Queue:

Queues is the ordered collections of homogenous elements. A queue is the linear list in which all the insertions are made at one end of the list known as rear or tail of the queue, all the deletions are made at other end known as front or head of queue.

⇒ It is non-primitive DS.

This mechanism is called first-in and first-out (FIFO).

The total number of elements in queue = rear - front + 1.

Operation in Queue:

The operation in queue are,

→ To insert an element in queue

→ Delete an element from queue.

States in Queue:

The number of states in queue are as follows.

i) Queue is Empty (FRONT = REAR)

ii) Queue is FULL (REAR = n)

iii) Queue contains element >=1

FRONT < REAR

No. of element = REAR - FRONT + 1.

Drawbacks of linear Queue:

When we delete same data from from full queue, then you are not able to insert data at that place again, that is the drawback of it

Prgram of linear Queue:

// Insert Algorithm.

```
{ int Q[4];
  int r=-1;  int f=-1;
```

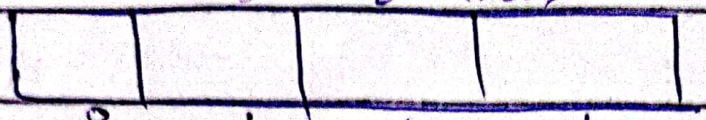
```
void Insert() {
    If (r==max-1)
    { cout<< "Queue is full";}
else{
        r=r+1;
        cout<<"Enter value";
        cin >> Queue[r]; }
```

// DELETE Algorithm

```
void DELETE() {
    If (f==r){
        cout<< "Queue is Empty"; }
else{
        f = f+1;
        cout<< Queue(f); }}
```

Dry Run:

Consider this array:

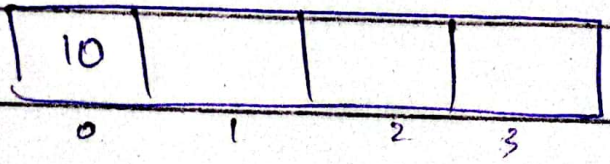


// Inserting values.

Pass-1: r = -1, f = -1, -1 == 3 (F)

r = -1 + 1 = 0

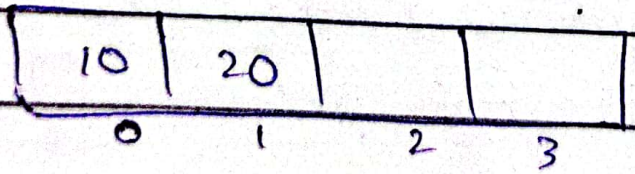
Q[0] = 10



Pass-2 r = 0; f = -1, 0 == 3 (F)

r = 0 + 1 = 1

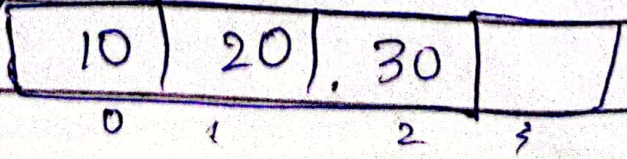
Q[1] = 20



Pass-3 r = 1, f = -1, 1 == 3 (F)

r = 1 + 1 = 2

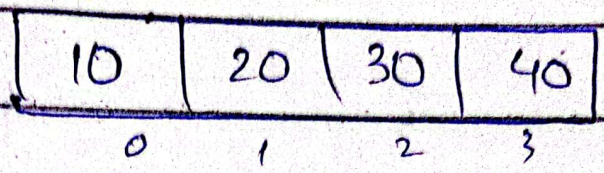
Q[2] = 30



Pass-4 | r=2, f=-1 , 2==3 (F)

r = 2+1 = 3

Q[3] = 40



Pass-5 | r = 3, f=-1 3==3 (T)

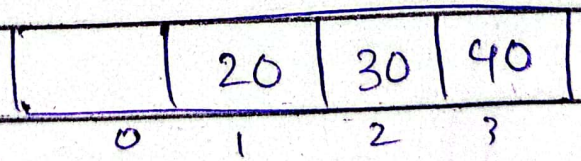
" Queue is full ".

" DELETE the values

Pass-1 | r=3, f=-1 -2==3(F)

r = -1+1 = 0

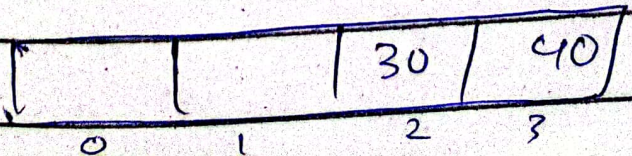
Q[0] = 10



Pass-2 | r=3, f=0 , 0==3 (F)

r = 0+1 = 1

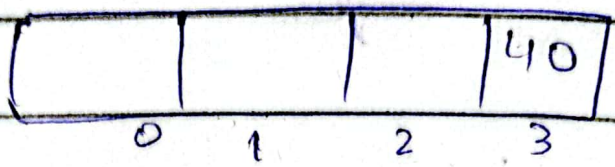
Q[1] = 20



Pass-3 | r = 3, f = 1 , 2 == 3 (F)

f = 1 + 1 = 2

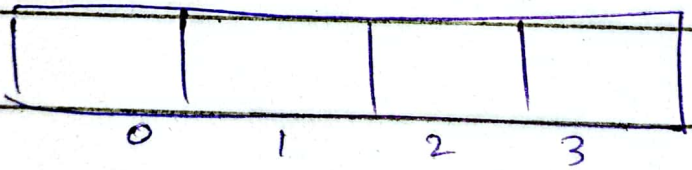
Q[2] = 40



Pass-4 | r = 3, f = 2 , 2 == 3 (F)

f = 2 + 1 = 3

Q[3] = 40



Pass-5 | r = 3 , f = 3 , 3 == 3 (T)

"Queue is Empty".

Circular Queue:

Circular queue is created to rectify the limitation of simple queue. In circular queue, rare and front are initialized by '0'.

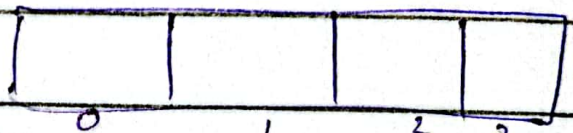
Program:

```
// Insert Algorithm.
{
  int Q[4];   int r=0; f=0, P;
void Insert() {
    P= r;  r= (r+1) % 4;
if (r==f) {
  cout << "Queue is full"; }
        r=P; }
else {
    cout << " Enter value";
    cin >> Q[r]; } }
```

// Delete algorithm.

```
void Delete() {
    If (f==r) {
    cout << " Queue is Empty ";}
else {
    f = (f+1) %n;
    cout << CQ[f];} }
```

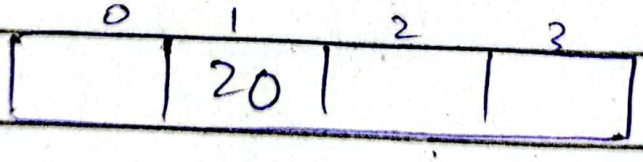
Dry Run:



// Inserting values.

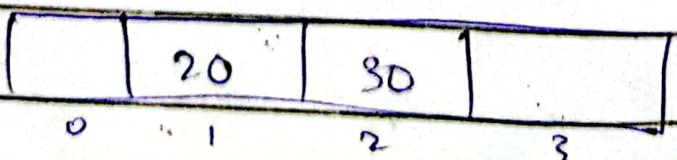
Pass-1 | r=0, f=0; P=0, r=(0+1)%4

CQ[1] = 20 1==0(F)



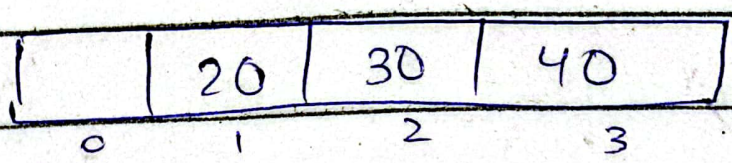
Pass-2 | r=1, f=0, P=1, r=(1+1)%4

CQ[2] = 30 = 2%4=



Pass-3| p=2, r=2, f=0, r=(2+1)%4=
= 3%4 = 3

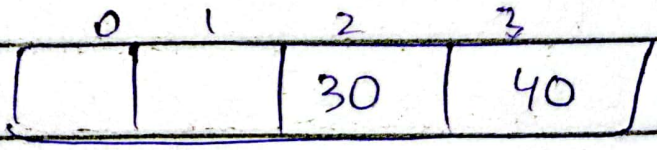
CQ[3] = 40 3 == 0 (F)



Pass-4| r=0, f=0; p=3
r=(3+1)%4=4%4=0
0==0(T)
"Array is Full".

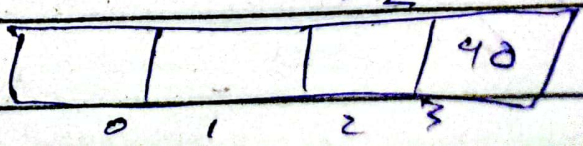
// Deleting the values:

f=(0+1)%4=1
Pass-1| r=3, f=0, p=3, 0==3(F)
CQ[1] = 20



Pass-2| r=3, f=1, p=3, f=(1+1)%4
=2

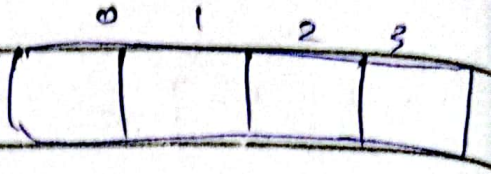
CQ[2] = 30



Pass-4 | r=3, f=2, p=3

f = (2+1)%4 = 3

CQ[3] = 40



Pass-5 | r=3, f=3, p=3 (3==3) (T)

"Array is Empty"

D-Queue (Double-Ended queue):

A dequeue or deck is double ended queue, allow elements to be added or removed on either the ends.

Types of Dqueue:

→ Input restricted queue.

→ output restricted queue

As STACK.

(when insertions and deletions made at same side).

As queue.

(when items are inserted at one end and removed at other end).

Operation in DQueue:

→ Insert element at back.
→ Insert element at front
→ Remove element at front
→ Remove element at back.

Application:

The applications of dequeue are as follows:

i) Palindrome - checker
ii) A steal job scheduling algo
iii) Undo - Redo operations in Software Eng.

Recursion:

Recursion is a technique can be defined as a routine that calls itself directly or directly. A recursive function is a function that solves a problem by solving smaller instances of the same problem; this technique

commonly used in programing to solve the problems that can be broken down into similar, sub-problems.

=> A recursive method must have at least one base or stopping case. [The base case doesn't execute a recursive call - [stops the recursion].

Recursion example:

Consider the example of find the factorial of a numbers.

```
=> public static int factorial (int n).
   { int fact;
     if (n => 1) {
       fact = factorial (n-1)*n; }
     else {
           fact = 1; }
     return fact; }
```

Dry Run
'Find factorial of 3'.
(Decomposition)

```
Public static int factorial (int 3)
{ int fact;
 if (n>1); fact = factorial (2) * 3;
else { fact = 1; return fact; }
```

```
Public static int factorial (int 2)
{ int fact;
if (n>1); fact = factorial (1) * 2;
 else { fact = 1;
        return fact; }
```

```
Public static void factorial (int 1)
{ int fact;
 if (n>1) {
     fact = factorial (n-1) * n;
else { fact = 1;
       return fact; }
```

```
public static int factorial (int 2)
{ int fact;
if (n>1){ fact= factorial(1)*2;
                           1* 2
else {   fact =1;
                 2
         return fact; }
```

```
public static int factorial (int 3)
{ int fact;
                                2*3
if (n>1){ fact = factorial(2)*3;
else{ fact =1;   return fact; }
                              6
```

The final result value is '6'.

Dynamic Data Structures:

"Linked List"

Data structure in which the memory occupied only at the time when data remains into the memory. As well as the data is removed then memory also is free. Its means that the size of data structure increases as the data comes and size reduces as the data is deleted.

Single linked list:

A simple or single data structure is one in which each node of which has one or more data fields (Data but only a single link field (LINK).

The advantages of linked list:

1. Easy to insert & delete at any time anywhere
2. Efficient memory usage.
3. Fast operations

Whenever we create a single link list, it must have the following characteristics.

→ Method to create node.

→ Method to find which node is full and which one is free.

→ Find the proper location of node.

→ A method to delete node.

Nodes in Linked List:

A node is basically a chunk of memory that carries the address, when a set of data elements to be used and application represented using a linked list.

The first node in linked list is called head node and last node is called null node.

→ Each node contain at least:

- A piece of data
- Pointer to the next node.

Linked list Program:

```
// Java program to implement simple
                                linked list.

import java.util.*;
class Node {                          // class Node
    int data;      Node link;
Node (int dd){                        // constructors
        link = null;   data = dd;}
Node (int dd , Node next)
    {   link = next;     data = dd;}}

class LinkedList {
        Node start;
        Node end;
        int size;
public LinkedList()
        {   start =null;
            end= null;
            size=0;}
```

```
// Function to check if list is Empty.
public boolean isEmpty()
  { return start == null; }
// Function to insert an element at start

Public void insertAtStart (int val)
  { Node nptr = new Node (val);
      size ++;
  if (isEmpty ())
    { start = nptr; end = start;
  else {
      nptr.link = start;
        start = nptr; } }

// function to insert element at end.
public void insertAtEnd (int val)
  { Node nptr = new INode (val);
      size ++;
  if (isEmpty ())
    { start = nptr; end = start; }
  else {
      end.link = nptr; end = nptr; }
```

```
// Function to insert at position.
public void insertATPos (int val, int x)
  { Node nptr = start;
while (nptr.link != null)
  { if (nptr.data == x)
      { break; }
    nptr = nptr.link; }
nptr.link = new Node (val, nptr.link);
      size ++;) }

// Function to delete an element at position.
  public void deleteATPos (int x)
    { int done = 0;
if (start == Null)
    System.out.println ("Empty");
else if (start.data == x)
  { start = start.link;
else {
  for (Node p = start; p.link != null;
                        p = p.link)
{ if (p.link.data == x)
  { done ++;
```

```
system.out.println("data inserted");
p.link = p.link.link; break; } } }

// Function to display elements.
public void display()
{ system.out.println("single link list=");
for( Node p = start; p != null; p = p.link)
{ sout("\t" + p.data");
system.out.println("Total no. of
nodes are" + size);

// class Single linked list
public class Single LinkedList
{ public static void main(String[] args)
{ Scanner in = new Scanner(System.in);
// creating object of class linked list
LinkedList list = new LinkedList();
system.out.println("single linked list");
int choice;
// Perform operation
do { sout("single linked list operation
```

```
sout ("1. Insert at Beginning");
Sout ("2. Insert at End");
Sout ("3. Insert at position");
Sout ("4. Delete at position");
Sout ("5. Display List");
sout ("0. To Exit");
choice = in.nextInt();
switch (choice)
{
case 0;
        sout ("Good by programing is ending");
        break;
case 1:
    sout ("Enter integer element to insert");
    List.insertAtStart (in.nextInt());
        break;
case 2;:
    sout ("Enter integer elemet to insert");
    list.insertAtEnd (in.nextInt());
case 3:
    sout ("Enter integer element to insert")
    int num = in.nextInt();
    sout ("Enter position");
```

```
int pos = in.nextInt();
list.insertAtPos(num, pos); break;
case 4;
system.out.println("Enter position");
int p = in.nextInt();
list.deleteAtPos(p); break;
case 5:
list.display(); break;
case 6:
sout("End program");
Default:
System.out.println("Wrong Entry"
break; }}
```

Analysis of Algorithim:

Analysis of algorithim means analyzing that which algorithim is upTO the mark. We analyze the algorithim in two ways.

→ Time complexity

→ Space complexity

As space complexity doesn't matter alot now adays. So main concern is speed complexity described by two method.

i) Imperical method
ii) Theoretical method.

Theoretical approach:

In order to perform the theoretical analysis of a algorithim.

→ we have to find the number of times each statement executes in the program.

→ Time taken by a statenet to execute it.

Efficiency = $\Sigma T \times C$

∴ $f(n) = an^2 + bn + c$

Imperical approach:

Emperical analysis in algorithim analysis involves the emperical evaluation of an algorithim's performance using actual data or simulation. Involves running algorithim multiple time on different input sizes.

Analysis of Program:

Example -1:

program	R	C
x=15;	1	1
cout << x;	1	1
for(i=1; i<=n; i++)	(n+1)	1
{ cout << i;	n	1
cout << x+i; }	n	1
cout << "Good bye"	1	1
	3n+4	6

∴ Loop running time = (UB-LB+1) + 1 = (n-1+1)+1 = n+1

∴ Loop body time = (UB-LB+1) = n-1+1 = n

Efficiency = (3n+4)(6) = 18n+4

Comparing it with equation

∴ $f(n) = an^2 + bn + c$

= 18n+4 (Neglect)

= 18n

[Analysis = O(n)]

Example-2:

Program	R	C
for(i=0; i<=n; i++)	n+2	1
{ if (a[i]==x)	n+1	1
{ loc = i; }}	n+1	1
if (loc != -1)	1	1
{ cout<<"Found"<<loc;}	1	1
else{	1	1
cout<<"value not found"; }	1	1
	n+3	7

$= (n+3)(7) = 7n+21$

$\therefore f(n) = an^2+bn+c$

$= 7n+21 \quad = 7n$

$O(n)$

(Neglect '4')

$$\therefore \sum_{i=1}^{n} = n \qquad \therefore \sum_{i=1}^{n} = \frac{n(n+1)}{2}$$

$$\therefore \sum_{i=1}^{n} = n^2$$

$$\therefore \sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$$

Example -3:

Program:	R	C
int m=5, n=5, i, j;	1	1
for (i=1; i<=n; i++)	(n+1)	1
{ cout<<"\n";	n	1
for (j=1; j<=m; j++)	n(n+1)=n²+n	1
{ cout<< j;}	n×n = n²	1
}		
cout<< "END";	1	1
	2n²+3n+3	6

$= (2n^2+3n+3)/6) = 12n^2+18n+18$

$= 12n^2+18n = 12n^2$

$= O(n^2)$

Program	R	C
{ int i, j, n=5	1	1
cout << "Good Bye";	1	1
for(i=1; i<=n; i++)	(n+1)	1
{ cout << "\n";	n	1
for(j=1; j<=i; j++)	$\sum_{i=1}^{n} i+1$	1
{ cout << j; } }	$\sum_{i=1}^{n} i$	1
cout << "Good Bye"; }	1	1

$= 2n+3 + \sum_{i=1}^{n} i+1 + \sum_{i=1}^{n} i$

$= 2n+3 + \sum_{i=1}^{n} i + \sum_{i=1}^{n} 1 + \sum_{i=1}^{n} i$

$= 2n+3+2\sum_{i=1}^{n} i + \sum_{i=1}^{n} 1$

$= (2n+3) + 2(n(n+1)) + n$

$= 2n+3+n^2+n+n = 2 = (n^2+4n+3)/6$

$= 6n^2 + 24n + 18 \quad = 6n^2$

$= O(n^2)$.