STACK
STACK ADT

- A Stack is a linear data structure where insertion and deletion of items takes place at one end called top of the stack.
- **A Stack is defined as a data structure which operates on a last-in first-out basis. So it is also is referred as Last-in First-out(LIFO)-an element inserted last will be removed first.**
- Stack uses a single index or pointer to keep track of the information in the stack.
- The basic operations associated with the stack are: a) push(insert) an item onto the stack. b) pop(remove) an item from the stack.
- **The general terminology associated with the stack is as follows:**
- A stack pointer keeps track of the current position on the stack. When an element is placed on the stack, it is said to be pushed on the stack.
- When an object is removed from the stack, it is said to be popped off the stack.
- Two additional terms almost always used with stacks are **overflow-** which occurs when we try to push more information on a stack that it can hold, and **underflow-**which occurs when we try to pop an item off a stack which is empty.

Pushing items onto the stack: Assume that the array elements begin at 0
(because the array subscript starts from 0) and the maximum elements that can be placed in stack is max.
The stack pointer, top, is considered to be pointing to the top element of the stack. A push operation thus involves adjusting the stack pointer to point to next free slot and then copying data into that slot of the stack. Initially the top is initialized to -1.

Code to push an element on to stack:

```
void stack::push()
{
 if(top==max-1)
cout<<"Stack Overflow...\n";
else
{
cout<<"Enter an element to be pushed:";
top++;
cin>>data;
stk[top]=data;
 cout<<"Pushed Successfully... \n";
}
 }
```

Code to pop an element from a stack:

```
void stack::pop()
{
if(top==-1)
cout<<“stack underflow”;
else
{
cout<<“the elelment popped is:”<<stk[top];
top--;
}
}
```

Popping an element from stack:

To remove an item, **first extract the data from top position in the stack** and then decrement the stack pointer, top.

Program to demonstrate stack using array implementation:

```
#include <iostream>

using namespace std;
int stack[100], n=100, top=-1;
void push(int val)
{
  if(top>=n-1)

cout<<"Stack Overflow"<<endl;

else {
     top++;
      stack[top]=val;
  }

}

void pop()

{

 if(top<=-1)
cout<<"Stack Underflow"<<endl;
else
{
cout<<"The popped element is "<< stack[top] <<endl;
top--;
  }

}

void display()

{

if(top>=0)

{
cout<<"Stack elements are:";
for(int i=top; i>=0; i--) cout<<stack[i]<<" ";
cout<<endl;
  }
  else

cout<<"Stack is empty";

}

int main()

{
  int ch, val;
```

```
cout<<"1) Push in stack"<<endl; cout<<"2) Pop from stack"<<endl; cout<<"3) Display stack"<<endl; cout<<"4) Exit"<<endl;
  do
  {
cout<<"Enter choice: "<<endl;cin>>ch;
    switch(ch)
     {
    case 1:
cout<<"Enter value to be pushed:"<<endl;cin>>val;
        push(val);break;

      case 2:   pop();
        break;

      case 3: display();
        break;


case 4:

cout<<"Exit"<<endl;break;


      default:

cout<<"Invalid Choice"<<endl;


    }

}while(ch!=4);

return 0;
}
```

Input/Output:

1) Push in stack

2) Pop from stack

3) Display stack

4) Exit

Enter choice:

1

Enter value to be pushed:
10
Enter choice:

1

Enter value to be pushed:
20
Enter choice:

3

Stack elements are:20 10

A program to implement stack (its operations) using Linkedlists

```
#include <iostream> #include<stdlib.h>

using namespace std;

struct Node {
   int data;

   struct Node *next;

};

struct Node* top = NULL;

void push(int val)

 {
   struct Node* newnode = (struct Node*) malloc(sizeof(struct Node));
newnode->data = val;
newnode->next = top;
top = newnode;
}

void pop()

{

if(top==NULL)
cout<<"Stack Underflow"<<endl;
   else
   {

cout<<"The popped element is "<< top->data <<endl;

top = top->next;
   }

}

void display() {

struct Node* ptr;if(top==NULL)
cout<<"stack is empty";
else
{
ptr = top;

cout<<"Stack elements are: ";
```

```
while (ptr != NULL)
{
cout<<ptr->data <<" ";
ptr = ptr->next;
      }
   }
cout<<endl;
}
int main()
{
 int ch, val;
cout<<"1) Push in stack"<<endl; cout<<"2) Pop from stack"<<endl; cout<<"3) Display stack"<<endl; cout<<"4) Exit"<<endl;
   do {
cout<<"Enter choice: "<<endl;cin>>ch;
      switch(ch)
      {
      case 1:
cout<<"Enter value to be pushed:"<<endl;cin>>val;
            push(val);
            break;

         case 2:

          pop();
            break;

         case 3:

         display();
            break;
case 4:
 cout<<"Exit"<<endl;break;

         default:

 cout<<"Invalid Choice"<<endl;

 }while(ch!=4);

 return 0;
 }
```

Input/Output:

1) Push in stack

2) Pop from stack

3) Display stack

4) Exit

Enter choice:

1

Enter value to be pushed:
10
Enter choice:

1

Enter value to be pushed:
20
Enter choice:

3

Stack elements are: 20 10

Applications of Stack:

1. Stacks are used **in conversion of infix to postfix expression**.
2. Stacks are also used in **evaluation of postfix expression.**
3. Stacks are used to **implement recursive procedures.**
4. Stacks are used in **compilers.**
5. **Reverse String**

An arithmetic expression can be written in three different but equivalent notations, i.e., without changing the essence or output of an expression.

These notations are

1. Infix Notation

2. Prefix (Polish) Notation

3. Postfix (Reverse-Polish) Notation

INFIX TO POSTFIX CONVERSION USING STACKS

Basic:-

Infix Expression: The operator is in between the two operands

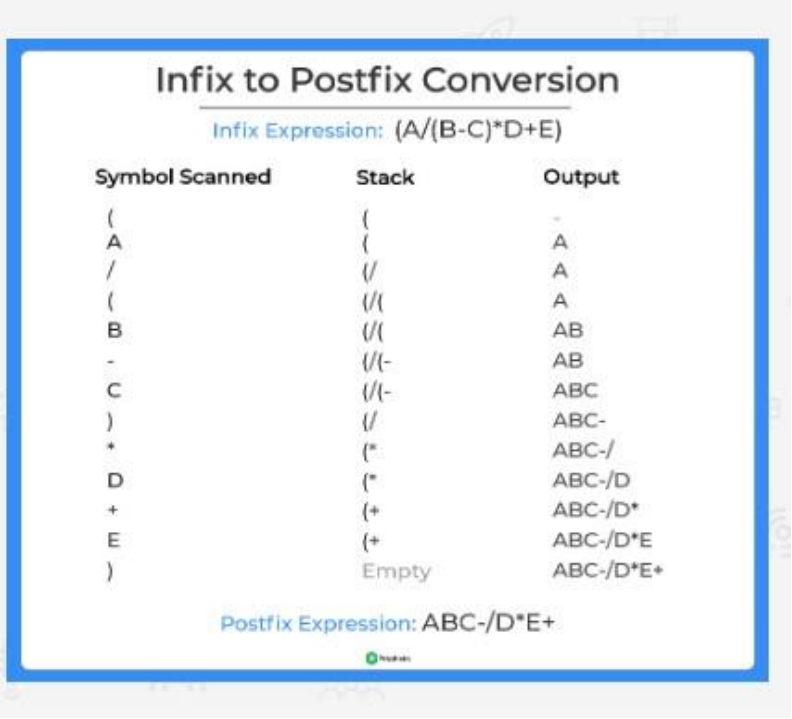
Example: A + B is known as infix expression.

Postfix Expression: The operator is after the two operands

Example: BD + is known as postfix expression.

Steps needed for infix to postfix conversion using stack in C++:-

1. First Start scanning the expression from left to right
2. If the scanned character is an operand, output it, i.e. print it
3. Else
    - If the precedence of the scanned operator is higher than the precedence of the operator in the stack(or stack is empty or has'(‘), then push operator in the stack
    - Else, Pop all the operators, that have greater or equal precedence than the scanned operator. Once you pop them push this scanned operator. (If we see a parenthesis while popping then stop and push scanned operator in the stack)
4. If the scanned character is an ‘(‘, push it to the stack.
5. If the scanned character is an ‘)’, pop the stack and output it until a ‘(‘ is encountered, and discard both the parenthesis.
6. Now, we should repeat steps 2 – 6 until the whole infix i.e. whole characters are scanned.
7. Print output
8. Do the pop and output (print) until the stack is not empty



Benefits of Postfix expression over infix expression

- In postfix any formula can be expressed without parenthesis.
- It is very useful for evaluating formulas on computers with stacks.
- Infix operators have precedence

Example:A program to implement Infix to Postfix Conversion using Stacks

```
#include<iostream>

#include<string.h>

#include<ctype.h>

using namespace std;

const int MAX = 50 ;

class infix
{

private :

char target[MAX], stack[MAX] ;

char *s, *t ;
int top ;
public :
infix( ) ;

void setexpr( char *str ) ;

void push ( char c ) ;

char pop( ) ;
void convert( ) ;

int priority ( char c ) ;

void show( ) ;
} ;

infix :: infix( )

{

top = -1 ;

strcpy( target, "" ) ;

strcpy( stack, "" ) ;

t = target ;
s = "" ;

}

void infix :: setexpr ( char *str )
{

s = str ;

}

void infix :: push ( char c )

{
```

```
if ( top == MAX )
cout<< "\nStack is full\n" ;
else
{
top++ ;
 stack[top] = c ;
}
}
char infix :: pop( )
{
if ( top == -1 )
{
cout<< "\nStack is empty\n" ;return -1 ;
}
else
{
char item = stack[top] ;
top-- ;
return item ;
}
}
void infix :: convert( )
{
while ( *s )
{
if ( *s == ' ' || *s == '\t' )
{
s++ ;
continue ;
}
if ( isdigit ( *s ) || isalpha ( *s ) )
{
while ( isdigit ( *s ) || isalpha ( *s ) )
{
*t = *s ;s++ ;
```

```
t++ ;
}
}
if ( *s == '(' )
{
push ( *s ) ;
s++ ;
}
char opr ;
if ( *s == '*' || *s == '+' || *s == '/' || *s == '%' || *s == '-' || *s == '$' )
{
if ( top != -1 )
{
opr = pop( ) ;
while ( priority ( opr ) >= priority ( *s ) )
{
*t = opr ;t++ ;
opr = pop( ) ;
}
push ( opr ) ;
push ( *s ) ;
}
else
push ( *s ) ;
s++ ;
}
if ( *s == ')' )
{
opr = pop( ) ;
while ( (opr ) != '(' )
{
*t = opr ;
t++ ;
opr = pop( ) ;
```

```
}
s++ ;
}

}

while ( top != -1 )

{

char opr = pop( ) ;

*t = opr ;

t++ ;
}

*t = '\0' ;

}

int infix :: priority ( char c )
{

if ( c == '$' )

return 3 ;
if ( c == '*' || c == '/' || c == '%' )
return 2 ;
else

{

if ( c == '+' || c == '-' )

return 1 ;
else
return 0 ;
}

}

void infix :: show( )

{

cout<<target ;

}

int main( )

{

char expr[MAX] ;
infix q ;
cout<< "\nEnter an expression in infix form: " ;
cin.getline ( expr, MAX ) ;
q.setexpr ( expr ) ;
q.convert( ) ;
```

```
cout<< "\nThe postfix expression is: " ;q.show( ) ;
return 0;

}
```

Input/Output:

Enter an expression in infix form: a+b*c

The postfix expression is: abc*+

EVALUATION OF POSTFIX EXPRESSION

1. Start reading the expression from left to right.
2. If the element is an operand then, push it in the stack.
3. If the element is an operator, then pop two elements from the stack and use the operator on them.
4. Push the result of the operation back into the stack after calculation.
5. Keep repeating the above steps until the end of the expression is reached.
6. The final result will be now left in the stack, display the same.

Program to demonstrate evaluation of postfix:

```
#include<iostream>

#include<stack>

#include<math.h>

using namespace std;

// The function calculate_Postfix returns the final answer of the expression after
calculation

int calculate_Postfix(string post_exp)

{

stack <int> stack;

int len = post_exp.length();

// loop to iterate through the expression

for (int i = 0; i < len; i++)

{

// if the character is an operand we push it in the stack

// we have considered single digits only here

if ( post_exp[i] >= '0' && post_exp[i] <= '9')

{

stack.push( post_exp[i] - '0');
```

```
}
// if the character is an operator we enter else block
else
{
// we pop the top two elements from the stack and save them in two integers
int a = stack.top();
stack.pop();
int b = stack.top();
stack.pop();
//performing the operation on the operands
switch (post_exp[i])
{
case '+': // addition
stack.push(b + a);
break;
case '-': // subtraction
stack.push(b - a);
break;
case '*': // multiplication
stack.push(b *  a);
break;
case '/': // division
stack.push(b / a);
break;
case '^': // exponent
stack.push(pow(b,a));
break;
}
}
}
//returning the calculated result
return stack.top();
```

```
}
//main function/ driver function
int main()
{
//we save the postfix expression to calculate in postfix_expression string
string postfix_expression = "59+33^4*6/-";
cout<<"The answer after calculating the postfix expression is : ";
cout<<calculate_Postfix(postfix_expression);
return 0;
}
```

Output

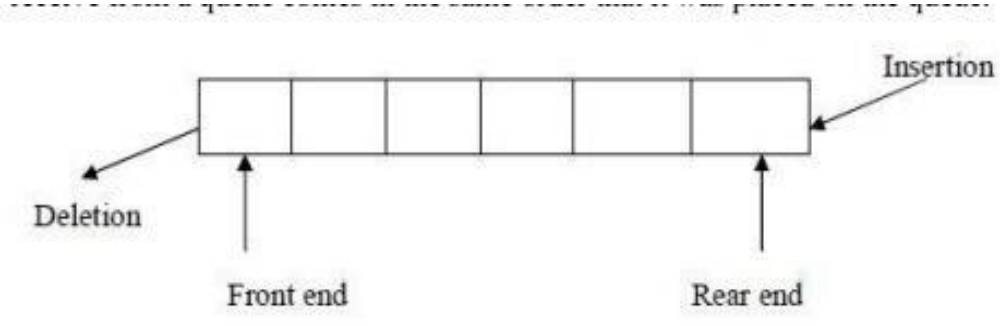
The answer after calculating the postfix expression is: -4

QUEUE ADT

A queue is an ordered collection of data such that the data is inserted at one end and deleted from another end.

The key difference when compared stacks is that in a queue the information stored is processed First-in First-out or **FIFO-an element inserted first, will be removed first.**

In other words the information receive from a queue comes in the same order that it was placed on the queue.



Representing a Queue:

One of the most common way to implement a queue is using array. An easy way to do so is to define an array Queue, and two additional variables front and rear.

The rules for manipulating these variables are simple:

- **Each time information is added to the queue, increment rear.**
- **Each time information is taken from the queue, increment front.**

- **Whenever front >rear or front=rear=-1 the queue is empty.**

Array implementation of a Queue do have drawbacks:

- The maximum queue size has to be set at compile time, rather than at run time.
- Space can be wasted, if we do not use the full capacity of the array.
- **Operations on Queue:**
- A queue have two basic operations: a)adding new item to the queue
- b) removing items from queue.
- **The operation of adding new item on the queue occurs only at one end of the queue called the rear or back.**
- **The operation of removing items of the queue occurs at the other end called the front**.
- For insertion and deletion of an element from a queue, the array elements begin at 0 and the maximum elements of the array is maxSize.
- **The variable front will hold the index of the item that is considered the front of the queue**, **while the rear variable will hold the index of the last item in the queue.**
- **Assume that initially the front and rear variables are initialized to -1.** Like stacks, underflow and overflow conditions are to be checked before operations in a queue.

Queue empty or underflow condition is

```
if((front>rear)||front= =-1)
    cout<"Queue is empty";
```

Queue Full or overflow condition is

```
if((rear==max)
    cout<"Queue is full";
```

-

Applications of Queue:

Queue, as the name suggests is used whenever we need to have any group of objects in an order in which the first one coming in, also gets out first while the others wait for there turn, like in the following scenarios :

1. Serving requests on a single shared resource, like a printer, CPU task scheduling etc.

2. In real life, Call Center phone systems will use Queues, to hold people calling them in an order, until a service representative is free.

3. Handling of interrupts in real-time systems. The interrupts are handled in the same order as they arrive, First come first served.

Program to implement Queue using Array:

```
#include<stdlib.h> #include<iostream>

 using namespace std;

#define max 5

 template <class T>

class queue
{

private:

T q[max],item;

int front,rear;
public: queue();
void insert_q();
void delete_q();
void display_q();
};

template <class T>
queue<T>::queue()
{

front=rear=-1;

}

//code to insert an item into queue;

template <class T>
void queue<T> ::insert_q()

{

if(rear>=max-1)

cout<<"queue Overflow...\n";

else
{

if(front>rear)

front=rear=-1;
else

{

if(front==-1)

front=0;

rear++;
cout<<"Enter an item to be inserted:";cin>>item;
q[rear]=item;
```

```
cout<<"inserted Sucesfully..into queue..\n";
}
}
}
template <class T>
void queue<T>::delete_q()
{
if(front==-1||front>rear)
{
front=rear=-1;
cout<<"queue is Empty....\n";
}
else
{
item=q[front];
front++;
cout<<item<<" is deleted Sucesfully... \n";
}
}
template <class T>
void queue<T>::display_q()
{
if(front==-1||front>rear)

{
front=rear=-1;
cout<<"queue is Empty....\n";
}
else
{
for(int i=front;i<=rear;i++)
cout<<"|"<<q[i]<<"|<--";
}
```

```
}

int main()

{

int choice; queue<int>
q;while(1)
{

cout<<"\n\n*****Menu for QUEUE operations*****\n\n";
cout<<"1.INSERT\n2.DELETE\n3.DISPLAY\n4.EXIT\n";
cout<<"Enter Choice:";cin>>choice;
switch(choice)
{

case 1: q.insert_q();
break;
case 2: q.delete_q();
break;
case 3:
 cout<<"Elements in the queue are .....\n";

q.display_q();break;
case 4:
exit(0);
default: cout<<"Invalid choice...Try again.  \n";

}

}

return 0;

}
```

Input/Output:

*****Menu for QUEUE operations*****

1.INSERT
2. DELETE

3. DISPLAY

4. EXIT

Enter Choice:3

Elements in the queue are....

queue is Empty....

*****Menu for QUEUE operations*****1.INSERT
2. DELETE

3. DISPLAY

4.EXIT

Enter Choice:1

Enter an item to be inserted:10

A program that implement Queue (its operations) using LinkedLists

```
#include<stdlib.h>
#include<iostream>

using namespace std; template <class
T>

class node
{

public:

T data; node<T>*next;
};

template <class T>

class queue
{

private:

T item;

friend class node<T>;node<T>
*front,*rear;

public:

queue();
void   insert_q();
void   delete_q();
void display_q();
};

template          <class          T>
queue<T>::queue()
{

front=rear=NULL;

}

//code to push an item into queue;

template <class T>
void queue<T>::insert_q()

{

node<T>*p;

cout<<"Enter an element to be inserted:";
```

```
cin>>item;
p=new node<T>;
p->data=item;
p->next=NULL;
if(front==NULL)
{

rear=front=p;

}

else

{

rear->next=p;

rear=p;
}

cout<<"\nInserted into Queue Succesfully . \n";

}

//code to delete an element

template <class T>
void queue<T>::delete_q()

{

node<T>*t; if(front==NULL)
cout<<"\nqueue is Underflow";
else
{

item=front->data;

t=front;
front=front->next;

cout<<"\n"<<item<<" is deleted Sucesfully from queue....\n";

}

delete(t);

}

//code to display elements in queuetemplate <class T>
void queue<T>::display_q()

{

node<T>*t;
if(front==NULL)
cout<<"\nqueue Under Flow";
else
{
```

```
cout<<"\nElements in the queue are ... \n";

t=front;

while(t!=NULL)
{

cout<<"|"<<t->data<<"|<-";t=t->next;
}

}

}

int main()

{

int choice;

queue<int>q1;

while(1)
{

cout<<"\n\n***Menu for Queue operations***\n\n";
cout<<"1.Insert\n2.Delete\n3.DISPLAY\n4.EXIT\n";        cout<<"Enter
Choice:";
cin>>choice; switch(choice)
{

case 1:
q1.insert_q();
break;
case 2: q1.delete_q();
break;
case 3: q1.display_q();
break;
case 4:
exit(0);

default:cout<<"Invalid choice...Try again...\n";

}

}

return 0;

}
```

Input/Output:

```
***Menu for Queue operations***
1.Insert
2.Delete
3.DISPLAY
4.EXIT
```

Enter Choice:1

Enter an element to be inserted:10

Inserted into Queue Succesfully....

Menu for Queue operations

1.Insert
2.Delete 3.DISPLAY
4.EXIT

Enter Choice:1

Enter an element to be inserted:20 Inserted into Queue Sucesfully....

Menu for Queue operations

1.Insert
2.Delete 3.DISPLAY
4.EXIT

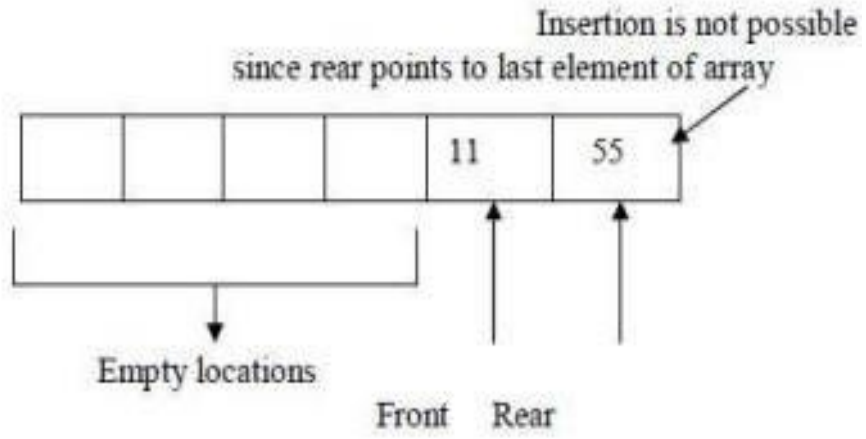
Enter Choice:3

Elements in the queue are....

|10|<-|20|<-

CIRCULAR QUEUE

Once the queue gets filled up, no more elements can be added to it even if any element is removed from it consequently. This is because during deletion, rear pointer is not adjusted.



When the queue contains very few items and the rear pointer points to last element. i.e. rear=maxSize-1, we cannot insert any more items into queue because the overflow condition satisfies. That means a lot of space is wasted.

Frequent reshuffling of elements is time consuming. One solution to this is arranging all elements in a circular fashion. Such structures are often referred to as Circular Queues.

A circular queue is a queue in which all locations are treated as circular such that the first

location CQ[0] follows the last location CQ[max-1].

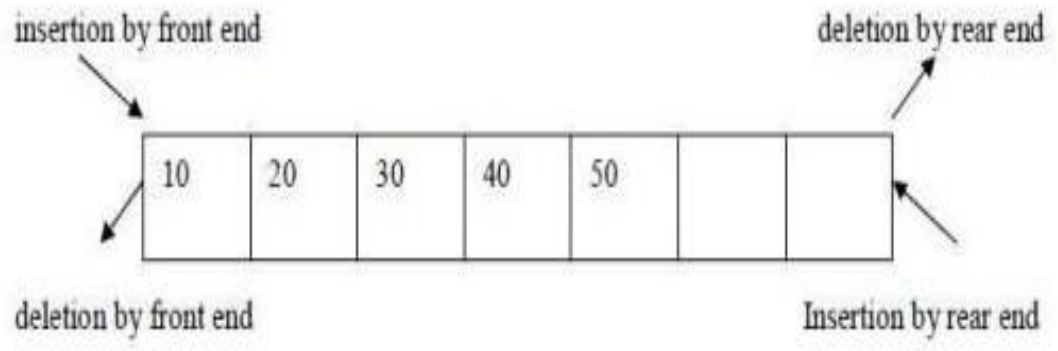
Circular Queue empty or underflow condition is

```
if(front==-1)
        cout<<"Queue is empty";
```

Circular Queue Full or overflow condition is

```
if(front==(rear+1)%max)
{
    cout<<"Circular Queue is full\n";
}
```

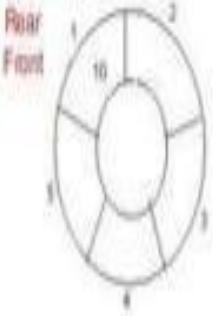
Normally insertion of elements is done at rear end and delete the elements from front end. For example elements 10,20,30 are inserted at rear end. To insert any element from front end then first shift all the elements to the right.



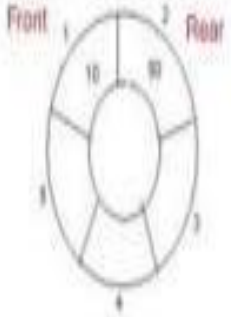
Example: Consider the following circular queue with N = 5.

1. Initially, Rear = 0, Front = 0.

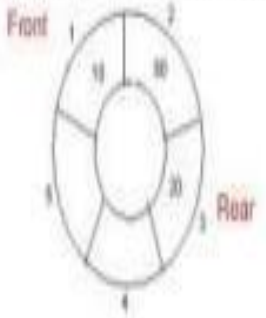
2. Insert 10, Rear = 1, Front = 1.



3. Insert 50, Rear = 2, Front = 1.



4. Insert 20, Rear = 3, Front = 0.



5. Insert 70, Rear = 4, Front = 1.

6. Delete front, Rear = 4, Front = 2.

Insertion into a Circular Queue:

```
Algorithm CQueueInsertion(Q,maxSize,Front,Rear,item)
Step 1: If Rear = maxSize-1 then
                Rear = 0
        else
                Rear=Rear+1
Step 2: If Front = Rear then
        print "Queue Overflow"
        Return
Step 3: Q[Rear] = item

Step 4: If Front = 0 then
        Front = 1
Step 5: Return
```

Deletion from Circular Queue:

```
Algorithm CQueueDeletion(Q,maxSize,Front,Rear,item)
Step 1: If Front = 0 then
        print "Queue Underflow"
        Return
Step 2: K=Q[Front]
Step 3: If Front = Rear then
        begin
                Front = -1
                Rear = -1
        end
        else
        If Front = maxSize-1 then
                Front = 0
        else
                Front = Front + 1
Step 4: Return K
```