in Fig. 7.13.



Figure 7.13 Linked stack

OPERATIONS ON A LINKED STACK

A linked stack supports all the three stack operations, that is, push, pop, and peek.

Push Operation

The push operation is used to insert an element into the stack. The new element is added at the topmost position of the stack. Consider the linked stack shown in Fig. 7.14.

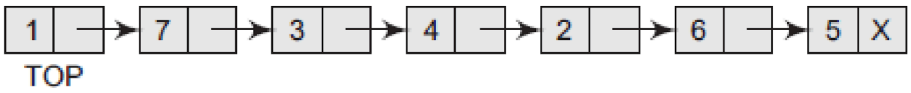


Figure 7.14 Linked stack

To insert an element with value 9, we first check if TOP=NULL. If this is the case, then we allocate memory for a new node, store the value in its DATA part and NULL in its NEXT part. The new node will then be called TOP. However, if TOP!=NULL, then we insert the new node at the beginning of the linked stack and name this new node as TOP. Thus, the updated stack becomes as shown in Fig. 7.15.



Figure 7.15 Linked stack after inserting a new node

Figure 7.16 shows the algorithm to push an element into a linked stack. In Step 1, memory is allocated for the new node. In Step 2, the DATA part of the new node is initialized with the value to be stored in the node. In Step 3, we check if the new node is the first node of the linked list. This is done by checking if TOP = NULL. In case the IF statement evaluates to true, then NULL is stored in the NEXT part of the node and the new node is called TOP. However, if the new node is not the first node in the list, then it is added before the first node of the list (that is, the TOP node) and termed as TOP.

```
Step 1: Allocate memory for the new
        node and name it as NEW_NODE
Step 2: SET NEW_NODE -> DATA = VAL
Step 3: IF TOP = NULL
            SET NEW_NODE -> NEXT = NULL
            SET TOP = NEW_NODE
        ELSE
            SET NEW_NODE -> NEXT = TOP
            SET TOP = NEW_NODE
        [END OF IF]
Step 4: END
```

Figure 7.16 Algorithm to insert an element in a linked stack

Pop Operation

The pop operation is used to delete the topmost element from a stack. However, before deleting the value, we must first check if TOP=NULL, because if this is the case, then it means that the stack is empty and no more deletions can be done. If an attempt is made to delete a value from a stack that is already empty, an UNDERFLOW message is printed. Consider the stack shown in Fig. 7.17.



Figure 7.17 Linked stack

In case TOP!=NULL, then we will delete the node pointed by TOP, and make TOP point to the second element of the linked stack. Thus, the updated stack becomes as shown in Fig. 7.18.

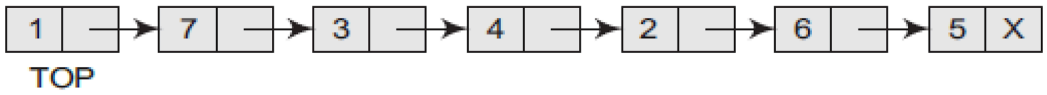


Figure 7.19 shows the algorithm to delete an element from a stack. In Step 1, we first check for the UNDERFLOW condition. In Step 2, we use a pointer PTR that points to TOP. In Step 3, TOP is made to point to the next node in sequence. In Step 4, the memory occupied by PTR is given back to the free pool.

```
Step 1: IF TOP = NULL
            PRINT "UNDERFLOW"
            Goto Step 5
        [END OF IF]
Step 2: SET PTR = TOP
Step 3: SET TOP = TOP -> NEXT
Step 4: FREE PTR
Step 5: END
```

Figure 7.19 Algorithm to delete an element from a linked stack

Applications of stack:

1. Reversing a list

2. Recursion

3. Evaluation of postfix expression

4. Conversion of infix expression to postfix expression

5. Conversion of infix expression to prefix expression

Reversing the list:

The stack usually works by inserting a new element into top of the stack and deleting the element from the top of the stack.

The elements of stack are displayed from top to bottom of stack, i.e. which displays the element reverse order based on insertion.

To understand this, imagine a pile of books usually, we pick up the topmost book from the pile. The elements in the list are displayed in reverse order.

Algorithm for displaying the elements using arrays:

```
display ()
begin
        Initialize i
         if top=-1
                write stack is empty
         otherwise
                for i=top to 0
                        print stack[i]
         end of if
end of display
```



Algorithm for Transforming Infix to Postfix Expression:

REVERSEPOLISH (Q, P)

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

1. Push "(" onto STACK and add ")" to the end of Q (Infix Expression).

2. Scan Q from left to right and repeat Steps 3 to6 for each element of Q until the STACK is empty

3. If an operand is encountered, add it to P (Postfix Expression).

4. If left parenthesis **"("** is encountered push it onto STACK

5. If an operator is encountered, than:

    (a) Repeatedly pop from STACK and add to P each operator (on the top of STACK) which has the same precedence as or higher precedence than operator

    (b) Add operator to STACK

    [End of If structure]

6. If right parenthesis is encountered, then

    (a) Repeatedly pop from STACK and add to P each operator (on the top of STACK) until a left parenthesis is encountered.

    (b) Remove the left parenthesis [do not add left parenthesis to P]

    [End of If structure]

[End of Step 2 loop]

7. Exit.

EXAMPLE:

Convert the following infix expression to postfix expression

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

First we push "(" onto STACK, and then we add ")" to the end of Q to obtain:

A	*	B	/	(	C	-	D	)	+	E	*	(	F	-	G	)	)
(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	(11)	(12)	(13)	(14)	(15)	(16)	(17)	(18)

Input (Symbol Scanned)	stack operations	postfix notation
(1) A	(	A
(2) *	(*	A
(3) B	(*	AB
(4) /	(/	AB*
(5) (	(/ (	AB*
(6) C	(/ (	AB*C
(7) -	(/ (-	AB*C

(8)	D	(/ (-	AB*CD
(9)	)	(/	AB*CD-
(10)	+	(+	AB*CD-/
(11)	E	(+	AB*CD-/E
(12)	*	(+ *	AB*CD-/E
(13)	(	(+ * (	AB*CD-/EF
(14)	F	(+ * (	AB*CD-/EF
(15)	-	(+ * (-	AB*CD-/EF
(16)	G	(+ * (-	AB*CD-/EFG
(17)	)	(+ *	AB*CD-/EFG-
(18)	)	EMPTY	AB*CD-/EFG-*+

The postfix for infix expression A*B/(C-D) +E*(F-G) is AB*CD-/EFG-*+

Conversion of an Infix Expression into a Prefix Expression:-

Algorithm to convert an infix expression into prefix expression

```
Step 1: Reverse the infix string. Note that
        while reversing the string you must
        interchange left and right parentheses.
Step 2: Obtain the postfix expression of the
        infix expression obtained in Step 1.
Step 3: Reverse the postfix expression to get
        the prefix expression
```

Figure 7.25 Algorithm to convert an infix expression into prefix expression

Example

For example, given an infix expression (A – B/C) * (A / K – L)

Step 1: Reverse the infix string. Note that while reversing the string you must interchange left and right parentheses.

(L – K / A) * (C / B – A)

Step 2: Obtain the corresponding postfix expression of the infix expression obtained as

a result of Step 1.

First we push "("onto STACK, and then we add ")" to the end of Q to obtain:

(L – K / A) * (C / B – A))

Input(symbol scanned)	**Stack operations**	**notation**
(1) **(**	**((**	
(2) **L**	**((**	**L**
(3) **-**	**((-**	**L**
(4) **K**	**((-**	**LK**

(5) /	**((- /**	**LK**
(6) **A**	**((- /**	**LKA**
(7) **)**	**(**	**LKA/-**
(8) *****	**(***	**LKA/-**
(9) **(**	**(* (**	**LKA/-**
(10) **C**	**(* (**	**LKA/-C**
(11) /	**(* (/**	**LKA/-C**
(12) **B**	**(* (/**	**LKA/-CB**
(13) **-**	**(* (-**	**LKA/-CB/**
(14) **A**	**(* (-**	**LKA/-CB/A**
(15) **)**	**(***	**LKA/-CB/A-**
(16) **)**		**LKA/-CB/A-***

Therefore the expression we get after step2 is L K A / – C B / A – *

Step 3: Reverse the postfix expression to get the prefix expression

Therefore, the prefix expression is * – A / B C – /A K L

Evaluation of arithmetic expression:-

The stacks are used in evaluation of arithmetic expression. The stacks will evaluate the

infix expression by converting into postfix expression. Then it evaluates the postfix expression.

Evaluation of a Postfix Expression

POSTFIXEVAL (P)

This algorithm finds the VALUE of an arithmetic expression P written in postfix notation.

1. Add a right parenthesis ")" at the end of P [this acts as a sentinel]

2 Scan P from left to right and repeat steps 3 and 4 for each element of P until the sentinel ")" is encountered

3. If an operand is encountered put it on STACK

4. If an operator is encountered Then:

   (a) Remove the top two elements of a STACK, where A is the top element and B is the next-to-top element

   (b) Evaluate B operator A

   (c) Place the result of (b) back on STACK.

   [End of If structure]

   [End of Step 2 loop]

5. Set VALUE equal to the top element on stack

6. Exit

Example 1: Evaluate the following postfix expression

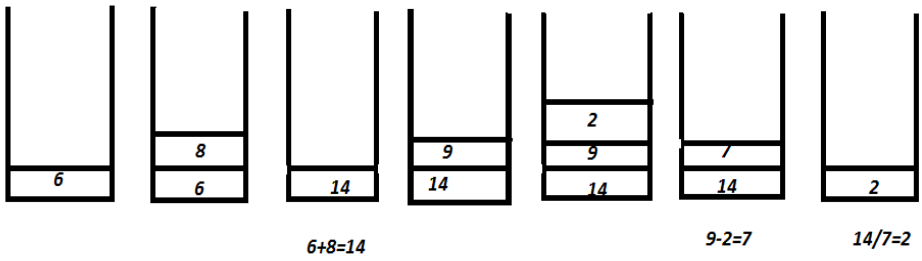
6 , 8 , + , 9, 2, -, /

-> add ")" to the end of expression

P: 6, 8, +, 9, 2, -, / ,)

Symbol Scanned	STACK
(1) 6	6
(2) 8	6,8
(3) +	14
(4) 9	14,9
(5) 2	14,9,2
(6) -	14,7
(7) /	2
(8))	

VALUE=2.



for the above example the result is 2.

EXAMPLE 2:

Evaluate the following postfix expression

5, 6, 2, +, *, 12, 4, /, -

add ")" to the end of expression

P: 5, 6, 2, +, *, 12, 4, /, - ,)

Symbol Scanned	STACK
(1) 5	5
(2) 6	5,6
(3) 2	5,6,2
(4) +	5,8
(5) *	40
(6) 12	40,12
(7) 4	40,12,4
(8) /	40,3
(9) -	37

(10))	

Result= 37

Queues:-

A Queue is a linear list of elements in which deletions can take place only at one end, called the front, and insertions can take place at the other end, called the rear.

Queue is a linear data structure that is accessed in the order of First in First out (FIFO), i.e. the 1st item inserted in a queue is also the first one to be accessed, and the last one to be inserted is also the last one to be accessed.

A queue which is similar to the way we queue up at train reservation counters (or) at banks.

Here all insertions are made at one end of the last known as rear (or) tail of the queue and all deletions are made at the other end of queues known as front (or) head of the queue.

Insertion operation is also referred as enqueuing a queue and deletion operation is known as dequeuing of the queue.

Operation	**contents of the queue**	**queue operation**
Insert a	a	a is inserted into the queue
Insert b	ab	b is inserted into the queue

Insert c	abc	c is inserted into the queue
Delete	bc	a is deleted from queue
Insert d	bcd	d is inserted into queue
Delete	cd	b is deleted from queue
Delete	d	c is deleted from queue
Insert e	de	e is inserted into the queue

In the above example, the insert operation adds a new item to queue and the delete operation removes an item from stack .The item which is inserted into queue at the first is also the first item to be removed from the queue.

 Implementation of queue using arrays:-

Queue is a linear data structure it can be implemented using arrays.

The variables **front** and **rear** keeps track of the front and rear ends in a queue to facilitate execution of insertion and deletion operations respectively.

Here front and rear are initialized to -1(NULL) and for every insert operation rear variable is increased by one and front variable is increased by one for every deletion operation from queue.

Algorithm for insert operation:-

```
INSERT (ITEM)          This algorithm inserts an item on to queue

BEGIN
1.   IF (REAR=MAX-1) THEN      /*WHERE MAX IS QUEUE SIZE
          Print OVERFLOW and GOTO STEP 4
     [END OF IF]
2.   IF FRONT= -1 AND REAR= -1
        SET    FRONT=0, REAR=0
     ELSE
        SET REAR=REAR+1;
     [ END OF IF]
3    SET QUEUE[REAR]=ITEM
4.   EXIT
```

Here in this algorithm the condition is checked whether the queue is full or not. If possible the element is added or inserted into queue by increasing the rear variable.

Algorithm for deleting an element from queue:-

```
1.  IF FRONT= -1 OR FRONT > REAR
        Write UNDERFLOW
    ELSE
        SET     ITEM=QUEUE [FRONT]
        SET     FRONT=FRONT+1
    [END OF IF]
2   EXIT
```

EXAMPLE:-

The following example illustrates the working of queue

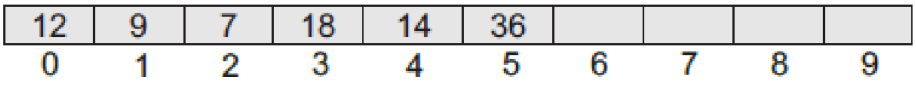


Figure 8.1 Queue

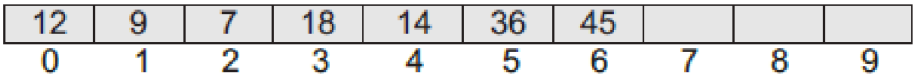


Figure 8.2 Queue after insertion of a new element

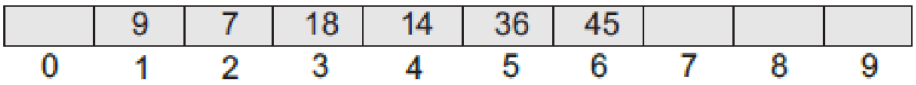


Figure 8.3 Queue after deletion of an element

Limitations of linear queue:-

10	25	15	45		
0	1	2	3	4	5

a) delete two elements

		15	45		

b) insert two elements 55,65

		15	45	55	65

c) insert 75

		15	45	55	65

Not possible because of overflow condition(rear=5).

In the above example the insertion operation was unsuccessful for inserting the element 75 into queue even the memory (0 and 1 location) is available, since the rear variable pointing "MAXIMIUM-1" position of queue. This is a major limitation when a queue full condition is involved it does not necessary to imply that the queue is 'physically full'. This leads to the limitation of rejecting insertions despite the space available. This limitation can be eliminated using circular queues.

Implementation of queue using linked lists:-

We have seen how a queue is created using an array. Although this technique of creating a queue is easy, its drawback is that the array must be declared to have some fixed size.

If we allocate space for 50 elements in the queue and it hardly uses 20–25 locations, then half of the space will be wasted.

Operations on Linked Queues:

A queue has two basic operations: insert and delete. The insert operation adds an element to the end of the queue, and the delete operation removes an element from the front or the start of the queue. Apart from this, there is another operation peek which returns the value of the first element of the queue.

Insert Operation

The insert operation is used to insert an element into a queue. The new element is added as the last element of the queue. Consider the linked queue shown in Fig. 8.7.

To insert an element with value 9, we first check if FRONT=NULL. If the condition holds, then the queue is empty. So, we allocate memory for a new node, store the value in its data part and NULL in its next part. The new node will then be called both FRONT and rear. However, if FRONT != NULL, then we will insert the new node at the rear end of the linked queue and name this new node as rear. Thus, the updated queue becomes as shown in Fig. 8.8.

Figure 8.9 shows the algorithm to insert an element in a linked queue.

In Step 1, the memory is allocated for the new node.

In Step 2, the DATA part of the new node is initialized with the value to be stored in the node.

In Step 3, we check if the new node is the first node of the linked queue. This is done by checking if FRONT = NULL. If this is the case, then the new node is tagged as FRONT as well as REAR. Also NULL is stored in the NEXT part of the node (which is

also the FRONT and the REAR node). However, if the new node is not the first node in the list, then it is added at the REAR end of the linked queue (or the last node of the queue).

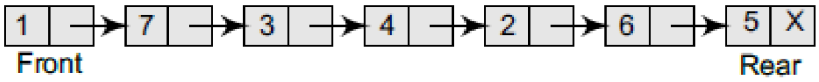


Figure 8.7 Linked queue

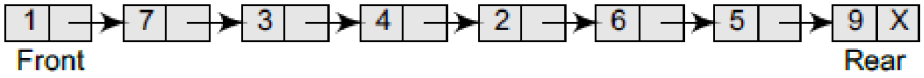


Figure 8.8 Linked queue after inserting a new node

Algorithm:

```
Step 1: Allocate memory for the new node and name
        it as PTR
Step 2: SET PTR -> DATA = VAL
Step 3: IF FRONT = NULL
            SET FRONT = REAR = PTR
            SET FRONT -> NEXT = REAR -> NEXT = NULL
        ELSE
            SET REAR -> NEXT = PTR
            SET REAR = PTR
            SET REAR -> NEXT = NULL
        [END OF IF]
Step 4: END
```

Figure 8.9 Algorithm to insert an element in a linked queue

Delete *Operation:*

The delete operation is used to delete the element that is first inserted in a queue, i.e., the element whose address is stored in FRONT. However, before deleting the value, we must first check if FRONT=NULL because if this is the case, then the queue is empty and no more deletions can be done. If an attempt is made to delete a value from a queue that is already empty, an underflow message is printed.

Consider the queue shown in Fig. 8.10.

To delete an element, we first check if FRONT=NULL. If the condition is false, then we delete the first node pointed by FRONT. The FRONT will now point to the second element of the linked queue. Thus, the updated queue becomes as shown in Fig. 8.11.

Figure 8.12 shows the algorithm to delete an element from a linked queue.

In Step 1, we first check for the underflow condition. If the condition is true, then an appropriate message is displayed, otherwise

in Step 2, we use a pointer PTR that points to FRONT.

In Step 3, FRONT is made to point to the next node in sequence. In Step 4, the memory occupied by PTR is given back to the free pool.



Figure 8.10 Linked queue

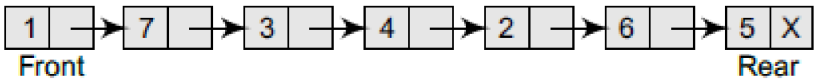
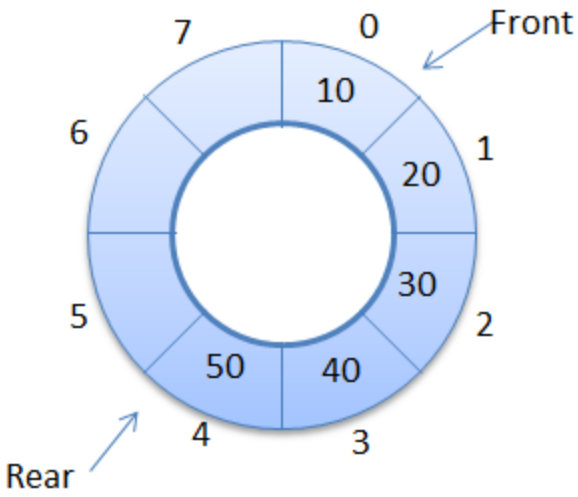


Figure 8.11 Linked queue after deletion of an element

```
Step 1: IF FRONT = NULL
            Write "Underflow"
            Go to Step 5
        [END OF IF]
Step 2: SET PTR = FRONT
Step 3: SET FRONT = FRONT -> NEXT
Step 4: FREE PTR
Step 5: END
```

Figure 8.12 Algorithm to delete an element from a linked queue

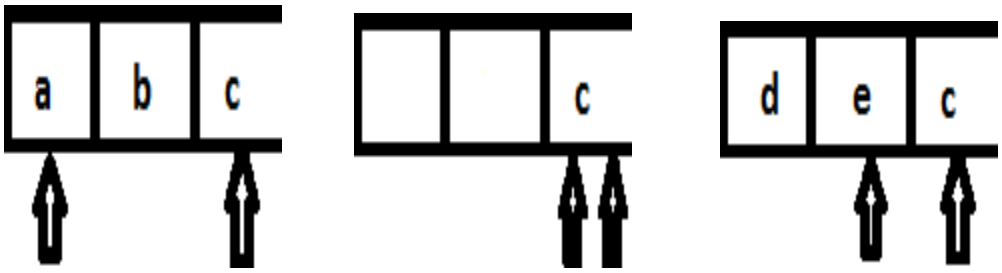
□□ **Circular Queue:** Circular Queue is a linear data structure in which the operations are performed based on FIFO (First In First Out) principle and the last position is connected back to the first position to make a circle.



In a normal Queue, we can insert elements until queue becomes full. But once queue becomes full, we can not insert the next element even if there is a space in front of queue.

Operations on a Circular Queue: Let CIRC_Q be a circular queue with a capacity of 3 elements as shown in figure. The queue is obviously full with front pointing to the element at the beginning of the queue and rear pointing to the element at the end of the queue.

a. Represents the initial circular queue with 3 elements.
b. After two deletions, front moves towards rear and points to 'c' as the current front element of CIRC_Q.
c. When'd' is inserted, unlike linear queues, rear moves back in a clockwise fashion to store'd' in the vacant space available.



(a)Initial Circular Queue.

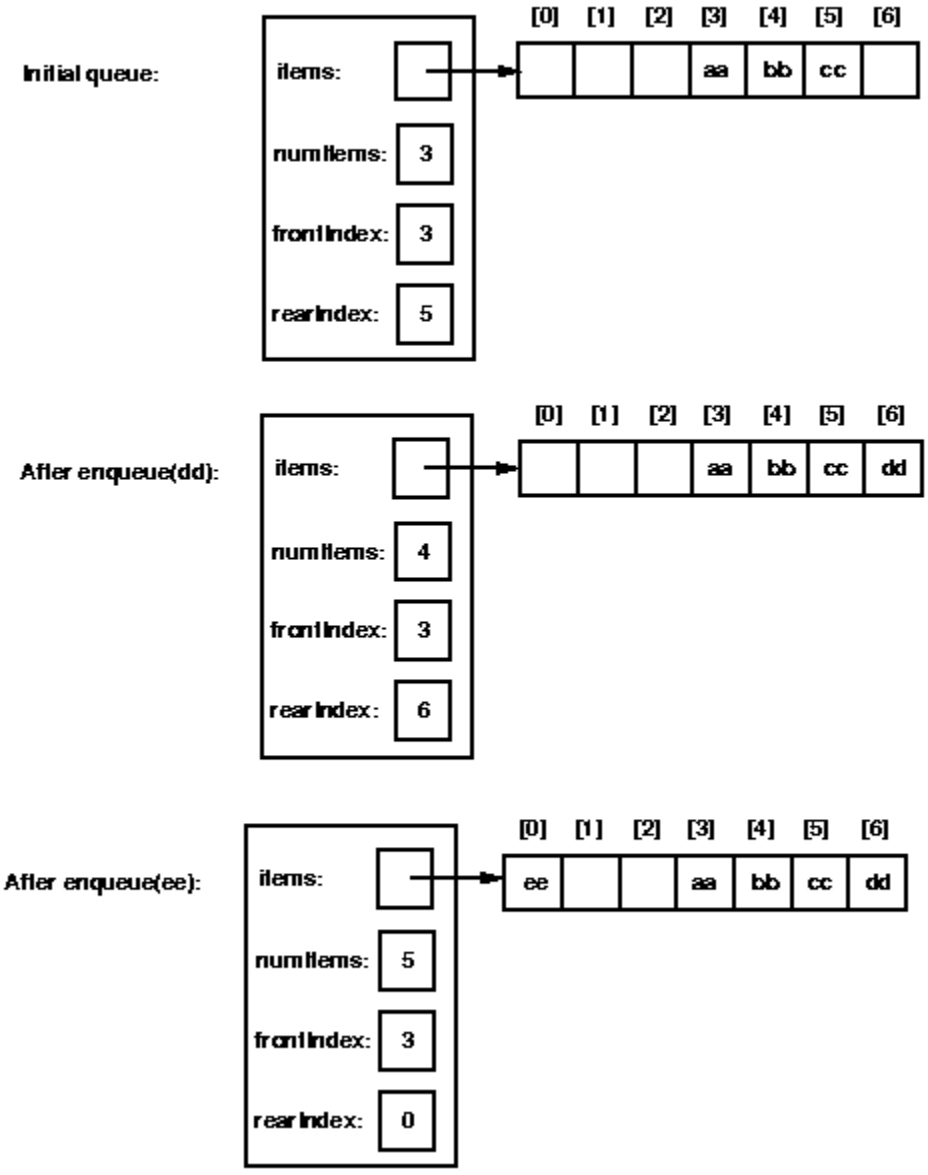
(b) Circular queue after two deletions.

(c) Circular queue after insertion of d, e.

The Circular queues are using the memory more efficiently than the linear queues.

In the above we can see that the memory is used efficiently than linear queue data structure.

EXAMPLE2:


Initial queue:
items:
numItems: 3
frontIndex: 3
rearIndex: 5
[0] [1] [2] [3] [4] [5] [6]
aa bb cc
After enqueue(dd):
items:
numItems: 4
frontIndex: 3
rearIndex: 6
[0] [1] [2] [3] [4] [5] [6]
aa bb cc dd
After enqueue(ee):
items:
numItems: 5
frontIndex: 3
rearIndex: 0
[0] [1] [2] [3] [4] [5] [6]
ee aa bb cc dd

OPERATIONS:

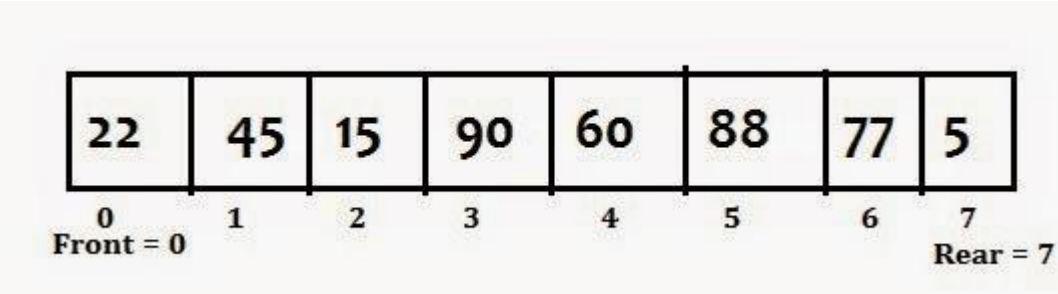
Inserting an element in a circular queue:

ALGORITHM:

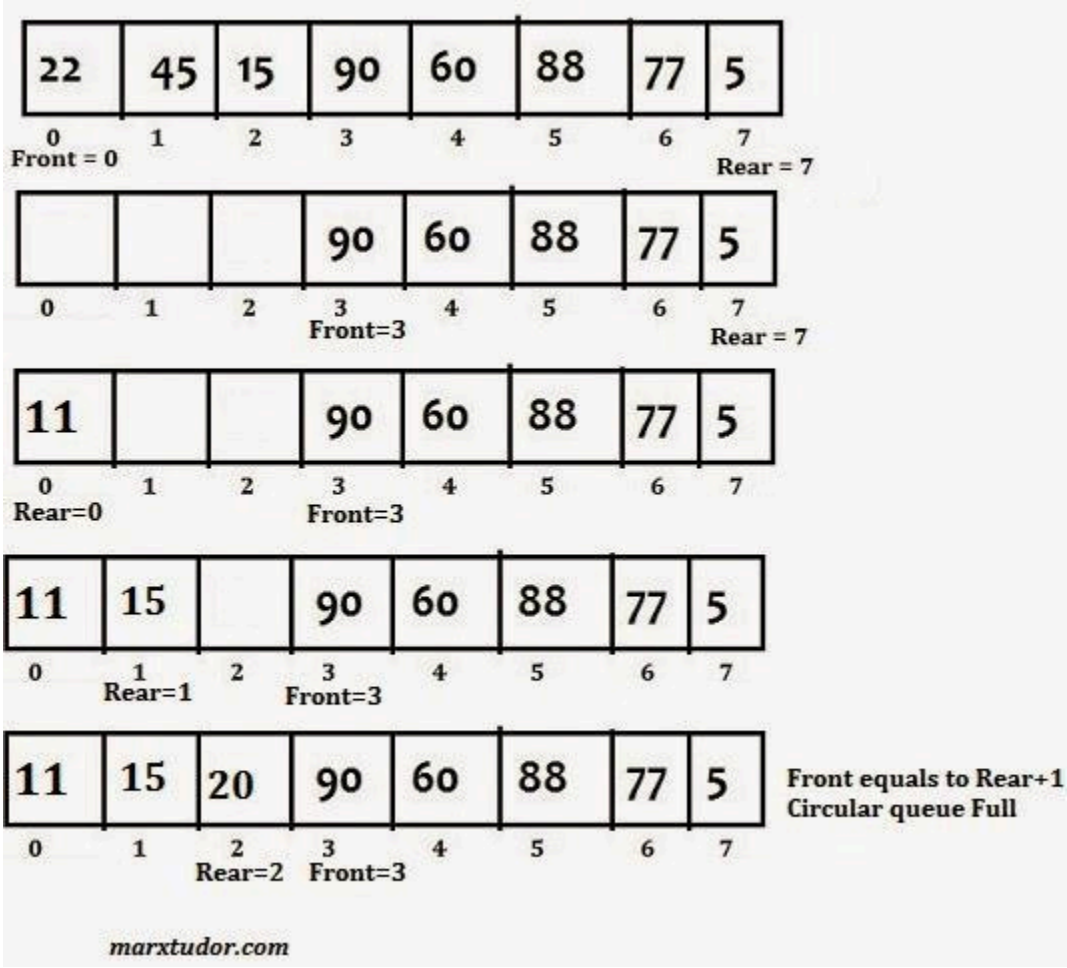
```
Step 1: IF FRONT =0 and Rear = MAX – 1 (or) front=rear+1 [Queue already filled ]
Write OVERFLOW
Step 2: IF FRONT = -1 and REAR = -1[Queue initially empty]
            SET FRONT = REAR =0
          ELSE IF REAR = MAX - 1 then
            SET REAR =0
          ELSE
                    SET REAR = REAR + 1
     [END OF IF]
Step 3: SET QUEUE[REAR] = VAL
Step 4: EXIT
```

Step 1: The circular queue will be full only when `front = 0` and `rear = Max - 1`. Or if front=rear+1;

Example:



Front=0 and rear=max-1



Step 2:a) if Queue is empty (front=-1 and rear=-1) then insert the item at 0^{th} location and set SET FRONT = REAR =0

b) if rear=max-1(rear points to the last element) then set rear=0 othewise set rear=rear+1

Step 3: store the value at rear position by using SET QUEUE[REAR] = VAL

Deleting an element in a circular queue:

Algorithm to delete an element from a circular queue

To delete an element, again we check for three conditions.

Look at Fig. 8.20. If front = –1, then there are no elements in the queue. So, an underflow condition will be reported.

If the queue is not empty and front = rear(Queue contains only one element), then after deleting the element at the front the queue becomes empty and so front and rear are set to –1. This is illustrated in Fig. 8.21.

If the queue is not empty and front = MAX–1,then after deleting the element at the front, front is set to 0. This is shown in Fig. 8.22.

Let us look at Fig. 8.23 which shows the algorithm to delete an element from a circular queue.

In Step 1, we check for the underflow condition.

In Step 2, the value of the queue at the location pointed by FRONT is stored in VAL.

In Step 3,we make two checks. First to see if the queue has become empty after deletion and second to see if FRONT has reached the maximum capacity of the queue. The value of FRONT is then updated based on the outcome of these checks.

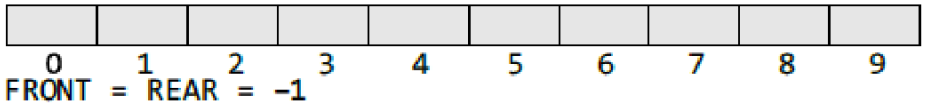


Figure 8.20 Empty queue

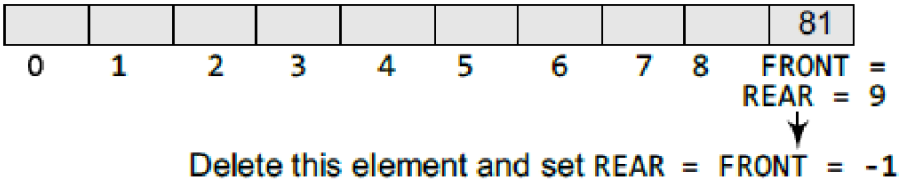


Figure 8.21 Queue with a single element

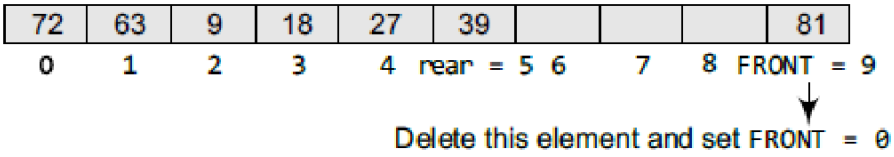


Figure 8.22 Queue where FRONT = MAX–1 before deletion

```
Step 1: IF FRONT = -1
            Write "UNDERFLOW"
            Goto Step 4
        [END of IF]
Step 2: SET VAL = QUEUE[FRONT]
Step 3: IF FRONT = REAR
            SET FRONT = REAR = -1
         ELSE
            IF FRONT = MAX -1
                  SET FRONT = 0
            ELSE
                  SET FRONT = FRONT + 1
            [END of IF]
        [END OF IF]
Step 4: EXIT
```

Figure 8.23 Algorithm to delete an element from a circular queue

->APPLICATIONS OF QUEUES

- Queues are widely used as waiting lists for a single shared resource like printer, disk, CPU.
- Queues are used to transfer data asynchronously (data not necessarily received at same rate as sent) between two processes (IO buffers), e.g., pipes, file IO, sockets.
- Queues are used as buffers on MP3 players and portable CD players, iPod playlist.
- Queues are used in Playlist for jukebox to add songs to the end, play from the front of the list.
- Queues are used in operating system for handling interrupts.

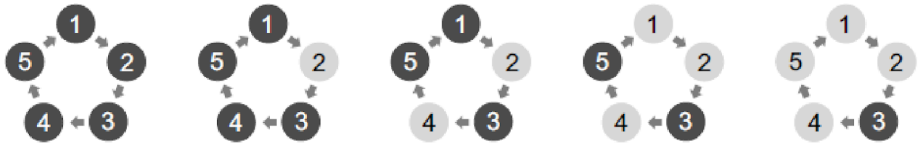
->Josephus Problem

Let us see how queues can be used for finding a solution to the Josephus problem.

In Josephus problem, n people stand in a circle waiting to be executed. The counting starts at some point in the circle and proceeds in a specific direction around the circle. In each step, a certain number of people are skipped and the next person is executed (or eliminated). The elimination of people makes the circle smaller and smaller. At the last step, only one person remains who is declared the 'winner'.

Therefore, if there are n number of people and a number k which indicates that k–1 people are skipped and kth person in the circle is eliminated, then the problem is to choose a position in the initial circle so that the given person becomes the winner

For example, if there are 5 (n) people and every second (k) person is eliminated, then first the person at position 2 is eliminated followed by the person at position 4 followed by person at position 1 and finally the person at position 5 is eliminated. Therefore, the person at position 3 becomes the winner.



Try the same process with $n = 7$ and $k = 3$. You will find that person at position 4 is the winner. The elimination goes in the sequence of 3, 6, 2, 7, 5 and 1.

□□ **Round Robin Algorithm: -** The round robin is a simple scheduling algorithm in which the queue data structure is used. A small unit of time called time quantum is defined. All the runnable processors are kept in queue. The CPU scheduler goes around this queue and allocates the CPU for each process one by one for that time interval. New processes are added at the rear end of the queue.

The Scheduler chooses the 1st process from the queue and assigns the CPU to it. If the process does not finish its job, then that process will be added at the rear end, if the process finishes its job in given time interval then it is deleted from the queue. The CPU scheduler assigns the CPU to the next process in the queue.

EXAMPLE

Time quantum=5

Process	Burst Time
P1	12
P2	8
P3	4
P4	10
P5	5

Round Robin scheduling algorithm
Gantt chart

P1	P2	P3	P4	P5	P1	P2	P4	P1

0 5 10 14 19 24 29 32 37 39