Data Structure

Dt:- 06/01/2020

Algorithm + Data structure = program

→ The organised collection of data is called data structure. The data are organised in the logical manner so that processing of data will be easier

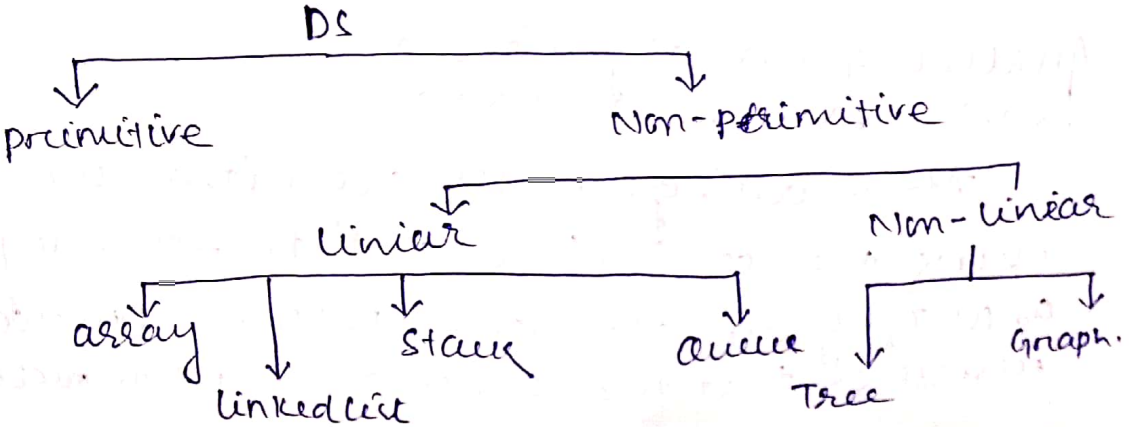
→ Data may be organised in different ways. The logical or mathematical model of a particular organisation is called datastructure

DS = organised data + operation

Different types of operations:-

→ Retrival/Traversing.
→ Insertion.
→ Deletion.
→ Modification.
→ Sorting.
→ Searching.
→ merging.

Classification of Data Structure:-



Dt:- 09/01/2020

Algorithm:-

→ It is well defined list of steps for solving a particular problem.

→ An algorithm is a sequence of computational steps that transforms inputs into outputs.

criteria/ conditions:-

1) Input:- 0 or more.

For ~~int~~ any operation input either 1 or 2.

For print anything ("Hello world") input is zero.

2) output:- one or more output ~~does~~ must should be there.

3) Definiteness:- Every step must be clear and be clear and unambiguous.

4) Finiteness:- The algorithm must terminate after a finite times.

5) Effectiveness:- Every step must be basic, so that anyone is possible.

Analizing an algorithm:-

After designing an algorithm we have to check the correctness and its efficiency. analyzing an algorithm means predicting the resources that the algorithm requires.

While designing an algorithm for a program we must consider.

1) Time complexity.
2) Space complexity.

Time complexity:- Amount of computer time it required.

Space complexity:- Amount of memory it required.

* Time complexity or running time of an algorithm on a particular input number of premitive operations or steps incurred

	frequency
Sum(A, n)	0
1. {	0
2. S = 0	1
3. for i = 0 to n / for(i=0; i<n; i++)	n+1
4. S = S + A[i]	n
5. return S	1
6. }	0

$T(n) = 2n+1$

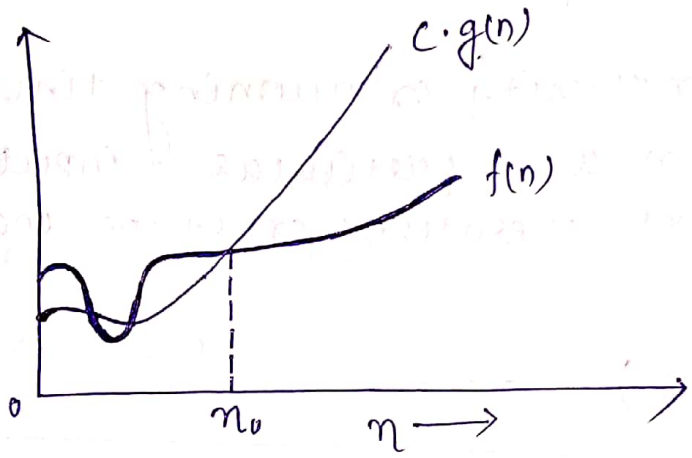
→ When the size of a problem increases it will increase the number of operations (T_n). In what order the total number of operation is increased is known as rate of growth. So when we increase the input size(n), the running time of the algorithm T(n) also increases.

To know the order T(n) we use some notation known as "Asymptotic notation". ($O, \Omega, \Theta, o, \omega$)

* O - Notation ("Big-oh" notation) :-

$$f(n) = O(g(n))$$

$O(g(n)) = \{ f(n) \mid$ There exist positive constant c and n_0 such that $0 \leq f(n) < c \cdot g(n)$ for all $n \geq n_0 \}$



Ex-1

$3n+2 = O(n)$, find c and n_0.

$3n+2 = O(n)$

$f(n)$ $\quad$ $g(n)$

$3n+2 \leq c \cdot n$

Let, $c = 4$

$3n+2 \leq 4n$

$\Rightarrow$ at $n = 2$

$8 \leq 8$

So, $c = 4$ and $n_0 = 2$

(Ans).

* Asymptotic Notations:-

Dt:- 13/01/2020

Ex:- prove that;

$2n^2+3n+4 = O(n^2)$

Here, $f(n) = 2n^2+3n+4$ and $g(n) = n^2$

So, $2n^2+3n+4 \leq cn^2$

$c = 9$	$c = 3$
$n_0 = 1$	$n_0 = 4$

Ex:- $3n+2 = O(n)$

Here, $f(n) = 3n+2$, $g(n) = n$

$c = 5$
$n_0 = 1$

Ex:- $2n^2+3n+2 = O(n)$

Here, $f(n) = 2n^2+3n+2$, $g(n) = (n)$

~~$c = 7$~~
~~$n_0 = 1$~~

$2n^2+3n+2 \not\leq c(n)$

(It can't be a big-oh notation.)

```
for(i=0; i<m; i++) ---------- m+1
{
    for(j=0; j<n; j++) ---------- n+1
    {
        printf("%d", i*j); ---------- m*n
    }
}
```

$T(n) = (m+1+n+1) + m*n$

$= O(m*n) \approx O(m^2)$ (or) $O(n^2)$

Ex:-

		frequency
1.	i = 1;	1
2.	while (i <= n)	n+1
3.	{	0
4.	x = x+1;	n
5.	i = i+1;	n
6.	}	0

$T(n) = 3n+2$

$T(n) = O(n)$

Ex:-

```
int f(int n)
{
    if (n==1)
    {
      return 1;
    }
    else
    {
      return n + f(n-1);
    }
}
```

$T(n) = 1$; if $n=1$

$= n + T(n-1)$ if $n>0$

$T(n) = n + T(n-1)$

$T(n-1) = n + (n-1) + T(n-2)$

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

$\Rightarrow T(n+1) = n + (n-1) + (n-2) + \cdots + 3 + 2 + 1 \quad (\because T(n) = 1)$

$\Rightarrow T(n) = \frac{n(n+1)}{2}$

$\Rightarrow T(n) = \frac{n^2 + n}{2}$

$\Rightarrow \boxed{T(n) = O(n^2)}$

Assignment:-

1. $4n^3 + 6n + 5 = O(n^3)$ ⟨prove that⟩
2. $2n^5 + 100n + 6 = O(n^5)$
3. $2^n + 3n^5 + 5 = O(2^n)$
4. Factorial of a number;

$T(n) = 1$ for $n = 1$

$= n * T(n-1)$ for $n \geq 2$

5. Fibonacci ↓

$T(n) = 0$ if $n = 1$

$= 1$ if $n = 1$

$= T(n-1) + T(n-2)$ $n \geq 2$

Have to submit on (25-/01/2020)

Dt:- 15-/01/2020

⇓

*Abstract Data type (ADT):-

The ADT consist of a set of definition, that allow programmers to use the functions while hiding the implimentation.

With an ADT users are not concerned with how the task is done rather they need to be concerned only what it can do!

An ADT is a data declaration packaged together with the operations that are meaningful on the data type.

Array:-

2D-array:-

int a[3][2];

Representation → Row-major order.

Representation → column-major order.

$$\begin{bmatrix} a[0][0] & a[0][1] \\ a[1][0] & a[1][1] \\ a[2][0] & a[2][1] \end{bmatrix}$$

(a) Row-major order:-

a[0][0] → a[0][1]

a[1][0] → a[1][1]

a[2][0] → a[2][1]

(b) column-major order:-

a[0][0] → a[1][0] → a[2][0] → a[0][1] → a[1][1] → a[2][1]

Address calculation:-

(a) Row-major order:-

Let, arr[u1][u2] be a 2-D array.

Add(arr[i][j]) = b + (i * u2 * j) * w

b ← Base address.

w ← width of the data.

(b) column major order :-

$$add(arr[i][j]) = b + (j * u_1 + i) * w$$

Ex:- consider the array with $u_1 = 10$ and $u_2 = 15$ and b is the address of a[0][0] and w = 2

add(a[9][8]) = ?

(a) Row-major :-

= b + 226

(b) column-major :-

= b + 174

Ex:- If the address of A[1][1] and A[2][1] is 1000 and 1010 respectively and each element occupies 2 bites. Then the array have been stored in which order.

Q2:- Asume 4 bites of storage required to store each element of an array A[10][20] which is stored in row-major order. Let the element A[0][3] is at the location 1012 in memory. Find out the address of the element A[8][15]

Ans:- A[10][20]

w = 4

Add(A[0][30]) = 1012

Add(A[8][15]) = ?

=> b + (8 * 20 + 15) * 4

=> b + (160 + 15) * 4

=> b + 700

=> 1000 + 700 = 1700

Q.1 (Ans):-

Add[A[1][1]) = 1000

Add[A[2][1]) = 1010

w = 2

Sparse Matrix:-

A m×n matrix, is called sparse matrix, if it contains of more number of '0' as compare to non-zero elements. A matrix which is not sparse is called dense matrix.

$$A = \begin{bmatrix} 1 & 0 & 0 & 0 & -2 & 0 \\ 0 & 0 & 3 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 4 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}_{6\times6}$$

6X6 X4 = 144 bytes

Alternate approach:-

No. of rows	No. of columns	No. of Non-zero
–	–	–
–	–	–
–	–	–

:) Alt → A ⇒

$$\begin{bmatrix} 6 & 6 & 5 \\ 0 & 0 & 1 \\ 0 & 4 & -2 \\ 1 & 2 & 3 \\ 2 & 2 & -1 \\ 4 & 2 & 4 \end{bmatrix}$$

6 X 3 X 4 = 72 bytes.

* Two types of n-square Spare matrix:-

1) Triangular Spare matrix.
2) Tridiagonal Spare matrix.

1) Triangular Spare matrix:-

All entries above the main diagonals are zero.

Ex:-
$$\begin{bmatrix} 5 & 0 & 0 & 0 & 0 \\ 2 & 3 & 0 & 0 & 0 \\ 1 & 2 & 4 & 0 & 0 \\ 2 & 1 & 2 & 3 & 0 \\ 4 & 5 & 6 & 3 & 2 \end{bmatrix} (5 \times 5)$$

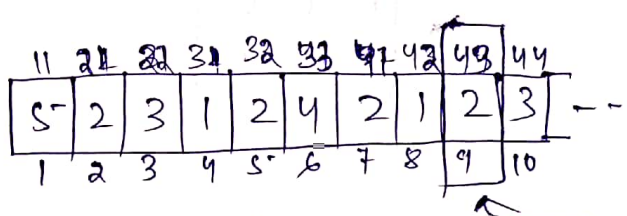
2) Tridiagonal Spare matrix:-

Non-zero entries can occure on the diagonal or on elements immidiedly above or below the diagonal.

$$\begin{bmatrix} 1 & 2 & 0 & 0 & 0 & 0 \\ 4 & 3 & 5 & 0 & 0 & 0 \\ 0 & 6 & 4 & 7 & 0 & 0 \\ 0 & 0 & 8 & 9 & 2 & 0 \\ 0 & 0 & 0 & 3 & 1 & 6 \\ 0 & 0 & 0 & 0 & 2 & 3 \end{bmatrix} (6 \times 6)$$

position of a perticular elements in an array:-

:) Triangular:-
$$\begin{bmatrix} 5 & & & & \\ 2 & 3 & & & \\ 1 & 2 & 4 & & \\ 2 & 1 & (2) & 3 & \\ 4 & 5 & 6 & 3 & 2 \end{bmatrix} 5 \times 5$$



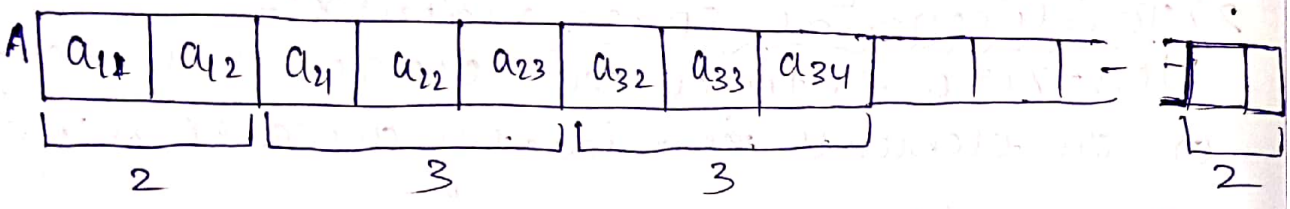
$$pos[a_{ij}] = 1+2+3+\cdots+(i-1)+(j-1)+1$$
$$= \frac{i(i-1)}{2} + j$$

Ex:-
$$pos[a_{43}] = \frac{4 \times 3}{2} + j$$
$$= 9$$

Tridiagonal:

$$\begin{bmatrix} 1 & 2 & & & & \\ 3 & 4 & 5 & & & \\ & 6 & 1 & 7 & & \\ & & 8 & 9 & 2 & \\ & & & 3 & 1 & 6 \\ & & & & 2 & 3 \end{bmatrix}_{(6\times 6)}$$

⇓ (1-D array)



$P[a_{ij}] = 2+3+3+\cdots+3+1$

— Dt:- 17/01/2020

*

$$A = \begin{bmatrix} 5 & 0 & 0 & 0 \\ 0 & 9 & 0 & 0 \\ 2 & 0 & 0 & 0 \end{bmatrix} \xrightarrow{\text{Triplets}} (B) = \begin{bmatrix} 3 & 4 & 3 \\ 0 & 0 & 5 \\ 1 & 1 & 9 \\ 2 & 0 & 2 \end{bmatrix}$$

⇓

$$A^T = \begin{bmatrix} 5 & 0 & 2 \\ 0 & 9 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \xrightarrow{\text{Triplets}} (C) = \begin{bmatrix} 4 & 3 & 3 \\ 0 & 0 & 5 \\ 0 & 2 & 2 \\ 1 & 1 & 9 \end{bmatrix}$$

1) convert sparse matrix to triplet form.

```
void triplet(int A[ ][10], int B[ ][3], int r, int c)
{
    int cnt = 0;
    B[0][0] = r; B[0][0] = c;
    for(i=0; i<r; i++)
    {
        for(j=0; j<c; j++)
        {
            if( A[i][j] != 0)
            {
                cnt++;
                B[cnt][0] = i;
                B[cnt][1] = j;
                B[cnt][2] = A[i][j];
            }
        }
    }
    B[0][2] = cnt++;
}
```

* $T(n) = O(n^2)$ (Due to nested loop)

2) Find out the transpose of a sparse matrix which is represented in triplet matrix form.

```
void Triplet_Transpose(int B[ ][3]; int C[ ][3])
{
    C[0][0] = B[0][1]; C[0][1] = B[0][0]; C[0][2] = B[0][2];
    m = 1;
    for(i=0; i<B[0][1]; i++)
    {
        for(j=1; j<=B[0][2]; j++)
        {
            if( i == B[j][1])
            {
```

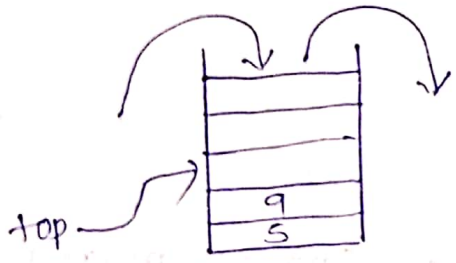
```
            C[m][0] = B[j][1];
            C[m][1] = B[j][0];
            C[m][2] = B[j][2];
            m++
        }
    }
}
}
```

Dt:- 22/01/2020

STACK:-

→ Stack is a linear data structure in which insertions and deletion of element will take place at one end called top of the stack.

→ Stack follows last-in-first-out (LIFO).



~~Insert an element on the stack~~

Operation:-

1) push(s, i): Insert an element (i) to the top of the stack.

2) pop (s): Delete the top element of the stack.

Implimantation of stack:-

→ Array (static allocation)

→ Linked-list (Dynamic allocation)

1) Array implimantation of stack:-

In the array implimantation, we will use an array and an element "top" which holds the index of the "top" element of the array. The limitation of this technique, ~~the~~ is we have to declare the size of the array, before start the operation.

* push operation:-

```
#define maxsize 5
int s[maxsize];
int top=-1; /* stack is empty */
void push(int s[maxsize], int i)
{
    if(top == maxsize-1)
    {
        printf("stack is fulfilled");
        exit(1);
    }
    else
    {
        top = top+1; / top++; / top += 1;
        s[top] = i;
    }
}
```

* pop operation:-

```
#define maxsize 5
int s[maxsize];
int top=-1;
int pop(int s[maxsize])
{
    if(top==-1)
    {
        printf("stack is empty\n");
        exit(1);
    }
    else
    {
        ele=s[top];
        pop--;
    }
    return ele;
}
```

Dt:- 23/01/2020

STACK-operation:-

```
#include<stdio.h>
#include<stdlib.h>
#define maxsize 10
int s[maxsize];
int top=-1;
void push(int [maxsize], int);
int pop(int [maxsize]);
void display(int [maxsize]);
```

```
int main()
{   int choice, ele,
    while (1)
    {
        printf(" 1 for push\n");
        printf(" 2 for pop\n");
        printf("3 for display\n");
        printf("4 for Quit\n");
        printf("Enter your choice\n");
        scanf("%d", &choice);
        switch(choice)
        {
            case 1:
                    printf("Enter an element\n");
                    scanf("%d", &ele);
                    push (s, ele);
                    break;
             case 2:
                    x = pop(s);
                    printf("%d is deleted\n", x);
                    break;
             case 3:
                    printf("The stack is:-\n");
                    display(s);
                    break;
              case 4:
                    exit(0);
              default:
                    printf("wrong option\n");
                    break;
        }
    }
}
```

Applications of stack:-

→ Implemantation of recursion.

→ Evaluation of arithmatic expression.

(i) infix to postfix expression.

(ii) infix to prefix expression.

(ii) evaluation postfix expression.

(iv) evaluation of prefix expression.

(v) check for balanced parenthesis.

→ Reversing a string.

Dt:- 24/01/2020

* Arithmatic operation:-

Infix expression (real expression) = A + B.

postfix expression : AB+

prefix expression = +AB.

Ex:- ((A+B) * (C-D)) = (AB+) * (CD-)

= AB+ CD- *.

(i) infix to postfix expression:-

```
void infix_to_postfix(E,P)
{
    push(S,C);
    while(stack is not empty\n");
    {   item = E, search( );
        if (item == operand)
        else if(item == '(')
        else if(item == ')')
        {
```

```
        x = pop(s);
        while (x != '(')
        {
            add top();
            x = pop(s);
        }
    }
    else
    {
        x = pop(s);
        while (precendence(x) >= precedence(item))
        {
            add top;
            x = pop(s);
        }
        push(s, x);
        push(s, item);
    }
while(x = pop(s) != '(')
{
    add top;
}
}
```

Ex:-

E: A+(B*C-(D/E^F)*G))*H.

	Symbol scanned	Stack(S)	Postfix(P)
		(	—
1	A.	(	A
2	+	(+	A
3	(	(+(	AB
4	B	(+((	AB
5	*	(+(*	ABC
6	C	(+(*	
7	-	(+(-	ABC*
8	(	(+(-(	ABC*
9	D	(+(-(	ABC*D
10	/	(+(-(/	ABC*D
11	E	(+(-(/	ABC*DE
12	^	(+(-(/^	ABC*DE
13	F	(+(-(/^	ABC*DEF
14	)	(+(-	ABC*DEF^/
15	*	(+(-*	ABC*DEF^/
16	G	(+(-*	ABC*DEF^/G
17	)	(+	ABC*DEF^/G*-
18	*	(+*	ABC*DEF^/G*-
19	H	(+*	ABC*DEF^/G*-H
20	)	—	ABC*DEF^/G*-H*+

Q convert the following Infix expression into postfix expression using STACK.

Q: 5 * (6+2) - 12/4

postfix - Evaluation:-

Symbol Scaned	A	B	Stack(s)
1. 5	—	—	5
2. 6	—	—	5, 6
3. 2	—	—	5, 6, 2
4. +	2	6	5, 8
5. *	8	5	40
6. 12	—	—	40, 12
7. 4	—	—	40, 12, 4
8. /	4	12	40, 3
9. —	3	40	37.

```
int postfix eva(char p[])
{
    int op1, op2;
    while (po[i] = '\0')
    {
        if(operand)
        {
          push(s, po[i]);
        }
        else
        {
            op1 = pop(s);
            op2 = pop(s);
            switch(po[i])
            {
                    case '+':
                            r = op2 + op1;
                            break;
                    case '-':
                            r = op2 - op1;
                            break;
                    case '*':
                            r = op2 * op1;
                            break;
                    case '/':
                            r = op2 / op1;
                            break;
                    case '^':
                            r = op2 ^ op1;
                            break;
            }
            push(s, r);
            i++
    }   return pop(s);
}
```

Assignment:- (Submission date:- 31/01/2020)

1) convert infix to postfix using stack.

a) A+[(B+C)+(D+E)*F]/G.

b) (A+B)*C/D+E$$F/G (where, $==^)

c) A-B/(C* D$E)

d) (A-B/C)*(D*E-F)

e) (C-D)+(E$F)+F/(H+W)* A$B

2) post-fix evaluation:-

4, 2, $, 3, *, 3, -, 8, 4, /, 1, 1, +, /, +

Queue:-

→ A queue is a linear Data-structure, in which elements are added, at one end and known as 'rear' and remove from other end 'front'.

→ It follows First in first out (FIFO)

operations:-

→ Insert(Q, n)
insert n into queue(Q) at rear end

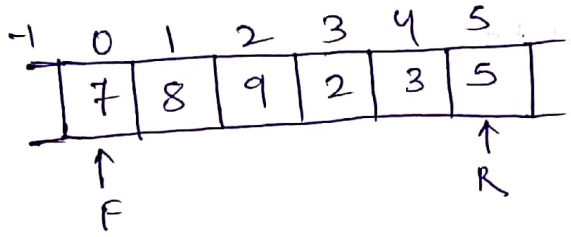
→ Delete(Q)
Delete an element from the front-end.

array Implementation:-

→ During initialisation of Q front = 0
rear = -1

→ on each addition (insert an new element) of Q rear is incremented by 1.

→ On ~~de~~ each delete operation front is incremented by 1.



maxsize = 6
front = 0
rear = -1

```
Insert ( int Q[maxsize], int n)
{
    if (rear == maxsize-1)
    {
        printf("Queue full");
        exit(0);
    }
    else
    {
        rear = rear+1;
        Q[rear] = n;
    }
}
```

```
int delete (int Q[maxsize])
{
    if (front > rear)
    {
        printf ("Queue is Empty\n");
        exit(0);
    }
    else
    {
        n = Q[front];
        front = front++;
    }
    return n;
}
display (int Q[maxsize])
{
    for (i = front; i <= rear; i++)
    {
        printf ("%d", Q[i])
    }
}
```

HW

Q write a menue drive program, to ~~find~~ implement of Queue.

Q Asume that there is a queue having maxsize = 5, the following operation;

Insert(10), Insert(20), Insert 30, Delete(), Insert (40), Insert (50), Delete(), find the final ~~value~~ result of Que as well as value of front and rear.

Ans; Result:- 30, 40, 50
rear;- 4.
front:- 2.

Again, insert (60)

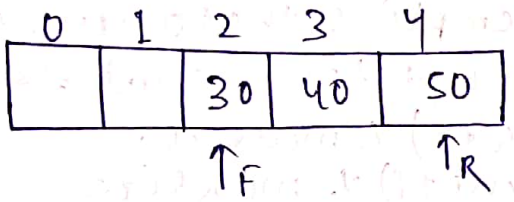
Result: 30, 40, 50, 60.

front = 2

rear = 5.

Q consider a linear Queue with MAXSIZE = 5. After performing the following operations, what will be result of the queue.

Insert(10), Insert(20), Insert(30), Delete, Delete, Insert(40 Insert(50), Insert(60)



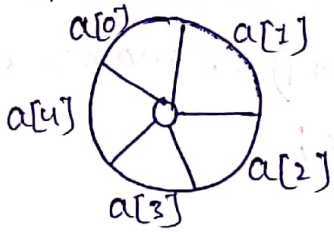
front = 2
rear = 4

* Circular Queue:-

A circular is one in which insertion of a new element is done at the very first location of the Queue if the last location of the queue is filled. ~~for the of~~

That is, for the queue of 'n' elements after inserting an element at the location of array, for a queue of n elements the next element will be inserted at very first location

It is possible if there locations are empty



Assumption-1:-

1) Front will always pointing to the first element

i.e, front = -1 ⇒ Queue is empty

front = rear = -1

2) Each time a new element is inserted into the queue.

rear = rear + 1

3) Each time an element is deleted from the queue.

front = front + 1

4) If (front == rear) => queue contains only one element except front = rear = -1.

5) If Front = rear is implies that queue contains only one element except front = rear = -1

-> rear = (rear + 1) % maxsize.

-> Front = (Front + 1) % maxsize

For insertion:-

```
int front = -1, rear = -1;
void insert(int Q[], int ele)
{
    if (front == (rear+1) % maxsize)
    {
        printf("Queue is filled\n");
        exit(0);
    }
    else if (front == -1) /* Empty Queue */
    {
        front = rear = 0;
    }
    else
    {
        rear = (rear+1) % maxsize;
        Q[rear] = ele;
    }
}
```

For Display :-

```
void display( )
{
    if (front <= rear)
    {
        for (i = front; i <= rear; i++)
        {
            printf("%d", Q[i]);
        }
    }
    else
    {
        for(i = front; i <= maxsize-1; i++)
        {
            printf("%d", Q[i]);
        }
        for(j=0; j <= rear; j++)
        {
            printf("%d", Q[j]);
        }
    }
}
```

Application of Circular Queue:-

a) Simulation of traffic control system.

b) CPU scheduling in multiprogramming.

c) Round - Robin Scheduling.

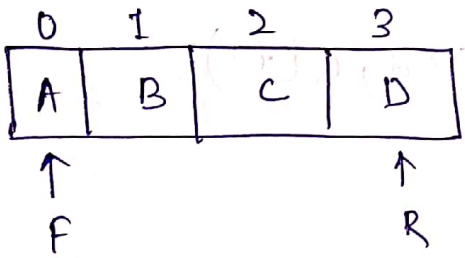
Q. Consider a circular queue implemented on an array of size 4. The front is pointing to element A and the rear is pointing to element D. Perform the following operations and show the content of circular queue after each operation.

1) Insert(F), 2) Delete(), 3) Delete()

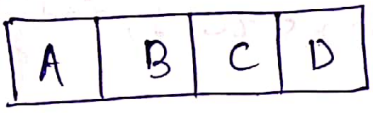
4) Insert(K), 5) Insert(L), 6) Insert(M)

Ans

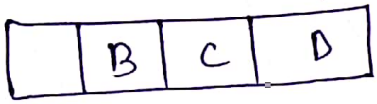
Q:



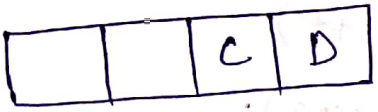
1)



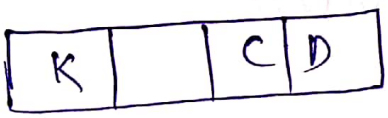
2)



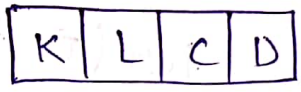
3)



4)



5)



6)

* Dqueue (Double-ended queue):-

A Dqueue is linear list in which elements can be added or removed at either end. but not in the middle.

Types:-

(a) Input restricted dqueue.

(b) output restricted dqueue.

(a) Input restricted dqueue:-

An input restricted dqueue is a D-queue which allows insertion at only one end of the list but allows deletion at both ends of the list.

(b) output restricted dqueue:-

An output restricted Dqueue is a Dqueue. which allows deletion operation at only one end of the list, but allows insertion at both the ends.

* Operations of the D-queue:-

1) Insertion of an element at the rear end.
2) Deletion of an element at the front end.
3) Insertion of an element at the front-end.
4) Deletion of an element at the rear end.

* priority Queue:-

It is a collection of elements such that each element has been assigned a priority and the order in which elements are deleted and processed comes from the following rules.

1) The element of an higher priority is ~~posse~~ processed over lower priority elemen
2) Two element with same priority are processed according to the order in which they were added to the queue.

Application:-

→ Jobs waiting to use printer on an LAN.

→ For checking the balanced parantheris, an expression is valied if it satisfais two conditions:-

(a) Number of left parantheris should equal to the total number of right parantheris in the equation.

(b) For every right parantheris, there should be a left parantheris of the same type

* Algorithm:-

1) Scan the symbole of expression from left to right
2) If the symbole is a left parantheris then push into the stack.

3) If the symbole is right paranthesis then, if the stack is empty then,

valid = false.

else pop the element from the stack. with

If the poped paranthesis does not match, the paranthesis being scan then,

valid = false.

4) After scanning all symbole if the stack is not empty then,

valid = false.

* Reverse of a string:-

```
main()
{
    char str[10];
    printf("Enter any string data\n");
    gets(str);
    for(i=0; i< strlen(str); i++)
    {
        push(str[i]);
    }
    for(i=0; i<strlen(str); i++)
    {
        str[i]= pop();
    }
    printf("Reverse string is : \n");
    puts(str);
}
```

* Tower of Hanoi :-
* conversion

→ Tower of Hanoi.
→ conversion of decimal to binary. } (Recursive function.)

Q) Consider a stack with maxsize = 4.
Perform the following operations:
push(1), push(2), push(3), push(4), push(5),
what is the output?

Linked-List

Array	Linked-List
→ It contains fixed memory location.	→ It is of variable length.
→ Easy to implement.	→ Little bit complicate to implement.
→ It has random access.	→ It has sequential access.
→ Chances of wastage of memory.	→ According to requirement memory is allocated so, no chance of waste of memory.
→ It has continious memory location.	→ memory is allocated randomly, sometimes it may contain congicutive location.

Dynamic memory allocation

→ Dynamic memory management techniques allow to allocate additional memory space or to release unwanted storage at run-time, They optimize the user storage space.

→ The memory management functions that can be use for allocation an freeing memory in program execution are:-

(i) malloc():- Allocates required size of bites.

(ii) calloc():- Allocates array of specified number of elements of a given size.

(iii) Realoc:- modefies the previously allocated space.

(iv) Free():- Releases a previously allocated space.

* Allocating a block of memory:-

→ The malloc() function can be used to allocated to block of memory. It reserves a specified size of memory and returns a pointer of type void.

- Syntax:-

```
ptr = (type cast *)malloc(byte(size));
```

Ex:- ptr = (int *) malloc(100*sizeof(int));

*Linked-List:-

It is an ordered collections of data in which each element contains the location of the next element.

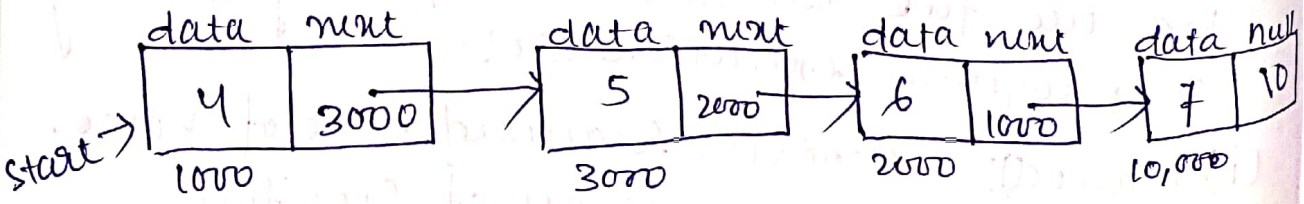
(or),

It is a collection of nodes in which each nodes has 2 parts:-

i) Data part, ii) Next part.

Eg:- A

4	5	6	7



→ Self Referential Structure:-

The nodes in the Linked-List are called "Self Referential Structures".

In this, each instance of the structure contains a pointer to another instance of the same structural type.

1) Single Linked-List:

Operation:-

a) Create Linked-List.

b) Insertion operation.

→ Insertion of a node at the begining of Linked-List.

→ Insertion of a node at the end of Linked-List.

→ Insertion of a node at a specification-location.

c) Deletion operation:-

→ Delete the first node.

→ Delete the last node.

→ Delete a node at Specific Location.

→ Traversing operation.

→ Searching operation.

→ Sorting operation.

→ Merging operation.

→ A "Single Linked-List" is a dynamic data structure and is a linear collection of data elements called "nodes" where each node is divided into two-parts:-

1) Data part 2) Next part.

A Linked-list is created using structures, pointers and dynamic memory allocation functions i.e, malloc().

* Node-declaration

```
struct node
{
    int data;
    struct node *next;
};
struct node *start;
```

(or)

```
typedef struct node NODE;
    NODE *start;
start = (struct node *) malloc (sizeof (struct node));
          NODE                          NODE
scanf ("%d", &start->data);
start -> next = NULL;
```

Q create a single Linked-list and display all nodes.

```
#include<malloc.h>
struct link
{
    int info;
    struct link *next;
};
void create L(struct link *);
void display (struct link *);
struct link
{   int data;
    struct link next;
};
```

```
int main()
{
    struct link *node, *start;
    node = (struct link*) malloc (size of (struct link));
    start = node;
    if (node == NULL)
    {
        printf ("out of memory\n");
        exit(0);
    }
    printf("Enter the data\n");
    scanf("%d", &node->data);
    node->next = NULL;
    create L(node);
    display (node);
}
void create L(struct link * node)
{
    char ch;
    printf("Enter the data\n");
    scanf("%d", &node->data);
    node->next = NULL;
    printf("Enter 'n' to exit else any key to
            continue\n");
    ch = getchar();
    while(ch != 'n')
    {
        node->next = (struct link*) malloc
                    (size of (struct link));
        if (node->next == NULL)
        {
            printf("out of memory\n");
            exit(0);
        }
```

```
        node = node -> next;
        printf("Enter the data \n");
        scanf("%d", &node -> data);
        node -> next = NULL;
        printf("Enter 'n' to exit else any key to
                continue\n");
        ch = getchar();
    }
}
void display (struct link *node)
{
    while (node != NULL)
    {
        printf ("%d\t", node -> data);
        node = node -> next;
    }
}
```

* Insertion of a node in a single linked list:-

→ The insertion operation is used to add an element in an existing linked list.

→ points to be remembered before make a insertion operation:-

(a) create a new node.

(b) If the list is empty, insert the node as the first node.

(c) If the node to be inserted at the beginning, insert the node and adjust the next pointer field.

(d) Insert the node at the desired place in the list.

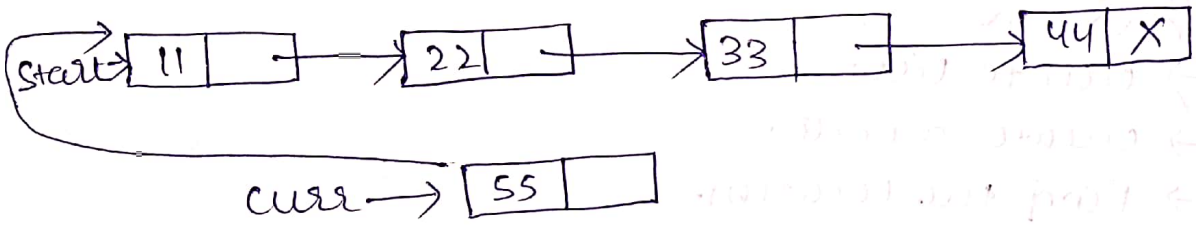
* various position where node can be inserted:-

1) Insert at begining.
2) Insertion at the end.
3) Insertion at specific location.
4) Insert a node after a given node.

1) Insertion of a node at the begining:-

* Algorithm:-

→ start holds the address of the first node.
→ create a node named curr.
→ If (curr == NULL). printf("out of memory\n") and exit
→ set curr → data = x.
→ set curr → next = start.
→ set start = curr.



Pseudo code:-

```
→ void InsertB (struct Link * node)
  {
      struct Link *curr;
      curr = (struct Link *) malloc (sizeof (struct Link));
      if (curr == NULL)
      {
          printf("out of memory\n");
          exit(0);
      }
      scanf("%d", &curr → data);
      curr → next = node;
      start = curr;
  }
```

2) Insert of a node at the end :-

* ~~pseudo code~~ *

* Algorithm :-

→ create a node named curr.
→ if (curr == NULL), printf("out of memory\n"); and exit.
→ set curr → data = x.
→ set curr → next = NULL.
→ while(node → next != NULL)
node = node → next.
→ node → next = curr;
→ Exit.

3) Insert of a node at specific location :-

* Algorithm :-

→ create list.
→ create a node.
→ Find the location.
→ Assign the data.
→ Find the previous node.
→ update the pointer.

* pseudo code :-

```
void insert (struct link * node)
{
    int loc, i=1;
    struct link * curr * prev;
    curr = (struct link *) malloc (sizeof (struct link));
```

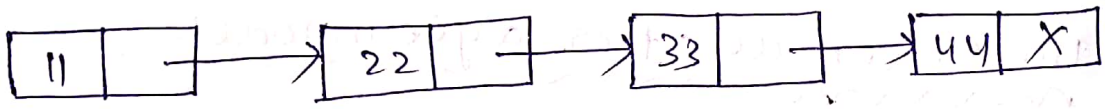
```
if(curr == NULL)
{
  printf("out of memory\n");
  exit(0);
}
printf("Enter the position\n");
scanf("%d", &loc);
printf("Input the value\n");
scanf("%d", &curr -> data);
while(i<loc)
{
   prev = node;
   node = node -> next;
   i++;
}
prev -> next = curr;
curr -> next = node;
}
```

4) Insert a node after a given node:-

```
void insert(struct link * node)
{
    int n;
    printf("Enter the element after which element
            to be inserted\n");
    scanf("%d", &n);
    while(node != NULL)
    {
        if(node -> data == n)
        {
            break;
        }
```

```
else
{
    node = node -> next;
}
if (node = NULL)
{
    printf("The element is not in the list\n");
    exit(0);
}
curr = (struct link *) malloc (sizeof (struct link));
printf("Enter the value\n");
scanf("%d", &curr -> data);
curr -> next = node -> next;
node -> next = curr;
}
}
```

* Deletion of a Node :-



1) Delete first node.
2) Delete last node.
3) Delete Specific location node.
4) Delete a node after a given data.

1) Deletion of first node :-

```
void deleteB(struct link *node)
{
    node = node -> next;
(or)
```

```
    if (node -> next = NULL)
    {
        struct = NULL;
        start = node -> next;
        free(node);
    }
}
```

2) Deletion of ~~node~~ last node:-

```
void deleteL (struct link * node)
{
    struct link * prev;
    if (node -> next == NULL)
    {
        start = NULL;
        free(node);
        return;
    }
    while (node -> next != NULL)
    {
        prev = node;
        node = node -> next;
    }
    prev -> next = NULL;
    free(node);
}
```

3) Deletion of a node at specific position:-

```
void delete s (~~xxxxxx~~ struct link * node)
{
    int i, loc;
    i = 1;
    struct link * prev;
```

```
printf("Enter the location of the element
        to be deleted \n");
scanf("%d", &loc);
while(i<loc)
{
    prev = node;
    node = node -> next;
    i++;
}
prev -> next = node -> next;
free(node);
}
```

4) Deletion of a node after a given data:-

Dt:- 19/02/2020

* Operations of Single Linked List Operation:-

1) Searching.
2) Sorting.
3) Reverse.
4) merging.

2) Sorting:- 

(a) Algorithm:-

Let start is pointing to the 1st node.

1) let node = start.
2) Repeat steps 3 to 10 while node→next != NULL
3) set ptr = node→next.
4) Repeat step 5 to 9 until ptr != NULL.
5) if (node→data > ptr→data)
6) set temp = node→data.
7) set node→data = ptr→data.
8) ptr→data = temp.

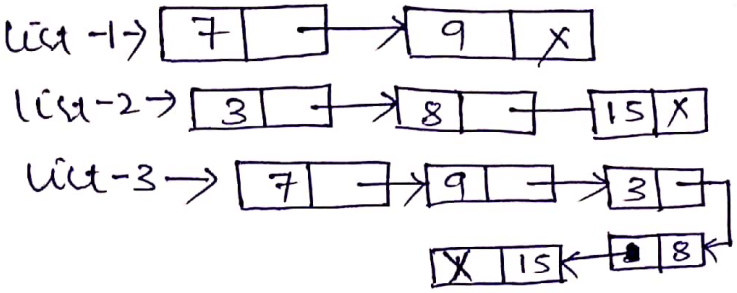
a) ptrc = ptrc → next.

b) node = node → next.

code:-

```
void sorting(struct link *node)
{
    int temp;
    struct link *ptrc;
    node = start;
    ptrc = node → next;
    while(node → next != NULL)
    {
        while(ptrc != NULL)
        {
            if(node → data > ptrc → data)
            {
                temp = node → data;
                node → data = ptrc → data;
                ptrc → data = temp;
            }
            ptrc = ptrc → next;
        }
        node = node → next;
    }
}
```

9) Merging :-



Algorithm :-

1. List 3 = List 1
2. while (List → next != NULL)
3. List = List → next;
4. List → next = List 2.

3) Reverse :-

* Algorith :-

Start will be pointing to the 1st node.

1) Set curr = start.

2) If curr → Next = = NULL then return.

3) prev = curr → next;

4) curr → next = NULL.

5) Repeat step 6 to 9 while (prev → next != NULL)

6) ptr = prev → next.

7) prev → next = curr.

8) curr = prev.

9) prev = ptr.

10) prev → next = curr.

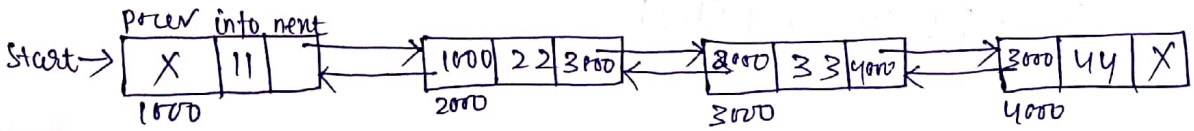
11) start = ptr.

* Double Linked-List:-

In single-Linked-list one can move starting from the first node to any node in one direction only in left to right, that is why it is termed as one-way list. on other hand a double linked-list is a two way list one can move in either direction, either from left to right or from right to left.

A two way list is a linier collection of data elements called nodes, where each node is devided into three parts.

1) Info (Information/data) field.
2) Next pointer (containing the address of next pointer)
3) previous pointer (containing the address of prev-pointer)



IMPLEMENTATION OF DOUBLE LINKED-LIST

```
struct d-link
{
    struct dlink *prev;
    int data;
    struct dlink *next;
};
void create_double(struct dlink*)
void display(struct dlink *)
int main()
{
    struct dlink *node;
    node = (struct dlink *)malloc(sizeof(struct dlink));
    if(node == NULL)
    {
        printf("out of memory \n");
        exit(0);
    }
    create_double(node);
    display(node);
}
void create_double(struct dlink *node)
{
    printf("Enter the data for the node\n");
    scanf("%d", &node->data);
    struct dlink *curr;
    node->prev = NULL;
    node->next = NULL;
    printf("Enter N to exit \n");
    scanf("%c", &ch);
```

```
        while(ch != n)
        {
            curr=(struct dlink*)malloc(sizeof(struct dlink));
            if(curr = NULL)
            {
                printf("Out of memory\n");
                exit(0);
            }
            node->next = curr;
            curr-> prev = node;
            node = node-> next;
            printf("Enter the data for node\n");
            scanf("%d", node-> data);
            node-> next = NULL;
            printf("Enter 'n' to exit\n");
            scanf("%d", &ch);
        }
}
void display(struct dlink *node)
{
    printf("The list is in the forward direction\n");
    Struct dlink *p;
    while(node != NULL)
    {
        p = node;
        printf("%d\t", node-> data);
        node = node-> next;
    }
    printf("The list is in the backward direction\n");
    while(p != NULL)
    {   printf("%d\t", p-> data);
        p = p-> prev;
    }
}
```