DATA Structures

Introduction :-

- Data may be organised in many different ways.
- A data structure is a particular way of organising data in a computer so that it can be used effectively.
- The idea is to reduce the space and time complexity of different task.

Ex- A list of items having same Data type can be stored using Array.

Classification of DS :-

Primitive Data Structure

→ int
→ float
→ char

Non-primitive Data Structure

In Linear DS, the data items are arranged in a linear sequence

Linear DS

→ Arrays, Linked list
→ Stack, queue

Non-linear DS

→ Tree
→ Graph

In Non-linear DS, data items are not arranged in sequence

DS Operations :-

(1) Create :- To create a DS
(2) Inserting :- Adding new records (details) to the structure.
(3) Deleting :- removing a record from the structure.
(4) Traversing :- Accessing each element for processing.
(5) Searching :- searching the record. (a particular element)
(6) Sorting :- Arranging records in some logical order.
(7) Merging :- (gets combined in a sorted manner)

Modifying
Copying
Concatinating
Splitting

Static DS

→ In Static DS, the size and structures associated memory locations are fixed at compile value.

Ex- Array

Dynamic DS

→ In Dynamic DS, it can expand or shrink as required during program execution and there associated memory locations can be changed.

Ex- Linked List created using pointers

DATA Structures

Introduction :-

- Data may be organised in many different ways.
- A data structure is a particular way of organising data in a computer so that it can be used effectively.
- The idea is to reduce the space and time complexity of different task.

Ex- A list of items having same Data type can be stored using Array.

Classification of DS :-

Primitive Data Structure

→ int
→ float
→ char

Non-primitive Data Structure

In Linear DS, the data items are arranged in a linear sequence

Linear DS

→ Arrays, Linked list
→ Stack, queue

Non-linear DS

→ Tree
→ Graph

In Non-linear DS, data items are not arranged in sequence

DS Operations :-

(1) Create :- To create a DS
(2) Inserting :- Adding new records (details) to the structure.
(3) Deleting :- removing a record from the structure.
(4) Traversing :- Accessing each element for processing.
(5) Searching :- searching the record. (a particular element)
(6) Sorting :- Arranging records in some logical order.
(7) Merging :- (gets combined in a sorted manner)

Modifying
Copying
Concatinating
Splitting

Static DS

→ In Static DS, the size and structures associated memory locations are fixed at compile value.

Ex- Array

Dynamic DS

→ In Dynamic DS, it can expand or shrink as required during program execution and there associated memory locations can be changed.

Ex- Linked List created using pointers

Abstract Data Types (ADT):-

→ An ADT is a mathematical model for data types where data is defined by its behaviour (possible values) and a set of possible operations on them.

→ An ADT refers to a set of data values and an associated operations on them.

Ex- int data type in C can be considered as an ADT which defines a set of numbers given by {-∞, ..., 0, 1, ..., ∞} and also specifies the operations that can be performed on them ie +, -, *, %, /.

Algorithms:-

→ An algorithm is a finite sequence of well defined instructions for solving a particular problem.

```
int add (int n1, int n2)
{ int s;
  s = n1 + n2;
  return s;
}
```

Algorithm analysis:-

→ Designing an efficient algorithm involves analysis of time and space complexities.

→ Time complexity:- amount of computer time the algorithm need to run (to) the complete the task.

→ Space complexity:- amount of memory the program needs to run.

Time complexity Measures (types):-

(a) Best case:- Best case time complexity of an algorithm is a measure of the minimum time that the algorithm will require for an input of size n.

(b) Worst case:- WC time complexity of an algorithm is a measure of the maximum time that the algorithm will require for an input of size n.

(c) Average case:- AC time complexity is the time that a algorithm will require to execute a typical input data of size n. i.e the value obtained by averaging the running time of an algorithm for all possible

inputs of size n.

Asymptotic Notations :-

→ Are the languages that allows to analyse algorithm's running time by identifying its behaviour as the input size increases.

→ This is also known as Algorithm Growth Rate.

1. Big Oh (O) Notation
2. Big Omega (Ω) "
3. Big Theta (Θ) " (Tight Bound)

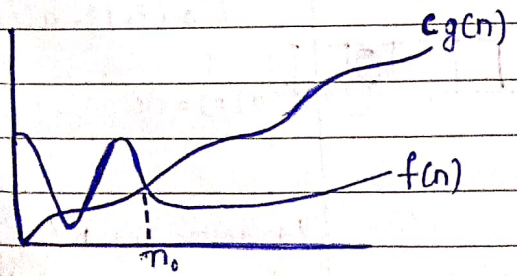
1. Big Oh (O) Notation :- Upper bound

→ defines an upper bound of an algorithm i.e a certain function will never exceed a specified time for any value of input n.

→ It represents the worst case of an algorithm.

For a non-negative function $f(n)$, we can say that

$O(g(n)) = \{ f(n) : \exists$ the constant c and n_0 such that $0 \le f(n) \le cg(n)$ $\forall\ n \ge n_0 \}$



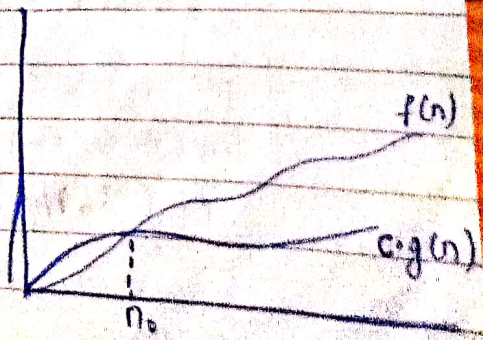
2. Big Omega (Ω) Notation :-

→ defines the lower bound of an algorithm i.e the minimum time required for any algorithm for all input values.

→ represents Best Case of an algorithm.

$\boxed{0 \le cg(n) \le f(n)}$ $\forall\ n \ge n_0$

$f(n) = \Omega(g(n))$

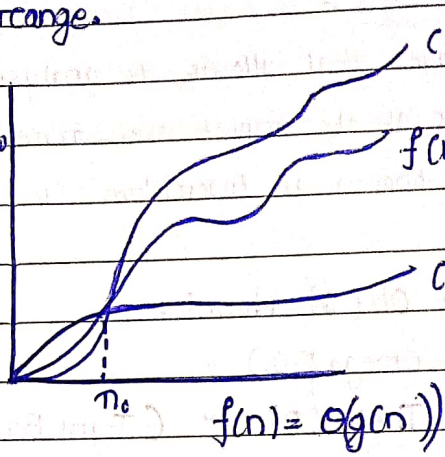


Homogenous - Similar type of data (array) elements
Heterogenous/Non-Homogenous - different type of data elements (structure) (record) (list)

3. Big Theta (Θ) Notations :-

→ It defines the average value or range within which the execution time of an algorithm will range.

$c_1 g(n) \le f(n) \le c_2 g(n) \quad \forall n \ge n_0$



Array :-

```
/* searching an element */
for (i=0; i<n; i++)
  if (x == ele  a[i])
    { pos = i;
      k++;
      break;
    }
if(k>0) pf("Element is present at %d", pos);
  else pf("not present.
```

```
/ sorting an element */
for(i=0; i<9; i++)
for (j=0; j<9-i, j++)
if (a[j] > a[j+1])
  { temp = a[j];
    a[j] = a[j+1];
    a[j+1] = temp;
  }
}
for(i=0; i<=9; i++)
  pf ("%d\n", a[i]);
```

```
/* insert beginning */
{ for(i=n; i>0; i--)
  { a[i+1] = a[i]
  } a[0] = ele;
}
```

```
/* insert specific */
{ for(i=n; i>=p; i--)
  { a[i+1] = a[i];
  }
a[p] = ele;
}
```

```
/* delete beg */
for(i=0; i<n; i++)
  { b[i] = b[i+1];
  }
n = n-1
```

```
/ * delete specific */
for(i=k; i<z; i++)
  { a[i] = a[i+1];
  }
z = z-1;
```

Sparse Matrix :- Most of the elements are zero.

Ex- $\begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 2 \end{bmatrix}$ → Sparse Matrix

	0	1	2	------	99
0	1	0	0	----	0
1	0	1	0	----	0
2					
59	0	0	1	----	0

60 × 4 = 240 bytes

Huge Memory Loss

60 × 100 = 6000 × 4 bytes

= 24,000 bytes

To avoid wastage of memory, an alternative representation is used which stores only non-zero elements in a triplet format

Sparse Matrix Triplet Form Representation :-

[row col val]

no. of columns = 3

no. of rows = no. of non-zero element + 1

	0	1	2
0	Total no. of rows in Sparse Matrix	Total no. of columns in Sparse Matrix	Total no. of non-zeros in Sparse Matrix
1			
2			
no. of non-zeros			

Ex:-

	0	1	2	3	4	5
0	1	0	0	0	0	0
1	0	0	0	0	0	0
2	0	0	2	0	3	0
3	0	0	0	0	0	0
4	0	5	0	0	0	0

5×6

Triplet form

5	6	4
0	0	1
2	2	2
2	4	3
4	1	5

5×3

Ex-

	0	1	2	3	4	5	6
0	0	0	0	2	0	0	0
1	0	1	0	0	0	0	0
2	0	0	0	0	0	0	0
3	0	0	5	0	3	0	0

4×7

Triplet form

4	7	4
0	3	2
1	1	1
3	2	5
3	4	3

5×3

TRIPLET PROGRAM

```
K=1;
for(i=0; i<row; i++)
  { for(j=0; j<col; j++)
    { if(a[i][j] != 0)
      { b[K][0] = i;
        b[K][1] = j;
        b[K][2] = a[i][j];
        K++;
      }
    }
  }
b[0][0] = row;
b[0][1] = col;
b[0][2] = K-1;
for(i=0; i<K; i++)
  { for(j=0; j<3; j++)
    { printf("%d", b[i][j]);
      \t
    }
    printf("\n");
  }
```

K is the no. of row of Sparse Matrix of triplet form

5	6	5
0	2	5
1	3	1
2	0	7
3	0	8
4	4	10

6x3

Triplet form

0	0	5	0	0	0
0	0	0	1	0	0
7	0	0	0	0	0
8	0	0	0	0	0
0	0	0	0	10	0

5x6

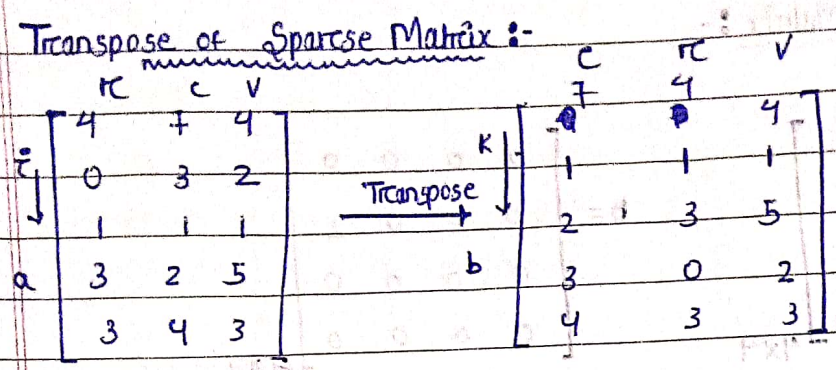
Triplet to original matrix :-

$$\begin{bmatrix} 4 & 7 & 4 \\ 0 & 3 & 2 \\ 1 & 1 & 1 \\ 3 & 2 & 5 \\ 3 & 4 & 3 \end{bmatrix} \xrightarrow[\text{form}]{\text{original}} \begin{bmatrix} 0 & 0 & 0 & 2 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 5 & 0 & 3 & 0 & 0 \end{bmatrix}_{4\times 7}$$

$$a = \begin{bmatrix} 3 & 3 & 2 \\ 0 & 0 & 5 \\ 1 & 2 & 11 \end{bmatrix} \xrightarrow[\text{form}]{\text{Original}} b = \begin{array}{c} \\ k \\ 0 \\ 1 \\ 2 \end{array} \begin{array}{c} \begin{array}{ccc} 0 & 1 & 2 \end{array} \\ \begin{bmatrix} 5 & 0 & 0 \\ 0 & 0 & 11 \\ 0 & 0 & 0 \end{bmatrix}_{3\times 3} \end{array}$$

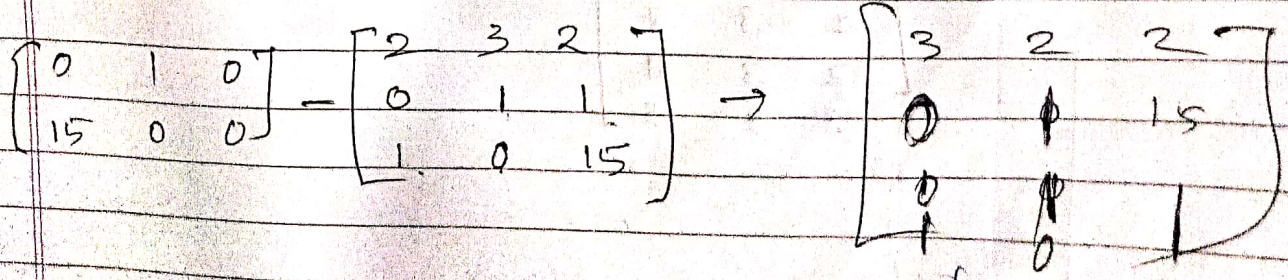
```
/* enter the triplet representation and convert it to original form */
row = a[0][0];
col = a[0][1];
k = 1;
for (i=0; i<row; i++)
{
  for (j=0; j<col; j++)
  { if (a[k][0] == i && a[k][1] == j)
    { b[i][j] = a[k][2];
      k++;
    }
  else
    b[i][j] = 0;
  }
}
```

Transpose of Sparse Matrix :-



The elements in [i][j] will move to [j][i]

```
rc = a[0][0];
c = a[0][1];
v = a[0][2];
K = 1;
b[0][0] = c;
b[0][1] = rc;
b[0][2] = v;
for (j=0; j<c; j++)
   { for (i=1; i<=v; i++)
      { if (a[i][1] == j)
         { b[K][0] = a[i][1];
           b[K][1] = a[i][0];
           b[K][2] = a[i][2];
           K++;
         }
      }
   }
```



Addition of Sparse Matrix :-

$$a = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 5 \\ 0 & 0 & 0 & 8 \\ 0 & 0 & 0 & 0 \end{bmatrix}_{4\times4} \qquad b = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}_{4\times4}$$

Triplet Representation

Triplet representation

$$m_1 = \begin{bmatrix} 4 & 4 & 3 \\ 0 & 0 & 1 \\ 1 & 3 & 5 \\ 2 & 3 & 8 \end{bmatrix} \qquad m_2 = \begin{bmatrix} 4 & 4 & 1 \\ 1 & 3 & 2 \end{bmatrix}$$

$$m_1 + m_2 = \begin{bmatrix} 4 & 4 & 3 \\ 0 & 0 & 1 \\ 1 & 3 & 7 \\ 2 & 3 & 8 \end{bmatrix}$$

Ex

$$m_1 = \begin{bmatrix} 4 & 4 & 3 \\ 0 & 0 & 1 \\ 1 & 3 & 5 \\ 2 & 3 & 11 \end{bmatrix} \qquad m_2 = \begin{bmatrix} 4 & 4 & 2 \\ 1 & 3 & 2 \\ 3 & 3 & 5 \end{bmatrix}$$

5-1

$$\begin{bmatrix} 4 & 4 & 4 \\ 0 & 0 & 1 \\ 1 & 3 & 7 \\ 2 & 3 & 11 \\ 3 & 3 & 5 \end{bmatrix}$$

Ex- $m_1 =$

7	7	6
0	5	1
1	4	1
3	4	3
4	1	2
5	4	4
6	4	2

$m_2 =$

7	7	9
0	0	1
0	4	9
1	4	2
2	1	4
2	5	8
3	0	6
3	2	7
5	0	5
6	4	9

$m_1 + m_2 =$

7	7	13
0	0	1
0	4	9
0	5	1
1	4	3
2	1	4
2	5	8
3	0	6
3	2	7
3	4	3
4	1	2
5	0	5
5	4	4
6	4	11

LS-2

Stack :-

→ It is a linear DS in which items are inserted or deleted only at one end called the top of the stack.

→ It follows LIFO (Last in Fast Out) mechanism where the last element inserted in the stack will be the first one to be removed from the stack.

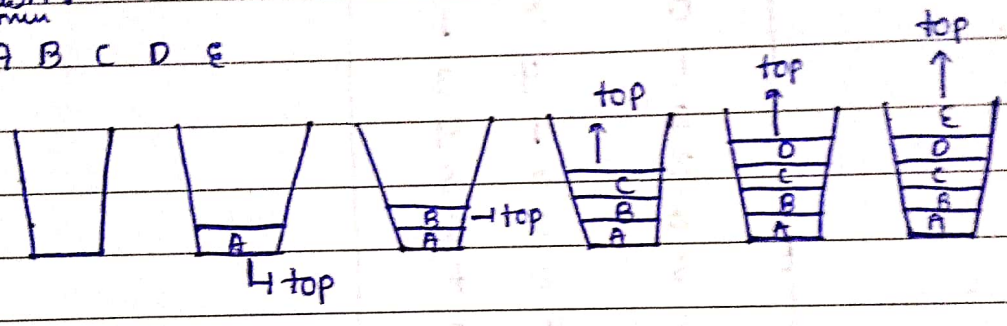
→ There are two basic operations associated with Stack :-

(1) Push - used to insert an element into the stack.

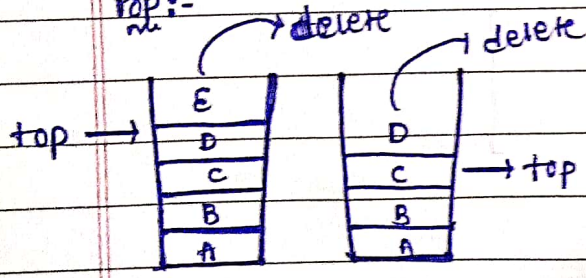
(2) Pop - used to delete an element.

Push :-

A B C D E

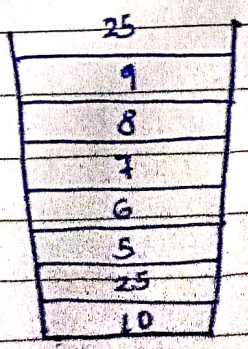


Pop :-



Ex :- For the given pattern find the contents of the stack after applying 4 insert, 3 delete operations.

10, 25, 5, 6, 7, 8, 9 (25, 60, 70, 80)



Overflow:- trying to insert element into the stack after maximum size is reached.

Underflow :- trying to delete elements from an empty stack.

Q. For the following sequence of operations, what will be the sequence of popped out values?

Push(1), Push(2), pop, push(1), push(2), pop, pop, pop, push(2), pop

Ans:- 2 2 1 1 2

Representation of Stack :-

Stack can be represented in 2 ways :-

(a) Array

(b) Linked list

Array representation of Stack :-

```
# include<stdio.h>
# include<stdlib.h>
void push();
void pop();
void display();
# define max 50
int stack[max];
int top=-1;
// int c=0;
main()
{ int ch;
 while(1) {
printf("1-push\n 2-pop\n 3-display\n 4-exit\n");
scanf("%d",&ch);
 Switch(ch)
      { case 1 :
                push();
                // c++; break;
```

```
        case 2:
            pop();
            break;
        case 3:
            display();
            break;
        case 4:
            exit(0);
        default:
            printf("Invalid option\n Enter a valid option\n");
    }
  }
}

void push()
{  int ele;
   if (top==max-1)
    printf("overflow");
   else {
        printf("Enter the element to be inserted\n");
        scanf("%d", &ele);
        top++;
        stack[top] = ele;
    }
}
void pop()
{  if (top==-1)
    printf("Underflow");
    else {
        int item = stack[top];
        top--;
        // c--;
        printf("%d is poped out\n", item);
    }
```

can be written as

```
stack[++top] = ele;
```

```
Void display ()  { int i;
    if (top == -1)
    printf ("stack is empty \n");
    else {
        pf ("Elements of stack are \n");
       // for (i=0; i<C; i++)    for (i=0; i<=top; i++)
          {
           printf ("%d \t", stack [i]);
          }
    }
}
```

* To print the stack elements from top to bottom :-

```
void display
{  int i;
   if (top == -1)
   pf ("Stack is empty \n");
   else
    { pf ("Elements of stack are \n");
      for (i= top; i >=0; i--)
        { pf ("%d \t", stack [i]);
        }
}
```

```
Void display ()  { int i;
    if (top == -1)
    printf ("Stack is empty \n");
    else {
        pf ("Elements of stack are \n");
       // for (i=0; i<c; i++)    for (i=0; i<=top; i++)
          {
           printf ("%d \t", stack[i]);
          }
      }
    }
```

* To print the stack elements from top to bottom :-

```
void display
{ int i;
  if (top == -1)
  pf ("Stack is empty \n");
  else
  { pf ("Elements of stack are \n");
    for (i= top; i>=0; i--)
      { pf ("%d \t", stack[i]);
      }
}
```

Algorithm for push operation:-

Step-1 :- If top = max size of the stack, then print overflow and return

Step-2 :- Otherwise read the element to be inserted. Increase top by 1 i.e top = top+1.

Step-3 :- Store the element at stack[top] position and return.

Algorithm for pop operation:-

Step-1 :- If top = -1 (Stack is empty), then print underflow and return.

Step 2 :- Otherwise set item = stack[top]. Decriment top by 1 i.e top = top-1 and then return

```
.
.
.
main()
{ int x, z;
  while(1)
  { pf("....");
    sf("%d", &ch);
  }  Menu driven part
  switch(ch)
  { case 1:
    pf("Enter the element to be pushed");
    sf("%d", &x);
    push(x);
    break;
  Case 2:
    z = pop();
    pf("%d is popped out", z);
    break;
  .
  .
  .
}
}
```

```
void push (int x)
{ // cond^n for overflow
   top++;
   stack[top] = x;
}
int pop()
{ // underflow cond^n
   int y;
   y = stack[top];
   top--;
   return y;
}
```

Application of Stack :-

(a) Reversing string
(b) Evaluation of Arithmatic expression
(c) Conversion of one form of arithmatic operation to another
(d) Recursion.

Reversing string using Stack :-

```
void push(char);
char pop();
#define max 50
char stack[max];
int top = -1;
main()
{  int i;
   char str[30];
   pf(" Enter the string");
   gets(str);
   for(i=0; i<strlen(str); i++)
```

```
{  push (str[i]);
}
for (i=0; i< strlen (str); i++)
 {
      str [i] = pop ( );
 } Pf(" Retrived string is ");
   puts (str);
}
Void push (char ch)
{  if (top == max)
    Pf(" [redacted] Overflow");
   else { top++;
          stack [top] = ch;
        }
}

Char pop ( )
  { // underflow cond^n.   char c;
    [redacted]              c = stack [top];
                           top--;
                           return c;
    }
```

Decimal to binary conversion using stack:-

```
void push (int);
int pop ();
# define max 50
    int stack [max];
    int top = -1;
main()
{ int i, num, rem, c = 0
  pf ("Enter a numbere");
  sf ("%d", &num);
  while ( num != 0)
    { rem = num % 2;
      push( rem);
      num = num / 2;
      c++;
    }
  i = 0;
  while (i < c)
    { [redacted] int a;
      a = pop( );

Void push ( int n)
 {
    top ++;
    stack [top] = n;
 }
int pop ( )
 {
```

Notations for arithmatic operations (expressions):-

There are 3 notations used to represent arithmatic expressions

(a) Infix :- operator is placed betn operands.

Ex- A + B → operands
↳ operator

(b) Postfix :- operators is placed after the operands.

Ex- AB+, CD-, GH*

(c) Prefix :- operator is placed before the operands.

Ex- +AB, -CD, *GH (Also called Polish Notation)

Ex- (A+B)*C → Infix

AB+C* → Postfix

Ex - (A+B)/(C-D) → Infix

AB+CD-/ → Postfix

Ex- A+B/C-D → Infix

ABC/+D- → postfix

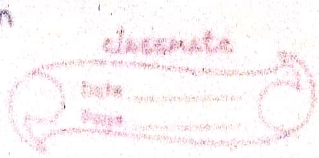
Ex- (A+B)*(C+D) → AB+CD+*

Addⁿ of Sparse Matrix

```
main()
{

  if(m1[0][0] != m2[0][0] || m1[0][1] != m2[0][1])
   { printf("Addⁿ not possible");
     exit(0);
   }
  nzero1 = m1[0][2];
  nzero2 = m2[0][2];
  k1 = k2 = k3 = 1;
  int c = 0;
while( k1 < nzero1 + 1 && k2 < nzero2 + 1)
  {
    if(m1[k1][0] < m2[k2][0])
     { m3[k3][0] = m1[k1][0];
       m3[k3][1] = m1[k1][1];
       m3[k3][2] = m1[k1][2];
       k3++;
       k1++;
     }
else if(m1[k1][0] > m2[k2][0])
     { m3[k3][0] = m2[k2][0];
       m3[k3][1] = m2[k2][1];
       m3[k3][2] = m2[k2][2];
       k3++;
       k2++;
     }
else if((m1[k1][0] == m2[k2][0]) && (m1[k1][1] < m2[k2][1]))
     { m3[k3][0] = m1[k1][0]
       m3[k3][1] = m1[k1][1]
       m3[k3][2] = m1[k1][2]
         k3++;
       k1++;
     }
```

```
else if((m1[k1][0] == m2[k2][0]) && (m1[k1][1] > m2[k2][1]))
{   m1[k3][0] = m2[k2][0];
    m1[k3][1] = m2[k2][1];
    m1[k3][2] = m2[k3][2];
        k3++;
        k2++;
}
else {
    m3[k3][0] = m2[k2][0];
    m3[k3][1] = m2[k2][1];
    m3[k3][2] = m2[k2][2] + m1[k1][2];
    c++;
    k3++;
    k2++;
    k1++;
}
m3[0][0] = m1[0][0];
m3[0][1] = m1[0][1];
m3[0][2] = (nzero1 + nzero2) - c;
```

(A+B) → parenthesis to form



Arithmetic expressions :-

Ex- A + B * C - D / E * H ⟶ ABC * + DE / H * -

Input Symbol	Output position String	Stack
A	A	
+		+
B	AB	+
*	AB	+ *
C	ABC	+ *
-	ABC * +	-
D	ABC * + D	-
/		- /
E	ABC * + DE	- /
*	ABC * + DE /	- *
H	ABC * + DE / H	- *
Input empty	ABC * + DE / H * -	

Algorithm for infix to postfix :-

Step 1:- Scan the infix expression from left to right

Step 2:- (a) If ~~stack~~ scanned symbol is a left parathesis, push it onto the stack

(b) If the scanned symbol is an operand, then place it directly in the postfix expression (output)

(c) If its a right parathesis, then pop all elements from the stack and place them in postfix expression till a matching left parenthesis is obtained.

(d) If its an operator, then (i) remove all operators from stack having higher precedence than the scanned one and then place it in postfix expression.

(ii) Otherwise, push the scanned operator onto the stack.

Step 3:- If the input is empty, pop out all stack elements and place in the output postfix expression.

(A+B) → parenthesis form

Arithmatic expressions :-

Ex- A+B*C - D/E*H ⟶ ABC*+DE/H*-

Input Symbol	Output position string	Stack
A	A	
+		+
B	AB	
*	AB	* +
C	ABC	* +
-	ABC*+	-
D	ABC*+D	
/		/ -
E	ABC*+DE	
*	ABC*+DE/	* -
H	ABC*+DE/H	
Input empty	ABC*+DE/H*-	

Algorithm for infix to postfix :-

Step 1:- Scan the infix expression from left to right

Step 2:- (a) If scanned symbol is a left parenthesis, push it onto the stack

(b) If the scanned symbol is an operand, then place it directly in the postfix expression (output)

(c) If its a right parenthesis, then pop all elements from the stack and place them in postfix expression till a matching left parenthesis is obtained.

(d) If its an operator, then (i) remove all operators from stack having higher precedence than the scanned one and then place it in postfix expression.

(ii) Otherwise, push the scanned operator onto the stack.

Step 3:- If the input is empty, pop out all stack elements and place in the output postfix expression.

Infix to prefix conversion :-

Ex- $(A+B)*C \longrightarrow *+ABC$
Infix → prefix

Ex- $(A+B)/(C-D) \rightarrow /+AB-CD$

Ex - $A+B/C-D \rightarrow -+A/BCD$

Evaluation of Postfix expression :- (Algorithm)

Step 1- Scan the expression (input) from left to right

Step 2- If its an operand, push it onto the stack.

Step 3- If its an operator, pop out two elements and apply the operator and push the result on the stack.

Step 4- Write the result if the input is empty.

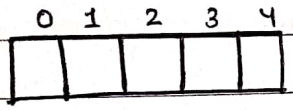
Ex - 6523+8*+3+*

Symbol	operands	operands	value	Stack
6				6
5				5 6
2				2 5 6
3				3 2 5 6
+	3	2	5	5 5 6
8				8 5 5 6
*	5	8	40	40 5 6
+	40	5	45	45 6
3				3 45 6
+	45	3	48	48 6
*	48	6	288	288

Answer = 288

Queue:- A queue is a linear list of elements in which insertion can take place only at one end called rear and deletion can take place only at the other end called front.

→ Queue follows FIFO Mechanism (First in First out) where the first element inserted into the queue will be the first element to be removed from the queue.



Queue

front = 0

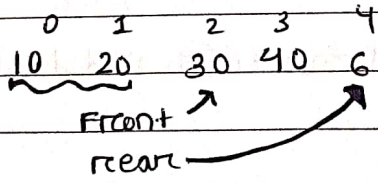
rear = -1

1. Enqueue:- Add elements into the queue. // rear = rear + 1
2. Dequeue:- Remove elements from the queue. // front = front + 1

Ex:-

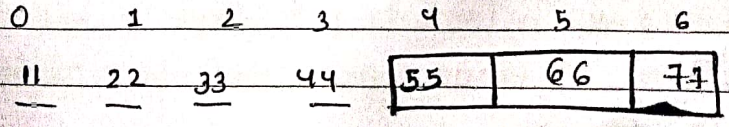
Assuming max. size of linear queue = 6. what is the value of front and rear after the following operations? (insert(10), insert(20), insert(30), delete(), delete(), insert(40), insert(60))

Ans:-



Ex- What is the content of the queue after applying operations (delete(), delete(), delete(), insert(66), delete(), insert(77))

Ans:-



Ex- If initial configuration of the queue is a, b, c, d and 'a' is the front end, to get the configuration d, c, b, a in the queue, how many min. no. of delete and insert function are requeired.

a, b, c, d

d c b a

Ans:- 3 delete(), 3 insert

Algorithm (Implementation of linear queue using array):-

(a) For inserting:-

Step-1:- If queue is full (rear = size-1), print overflow and return.

Step-2:- Otherwise increase rear by 1 and store the element to be inserted at queue[rear] and return.

(b) For deletion:-

Step-1:- If queue is empty (rear = -1), print underflow and return.

Step 2:- Otherwise, remove the element from queue[front] and increase front by 1 (front = front + 1) and then return.

```
#include<stdio.h>
#define max 50
int queue[max];
int front=0;
int rear = -1;  void insert(); void delete();
main()
{
   Menu Driven Program
}
Void insert()
{ int [redacted] ele;
  [redacted]
   if(rear == max-1)
   pf(" Overflow");
else { pf(" Enter the element to be inserted");
                                    sf("%d",&ele)
       rear++;
       queue[rear] = ele;
   }
}
```

```
void delete()
{  if (rear == -1)
     pf("Underflow");
   else {
          int item = queue[front];
          front ++;
          pf("%d is poped/deleted out \n", item);
        }
}
```

Note:- If rear == front, it indicates that there is only one element in the queue.

Limitation of linear queue :-

→ When rear = max - 1, it indicates overflow condition i.e the queue is full.

→ Once the queue is full, if some elements are deleted from the front, then some free spaces are available in the queue.

→ But these free spaces cannot be reused for storing new elements as rear = max - 1 which indicates overflow situation.

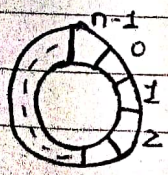
Solutions:-

→ Shifting all remaining elements in the queue towards front and updating rear position. However, it is not an effective method as shifting of all remaining elements each time a deletion is performed will be a time consuming process.

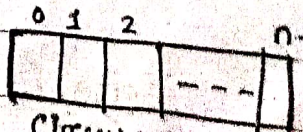
→ Convert linear queue into circular queue.

Circular queue:-

→ A circular queue is a queue in which the last and first positions are considered as adjacent position i.e the element next to the last element is the first element.

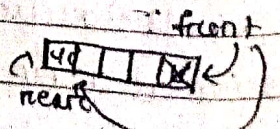


Circular queue (logical)



Find $\int_{-4}^{-5} e^{(x+5)^2} dx + 3\int_{1/3}^{2/3} e^{9\left(x-\frac{2}{3}\right)^2} dx$

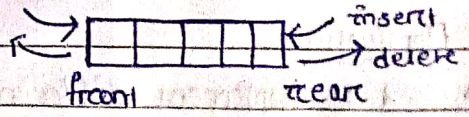
```
Void insert ()  {  int ele;  // Enter the element
  if ((front == 0 && rear == max-1) || (front > 0 &&
   rear == front-1))
   pf("Overflow");
  else {
        if ( front > 0 && rear == max-1 )
           { rear = 0;
             queue[rear] = ele;
           }
        else
           { rear ++;
             queue[rear] = ele;
           }
     }
  }

Void delete ()
{
  if ( front == 0 && rear == -1)
   pf("Underflow");
  if ( front == rear )    // one element
    { item = queue[front];
      front = 0;
      rear = -1;
    }
  else {
       if (front == max-1)
         { item = queue[front];
           front = 0;
         }
  else { item = queue[front];
         front ++;
```



```
pf("%d is deleted from the queue", item);
}
void display()
{  if (front == 0 && rear == -1)
    pf("Queue is empty");
    if (front > rear)
    {  for (i=0; i<=rear; i++)
         pf("%d\t", queue[i]);
       for (j=front; j<=max-1; j++)
         pf("%d\t", queue[j]);
    }
   else {
       for (i=front; i<=rear; i++)
       {  pf("%d\t", queue[i]);
       }
    }
}
```

Deque:- (Double ended queue)

A deque is a linear list in which elements can be added or removed at either ends but not in the middle.

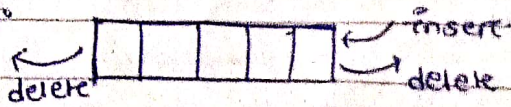


→ There are 2 variations of deque :-

(a) Input restricted deque

(b) Output restricted deque

(a) An Input restricted deque allows insertion at only one end of the list but allows deletion at both ends of the list.



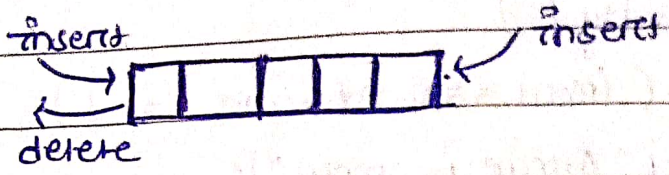
1- insert
2- delete front
3- delete rear
4- display
5- exit

} Menu

(b) An output restricted deque allows deletion at only one end of the list but allows insertion at both the ends.

1- insert front
2- insert rear
3- delete
4- display
5- exit

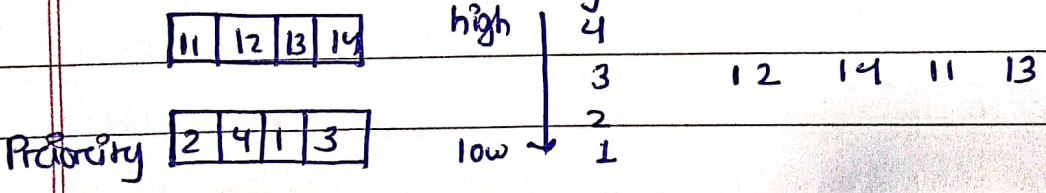
Menu



Priority Queue :-

A priority queue is a collection of elements where each element is associated with a priority value. The elements are deleted and processed using the following rules :-

→ An element with higher priority is processed before any element of lower priority.



→ Two elements with same priority are processed according to the order in which they are added to the queue

Application :-

→ Used to implement different CPU scheduling algorithms in multiprogramming and time sharing systems.

LL

Limitations of array :-

- Size needs to be defined when program is written, therefore additional space cannot be provided if required.
- Insertion and deletion in array are expensive as each insertion and deletion requires shifting of remaining elements.

Soln :-

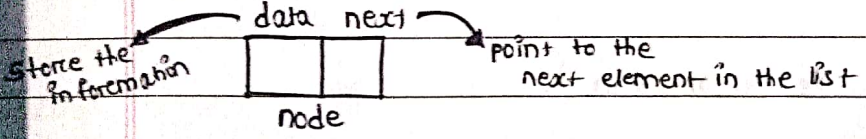
- Dynamic link representation of list

Linked List :-

→ It is a dynamic data structure where :-

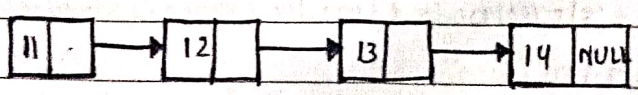
- amount of memory required can vary during use.
- adjacency betn the elements is maintained by links/pointers i.e the address of the subsequent elements.

In a linked list, data and link, both are required to be maintained. An element in a linked list is called a 'node'. A node consist of two fields i.e 'data' and 'next'



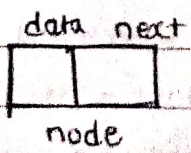
① Single linked list :- (one way list)

- Each node in SLL contains a data field and only one pointers field which will point to the adjacent node in the list



Structure of a node :-

```
Struct node
{ int data;
  Struct node * next;
};
```



Allocating blocks of memory to a node :-

```
struct node * newnode;
newnode = (struct node *) malloc (sizeof (struct node));
```

Program to create and Traverse a linked list :-

```
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>
struct node {
        int data;
        struct node * next;
    } * start = NULL, *newnode, * current;
void create ();
void display ();
void main ()
{ create ();
  pf ("The elements of the list are : \n");
  display ();
}
void create ()
{ char ch;
  do { newnode = (structnode *) malloc (sizeof (structnode));
  pf ("Enter the data \n");
  sf ("%d", &newnode → data);   or { int x;
                                     sf ("%d", &x)
                                     newnode → data = x;
  newnode → next = NULL;
  if (start == NULL)
  { start = newnode;
    current = newnode;
  }
else { current → next = newnode;
       current = newnode;
```

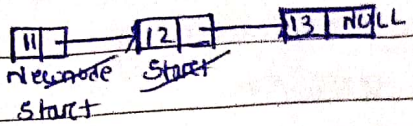
```
pf(" Press n to stop\n");
ch=getchare();
} while(ch != 'n');
}
Void display()
{   Struct node *ptr;
    ptr = start;
    While(ptr != NULL)
    {
        pf("%d\t", ptr->data);
        ptr = ptr->next;
    }
}
```

Menu driven program for insertion and deletion :-

```
#_ _ _ _ _ _ _
Struct node
   { int data;
     Struct node *next;
   } *start, *current, *newnode;
main()
{  int x, ch;

   newnode = (structnode*) malloc (sizeof (Structnode));
   pf("Enter the data for first node");
   sf("%d", &x);
   newnode -> data = x;
   newnode -> next = NULL;
   start = newnode;
   current = newnode;
   while(1) {
```

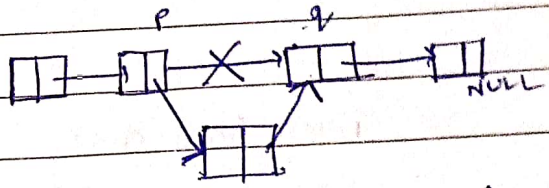
```
pf("Enter choice\n");
pf("1- insert end \n 2 - insert beginning \n 3- display \n 4- exit \n");
sf("%d",&ch);
switch(ch)
{   case 1: insertend();
            break;
    Case 2 : insertbeg();
            break;
    Case 3 : display();
            break;
    Case 4 : exit();
            break;
    default : pf("Enter a valid choice");
            break;
 }
}

void insertend()
{ int x;
newnode = (struct node *) malloc (sizeof(struct node));
pf("Enter the data \n");
sf("%d",&x);
 newnode → data = x;
        newnode → next = NULL;
        current → next = newnode;
        current = newnode;
  }
void insertbeg()
 { int x;
  newnode = (struct node *) malloc (sizeof(struct node));
  pf("Enter the data \n");
  sf("%d",&x);
    newnode → data = x;
```

```
newnode → next = start;
start = newnode;
}
```



Insert after a specific value :-

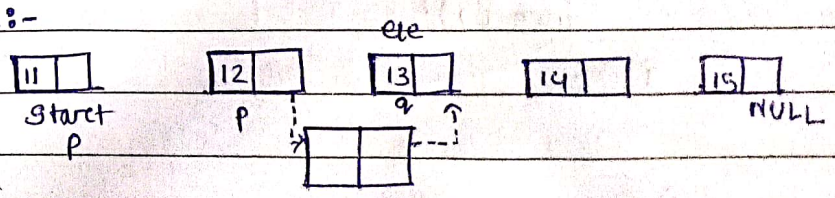
```
void insertafter()
{  int ele;
   struct node *p, *q;
   p = start;
// * enter the data which you want to insert * //
while (p → data != ele && p → next != NULL)
   { p = p → next;
   }
   q = p → next;
newnode : (struct node) * malloc (sizeof (struct node));
newnode → data = x;
p → next = newnode;
newnode → next = q;
```




```
printf("Enter the pos")
sf
printf("Enter the elemen
```

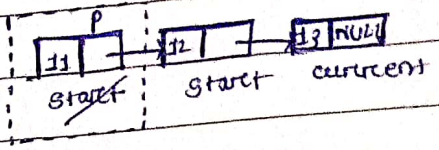
Insert before a specific value :-

```
while (p → next → data != ele)
{
   p = p → next;
}
q = p → next;
```



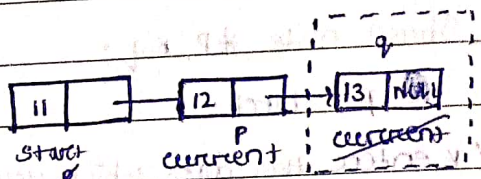
Delete from beginning :-

```
Void delete beg()
{ Struct node *P;
   p = start;
   start = start→next
   free(p);
}
```



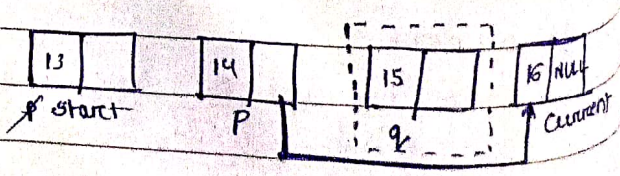
Delete from end :-

```
Void deleteend()
{
   Struct node *p, *q;
     p = start;
   while (p→next != current)
     {
       p = p→next;
     }
   q = p→next;
   p→next = NULL;
   current = p;
   free(q);
}
```



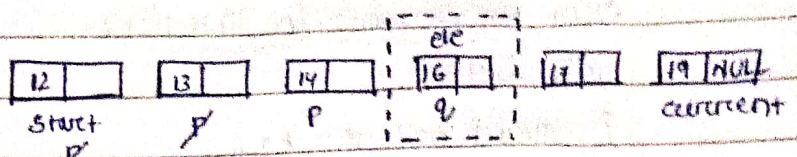
Delete after value :-

```
void deleteafter()
{ Struct node *P, *q;
    p = start;
  while (p→data != ele)
    { p = p→next;
    }
    q = p→next;
    p→next = q→next;
   free(q);
```



Delete a specific value:-

```
void deletevalue()
{   struct node *p, *q;
    p = start;
  while(p->next->data != ele)
    {  p = p->next;
    }
    q = p->next;
    p->next = q->next;
    free(q);
}
```



Q. To count the no. of nodes in a linked list (function)? void nodes() {

```
  struct node *p; , p = start;
 int c = 1;
  while(p->next != NULL)
  { p = p->next;
    c++;
   }
 // print c
}
```

Searching a specific element:-

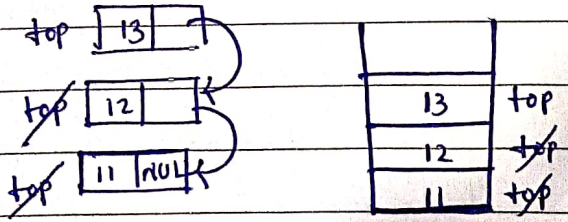
```
void search()
{  struct node *p; int ele;
    p = start;
  // enter the element to be searched
  while(p != NULL)
    { if(p->data == ele)
    { // element found
      // return
    }
    p = p->next;
  }
    pf("element not found");
}
```

Sort the elements in link list :-

```
void sort()
{  struct node *p, *q;
   int temp;
   for(p=start; p->next != NULL; p=p->next)
   for(q=p->next; q != NULL; q=q->next)
     { if(p->data > q->data)
            { temp = p->data;
              p->data = q->data;
              q->data = temp;
            }
     }
}
```

* reversing
* Concatination / Merging

Link list representation of Stack :-



Push :-

```
struct node {
    int data;
    struct node *next;
    } *top = NULL, *newnode;
void push() int x;
{   newnode = (struct node *) malloc(sizeof(struct node));
    newnode->data = x;          // Enter the value of x
    newnode->next = top;
    top = newnode;
}
```

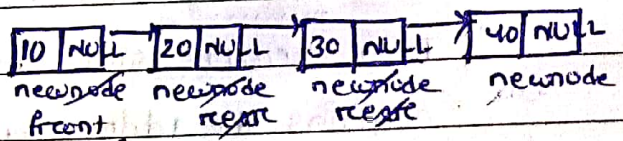
```
{ struct node *p;
  p= top;        // pf("%d is deleted from stack", top→data);
  top= top→next;
  free(p);
}
void display()
{ struct node *p;
  p= top;
 while (p != NULL)
  {
     // print p→data
     p= p→next;
  }
}
```

top → [12] → [11 | NULL]

```
if (top == NULL)
    pf ("stack is empty");
```

Link list representation of Linear Queue :-

```
struct node {
            int data;
            struct node *next;
          } *newnode, *front = NULL, *rear= NULL;
```

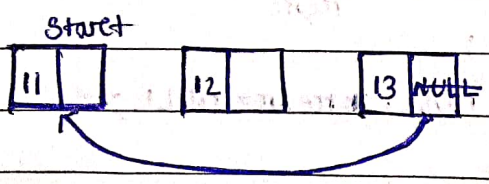



```
void insert()  int x;
{ newnode = (struct node *)malloc(sizeof(struct node));
  // Enter the value of x
  newnode → data = x;
  newnode → next = NULL;
  if (rear == NULL)
  { front = newnode;
    rear = newnode;
  }
  else {
```

```
void delete()
{ struct node *p;
    p = front;
    front = front -> next;
    item = p -> data;
    pf("%d is deleted", item);
}
void display()
{ struct node *p;
    p = front;
  while(p != NULL)
{ pf("%d\t", p -> data)
    p = p -> next;
}
}
```

Circular Single Linked List :-

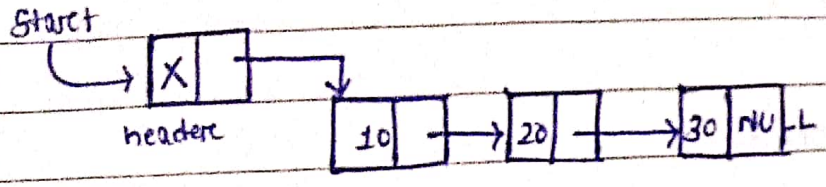
In a circular single linked list, the next field of last node contains the address of 1st node instead of NULL pointer.



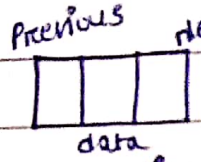

```
void insertend()
{ ---------
  newnode -> data = x;
  newnode -> next = start;
  ---------
}
Void display()
{ struct node *p;
    p = start;
    do
    { pf("%d\t", p -> data);
      p = p -> next;
    } while(p != start);
}
```

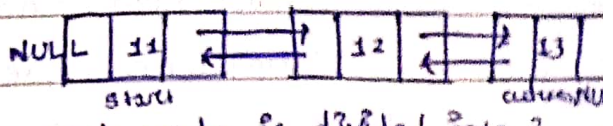
Header Linked List :-

A header linked list is a list which always contains a special node called the header node at the beginning of the list.



Double Linked List :-





- A double linked list is a two way list where each node is divided into 3 parts :- 1. Data
  2. Next – points to the next node in the list
  3. Previous - points to the previous of the list
- The previous of 1st node and next of last node contains NULL which indicates the beginning and end of the list.
- The advantage of Double LL over Single LL is that, traversal is possible in both directions in case of Double LL.

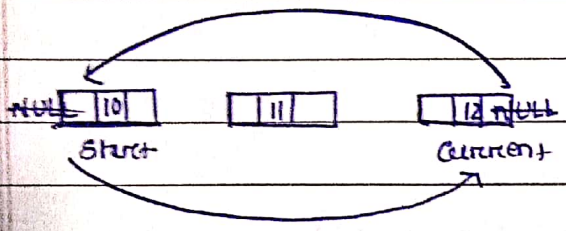
```
Struct node {
        int data;
        Struct node *next, *prev;
    } *start, *current, *newnode;
main() { int x;
    newnode = (Struct node*) malloc (sizeof(Struct node));
    // Enter the value of x
    newnode → data = x;
    newnode → next = NULL;
    newnode → prev = NULL;
    Start = newnode;
    current = newnode;
While(1) {
        Menu driven part (including switch)
}
```

```
void insertBeg()
{   int x;
    newnode = (struct node *) malloc(sizeof(struct node));
    //Enter the value of x
    newnode -> data = x;
    newnode -> prev = NULL;
    newnode -> next = start;
    start -> prev = newnode;
    start = newnode;
}
void insertEnd()
{ int x;
  newnode = (struct node *) malloc (sizeof(struct node));
   //Enter the value of x
  newnode -> data = x ;
  newnode -> next = NULL;
  newnode -> prev = current;
  current -> next = newnode;
  current = newnode;
 }
void Traverseforeword ()
{  struct node *p;
     p = start;
   while (p != NULL)
   {  pf ("%d \t", p -> data);
     p = p -> next;
   }
}
void Traverseback ()
{  struct node *p;
    p = current;
  while (p != NULL) { pf ("%d \t", p -> d
```

```
void deleteBeg()
{ struct node *p;
  p = start;
  start = start->next;
  free(p);
  start->prev = NULL;
}
```

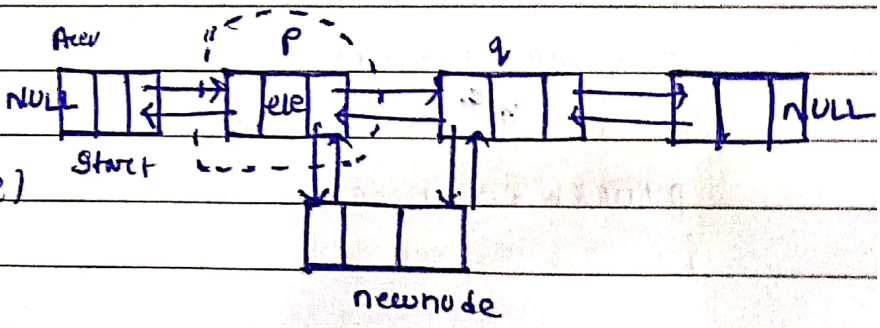
```
void deleteEnd()
{ struct node *p;
  p = current;
  current = current->prev;
  free(p);
  current->next = NULL;
}
```

Circular Double linked List :-




```
start->prev = current;
current->next = start;
```

Insert after specific value :-




```
void insertafter()
{ // Enter value after
  which you want to insert (ele)
  struct node *p, *q;
    p = start;
  while(p->data != ele)
    { p = p->next;
    }
  q = p->next;
  newnode = (struct node *) malloc(sizeof(struct node));
  newnode->data = x;
  p->next = newnode;
  newnode->prev = p;
  newnode->next = q;
  q->prev = newnode;
}
```

Q. Write a user defined function to insert after the ith node in a single linked list.

```
Void insertafterpos()
 { int x,i,pos;
   struct node *p;
   p=start;
   i=0;
   // Enter the node posn
  While(p->next != NULL && i < pos)
     { p=p->next;
       i++;
     }
  newnode =(struct node *) malloc (sizeof(struct node))
  // Enter data for newnode
  newnode->data =x;
  newnode->next=p->next;
  p->next=newnode;
 }
```

to delete after the ith node (user defined function) in single linked list.

Largest element in single linked list (user defined function)

```
Void largest()
 { struct node *p;
   p=start;
   int max;
   max = p->data;
  While(p != NULL)                    // for (p=start; p!=NULL; p=p
   { if(p->data > max)
          max=p->data;
          p=p->next;
  }
```

printf("Maximum = %d", max);
}

	0	1	2	3	4	5
0	8	0	0	0	0	9
1	0	3	0	0	0	0
2	0	5	0	0	2	0
3	4	0	0	0	0	7
4	0	0	7	0	0	0

Sparse

→

5	6	8
0	0	8
0	5	9
1	1	3
2	1	5
2	4	2
3	0	4
3	5	7
4	2	1

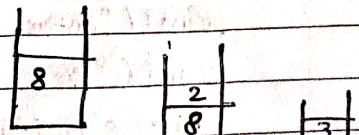
Triplet

↓

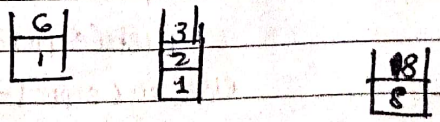
6	5	8
0	0	8
0	3	4
1	1	3
1	2	5
2	4	7
4	2	2
5	0	9
5	3	1

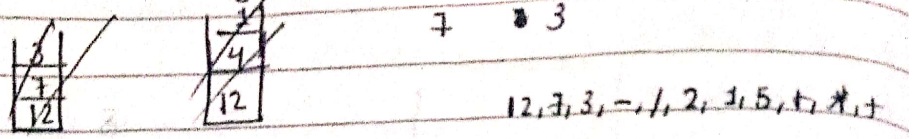
Treanspose

Q. Find the top two elements of the stack after first multiplication operation for the given postfix expression :- 823^/23*+51*-



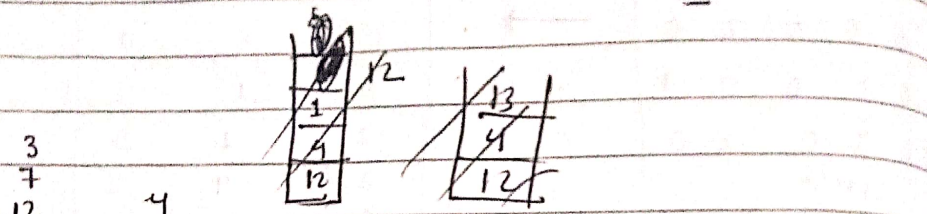
Ans:-



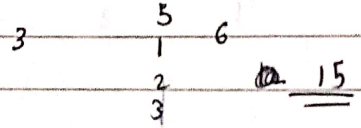


12, 3, 3, -, 1, 2, 3, 5, +, *, +

= 15



12 4



= 15

Infix to postfix :-

To check if the expression is balanced or not w.r.t parenthesis

((a+b)) → valid exp

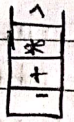
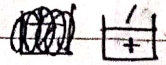
((a+b) → invalid exp

```
:- char stack[max];
   int top=-1;
   void push(char);
   void pop();
Main()
{  int i=0;
   char expr[30];
   printf("Enter the expression\n");
   scanf("%s", expr);
   while(expr[i]!='\0')
   {  if(expr[i]=='(')
         push(expr[i]);
      else if(expr[i]==')')
         pop();
      i++;
   }
```

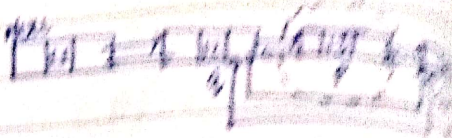
```
if (top == -1)
  pf("Valid expression");
else
  pf("Invalid expression");
}
void push(char ch)
{ top++;
  stack[top] = ch;
}
void pop()
{
  top--;
}
```

( )
^
* /
+ −

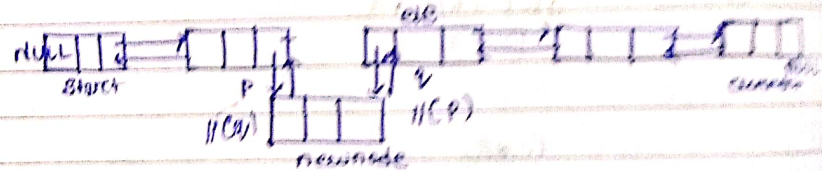
Q. Convert to postfix :- A-B+C*D^H/F+G-H

A	A	
-		
B	AB	
+		
C	ABC	
*		
D	ABCD	
^		
H	ABCDH	
/	ABCDH^*/	
F	ABCDH^*F	
+	ABCDH^*F/+	
G	ABCDH^*F/+G	
-	ABCDH^*F/+G-	
H	ABCDH^*F/+GH-+-	

```
void deletespecific value()
{ while (------)
  { p=p->next;
  }
  q= p->next;
  rr= p->next;
  q->next = rr;
  rr->prev = q;
  free(p);
}
```



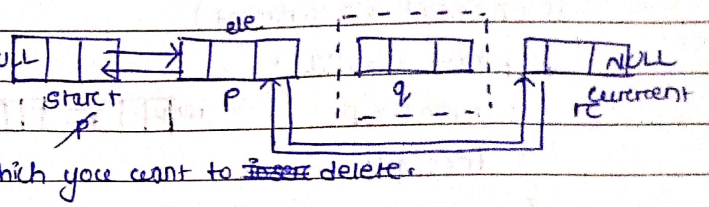
Q. Insert before a specific value




```
void insertbefore()
{ int ele,x;
  struct node *p,*q; p=start;
  // Enter the value before which you want to insert
  // Enter the value you wants to insert.
  while( p->next->data != ele) // while(p->data != ele)
     { p= p->next;
     }
     q= p->next;  // q= p->prev;
  newnode = (struct node *) malloc (sizeof (struct node));
  newnode->data = x;
   p->next = newnode;    // q->next = newnode;
   newnode->prev = p;    // newnode->prev = q;
   newnode->next = q;    // newnode->next = p;
   q->prev = newnode;    // p->prev = newnode;
}
```

Delete after a specific value :-

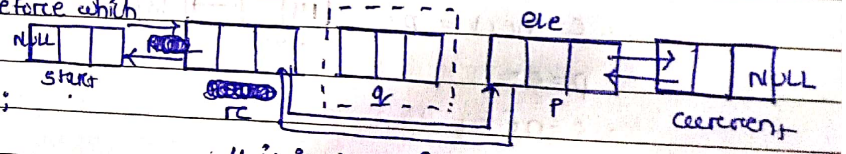
```
void deleteafter()
{ struct node *p, *q;
  p = start;
// enter the value after which you want to ~~insert~~ delete.
  while(p->data != ele)
  { p = p->next;
  }
  q = p->next;
  rr = q->next;
  p->next = rr;
  rr->prev = p;
  free(q);
}
```

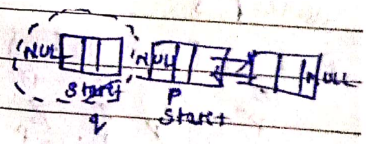


Delete before a specific value :-

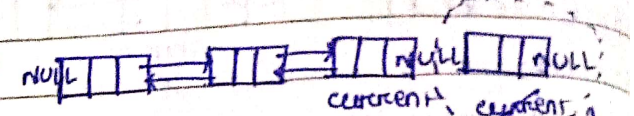
```
void deletebefore()
{ // Enter the value before which
  you want to delete
  struct node *p, *q;
  p = start;
  while(p->data != ele)
  { p = p->next;
  }
  q = p->prev;
  rr = q->prev;
  rr->next = p;
  p->prev = rr;
  free(q);
}
```

```
// if it is a first node
if (------)
{ q = p->prev;
  p->prev = NULL;
  free(q)
  start = p;
}
else {
  delete before value program
}
```





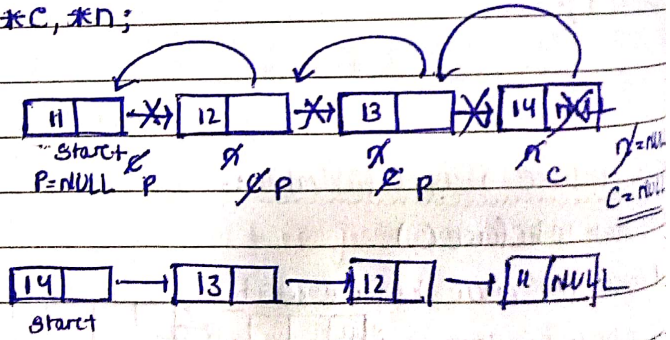
```
// if it is the last node        q = current
if (p->next == current)
{ p->next = NULL;
  current = p;
    free(q);
}
else
{ delete after value program
}
```

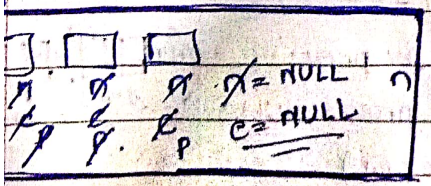


Reversing using Linked List :- (Single)

P = previous
C = current
n = next

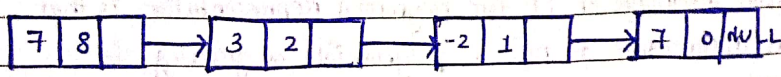
```
Void reverse()
{ struct node *p, *c, *n;
    p = NULL;
    c = start;
  while (c != NULL)
  { n = c->next;
    c->next = p;
    p = c;
    c = n;
  }
    start = p;
```



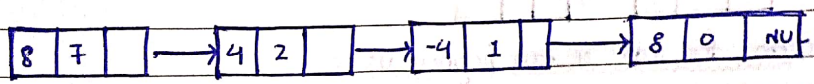


Polynomials :-

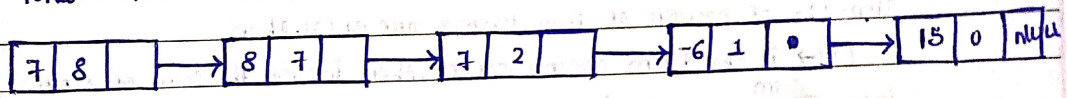
$P_1(x) = 7x^8 + 3x^2 - 2x + 7$



$P_2(x) = 8x^7 + 4x^2 - 4x + 8$



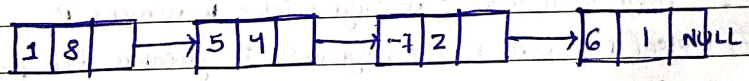
$P_{total} = P_1(x) + P_2(x) = 7x^8 + 8x^7 + 7x^2 - 6x + 15$



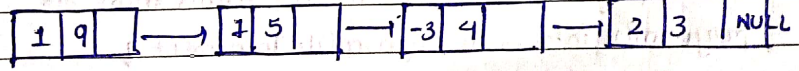
Q. $P_1(x) = x^8 + 5x^4 - 7x^2 + 6x$

$P_2(x) = x^9 + 7x^5 - 3x^4 + 2x^3$

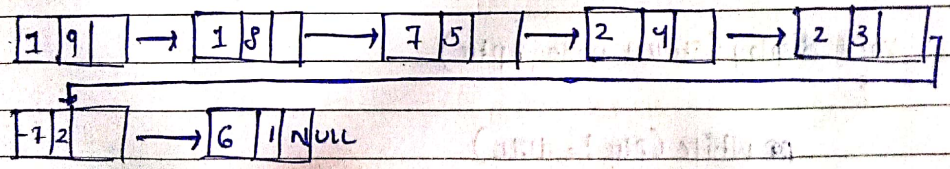
$P_1(x)$



$P_2(x)$



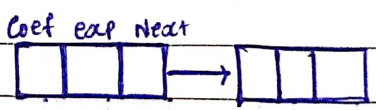
$P(x) = x^9 + x^8 + 7x^5 + 2x^4 - 7x^2 + 2x^3 + 6x$




```
Struct node {
        int coef, exp;
        Struct node *next;
    }
```

An important application of LL is to represent polynomials and manipulations.

The main advantage of LL for polynomial representation is that it can accomodate a no. of polynomials of growing size so that there combined size doesn't exceed the total memory size.



Algorithm for add^{n} of two polynomials :-

Let p and q are two polynomials represented by LL.

Step-1 :- While p and q are not NULL, repeat step 2.

Step-2 :- If powers of two terms are equal, then.

(i) → if the term doesn't cancel, then insert the sum of terms into sum polynomial (polynomial-3/rr). Increment p and q.

(ii) → else if power of first polynomial is greatere than power of 2nd polynomial, then insert the term from 1st polynomial into sum polynomial. Increment p.

(iii) → else (if) power of 2nd polynomial is greatere than power of 1st polynomial, then insert the terms from 2nd polynomial into sum polynomial. Increment q.

Step-3 :- Copy the remaining terms from the non-empty polynomial into the sum polynomial.

```
void display (struct node *ptrr)
{
    while (ptrr != NULL)
    { pf ("%dx^%d", ptrr->coef, ptrr->exp);
      ptrr = ptrr->next;
                        if (ptrr != NULL)
                        { pf ("+");
```

Q. $P_1(x) = 4x^3 + 5x^2 - 3x$ Multiplication

$P_2(x) = 2x^5 + 6x^4 + x^2 + 8$

$P(x) = 4x^3(2x^5 + 6x^4 + x^2 + 8) + 5x^2(2x^5 + 6x^4 + x^2 + 8) - 3x(2x^5 + 6x^4 + x^2 + 8)$

$= 8x^8 + 24x^7 + 4x^5 + 32x^3 + 10x^7 + 30x^6 + 5x^4 + 40x^2$

$- 6x^6 - 18x^5 - 3x^3 - 3x^2$

$= 8x^8 + 24x^7 + 10x^7 + 30x^6 - 6x^6 + 4x^5 - 18x^5 + 5x^4 + 32x^3 - 3x^3$

$- 3x^2$

$= 8x^8 + 34x^7 + 24x^6 - 14x^5 + 5x^4 + 29x^3 - 3x^2$