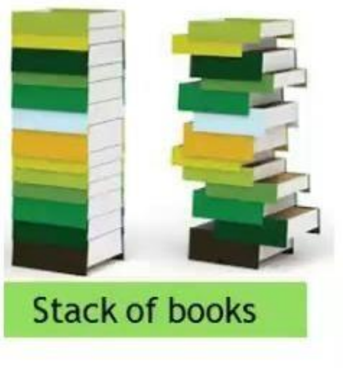
UNIT-II
STACKS AND QUEUES

STACKS

A Stack is linear data structure. A stack is a list of elements in which an element may be inserted or deleted only at one end, called the **top of the stack**. Stack principle is **LIFO (last in, first out)**. Which element inserted last on to the stack that element deleted first from the stack.

As the items can be added or removed only from the top i.e. the last item to be added to a stack is the first item to be removed.

Real life examples of stacks are:



Operations on stack:

The two basic operations associated with stacks are:
1. Push
2. Pop

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

a) Stack is empty or not b) stack is full or not

1. 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".

2. 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".

All insertions and deletions take place at the same end, so the last element added to the stack will be the first element removed from the stack. When a stack is created, the stack base remains fixed while the stack top changes as elements are added and removed. The most accessible element is the top and the least accessible element is the bottom of the stack.

Representation of Stack (or) Implementation of stack:

The stack should be represented in two ways:

1. Stack using array
2. Stack using linked list

1. Stack using array:

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.

1.push(): When an element is added to a stack, the operation is performed by push(). Below Figure shows the creation of a stack and addition of elements using push().

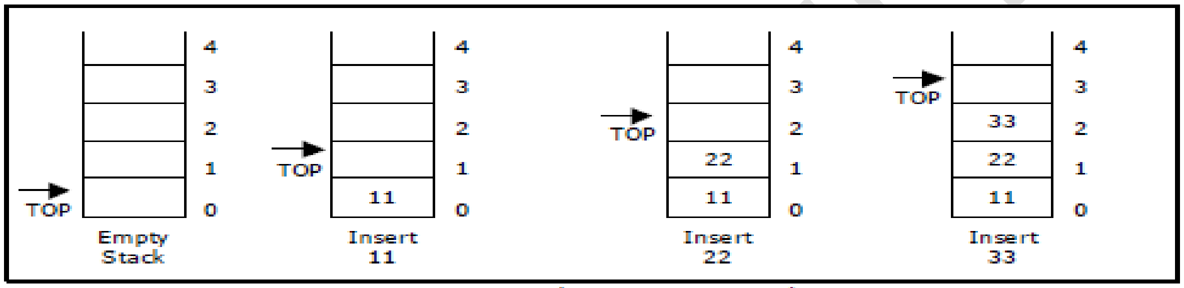


Figure . Push operations on stack

Initially **top=-1**, we can insert an element in to the stack, increment the top value i.e **top=top+1**. We can insert an element in to the stack first check the condition is stack is full or not. i.e **top>=size-1**. Otherwise add the element in to the stack.

void push()
{
int x;
if(top >= n-1)
{
printf("\n\nStack Overflow..");
return;
}
else
{
printf("\n\nEnter data: ");
scanf("%d", &x);
stack[top] = x;
top = top + 1;
printf("\n\nData Pushed into the stack");
}
}
	Algorithm: Procedure for push():
	
	Step 1: START
	Step 2: if top>=size-1 then
	Write " Stack is Overflow"
	Step 3: Otherwise
	3.1: read data value 'x'
	3.2: top=top+1;
	3.3: stack[top]=x;
	Step 4: END

2.Pop(): When an element is taken off from the stack, the operation is performed by pop(). Below figure shows a stack initially with three elements and shows the deletion of elements using pop().

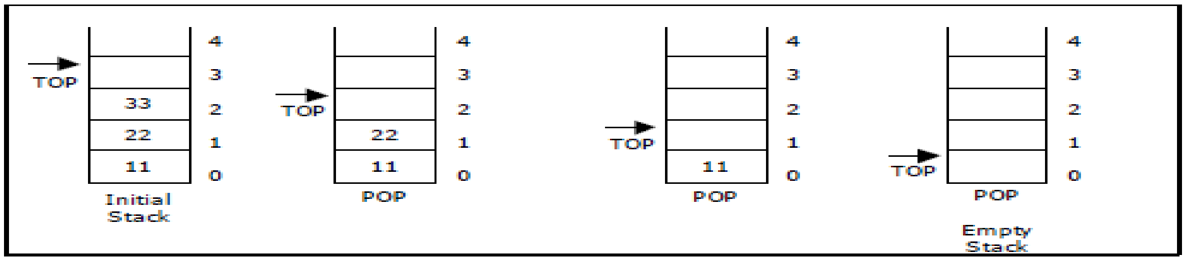
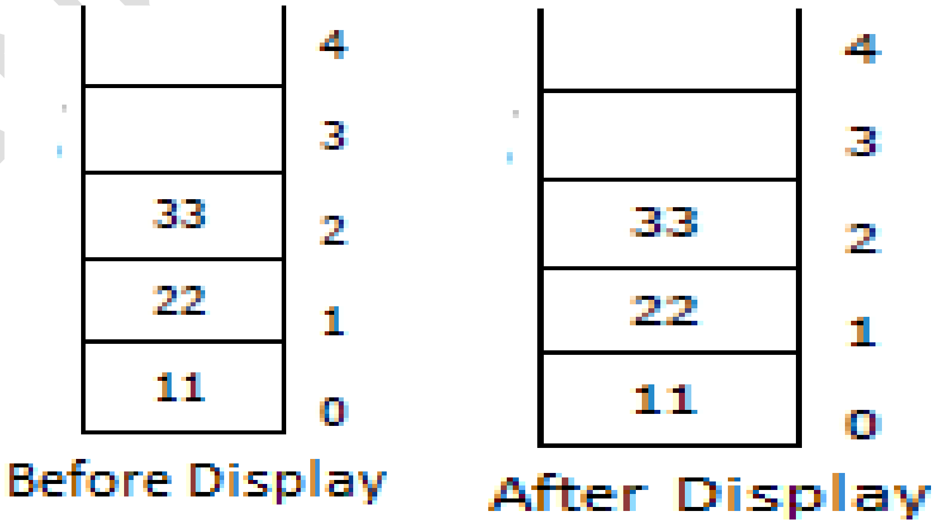


Figure Pop operations on stack

We can insert an element from the stack, decrement the top value i.e **top=top-1**.
We can delete an element from the stack first check the condition is stack is empty or not. i.e **top==-1**. Otherwise remove the element from the stack.

Void pop()
{
    If(top==-1)
    {
        Printf("Stack is Underflow");
    }
    else
    {
        printf("Delete data %d",stack[top]);
        top=top-1;
    }
}
	Algorithm: procedure pop():
	Step 1: START
	Step 2: if top==-1 then
	        Write "Stack is Underflow"
	Step 3: otherwise
	        3.1: print "deleted element"
	        3.2: top=top-1;
	Step 4: END

3.display(): This operation performed display the elements in the stack. We display the element in the stack check the condition is stack is empty or not i.e top==-1.Otherwise display the list of elements in the stack.



void display()
{
    If(top==-1)
    {
        Printf(“Stack is Underflow”);
    }
    else
    {
        printf(“Display elements are:);
        for(i=top;i>=0;i--)
            printf(“%d”,stack[i]);
    }
}
	Algorithm: procedure pop():
	Step 1: START
	Step 2: if top==-1 then
	        Write “Stack is Underflow”
	Step 3: otherwise
	        3.1: print “Display elements are”
	        3.2: for top to 0
	            Print ‘stack[i]’
	Step 4: END

Source code for stack operations, using array:

```
#include<stdio.h>
#inlcude<conio.h>
int stack[100],choice,n,top,x,i;
void push(void);
void pop(void);
void display(void);
int main()
{
   //clrscr();
   top=-1;
   printf("\n Enter the size of STACK[MAX=100]:");
   scanf("%d",&n);
   printf("\n\t STACK OPERATIONS USING ARRAY");
   printf("\n\t..................................");
   printf("\n\t 1.PUSH\n\t 2.POP\n\t 3.DISPLAY\n\t 4.EXIT");
   do
   {
      printf("\n Enter the Choice:");
      scanf("%d",&choice);
      switch(choice)
      {
         case 1:
         {
            push();
            break;
         }
         case 2:
         {
            pop();
            break;
         }
         case 3:
         {
            display();
            break;
          }
          case 4:
          {
            printf("\n\t EXIT POINT ");
            break;
          }
          default:
          {
            printf ("\n\t Please Enter a Valid Choice(1/2/3/4)");
          }

      }
    }
    while(choice!=4);
    return 0;
}
void push()
{
    if(top>=n-1)
    {
        printf("\n\tSTACK is over flow");

    }
    else
    {
        printf(" Enter a value to be pushed:");
        scanf("%d",&x);
        top++;
        stack[top]=x;
    }
}
void pop()
{
    if(top<=-1)
    {
        printf("\n\t Stack is under flow");
    }
    else
    {
        printf("\n\t The popped elements is %d",stack[top]);
        top--;
    }
}
void display()
{
    if(top>=0)
    {
        printf("\n The elements in STACK \n");
        for(i=top; i>=0; i--)
            printf("\n%d",stack[i]);
        printf("\n Press Next Choice");
    }
    else
    {
        printf("\n The STACK is empty");
    }

}
```

2. Stack using Linked List:

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. The linked stack looks as shown in figure.

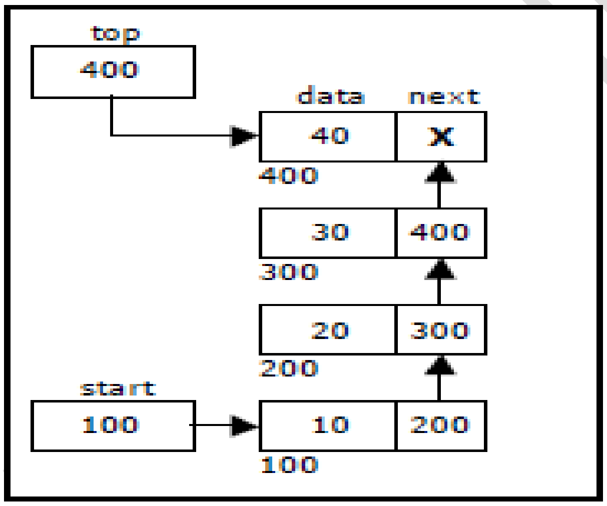


Figure Linked stack representation

Applications of stack:

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.

Converting and evaluating Algebraic expressions:

An **algebraic expression** is a legal combination of operators and operands. Operand is the quantity on which a mathematical operation is performed. Operand may be a variable like x, y, z or a constant like 5, 4, 6 etc. Operator is a symbol which signifies a mathematical or logical operation between the operands. Examples of familiar operators include +, -, *, /, ^ etc.

An algebraic expression can be represented using three different notations. They are infix, postfix and prefix notations:

Infix: It is the form of an arithmetic expression in which we fix (place) the arithmetic operator in between the two operands.

Example: A + B

Prefix: It is the form of an arithmetic notation in which we fix (place) the arithmetic operator before (pre) its two operands. The prefix notation is called as polish notation.

Example: + A B

Postfix: It is the form of an arithmetic expression in which we fix (place) the arithmetic operator after (post) its two operands. The postfix notation is called as *suffix notation* and is also referred to *reverse polish notation*.

Example: A B +

Conversion from infix to postfix:

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 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.

The three important features of postfix expression are:

1. The operands maintain the same order as in the equivalent infix expression.
2. The parentheses are not needed to designate the expression unambiguously.
3. While evaluating the postfix expression the priority of the operators is no longer relevant.

We consider five binary operations: +, -, *, / and $ or ↑ (exponentiation). For these binary operations, the following in the order of precedence (highest to lowest):

OPERATOR	PRECEDENCE	VALUE
Exponentiation ($ or ↑ or ^)	Highest	3
*, /	Next highest	2
+, -	Lowest	1

Example 1:

Convert ((A – (B + C)) * D) ↑ (E + F) infix expression to postfix form:

SYMBOL	POSTFIX STRING	STACK	REMARKS
(		(	
(		((	
A	A	((	
-	A	((-	
(	A	((- (	
B	A B	((- (	
+	A B	((- (+	
C	A B C	((- (+	
)	A B C +	((-	
)	A B C + -	(	
*	A B C + -	(*	
D	A B C + - D	(*	
)	A B C + - D *		
↑	A B C + - D *	↑	
(	A B C + - D *	↑ (	
E	A B C + - D * E	↑ (	
+	A B C + - D * E	↑ (+	
F	A B C + - D * E F	↑ (+	
)	A B C + - D * E F +	↑	
End of string	A B C + - D * E F + ↑	The input is now empty. Pop the output symbols from the stack until it is empty.	

Example 2:

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.	

Evaluation of postfix expression:

The postfix expression is evaluated easily by the use of a stack.

1. When a number is seen, it is pushed onto the stack;
2. 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.
3. When an expression is given in postfix notation, there is no need to know any precedence rules; this is our obvious advantage.

Example 1:

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

Example 2:

Evaluate the following postfix expression: 6 2 3 + - 3 8 2 / + * 2 ↑ 3 +

SYMBOL	OPERAND 1	OPERAND 2	VALUE	STACK
6				6
2				6, 2
3				6, 2, 3
+	2	3	5	6, 5
-	6	5	1	1
3	6	5	1	1, 3
8	6	5	1	1, 3, 8
2	6	5	1	1, 3, 8, 2
/	8	2	4	1, 3, 4
+	3	4	7	1, 7
*	1	7	7	7
2	1	7	7	7, 2
↑	7	2	49	49
3	7	2	49	49, 3
+	49	3	52	52

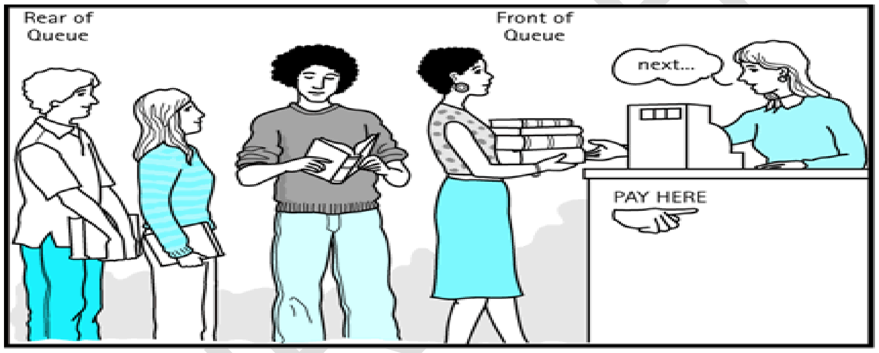
QUEUE

A queue is linear data structure and collection of elements. 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**. The principle of queue is a **"FIFO" or "First-in-first-out"**.

Queue is an abstract data structure. A queue is a useful data structure in programming. **It is similar to the ticket queue outside a cinema hall**, where the first person entering the queue is the first person who gets the ticket.

A real-world example of queue can be a single-lane one-way road, where the vehicle enters first, exits first.

More real-world examples can be seen as queues at the ticket windows and bus-stops and our college library.



The operations for a queue are analogues to those for a stack; the difference is that the insertions go at the end of the list, rather than the beginning.

Operations on QUEUE:

A queue is an object or more specifically an abstract data structure (ADT) that allows the following operations:

- **Enqueue or insertion:** which inserts an element at the end of the queue.
- **Dequeue or deletion:** which deletes an element at the start of the queue.

Queue operations work as follows:

1. Two pointers called FRONT and REAR are used to keep track of the first and last elements in the queue.
2. When initializing the queue, we set the value of FRONT and REAR to 0.
3. On enqueing an element, we increase the value of REAR index and place the new element in the position pointed to by REAR.
4. On dequeueing an element, we return the value pointed to by FRONT and increase the FRONT index.
5. Before enqueing, we check if queue is already full.
6. Before dequeuing, we check if queue is already empty.
7. When enqueing the first element, we set the value of FRONT to 1.
8. When dequeing the last element, we reset the values of FRONT and REAR to 0.

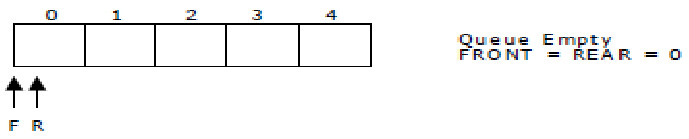
Representation of Queue (or) Implementation of Queue:

The queue can be represented in two ways:

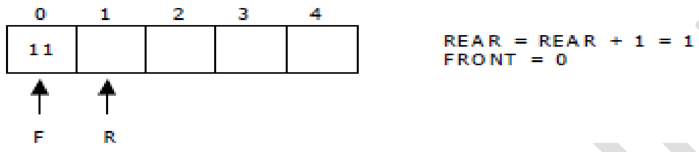
1. Queue using Array
2. Queue using Linked List

1.Queue using Array:

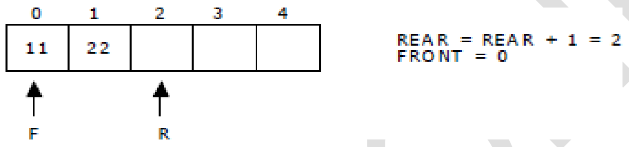
Let us consider a queue, which can hold maximum of five elements. Initially the queue is empty.



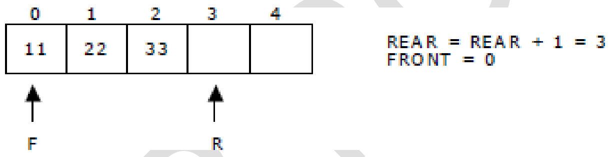
Now, insert 11 to the queue. Then queue status will be:



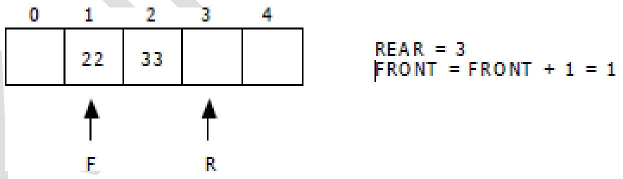
Next, insert 22 to the queue. Then the queue status is:



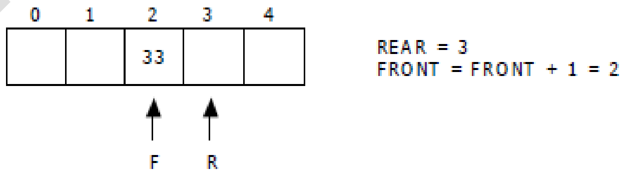
Again insert another element 33 to the queue. The status of the queue is:



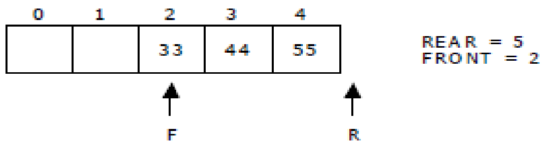
Now, delete an element. The element deleted is the element at the front of the queue.So the status of the queue is:



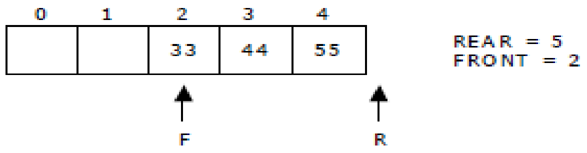
Again, delete an element. The element to be deleted is always pointed to by the FRONT pointer. So, 22 is deleted. The queue status is as follows:



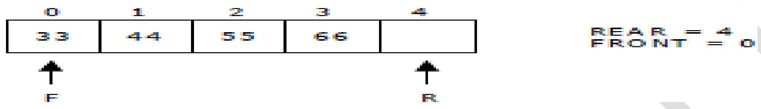
Now, insert new elements 44 and 55 into the queue. The queue status is:



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 overcome 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.**

Queue operations using array:

a.enqueue() or insertion():which inserts an element at the end of the queue.

```
void insertion()
{
   if(rear==max)
       printf("\n Queue is Full");
   else
   {
        printf("\n Enter no %d:",j++);
        scanf("%d",&queue[rear++]);
    }
}
```
	Algorithm: Procedure for insertion():
	Step-1:START
	Step-2: if rear==max then
	        Write 'Queue is full'
	Step-3: otherwise
	  3.1: read element 'queue[rear]'
	Step-4:STOP

b.dequeue() or deletion(): which deletes an element at the start of the queue.

```
void deletion()
{
    if(front==rear)
   {
     printf("\n Queue is empty");
   }
   else
    {
      printf("\n Deleted Element is
               %d",queue[front++]);
           x++;
     }}
```
	Algorithm: procedure for deletion():
	Step-1:START
	Step-2: if front==rear then
	        Write' Queue is empty'
	Step-3: otherwise
	        3.1: print deleted element
	 Step-4:STOP

c.dispaly(): which displays an elements in the queue.

```
void deletion()
{
    if(front==rear)
   {
      printf("\n Queue is empty");
   }
          else
          {
             for(i=front; i<rear; i++)
             {
                printf("%d",queue[i]);
                printf("\n");
             }
          }
}
```

Algorithm: procedure for deletion():

```
Step-1:START
Step-2: if front==rear then
          Write' Queue is empty'
Step-3: otherwise
         3.1: for i=front to rear then
         3.2: print 'queue[i]'
Step-4:STOP
```

2. Queue using Linked list:

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 alist. We use two pointers *front* and *rear* for our linked queue implementation.

The linked queue looks as shown in figure:

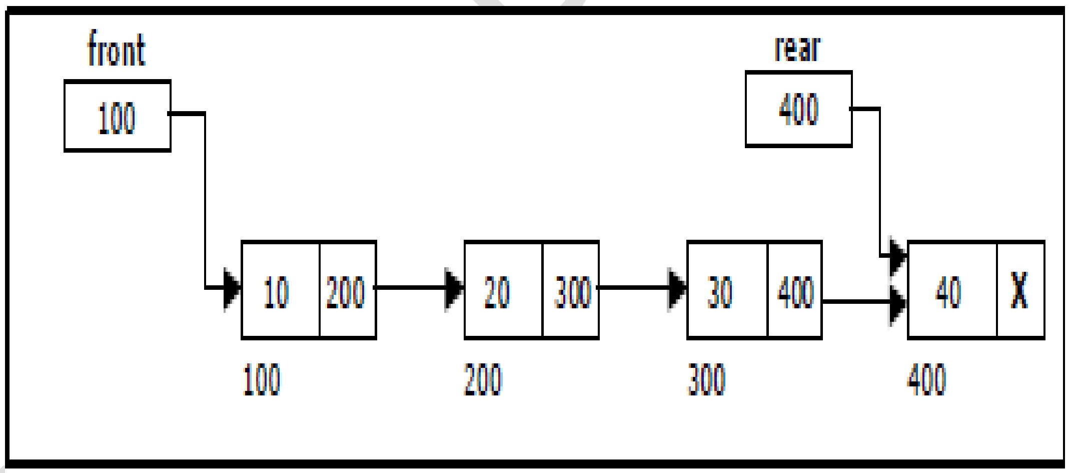


Figure : Linked Queue representation

Applications of Queue:

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.

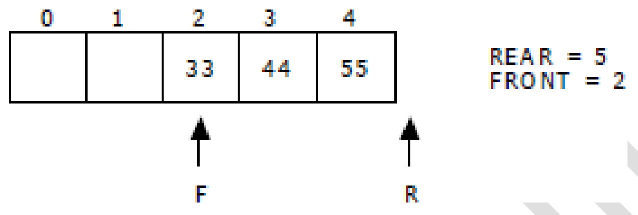
CIRCULAR QUEUE

A more efficient queue representation is obtained by regarding the array Q[MAX] as circular. Any number of items could be placed on the queue. This implementation of a queue is called a circular queue because it uses its storage array as if it were a circle instead of a linear list.

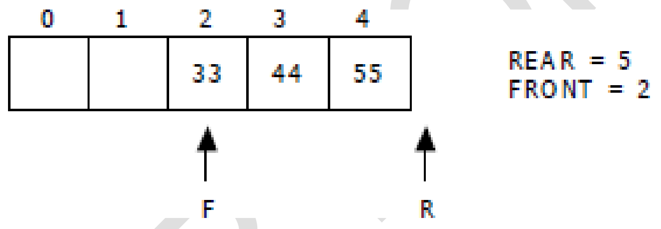
There are two problems associated with linear queue. They are:

- 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.

For example, let us consider a linear queue status as follows:



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:

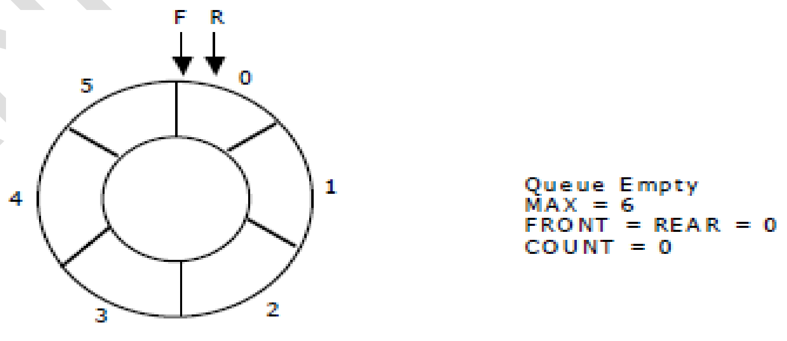


This difficulty can be overcome if we treat queue position with index zero as a position that comes after position with index four then we treat the queue as a **circular queue.**

In circular queue if we reach the end for inserting elements to it, it is possible to insert new elements if the slots at the beginning of the circular queue are empty.

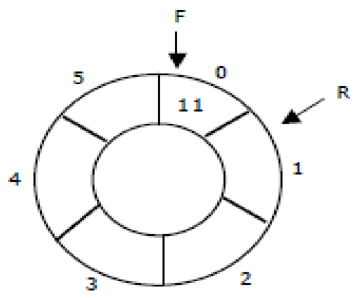
Representation of Circular Queue:

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



Circular Queue

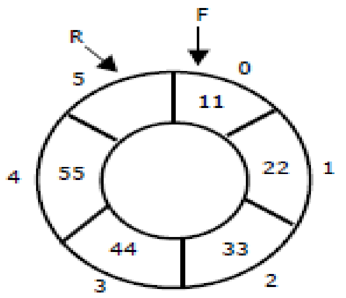
Now, insert 11 to the circular queue. Then circular queue status will be:



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

Circular Queue

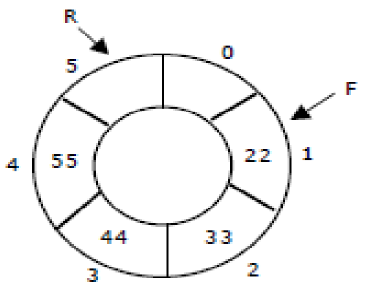
Insert new elements 22, 33, 44 and 55 into the circular queue. The circular queue status is:



```
FRONT = 0
REAR = (REAR + 1) % 6 = 5
COUNT = 5
```

Circular Queue

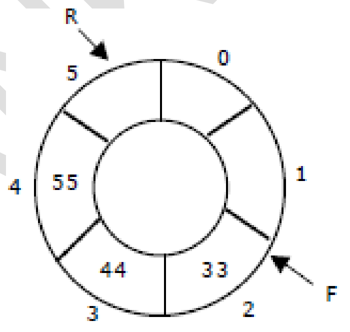
Now, delete an element. The element deleted is the element at the front of the circular queue. So, 11 is deleted. The circular queue status is as follows:



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

Circular Queue

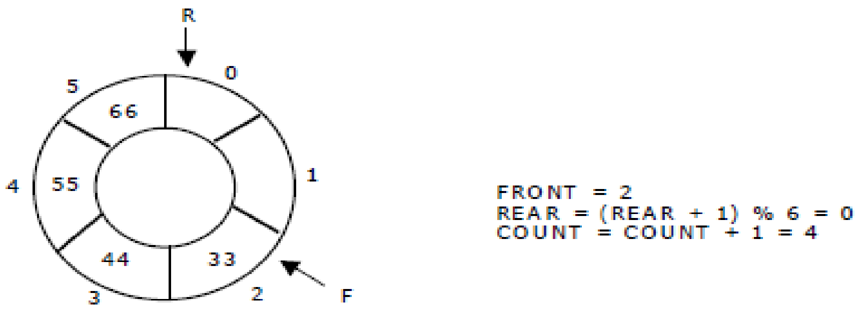
Again, delete an element. The element to be deleted is always pointed to by the FRONT pointer. So, 22 is deleted. The circular queue status is as follows:



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

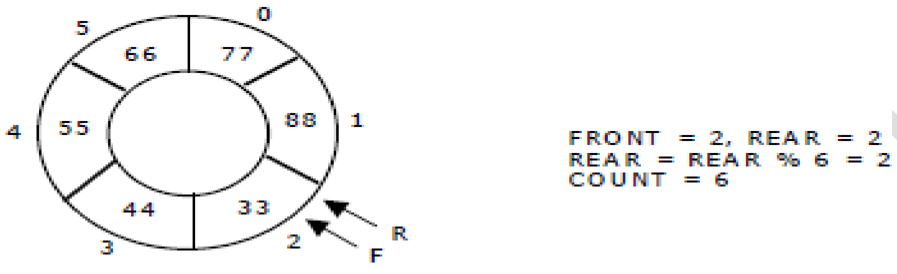
Circular Queue

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



Circular Queue

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



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*.

Operations on Circular queue:

a.enqueue() or insertion(): This function is used to insert an element into the circular queue. In a circular queue, the new element is always inserted at Rear position.

```
void insertCQ()
{
      int data;
      if(count ==MAX)
     {
         printf("\n Circular Queue is Full");
      }
     else
     {
         printf("\n Enter data: ");
         scanf("%d", &data);
         CQ[rear] = data;
         rear = (rear + 1) % MAX;
         count ++;
printf("\n Data Inserted in the Circular
Queue ");
    }
}
```

Algorithm: procedure of insertCQ():

```
Step-1:START
Step-2: if count==MAX then
           Write "Circular queue is full"
Step-3:otherwise
        3.1: read the data element
        3.2: CQ[rear]=data
        3.3 : rear=(rear+1)%MAX
        3.4 : count=count+1
Step-4:STOP
```

b.dequeue() or deletion(): This function is used to delete an element from the circular queue. In a circular queue, the element is always deleted from front position.

```c
void deleteCQ()
{
if(count ==0)
{
printf("\n\nCircular Queue is Empty..");
}
else
{
printf("\n Deleted element from Circular
Queue is %d ", CQ[front]);
front = (front + 1) % MAX;
count --;
}
}
```

Algorithm: procedure of deleteCQ():

```
Step-1:START
Step-2: if count==0 then
        Write “Circular queue is empty”
Step-3:otherwise
      3.1: print the deleted element
      3.2: front=(front+1)%MAX
      3.3: count=count-1
Step-4:STOP
```

c.dispaly(): This function is used to display the list of elements in the circular queue.

```c
void displayCQ()
{
int i, j;
if(count ==0)
{
printf("\n\n\t Circular Queue is Empty ");
}
else
{
printf("\n Elements in Circular Queue are:
");
j = count;
for(i = front; j != 0; j--)
{
printf("%d\t", CQ[i]);
i = (i + 1) % MAX;
}
}
}
```

Algorithm: procedure of displayCQ():

```
Step-1:START
Step-2: if count==0 then
        Write “Circular queue is empty”
Step-3:otherwise
      3.1: print the list of elements
      3.2: for i=front to j!=0
      3.3: print CQ[i]
      3.4: i=(i+1)%MAX
Step-4:STOP
```

<u>Deque:</u>

In the preceding section we saw that a queue in which we insert items at one end and from which we remove items at the other end. In this section we examine an extension of the queue, which provides a means to insert and remove items at both ends of the queue. This data structure is a ***deque***. The word ***deque*** is an acronym derived from ***double-ended queue***. Below figure shows the representation of a deque.

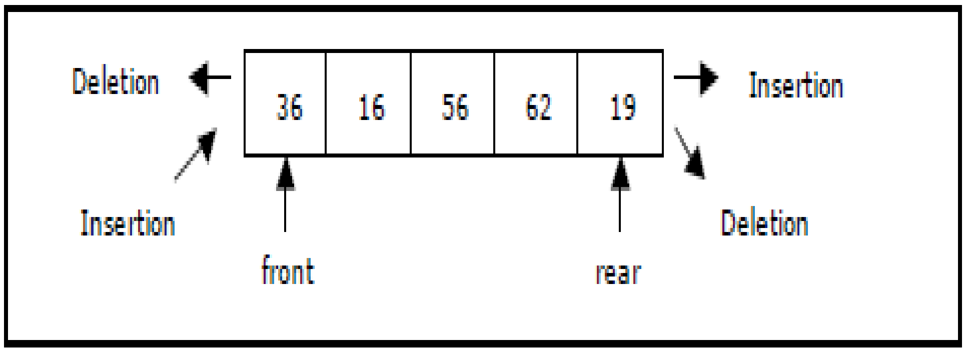


Figure Representation of a deque.

deque provides four operations. Below Figure shows the basic operations on a deque.

• enqueue_front: insert an element at front.
• dequeue_front: delete an element at front.
• enqueue_rear: insert element at rear.
• dequeue_rear: delete element at rear.

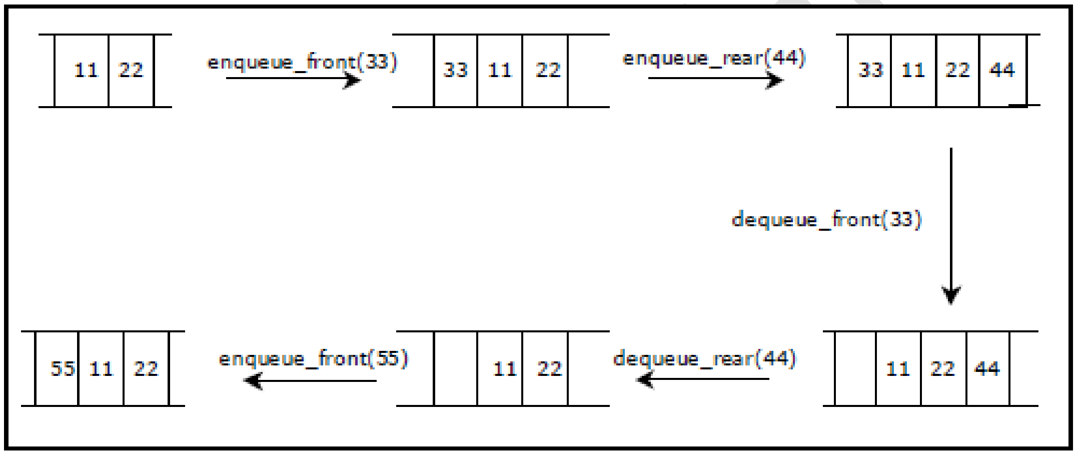


Figure ¯ ¯. Basic operations on deque

There are two variations of deque. They are:

• Input restricted deque (IRD)
• Output restricted deque (ORD)

An Input restricted deque is a deque, which allows insertions at one end but allows deletions at both ends of the list.

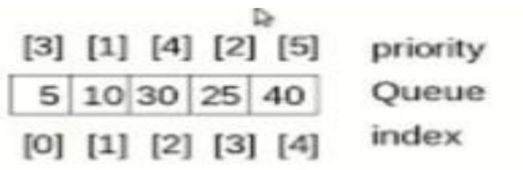
An output restricted deque is a deque, which allows deletions at one end but allows insertions at both ends of the list.

Priority Queue:

A **priority queue** is a collection of elements such that each element has been assigned a priority. We can insert an element in priority queue at the rare position. We can delete an element from the priority queue based on the elements priority and such that the order in which elements are deleted and processed comes from the following rules:

1. An element of higher priority is processed before any element of lower priority.
2. Two elements with same priority are processed according to the order in which they were added to the queue. It follows FIFO or FCFS(First Comes First serve) rules.

We always remove an element with the highest priority, which is given by the minimal integer priority assigned.



A prototype of a priority queue is time sharing system: programs of high priority are processed first, and programs with the same priority form a standard queue. An efficient implementation for the Priority Queue is to use heap, which in turn can be used for sorting purpose called heap sort

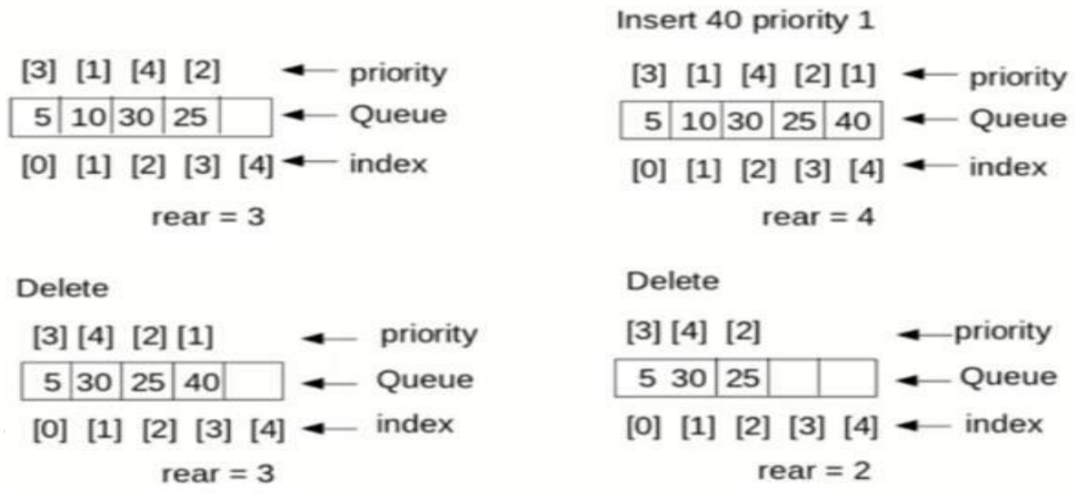
Priority queues are two types:

1. Ascending order priority queue
2. Descending order priority queue

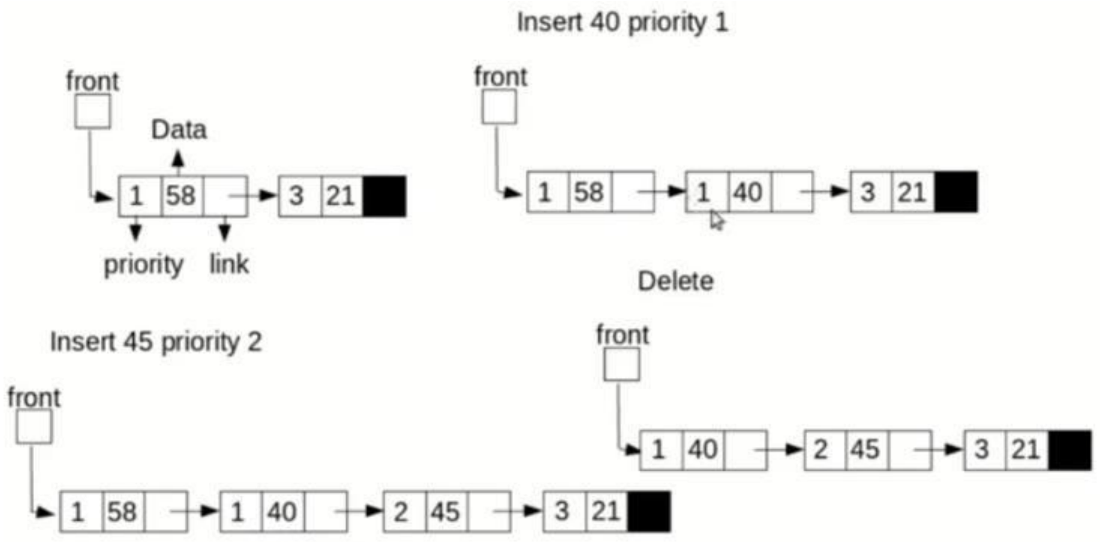
1. Ascending order priority queue: It is Lower priority number to high priority number.
Examples: order is 1,2,3,4,5,6,7,8,9,10
2. Descending order priority queue: It is high priority number to lowest priority number.
Examples: Order is 10,9,8,7,6,5,4,3,2,1
Implementation of Priority Queue:
Implementation of priority queues are two types:

1. Through Queue(Using Array)
2. Through Sorted List(Using Linked List)

1. Through Queue (Using Array): In this case element is simply added at the rear end as usual. For deletion, the element with highest priority is searched and then deleted.

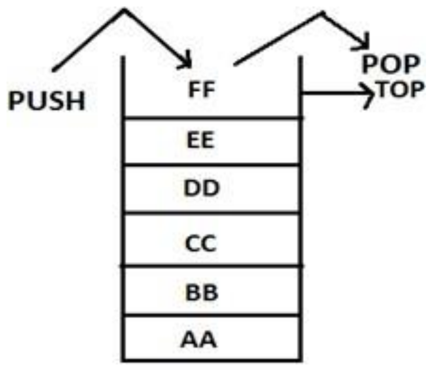
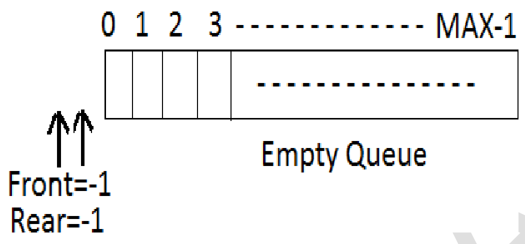


2. Through sorted List (Using Linked List): In this case insertion is costly because the element insert at the proper place in the list based on the priority. Here deletion is easy since the element with highest priority will always be in the beginning of the list.



1. Difference between stacks and Queues?

stacks	Queues
1.A stack is a linear list of elements in which the element may be inserted or deleted at one end. 2. In stacks, elements which are inserted last is the first element to be deleted. 3.Stacks are called LIFO (Last In First Out)list 4.In stack elements are removed in reverse order in which thy are inserted. 5.suppose the elements a,b,c,d,e are inserted in the stack, the deletion of elements will be e,d,c,b,a. 6.In stack there is only one pointer to insert and delete called "Top". 7.Initially top=-1 indicates a stack is empty. 8.Stack is full represented by the condition TOP=MAX-1(if array index starts from '0'). 9.To push an element into a stack, Top is incremented by one 10.To POP an element from stack,top is decremented by one.	1.A Queue is a linerar list of elements in which the elements are added at one end and deletes the elements at another end. 2. . In Queue the element which is inserted first is the element deleted first. 3. Queues are called FIFO (First In First Out)list. 4. In Queue elements are removed in the same order in which thy are inserted. 5. Suppose the elements a,b,c,d,e are inserted in the Queue, the deletion of elements will be in the same order in which thy are inserted. 6. In Queue there are two pointers one for insertion called "Rear" and another for deletion called "Front". 7. Initially Rear=Front=-1 indicates a Queue is empty. 8.Queue is full represented by the condition Rear=Max-1. 9.To insert an element into Queue, Rear is incremented by one. 10.To delete an element from Queue, Front is

11.The conceptual view of Stack is as follows: 	incremented by one. 11.The conceptual view of Queue is as follows:  0 1 2 3 ------------- MAX-1 Front=-1 Rear=-1 Empty Queue