< Analysis of Algorithms >

Analysing an algorithm means determining amount of space and time (space means memory) needed to execute it.

* Time complexity and space complexity.

Time complexity means running time of program which is independent on input size.

Space complexity of an algorithm means the amount of memory required for its execution. which is again dependent on input size.

OR

Time complexity of an algorithm is a simple measurement of how fast the time taken by a program grows, if the input size increases.

* Time complexity - why is it important ?

Suppose, to find whether a number is prime or not, these two algos are used.

• Algo-I :
```
for i ← 2 to i ← n-1
    if i divides n
        n is not prime.
```

• Algo-2 :
```
for i ← 2 to i ← √n
    if i divides n
        n is not prime.
```

let us ~~see~~ assume that a division operation takes 1ms.

Then, for the worst-case ie. if the no. is prime.

	loop will run $(n-2)$ times	$(\sqrt{n}-1)$ times
$n = 11$	9 ms	$(3-1) = 2$ ms.
$n = 101$	99 ms	9 ms
$n = 1000003 = 10^6+3$	$\simeq 10^6$ ms $= 16.66$ min	$\sqrt{10^6+3} - 1 \simeq 10^3$ ms $= 1$ sec.
$n = 1000000019 = 10^{10}+19$	$= 10^{10}$ ms $= 10^7$ sec $= 115$ days!	$\simeq 10^5$ ms $= 10^3 \times 10^2$ ms $= 10^2$ s $= 100$s

Now, there is an exceptionally larger difference in rate of growth of running time for both algos for large value of n.

Scanned by CamScanner

So, for large values of n, for worst case, algo 2 is more efficient as it has less time complexity.

- Worst-case, Avg-case & Best-Case time complexity:

→ worst-case running time corresponds to that value of input for which the execution of algorithm takes maximum time. It is an upper-Bound on running time for an algo.

→ Avg-case running time of an algorithm corresponds to the expected behaviour of algorithm when input is randomly drawn from given distribution.

→ Best-case running time: corresponding to the minimum time the algo will take to execute. for eg. Best case for simple linear search on array occurs when desired element is found at the starting position. and worst-case happens when it is not-present

- Time-space Trade-off.

The best algorithm to solve a particular problem is the one which takes less time as well as less computer memory. Among all the possible algorithms to a problem, some of them may require less memory & other may take less time. Sometimes, one thing is to be sacrificed at the cost of other. ∴ there is a time-space trade-off among algorithms.

So, if space is a big constraint, then one may choose an algo which takes less memory at the cost of more CPU time & if time is a big constraint, then one may choose an algo which takes less CPU time at cost of more computer memory.

Scanned by CamScanner

- Algorithm Efficiency. (or running time of Algorithm)

In case of loops, efficiency depends on no. of iterations or in other words, it depends on no. of loop factor. (n).

i) linear loops :

```
for (i=0 ; i<n ; i++)
{   statements ;
    :
}
```

loop factor is n and hence $f(n) = n$.

if → for (i=0 ; i<n ; i+=2)
{ statements ;
:
}

∴ no. of iterations $= n/2$

$f(n) = n/2$.

ii) logarithmic loops.

```
for (i=1 ; i<1000 ; i*=2)
    { statements ; }
```

or

```
for (i=1000 ; i>=2 ; i/=2)
    { statements ; }
```

loop in either case will run 10 times.

∴ $f(n) = \log n$.

* Nested loops : The total efficiency is obtained as product of no. of iterations in inner loop & no. of iterations in outer loop.

```
for (i=0 ; i<10 ; i++)
    for (j=01 ; j<10 ; j*=2)
        statements ;
```

$f(n) = n \log n$

```
for (i=0 ; i<10 ; i++)
    for (j=0 ; j<=i ; j++)
        statements ;
```

$f(n) = n^2$.

* Dependent quadratic loop → Inner loop will run once, twice etc. for i=1, 2, ...

```
for (i=0 ; i<10 ; i++)
    for (j=0 ; j<=i ; j++)
        statements;
```

$avg = \frac{1+2+\cdots+10}{10} = 5.5$.

∴ inner loop iterates $= \left(\frac{n+1}{2}\right)$ times

$f(n) = n\left(\frac{n+1}{2}\right)$

Scanned by CamScanner

* How to analyse time complexity?

Let CPU takes 1 unit time for each arithmetical operation & 1 unit for assignment & return statement.

∴ sum(a,b)

{ return a+b;

}

$T_{sum} = 2$

↓

Constant time complexity.

∴ Sum of list (A, n)

{ total = 0

for i=0 to n-1

total = total + A_i

return total

}

	Time/each	no. of times
	c_1 (1)	1
	c_2 (2)	n+1
	c_3 (2)	n
	c_4 (1)	1.

(1 for addition & 1 for assignment)

(1 for updation & 1 for test condition).

$T_{sum\ of\ list} = c$

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

$= 4n + 4.$

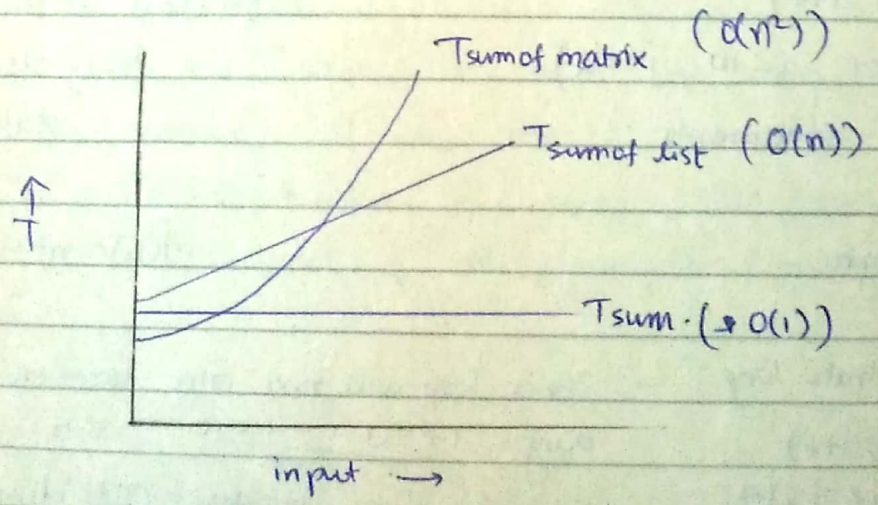
$T(n) = c_1 + c_2(n+1) + c_3 n + c_4$

$= n(c_2 + c_3) + c_1 + c_2 + c_4.$

$= cn + c'$

where $c = c_2 + c_3$ & $c_1 + c_2 + c_4 = c'$.

Similarly, $T_{sum\ of\ matrix} = an^2 + bn + c.$



Scanned by CamScanner

- Asymptotic notations. (O, Ω, θ).

It is a way to divide the efficiency (or running time $f(n)$) of different algos in broad classes.

Suppose, we have two algos with efficiency:

Algo 1 : $f(n) = 5n^2 + 7$

Algo 2 : $f(n) = 17n^2 + 6n + 8$

we are more interested in finding rate of growth of $f(n)$ with larger values of n,

So, as $n \to \infty$, $f(n) = \underbrace{5n^2 + 7}_{\text{will becomes insignificant}} = O(n^2)$.

and $f(n) = \underbrace{17n^2 + 6n + 8}_{\text{will become insignificant}} = O(n^2)$

* i) Big 'O' notation.

If the function $f(n)$ is complex, a dominant factor is sufficient to determine order of $f(n)$ & hence, the efficiency & this factor is called big 'O'.

definition: for non-negative n & non-Negative func^n. $g(n)$

$O\{g(n)\} = \{ f(n) : f(n) \leq c\,g(n) , \forall n \geq n_0 \}$

eg: $f(n) = 5n^2 + 2n + 1$

$g(n) = n^2$.

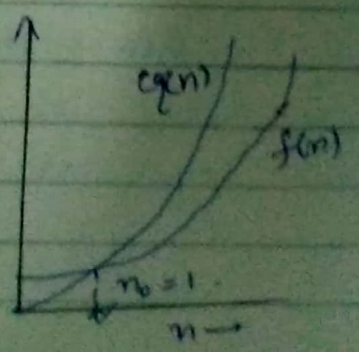
for $c = 8$ and $n_0 = 1$ $\therefore f(n) \leq 8n^2$ ie. $f(n) \leq c\,g(n)$.

there may exists more pairs of c & n_0 for which this relation will hold true.

$\therefore f(n) = O(n^2)$

Big 'O' provides an upper bound And tells that running time will not exceed $g(n) \times c$.

$\longrightarrow$ Big 'O' represents worst case.



Scanned by CamScanner

* OMEGA NOTATION (Ω).

It provides lower bound for $f(n)$. It represents best case.

definition :

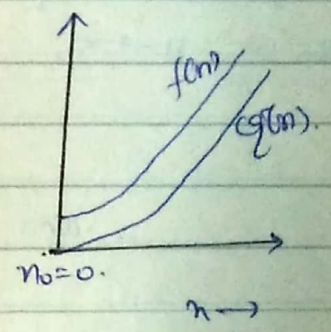
$$\Omega(g(n)) = \{ f(n) : cg(n) \le f(n), \forall n \ge n_0 \}.$$

$f(n) = 5n^2 + 2n + 1$

$g(n) = n^2$.

for $c = 5$ & $n_0 = 0$ $\Rightarrow$ $cg(n) \le f(n)$ $\forall$ $n \ge n_0$.

$\therefore$ $f(n) = \Omega(n^2)$.



* θ (Theta notation).

$$\theta\{g(n)\} = \{ f(n) : c_1 g(n) \le f(n) \le c_2 g(n) \ \forall \ n \ge n_0 \}.$$

$f(n) = 5n^2 + 2n + 1$ & $g(n) = n^2$

$c_1 = 5$ & $c_2 = 8$ & $n_0 = 1$.

$\Rightarrow$ $f(n) = \theta(n^2)$.

NOTE : We analyse time complexity : for

a) very large input

b) worst - case.

$f(n) = 17n^4 + \underbrace{3n^3 + 4n + 8}$ $= O(n^4)$

as this will be negligible for $n \to \infty$.

& drop constant term.

$f(n) = 16n + \log n$. $= O(n)$.

insignificant for $n \to \infty$

Scanned by CamScanner

< Analysis of Bubble sort Algorithm. >

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

2	4	1	5	3	7

2	1	4	3	5	7

1	2	3	4	5	7

unsorted array

For each pass through the array, the largest element in the unsorted array is shifted to the highest index of the unsorted Array

```
void Bubble_Sort()
{
    for( int round = 1 ; round < n ; round++)        — C1
    {
        flag = 0                                      — C2
        for (i = round ; i < n ; i++)                 — C3
        {
            if ( A[i-1] > A[i] )                      — C4
            {   t = A[i-1];                           — C5
                A[i-1] = A[i];                        — C6
                A[i] = t;                             — C7
                flag = 1 ;                            — C8
            }
        }
        if (flag == 0) break;                         — C9
    }.
}.
```

[SIR'S METHOD]

Now,

Since, some part of array gets sorted in each pass through the array, we need not to again scan the sorted Array, so, above algorithm can be modified as.

Scanned by CamScanner

```
void BubbleSort (A, n).
{
        for ( k=1 ; k<n ; k++)                          — c1
        {
                flag = 0 ;                              — c2
                for ( i=0 ; i<= n-k-1 ; i++)            — c3
                {
                        if (A[i] > A[i+1])              — c4
                        {
                                t = A[i] ;              — c5
                                A[i] = A[i+1] ;         — c6
                                A[i+1] = t ;            — c7
                                flag = 1 ;              — c8
                        }
                }
                if (flag==0) break ;                    — c9.
        }
}
```

Now, for worst case : [Sir's method]. [array is oppositely sorted].

$c_1, c_2, \ldots, c_9$ be the time taken by respective statements.

No. of times, the statements will get executed in worst case :

c_1 — n times

c_2 — $(n-1)$

c_3 — $n + n-1 + \cdots + 1 = \frac{n(n+1)}{2}$

c_4 — $(n-1) + (n-2) + \cdots + 1 = \frac{n(n-1)}{2}$

c_5 — $(n-1) + (n-2) + \cdots + 1 = \frac{n(n-1)}{2}$

c_6, c_7, c_8 — "

c_9 — $(n-1)$ times.

$$\therefore \underbrace{f(n)}_{\text{time } f^n \text{ or efficiency}} = c_1 \times n + c_2 \times (n-1) + c_3 \frac{n(n+1)}{2} + (c_4 + c_5 + c_6 + c_7 + c_8)\frac{n(n-1)}{2} + c_9(n-1)$$

$$= (\quad)n^2 + (\quad)n + k$$

$f(n) = an^2 + bn + k$ → represents time in worst case. i.e. when list is in descending

Scanned by CamScanner

let us suppose that $f(n) = 5n^2 + 7n + 3$.

let $n = 10^6$ & 1 million data.

$\therefore f(n) = 5 \times 10^{12}\, ns = 5000\, s = 1\text{-}2$ se hrs.

$\therefore$ huge amount of time is required.

i) Big O ('O') → represents time complexity of algo in worst case.

$\therefore f(n) = O(n^2)$

iff $f(n) \le cg(n)$ $\forall\, n \ge n_0$. ①

let $c = 7$.

n	$f(n) = 5n^2 + 7n + 3$		$7n^2$ ($cg(n)$)
1	15		7
2	37		28
3	69		63
4	111	$\le$	112
5	163	$\le$	175

$\therefore$ for $\forall\, n \ge 4$, $cg(n) \ge f(n)$. $\therefore f(n) = O(n^2)$

for $c = 7$, $n_0 = 4$.

for exp. ①,

time taken by algo in worst case is always less than or equal to $cg(n)$.

* Best-case Analysis.

Best case means the input in which array given is already sorted.

$c_1 = 1$, $c_2 = 1$, $c_3 = n$, $c_4 = n-1$, $\underbrace{c_5 \cdots c_8}_{(x)} = 0$, $c_9 = 1$

these steps will not be executed.

$\therefore f(n) = c_1 + c_2 + nc_3 + (n-1)c_4 + c_9$

$f(n) = cn + b$.

$f(n) = \Omega(n)$ iff $f(n) \ge cg(n)$ $\forall\, n \ge n_0$

Scanned by CamScanner

let $f(n) = 11n + 6$. $\because$ $f(n) \geq 11n$, $n \geq 1$.

$\therefore$ $f(n) = \Omega(n)$, for $c = 11$ & $n_0 = 1$.

$n = 10^6$; $f(n) = 11 \times 10^6$ ns = 11 ms $\rightarrow$ Best case.

* Some more concepts.

```
fun A()
{       for(m=i; m≤n; m++)
            {  funB();  → O(n)
            }
}.
```

$\therefore$ complexity for func(A) $= O(n^2)$.

{ n from loop & n from funcB() }.

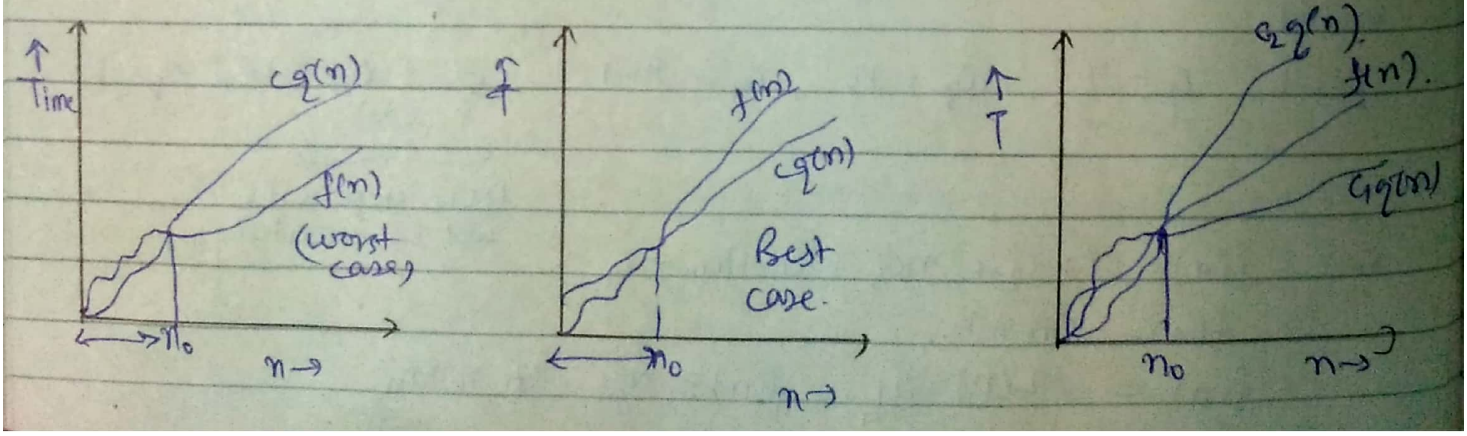
*

```
funA()
{       for(1→n)
        {       funB();   → O(n)
                funC();   → O(n²).
        }
}
```

$\therefore$ complexity of funcA() $= O(n^3)$.

NOTE: $f(n) = O(n^2) = \Omega(n^2)$

$f(n) = \Theta(n^2)$ when func is both bigh-oh & big-omega.



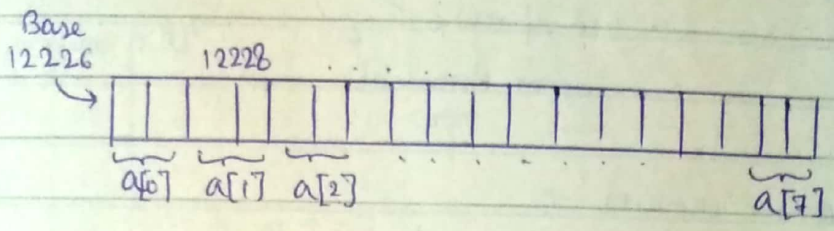
Scanned by CamScanner

* REPRESENTATION OF ARRAY IN MEMORY.

int a[7] : it is a request to compiler to reserve 7 cells of 2 bytes each for 7 integers.

(Contiguous memory allocation occurs in case of Array)

$\& a[i]$ = Base address + $i *$ size



$\& a[0]$ = base $+ 0 * 2$ = 12226

$\& a[1]$ = base $+ 1 \times 2$ = 12228

and so on.

* 2-Dimensional Array.

i> Row-major order

Elements are stored Row-wise and rows are listed on basis of column

$$a = \begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{bmatrix} \longrightarrow \underbrace{\boxed{a_{00}}\boxed{a_{01}}\boxed{a_{02}}}_{\text{row 1}}\underbrace{\boxed{a_{10}}\boxed{a_{11}}\boxed{a_{12}}}_{\text{row 2}}\underbrace{\boxed{a_{20}}\boxed{a_{21}}\boxed{a_{22}}}_{\text{row 3}}$$

Address of an element a[i][j] = $B + w(n(i-L_1)+(j-L_2))$

B → Base address

w → size of element

n → no. of columns or no. of elements in a row

L_1 → Lower Bound of row

L_2 → Lower Bound of column

when $L_1 = 0$ and $L_2 = 0$

Address of an element a[i][j] = $B + w(i * n + j)$

Scanned by CamScanner

Starting index of an Array = Lower Bound (LB)
Higher index − 1 = upper Bound
no. of elements = UB − LB + 1.

If array is given in this form : Arr [10:15 , −2 : 2]

LB of row

upper Bound of row

LB of column

Upper bound (UB)

no. of rows = UB − LB + 1 = 15 − 10 + 1 = 6

no. of columns = 2 − (−2) + 1 = 5

∴ it is equivalent to Arr[6][5].

ii) Column major.

Elements are stored column-wise. i.e. first column is stored, then 2nd column and so on.

a_{00}	a_{10}	a_{20}	a_{01}	a_{11}	a_{21}	a_{03}	a_{12}	a_{22}

first column 2nd column 3rd column.

∴ Address of $a[i][j] = B + w\,((i - L_1) + m(j - L_2))$

m → no. of elements in each column. or no. of rows.

(order of matrix = m×n).

For a 'k'-dimension matrix,

[Row-major form] , int $a[x_1][x_2][x_3]\cdots[x_k]$;

& $a[i_1][i_2][i_3]\cdots[i_k]$ = Base + $(i_1 \times x_2 * x_3 * \cdots * x_k + i_2 * x_3 * x_4 \cdots + i_{k-1} * x_k + i_k) * \text{size}$.

[Column-Major form]

&

→ SPARSE MATRIX.

A matrix with most of its elements as zero is called Sparse matrix.

Now, the sparse matrix is not stored in the memory in the way in which normal 2D- Array is stored. Instead, it is stored in memory in the following structure:

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

no. of elements = 4 × 5 = 20

size = 20 × 2 = 40 bytes

[size of int = 2 bytes]

(Form I)

→

4	5	5
0	2	5
2	0	2
2	2	3
3	1	3+1
3	4	7.

no. of columns of sparse matrix

total no. of non-zero values

no. of rows of sparse matrix

individual non zero values

6 × 3 × 2 = 36 bytes

if non-zero elements would be 7 then, 7 × 3 × 2 = 42 bytes

⇒ size may get increased.

∴ space ∝ non-zero elements.

(Form 2).

Ques 1: why to store sparse matrix in such an unsual way?

Sol: i) Storage: Space can be saved by storing only non-zero values.

ii) Computer time: Computer time can be saved by logically designing a data structure traversing only non zero elements.

Let a sparse matrix be int a[n][n];

⇒ Equating space by original method & our method of storing non-zero values.

$n \times n \times 2 = (nz+1) \times 3 \times 2$. where nz = no. of non zero elements.

$2n^2 = (nz+1) \times 6$.

$\frac{n^2}{3} = nz+1$

$nz = \frac{n^2}{3} - 1 \approx \frac{n^2}{3}$

Scanned by CamScanner

So, if no. of non-zero elements is less than 33% of total elements then we will be saving space by creating our array of the new form as we have created.
So, it is a data structure (for saving space).

PROGRAM 1 : Displaying the contents of a Sparse Matrix
We will take input in form 2. and print output as form 1.

```
struct sparse
  {     int row;
        int column;
        int value;
  };

int main()
{     struct sparse A[20];
      printf("Dimensions of matrix ie. mxn ?");
      scanf("%d%d", &A[0].row, &A[0].column);
      printf("How many non zero elements?");
      scanf("%d", &A[0].value);
      int i;
      for(i=1 ; i<=A[0].value ; i++)
       {
            scanf("%d%d%d", &A[i].row, &A[i].column, &A[i].value)
       }
      display(A);
}

void display(struct sparse A[])
{     int m, n, nz, i, j, k;
      m = A[0].row; n = A[0].column; nz = A[0].value; k=1;
      for(i=0 ; i<m ; i++)
      {
```

Scanned by CamScanner

```
        for(j=0 ; j<n ; j++)
        {       if( k<=Anz)
                {
                        if ( i == A[k].row && j== A[k].column)
                                {
                                        printf("%6d", A[k].value);
                                                k++;
                                }
                        else
                                {       printf("%6d",0);
                                }
                }
        }
        printf("\n");
}.
```

OR.

[appending after start of i -loop]

```
        for(j=0 ; j<n ; j++)
        {
                if ( i == A[k].row && j == A[k].column && k<=n
                        {
                                printf("%6d", A[k].value);
                                        k++;
                                }
                else
                        {       printf("%6d",0);
                        }
        }
        printf("\n");
    }
}.
```

Scanned by CamScanner

< DATA STRUCTURES >

A data structure is a way to store and organize data in a computer, so that it can be used efficiently.
for eg: we are able to find a particular word among millions of words in a dictionary because they are sorted in alphabetical order. If words would not have been stored in such a way, it would be impractical and impossible for us to search a word.
So, dictionary can be seen as a sorted list of words.

eg 2: A city map: we are able to locate any landmark because (data: position of landmards, routes etc.) are organized in form of geometries in a 2D-Map. (proper scales are used).

i) Abstract data type (ADT):
[Mathematical / logical model]
ADT defines data and operations, but ~~have~~ no its a logical concept and not any implementation.
eg: List: as an ADT.
- Store a given no. of elements of any type
- Read elements by position
- Modify element at position
etc...

So, here we define some data and operations which can be performed on that data ie. we define a logical model without going into details like implementation in any programming language. It is called an ADT.

ii) Concrete datatype
ie. Arrays, structures, linked lists, trees, graphs etc.
these are implementation of ADT.

Scanned by CamScanner

PROGRAM 2: Addition of two Sparse Matrices

4	5	7
0	2	11
1	3	12
1	4	5
2	0	6
2	4	9
3	0	2
3	3	3

+

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

↓

4	5	10
0	0	6
0	2	11
1	2	5
1	3	18
1	4	5
2	0	6
2	4	16
3	0	5
3	3	3
3	4	5

⟷ equivalent 2D-Array

6	0	11	0	0
0	0	5	18	5
6	0	0	0	16
5	0	0	3	5

Sol>

```
struct Sparse
  {  int row;
     int col;
     int val;
  };
struct sparse C[20];
```

Scanned by CamScanner

```
struct sparse* addsparse ( struct sparse A[], struct sparse B[])
{
        int i,j,k;
        if ( A[0].row == B[0].row && A[0].col == B[0].col )
        {       i=j=k=1;
            while ( i <= A[0].val && j <= B[0].val )
            {
                if( A[i].row < B[j].row || ( A[i].row==B[j].row &&
                                                A[i].col < B[j].col ))
                {
                        C[k].row = A[i].row;
                        C[k].col = A[i].col;
                        C[k].val = A[i].val;
                        k++; i++;
                }
                else if ( A[i].row > B[j].row || ( A[i].row == B[j].row &&
                                                A[i].col > B[j].col))
                {
                        C[k].row = B[j].row;
                        C[k].col = B[j].col;
                        C[k].val = B[j].val;
                        k++; j++;
                }
                else
                {
                        C[k].row = A[i].row;
                        C[k].col = A[i].col;
                        C[k].val = A[i].val + B[j].val;
                        k++; j++; i++;
                }
        }
```

Scanned by CamScanner

```
while (i<=A[0].val)
{
        C[k].row = A[i].row;
        C[k].col = A[i].col;
        C[k].val = A[i].val;
        k++; i++;
}
while (j<=B[0].val)
{
        C[k].row = B[j].row;
        C[k].col = B[j].col;
        C[k].val = B[j].val;
        k++; j++;
}

C[0].row = A[0].row;
C[0].col = A[0].col;
C[0].val = k-1;

return (c)

}

else
{ cout << " cannot be Added "; }
        exit(0);

}

int main()
{
    struct sparse A[20], B[20];
    struct sparse *p; int i;
    printf ("\nDimensions mxn of matrix A ");
    scanf ("%d%d ", &A[0].row, &A[0].col);
```

Scanned by CamScanner

```
printf(" How many non - zero elements");
scanf ("%d", &A[0].val);
for (int i = 1 ; i <= A[0].val ; i++)
  {
      scanf("%d%d%d", &A[i].row, &A[i].col, &A[i].val);
  }

printf(" \n Dimensions mxn of matrix B ");
scanf (" %d %d ", &B[0].row , &B[0].col);
printf(" How many non-zero elements");
scanf (" %d ", &B[0].val );
for (i=1; i<= B[0].val ; i++)
  {
      scanf("%d%d%d", &B[i].row, &B[i].col, &B[i].val);
  }

p = addsparse(A, B) ;

}.
```

* COUNTING SORT.

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

→ size n.

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

→ size = max of range of A[]+1
let size = k+1

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

→ C is to count the occurrence of distinct elements.

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

→ Cumulative sum ie.
A c[i] = c[i] + c[i-1]

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

k = max Range.

indexing from 1 of real & output array.

Scanned by CamScanner

Counting sort is not a comparision-based-sorting algo.
Indexing of given unsorted array is done from 1 to size n.
Indexing of C array is done from 0 to k.
C array i.e. count Array is used here to count the no. of occurences of distinct elements in the unsorted Array.
Then, cumulitive sum is taken for C[] array.

Algo in form of Pseudocode is :

```
counting_sort (A, B, k)
for i ← 0 to k                    // k times
    do C[i] ← 0                   // k times
for j ← 1 to n                    // n times
    do C[A[j]] ++                 // n times
for i ← 1 to k                    and
    do C[i] ← C[i-1] + C[i] ;     so
                                  on.
for j ← n to j ← 1
    do B[C[A[j]]] ← A[j].
       C[A[j]] -- ;
```

time complexity = $O(4n+4k)$
= $O(n)$ → efficient algorithm.

COUNTING SORT (SIR'S METHOD) :

- n is the size of given unsorted array.
- k is the size of count array C[] which is equal to [maximum value in unsorted Array]+1 i.e. k = max+1.
- array C[] is used to count the occurences of distinct elements in unsorted array.
- array t[] is used to store place values.

Scanned by CamScanner

	0	1	2	3	4	5	6	7	8	9	10	11	12
A[] →	2	0	5	6	0	4	3	2	0	1	4	5	6

→ unsorted Array (size n=13).

	0	1	2	3	4	5	6
B[] →	0	0	0	0	0	0	0

→ array C[k] initialized with 0.
k = max+1 = 6+1 = 7.

	0	1	2	3	4	5	6
C[] →	3	1	2	1	2	2	2

→ counts of distinct elements.

	0	1	2	3	4	5	6
t[] →	0	3	4	6	7	9	11
	1	4	5	7	8	10	12
	2		6		9	11	13.
	3						

→ cumulitive sum. to generate place values.

	0	1	2	3	4	5	6	7	8	9	10	11	12
B[] →	0	0	0	1	2	2	3	4	4	5	5	6	6

Pseudocode for Counting sort :

```
for i=0 to k-1                    } O(k)
        C[i] = 0 ;
for i=0 to n-1                    } O(n)
        C[A[i]] ++ ;
for
        t[0]=0      // because we have to start filling B[]
                    // in sorted manner from index 0.
for i=1 to k-1                    } O(k)
        t[i]= t[i-1]+ C[i-1] ;

for i=0 to i=n-1                  } O(n).
  B[t [A[i]]] = A[i] ;

        t[A[i]] ++ ;
```

Time complexity = $O(2n+2k)$
$= O(n)$. → efficient algo.

Scanned by CamScanner

PROGRAM : Transpose of a sparse matrix.

4	5	8
0	1	6
0	3	12
1	2	11
2	0	2
2	4	3
3	1	7
3	3	5
3	4	9

A[].

mathematically

Transpose. →

5	4	8
0	2	2
1	0	6
1	3	7
2	1	11
3	0	12
3	3	5
4	2	3
4	3	9

B[].

mathematically

0	6	0	12	0
0	0	11	0	0
0	0	0	0	3
0	7	0	5	9

0	0	2	0
6	0	0	7
0	11	0	0
12	0	0	5
0	0	3	9

Logic :- Since, during a transpose operation, row of given matrix becomes columns of required matrix and columns of given matrix becomes rows of required matrix., ~~so~~ Now, in sparse matrix row should be arranged in ascending order i.e. they are needed to be sorted. So, B[i]. row should be sorted. This is done using concept of counting sort.

* transpose sparse () is to be created.

Scanned by CamScanner

```
# define MAXS 20 ;
// # define directive statement to declare & initialize global constant

struct sparse* TransposeSparse (struct sparse A[], struct sparse B[])
{
        int c[MAXS], d[MAXS], i, m, n, t ;
        m = A[0].row ;
        n = A[0].col ;
        t = A[0].val ;

    for (i=0 ; i<n ; i++)
            c[i]=0 ;
    for (i=1 ; i<=t ; i++)
      { // c[A[i]]++ ;  // wrong statement
            c[A[i].col]++ ;  // Correct statement
      }
            d[0]=1 ;

    for (i=1 ; i<n ; i++)
      {
            d[i] = d[i-1] + c[i-1];
      }

    for (i=1 ; i<=t ; i++)
      {
            B[t[A[i].col]].row = A[i].col ;

            B[t[A[i].col]].col = A[i].row ;
            B[t[A[i].col]].val = A[i].val ;
            t[A[i].col]++ ;
      }.

            B[0].row = n ;
            B[0].col = m ;
            B[0].val = t ;
}
```

Scanned by CamScanner

PROGRAM: Addition of two polynomials. assuming. that all coefficients are +ve and that case in which coeff. add up to give zero resultant coeff. may be ignored.

LOGIC: we will compare the coefficients of both polynomials, (x) We will compare the exponential part of both polynomials and add that element of Array (or that term) which has larger exponential part. In case of same exponential part, we will add the coefficients of given two polynomials and resultant value of coeff along with expo part will be stored in the required polynomial.

(i is the counter for P_1 and j for P_2, if suppose expo of first term of P_2 > expo of first term of P_1, then, we will add the term having greater expo part in required polynomial P3 & will increase j counter & (k counter (k counter for P_3)).

$P_1 = 10x^9 + 3x^6 + 5x^3 + 4x^2 + 6x + 8$

0	1	2	3	4	5	
10	3	5	4	6	8	← coeff
9	6	3	2	1	0	← expo

i

size of array of structure = 6.
S1 = 6.

$P_2 = 2x^{10} + 7x^6 + 9x^4 + 3x^2 + 5$

0	1	2	3	4
2	7	9	3	5
10	6	4	2	0

j

Similarly, S2=5.

0	1	2	3	4	5	6	7
10	9	6	4	3	2	1	0
2	10	10	9	5	7	6	13

k

S3 = 8.

Scanned by CamScanner

Dry run:

Initially, as the function add_polynomial is invoked, $i=j=k=0$ then as test condition is satisfied, so control will enter into loop and check expo part of ith element of P_1 & j^{th} element of P_2 ($i=0$ & $j=0$ at this instant) ∵ expo part of P_2 > that of P_1, so $P_2[0]$ will be added to $P_3[0]$ & j & k will get increased. Similarly, $P_1[0]$ will be added to $P_3[1]$ and i & k will get ↑. Now, for $i=j=1$, ∵ expo part of $P_1[1]$ = that $P_2[1]$ ∴ coeff sum i.e. $P_1[1].coeff + P2[1].coeff$ should be assigned to $P_3[k].coeff$ & $P1[i].expo$ should be assigned to $P3[k].expo$. & so on.

* CODE:

```
struct node
{
    int ~~float~~ coeff;
        int expo;
};
typedef struct node poly;   // now objects of structure node can
                            // be declared using poly.

add_polynomial ( poly P1[], poly P2[], poly P3[], int s1, int s2, int *s3
→ void.
int main()
{       poly P1[20], P2[20], P3[40];
        int s1, s2, s3;     //sizes of P1, P2, P3.
        printf(" Enter ~~siz~~ no. of terms in polynomial one:");
        scanf("%d", &s1);
        printf(" Enter coff. and expo part of each term successively")
        int i;
        for(i=0; i<s1; i++)
        {   scanf("%d%d", &P1[i].coeff, &P1[i].expo);
        }.
```

Scanned by CamScanner

```
printf ("Enter size of P2 ie : no. of terms of P2");
scanf ("%d", &S2);
printf ("Enter coeff & expo part of each term of polynomial succesively");
for (i=0; i<S2; i++)
{
    scanf("%d%d", &P2[i].coeff , &P2[i].expo);
}.

add_polynomial (P1, P2, P3, S1, S2, &S3);

for (i=0 ; i<S3 ; i++)
{   printf("%d x %d + ", P3[i].coeff, P3[i].expo );
    if (i != S3-1)

  else
    {
        printf ("%d x %d ", P3 [i].coeff, P3[i].expo);
    }
}

}. //end of main().

void add_polynomial ( Poly P1[], poly P2[], poly P3[], int S1, int S2, int *S3
{
    int i,j, k ;   i=j=k=0 ;
  while ( i<S1 && j<S2)
{
    if ( P1[i].expo == P2[j].expo)
      {
            P3[k].expo = P1[i].expo ;
            P3[k].coeff = P1[i].coeff + P2[j].coeff ;
            i++; j++; k++;
      }
```

Scanned by CamScanner

```
else if (P1[i].expo > P2[j].expo)
    {
        P3[k].coeff = P1[i].coeff;
        P3[k].expo = P1[i].expo;
        i++; k++;
    }
else
    {
        P3[k].coeff = P2[j].coeff;
        P3[k].expo = P2[j].expo;
        j++; k++;
    }
}

while (i < S1)
{
        P3[k].expo = P1[i].expo;
        P3[k].coeff = P1[i].coeff;
        i++; k++;
}
while (j<S2)
{
        P3[k].coeff = P2[j].coeff;
        P3[k].expo = P2[j].expo;
        k++; j++;
}

    *S3 = k ; // Not k-1 because indix ~~from~~ starts from 0
}.
```

Scanned by CamScanner

PROGRAM : Multiplication of two polynomials.

Logic : let two polynomials be P_1: x^3+2x^2+1. & P_2: $x+1$.
Then, they will be stored in form of array of structure ie

P_1:

0	1	2
1	2	1
3	2	0

i →

P_2:

0	1
1	1
1	0

j →

let P_3[] be the array of structure in which we will store our result. So, we will multiply each term of P_1[] with all terms of P_2[] and result will be stored in other array of structure P_4[] (for i != 0) and in P5[] (for i=0). ie. for only first term of P1, we will store in P5[] & otherwise in P4[].
After this, we will add P_5[] and P4[] and then we will store the result in P3[]. Now, we will copy P3[] to P5[] so that next time, resultant can be added to P4[].
All this work is done only for one iteration of i counter so, when whole of P1 is multiplied we will get the result in P3[].

P_5[]

0	1
1	1
4	3

(for i=0, multiplication saved in P5[])

P_4[]:

0	1
2	2
3	2

(for i=1).

For i=1: P_4[] is added to P5[] & result is stored in P3[].

∴ after i=1: P3[]

0	1	2
1	3	2
4	3	2

Now, P_3[] will be copied to P_5[].

P_5[]:

0	1	2
1	3	2
4	3	2

Now, i increases to i=2.
then, P_4[] will be :

0	1
1	1
1	0

Scanned by CamScanner

Now, $P_4[\,]$ and $P_5[\,]$ will be added to $P3[\,]$ and result will be saved in $P_3[\,]$.

So, status of $P_3[\,]$ will be:

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

Now, i increase to $i=3$, Now, $i < s1$ condition will not be satisfied (as $s1=3$). ∴ Our multiplication of $P_1[\,]$ & $P2[\,]$ is $P3[\,]$. The code for the same is given below:

* CODE:

```
void multPoly ( poly P1[], poly P2[], poly P3[] ,int s1 , int s2 , int *s3)
{
    poly P4[100], P5[100];
    int s4, s5;
    int i, j, k, l, m;
    i = j = k = l = m = 0;
    for (i=0 ; i<s1 ;i++)
    {
        l=0;
        for (j=0 ;j<s2 ; j++)
        {
            if (i==0)
            {
                p5[k].coeff = p1[i].coeff * P2[j].coeff;
                P5[k].expo = p1[i].expo + P2[j].expo;
                k++;
            }
            else
            {
                P4[l].coeff = P1[i].coeff * P2[j].coeff;
                P4[l].expo = P1[i].expo + P2[j].expo;
                l++;
            }
        }
```

Scanned by CamScanner

```
        if(i != 0)
        {
            s5 = k ; s4 = l ;
            addpoly (p5, p4, p3, s5, s4, s3);
            k=0 ; m=0 ;
            for (m=0; m < * s3 ; m++, k++)
            {
                p5[k].coeff = p3[m].coeff ;
                p5[k].expo = p3[m].expo ;
            }
        }
    }
}

int main()
{   poly p1[10], p2[10], p3[100] ;
    int s1, s2, s3 ;
    printf ("\n Enter no. of terms of first polynomial : ");
    scanf ("%d", &s1) ;
    printf (" Enter coeff. & expo part of poly m each term successivly : ");
    int i ;
    for (i=0 ; i<s1 ; i++)
    {
        scanf ("%d%d", &p1[i].coeff , &p1[i].expo);
    }
    printf ("\n Enter two no. of terms of second polynomial");
    scanf ("%d", &s2) ;
    printf (" Enter coeff. & expo part of each term successively : ");
    for (i=0; i<s2 ; i++)
    {
        scanf ("%d%d", &p2[i].coeff , &p2[i].expo);
    }
```

Scanned by CamScanner

```
multPoly (P1, P2, P3, s1, s2, &s3):
for (i=0; i<s3; i++)
{
        if(i!=s3-1)
        {
                printf("%dx%d + ", P3[i].coeff, P3[i].expo);
        }.

        else
        {       printf("%d x %d ", P3[i].coeff, P3(i).expo);
        }

    }

}.
```

Scanned by CamScanner

<< LINKED LIST >>.

* Like arrays, linked list is a linear data structure.
* Linear data structure means that data is stored in one-Dimensional fashion or in other works, data structure is said to be linear, if elements form a sequence. eg. Arrays, queues etc.
* Unlike arrays, linked list are not stored at contigous location, the elements are linked using pointers. Elements are scattered at random locations.
* A linked list is represented by a pointer to the first node of the linked list. The first node is called a head. If the linked list is empty, then value of head is NULL.
* Each node in a list consists of at least two parts:-
  a) data
  b) pointer to the next node.
* In C, we represent nodes using structures.

* Structure of a node :

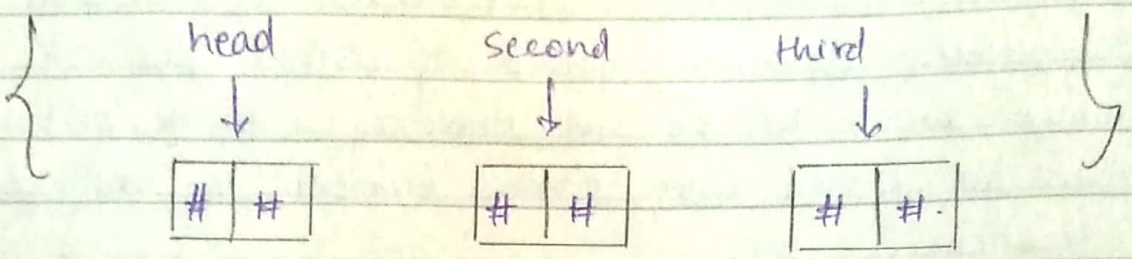
```
// Simple program to introduce linked list
# include <stdio.h>
# include <stdlib.h>
struct node
{   int data ;
    struct node * next ;
};
```

* Lets see some programs to build our concept :

```
struct node * head = NULL ;        // will create three pointers
struct node * second = NULL ;      // which will not point to any
struct node * third = NULL ;       // memory location initially.
```

Scanned by CamScanner

```
//allocate 3 nodes in the heap memory.
head = (struct node*) malloc (sizeof (struct node)); //①
second = (struct node*) malloc (sizeof (struct node));
third = (struct node*) malloc (sizeof (~~siz~~ struct node));
```

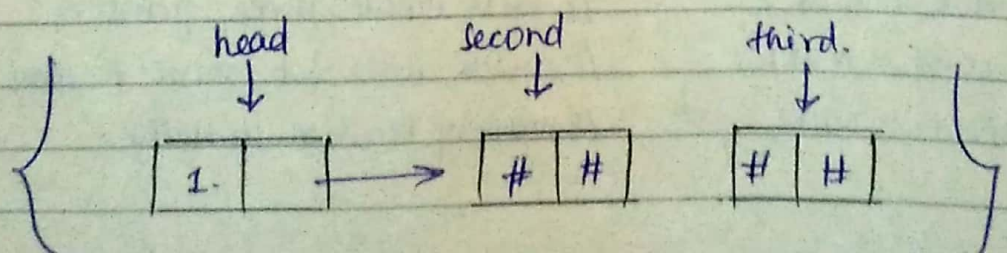


/* Now, what function does malloc serve here? Malloc ( ) allocates memory dynamically at runtime. Now, ∵ some memory has to be allocated, the question arises what will be the size of the memory to be allocated? Since, we requi a node so size of this node will be passed as arguement to malloc function i.e. (struct node) will be passed as arguement. (struct node*) part in statement 1 signifies that pointer 'head' is of struct node* type. */

// # means random (garbage) values present at the locations.

Now, let's fill in data part and pointer part of each node

```
head → data = 1 ; //1 is assigned to data part of that node
head → next = second ;   // to which head is pointing.
// pointer part of that node to which head is pointing contai
// address of second node.
```

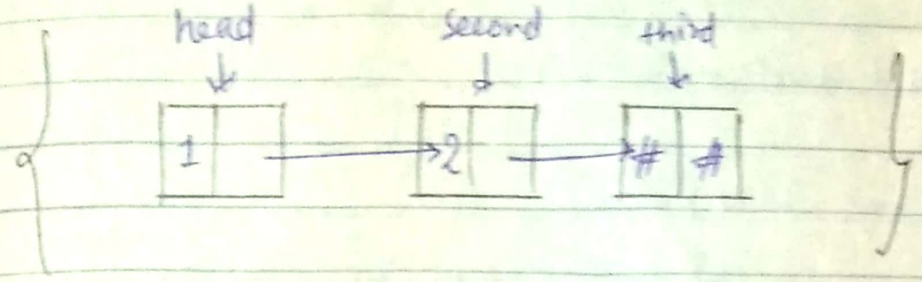


Scanned by CamScanner

Similarly,

second → data = 2 ; // assign data to 2nd node

second → next = third ; // link second node with 3rd node

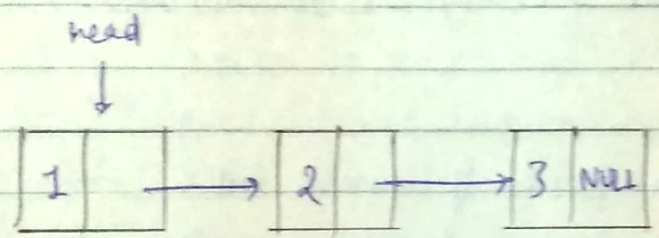


Similarly,

third → data = 3 ; // assign data to 3rd node.

third → next = NULL ;

/* pointer part of last node is always NULL to indicate that our list has terminated here
*/



/* In above fig, we haven't shown second and third pointer because now they are of no use as the whole list can be traversed using head pointer & next pointers of each node
*/

NOTE : Malloc() can be used only when #include <stdlib.h> statement occurs in the program. OR.
Header file of Malloc() is Stdlib.h.

Suppose, we wish to allocate array using malloc, can be done as:

```
int * ptr;
ptr = malloc ( 10 * sizeof(int) );        // int a ptr[10];
```

both are equivalent.

Scanned by CamScanner

* Let us traverse the created list and print data of each node
* PrintList () is a general function which prints any list.

```
void printList (struct node * n)
{
        while (n != NULL)
        {
                printf ("%d", n->data) ;
                n = n->next ;
        }
}
```

<< Linked list v/s Arrays >>.

Both Arrays and linked list are used to store data of simil type, but Both have some advantages & disadvantages over each other.

following are points in favour of linked list :

* The size of array is fixed : So, we must know the upper limit of el no. of elements in advance. Also, allocated memory is equal to the upper limit, irrespective of usage, ∴ some memory is wasted.

```
eg:     int a[10] ;
        int n ;  cin>>n ;  // 2.
                for (int i = 0 ; i < n ; i++)
                        { cin >> a[i];
                        }
        // So, rest 8 blocks of memory will be wasted.
```

* Inserting or deleting an element in an array can be expensive (means high Time Complexity). because shifting occurs.

Scanned by CamScanner

eg: Suppose, we maintain sorted list of IDs in an array ID[].
ID[] = [1000, 1010, 1050, 2000, 2040].
if we want to add new ID 1005, st. sorted order is retained, we have to move all elements after 1000 & then put at 1st index.

Similarly, for deletion (), suppose 1010 is to be deleted, ~~so~~ after 1010, everything is to be moved.

* Advantages of Linked list over Arrays.
→ Dynamic size
→ Insertion / deletion is quite easy.

* Drawbacks of Linged list.
→ Random Access is not allowed. eg: If we have to reach 5th node, then we have to go through 1st, 2nd, 3rd, 4rd node & then to 5th node. ie. we have to access elements sequentially starting from first node. So, binary search is not performed with linked lists. (unable to random access middle element).

→ Extra memory space for a pointer is required with each element (node) of list.

→ Due to contiguous memory Allocation (ie. elements are stored sequentially in memory next to each other). in case of Arrays, they have better cache locality that can make pretty big- difference in performance.

Scanned by CamScanner

PREREQUISITES - FOR STUDING LINKED LIST.

* void pointer (void*)

whenever, we declare a pointer, we also tell the compiler that what is type of pointer ie. int*, char* etc ie. we tell the compiler that what type of variable it will point to.
Now, what is the need of doing so? Eventually, a pointer stores only the memory address irrespective of the type of variable it is pointing to. So, is there any generic pointer which will point to all types of variable? Yes, there's a generic pointer called void pointer (void*) but using it, has some limitations ie. we cannot dereference the void pointer. ie we cannot access the value stored in the variable to which the void pointer is pointing. That's why we tell the compiler that what will be type of variable to which pointer will be pointing.

Examples:

```
#include <stdio.h>
int main()
{
    int a = 1025;
    int *p;
    p = &a;
    printf("size of integer is %d", sizeof(int));
    printf("Address = %d", p, *p);      // Statement 1.
    char *p0;
    p0 = p;  // Error as p0 is a char pointer & a char poin
             // cannot store address of an integer variable.
    p0 = (char*)p;  // So, to avoid above error, we type cast the
                    // pointer p.
    printf("size of char is %d \n", sizeof(char));
    printf("Address = %d , value = %d \n", p0, *p0);  // statement 2.
}
```

Scanned by CamScanner

OUTPUT: Size of integer is 4 bytes
Address = 5373032 , value = 1025
Size of char is 1 bytes
Address = 5373032 , value = 1.

Explanation: Binary equivalent of decimal no. 1025 is:

Byte 3	Byte 2	Byte 1	Byte 0	
00000000	00000000	00000100	00000001	
203	202	201	200	← Address (let).

(Each Byte will have some address)

- *p means go to the address stored in p and consider 4 bytes ahead from that memory address (4 bytes because p is int* type) ie. byte 0, byte 1, byte 2, byte 3 & print it.
  So, combination of 4 bytes ie 1025 will be printed.

- *p0 means go to the address stored in p0 and consider 1 byte only ahead from that memory stored (1 bytes because p0 is char*) ie. byte 0 & print it.
  Decimal equivalent of byte 0 ie binary equivalent of 00000001 ie. 1 will be printed.
  decimal

* when statement 1 is replaced by printf("Address = %d, value = %d\n", p+1, *(p+1));
  & Statement 2 by printf("Address = %d , value = %d\n", p0+1, *(p0+1));

Output: size of integers is 4 bytes
Address = 4456036 , value = 1025
Address = 4456046 , value = -858993460
Size of char is 1 Bytes
Address = 4456036 , value = 1
Address = 4456037 , value = 4.

control moves from address p0 to p0+1 ie. 1 byte ahead & prints decimal equivalent of Byte 1.

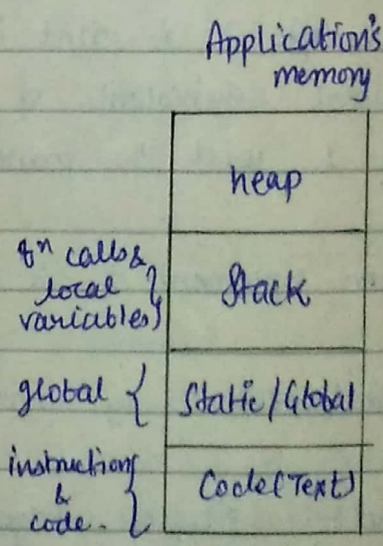
Scanned by CamScanner

Example 2:

```
int main()
{   int a = 1025;
    int *p;
    p = &a;
    printf("size of integer is %d", sizeof(int));
    printf("Address = %d, value = %d\n", p, *p);
    void *p0;
    p0 = p;  // void pointer p0 can store address of any type of variable.
    printf("Address = %d", p0);      // allowed.
    printf("Address = %d", p0+1);    // not allowed because p0 will
                                     // not know that how many
                                     // bytes are to be considered ahead
                                     // from P0.
    printf("value = %d", *p0);       // Not Allowed.

}.
```

* POINTERS & DYNAMIC MEMORY.

Memory reserved in computer system is divided into four sections i.e (code/text) memory, static/global section, stack section & heap section.

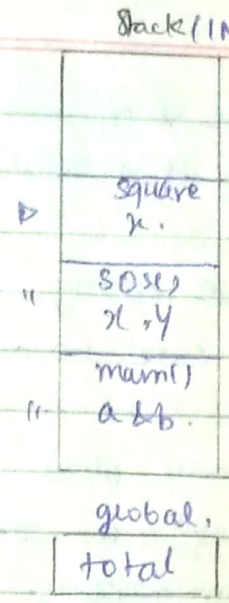


Example:

```
int total;
int square (int x)
{   return x*x; }
int square of sum (int x, int y)
{   int z = square(x+y);
        return z;
}
```

Scanned by CamScanner

```
int main()
{   int a=4, b=4;
    total = square of sum(a,b);
    printf("output = %d", total);
}
```



During compilation, stack section of memory is allocated & its size is always fixed (say 1MB). when ~~function is executed~~ program execution starts, first main() is executed & for its execution, some part of stack segment of memory is given to main() and all local variables (a & b) are created within this stack frame of main(). Then, within main(), square of sum (or fn() is called and then this f^n is pushed over main() in stack memory and then execution of SOS() starts. within SOS(), square() is called and hence square() is pushed over SOS() in stack memory. when whole execution of square() ends, then it returns the square and square() is poped from stack memory & then SOS() comes on top on stack. & hence its execution begins. Then, SOS() returns sum of squares, get poped from stack & main() comes on top of stack & its execution is continued. Then, it calls printf(), which comes on top, gets executed, prints output & then poped from stack & hence main() execution continues and then main() is poped from stack, when all of its statements are executed. And stack memory becomes free.

But still, stack memory has fixed size. so, when there are a lot of f^n calls inside function, then, when stack memory is filled & then condition is called stack overflow.

Scanned by CamScanner

Suppose, there's a long recursive code in our program, then stack overflow may happen.
So, in case of Stack memory, dynamic memory allocation can't be done.

f().
E()
D()
C()
B()
A()

There's another memory called heap memory. (free pool of memory) i.e. memory allocated to any variable at runtime is always allocated in heap memory. It is used (implemented) through, malloc(), calloc(), realloc(). Memory allocated dynamically can be accessed anywhere we want, i.e. we can manipulate the scope of dynamic memory. So, ~~to~~ deallocation of this memory when our use is over, is the responsibility of programmer. Deallocation is implemented through free().

* MALLOC(). {to allocate Block of memory}.
- signature:

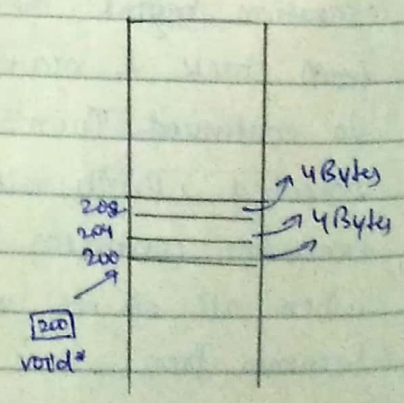
void* malloc (size_t size)

unsigned int. (so that size cannot be -ve).

takes size of memory block to be allocated dynamically as argument.

returns memory address of allocated memory

```
int *p = (int*)malloc (3* sizeof(int));
    printf("%d", p) ; // 200
```



- calloc()
- signature

void * calloc (size_t num, size_t size).

- implementation

int *p = (int*) calloc (3, sizeof(int)) ;

no. of elements

size of datatype in bytes.

Scanned by CamScanner

* Difference b/w malloc() & calloc()

* Memory allocated using calloc() are initialized with zero implicitly and mememory allocated using Malloc() contains garbage value.
* Syntax.

* REALLOC()

It is used to manipulate the size of already existing block of memory created dynamically.

Signature : void* realloc (void * Ptr , size-t size) ;

↓ Pointer to already existing block of memory.

↓ size of new memory

NOTE : Any memory that is dynamically allocated remains allocated till lifetime of program unless we explicitly deallocate it.

* FREE ()

Syntax : If p be the pointer to memory address allocated dynamically, then, that memory can be made free using,

free (P) ;

By this statement, that memory becomes unreserved i.e. malloc(), if used again, can give that memory address to any other dynamically allocated Block.

P pointer still have memory address of deallocated Block, so it can still access that memory Address. But, it is not a good practice to manipulate or print values at deallocated blocks.

So, pointer P is set to NULL after free. i.e.

```
free(P);
P = NULL;
```

Even after using free() statement, we can modify the values at that Address using pointer P if its ~~not~~ is not set to NULL.

Scanned by CamScanner

Examples: use of malloc(), free();

```
int main()
{   int n;
    printf("Enter size of Array \n");
    scanf("%d",&n);
    int * A = (int*) malloc (n*sizeof(int)); //dynamically allocated ar
    for (int i=0 ; i<n ; i++)
     {
        A[i]=i+1;
     }
    free(A);
    A = NULL ;  // After free, adjust pointer to NULL.
    for (int i=0 ; i<n; i++)
     {
        printf ("%d", A[i]);
     }

}.
```

Example 2: using of realloc()

```
int main()
{   int i,n;
    printf("Enter size of Array \n");
    scanf ("%d",&n);
    int *A = (int*) malloc (n*sizeof(int)); //dynamically allocated
                                            // Array
    for (i=0 ; i<n; i++)
    {
        A[i] = i+1;
    }
    int *A = (int*) realloc (A, 2*n*sizeof(int)); // Statement 1.
    Printf ("Prev Block address = %d , new address = %d \n", A, B[]);
```

Scanned by CamScanner

```
    for (int i= 0 ; i< 2*n ; i++)
    {
        printf("%d\n", A[i]);
    }

}.
```

OUTPUT:

Enter size of Array

5

Prev Block Address = 9900128 , new address = 9900128

1

2

3

4

5

-842150451

-842150451

-842150451

-842150451

-842150451.

smaller size.

• If statement 1 would be : int *A = (int*) realloc (A, (n/2)*sizeof(int));

Output:

Enter size of Array

5

Prev Block Address = 9461736 , new Address = 9461736.

1 } (n/2) = (5/2) = 2.5 ≃ 2 ∴ first 2 blocks are

2 } retained & others are free now.

-3368019 } ∴ random values.

-1414812757

-1414812757

Scanned by CamScanner

• Realloc() can be used as free() :

In previous example, if statement 1 would be:

```
int *A = (int *) realloc(A, 0);   // Equivalent to free()
```

decreasing size to zero ⇒ deallocating memory blocks.

• Realloc() can be used as malloc()

In previous ex, if statement 1 would be :

```
int *B = (int *) realloc(NULL, n*sizeof(int));
                 // equivalent to
                 // malloc().
```

* In linked lists, we will use following structure for node:

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

Scanned by CamScanner

<< LINKED LIST >>

<< OPERATIONS >>

Program 1 : Printing a linked list.

```
void display ( struct node * p )
{
        while (p != NULL)
        {
            printf ( " %d " , p→ data);
            p = p → next ;
        }
        printf("\n");
}.
```

Program 2: Count no. of nodes in a linked list.

```
int countnodes ( struct node *p )
{       int count = 0 ;
        while ( p != null )
        {       count ++;
                p = p1 → next ;
        }
        return count;
}
```

Program 3: Insert an element in linked list st. ascending order of linked list is retained.

```
struct node* insert ( struct node * start , int x)
{       struct node * q ;
        q = (struct node*) malloc ( sizeof( struct node) ) ;
        q → data = x ;
        if ( start == NULL || (start != NULL && x < start → data))
        {       q → next = start ;
                start = q ;
        } // For the case when list is empty or element is to be
```

Scanned by CamScanner

```
// inserted at the first position
else
{       struct node * start1 = start ;
        while (start1->next != NULL && start1->next->data < x )
        {
                p = start1 = start1->next ;
        } // we will not traverse using start because that
          // will change the head of our linked list.

        q->next = start1->next ;
        start1->next = q ;
}
    return start ;
}.
```

PROGRAM 4: Dellete a node whose data is given.

```
struct node* deletenode ( struct node * start , int x)
{
        struct node * p = start ;   struct node * q ;
        if ( start == NULL )
            return start ;
        if ( start -> data == x)
        {       start = start -> next ;
                free(p);
                return(); start .
        }
        while ( p->next != NULL && p-> next ->data != x )
        {
                p = p->next ;
        }
        q = p->next ;
```

Scanned by CamScanner

```
    if (q != NULL)
    {
        p->next = q->next;
        free(p);
    }
    return start;
}
```

PROGRAM 5: Insert a node at the end of linked list (linked list data parts are not sorted).

```
struct node* insertEnd (struct node* start, int x)
{
    struct node* q = (struct node*) malloc (sizeof(struct node));
    q->data = x;   q->next = NULL;
    if (start == NULL || start->next == NULL)
    {
        q->next = start;
        start = q;
    }
    else
    {
        struct node *p;
        p = start;
        while (p->next != NULL)
        {
            p = p->next;
        }
        p->next = q;
    }
    return (start);
}
```

Scanned by CamScanner

PROGRAM 6: Inserting a node at beginning.

```
struct node * InsertAtBeginning ( struct node * start, int x).
{
        struct node *q = (struct node*) malloc (sizeof (struct node));
            q -> data = x ;
            q -> next = start ;
            start = q ;
            return start ;
}

int main ()
{       struct node* start ;
            start = NULL ;    // Empty linked list.
        printf ("How many nos.");
        int n, i, x ;
        scanf ("%d", &n);
        for (i=0 ; i<n; i++)
        {       printf(" Enter the no.\n");
                scanf ( " %d ", & x) ;
                start = InsertAtBegining ( start, x);
        }
}.
```

PROGRAM 7: Inserting a node at n^{th} position.

```
struct node *  Insert ( struct node * start, int n, int x)
{       struct node *q = (struct node*) malloc (sizeof (struct node));
            q -> data = x ;
        if (n==1)
        {       q -> next = start ;
                Start = q ;
        } // If element is inserted at 1st posn.
```

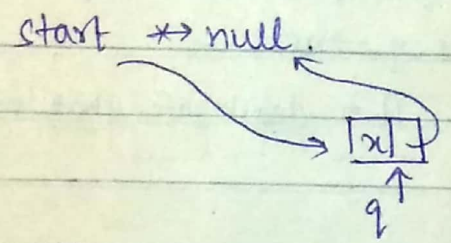
Scanned by CamScanner

```
else
{       struct node *p = start;
        int i=0 ;
        for ( i=0 ; i<n-2 ; i++)
        {
                p = p->next;
        }
            q->next = p->next;
            p->next = q;
}

Return (start);
}.
```

logical view of above code:

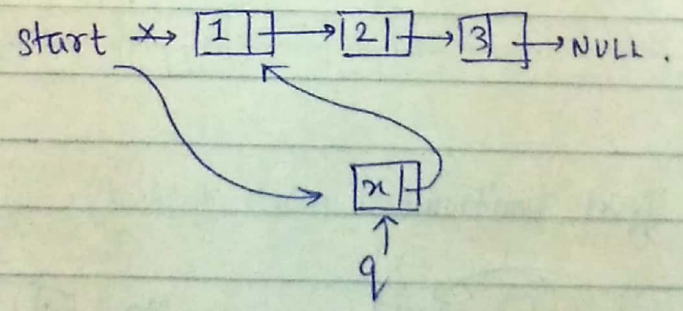
Case i) n = 1. ie

i)



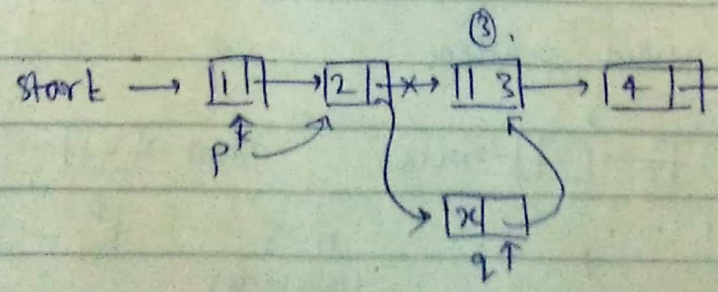
(✓).

ii) Filled linked list



(✓)

Case ii) n != 1 ie n > 1.



(✓)

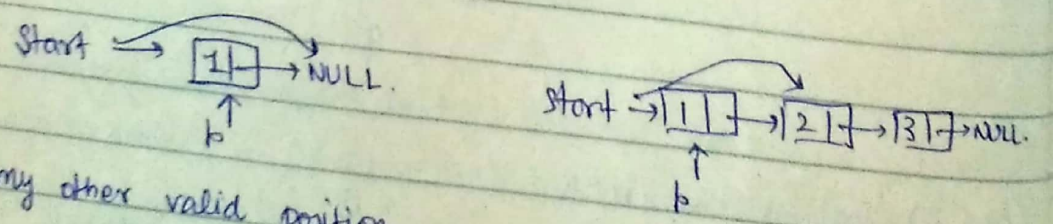
Scanned by CamScanner

PROGRAM 8: Delete a node at n^{th} position.

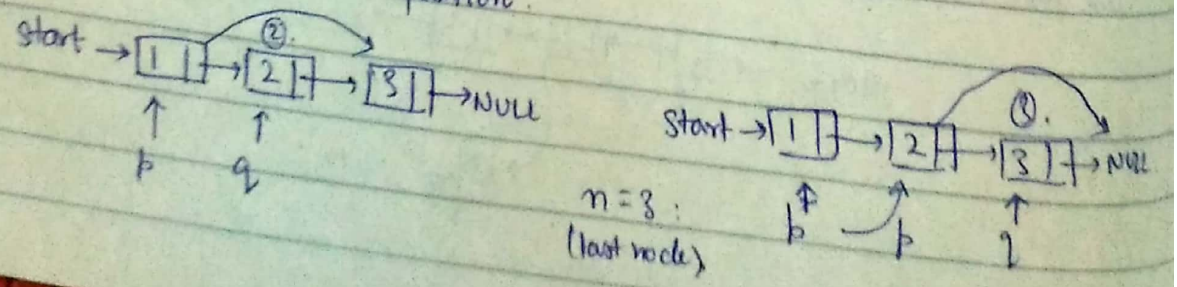
```
struct node * Delete ( struct node *start , int n)
  {
        struct node * p = start ;
        if (n==1)
         {      Start = start -> next ;
                free(p);
         }
       else
         {      int i;
                for (i=0 ; i< n-2 ; i++) // to reach (n-1)th node.
                {
                        p = p -> next ;
                }
                struct node *q = p-> next ; //q points to nth node.
                p-> next = q -> next ;
                free (q) ; // to dealbocate that node.
         }.

        return (start);
  }.
```

logical view:

Case i) deletion at first position

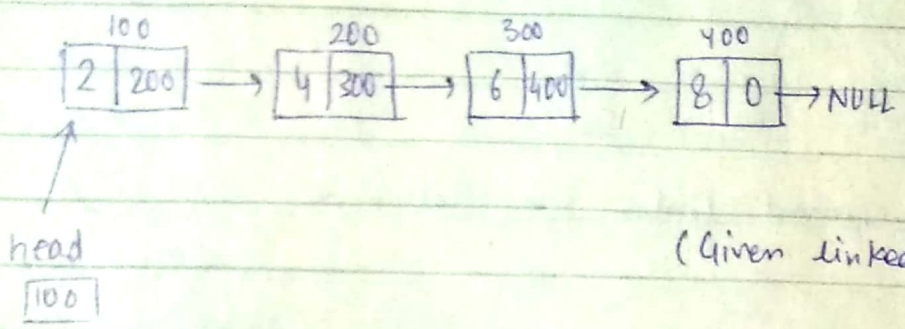


Case ii) at any other valid position.



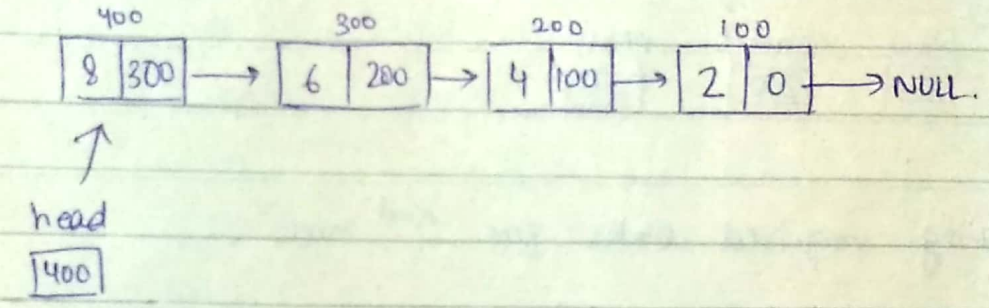
PROGRAM 9 : Reverse a linked list using iterative method.

logic :

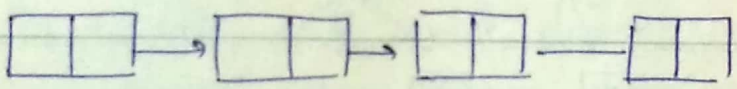


(Given linked list).

Output should be as shown:

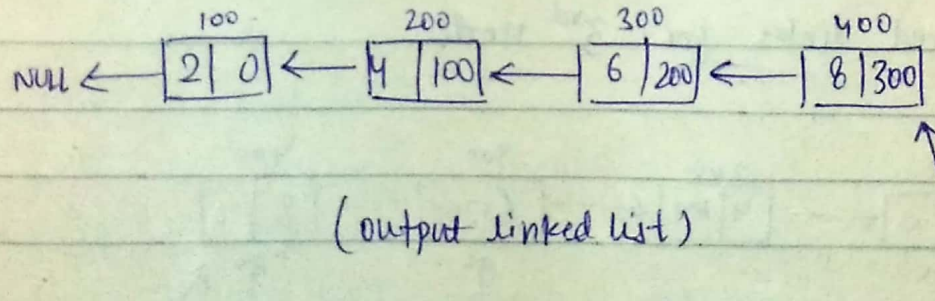


OR



(x).

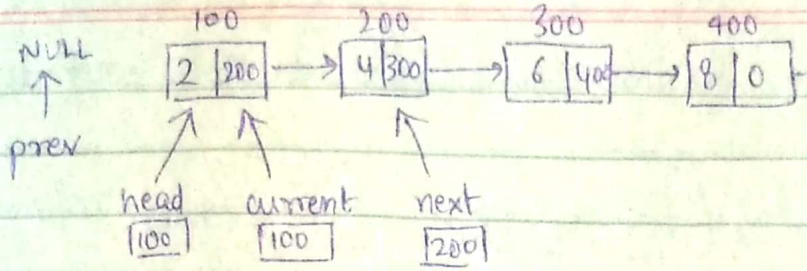
OR



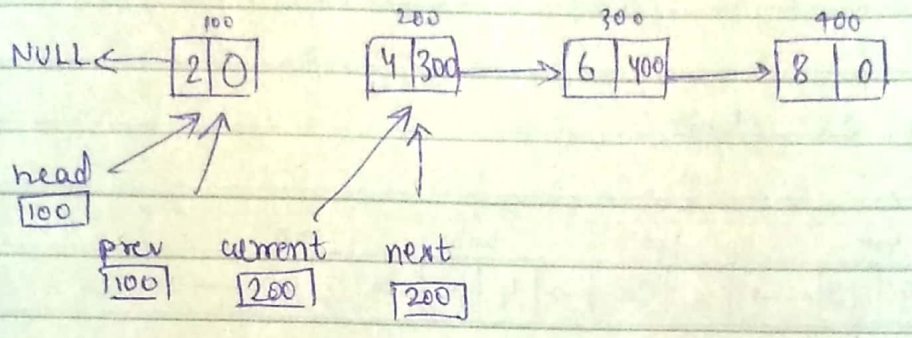
(output linked list).

So, we will traverse through each node and set the required links. Since, we have to keep track of previous node to make links with current node,

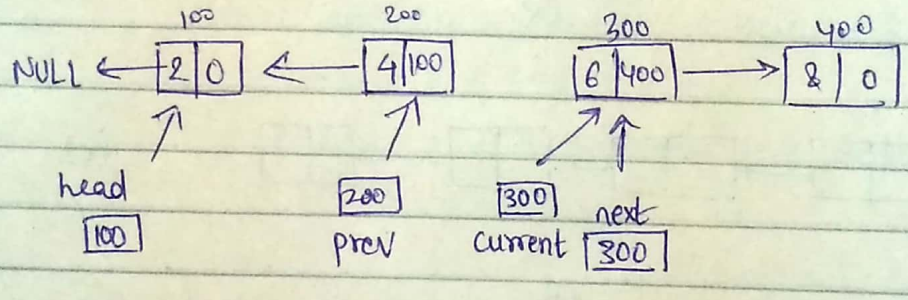
Scanned by CamScanner



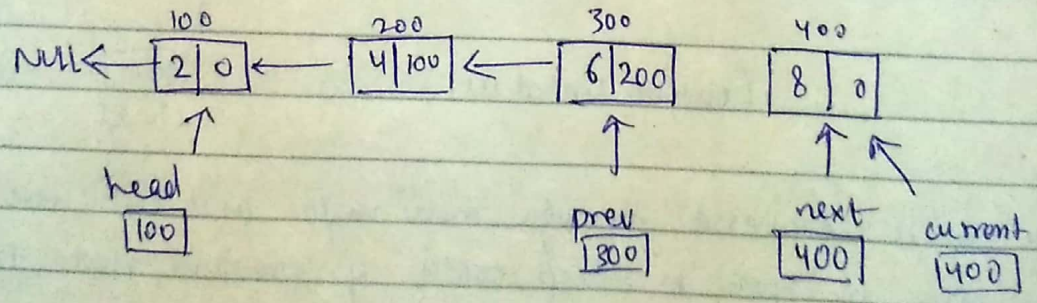
Step 1: Setting required links for first node



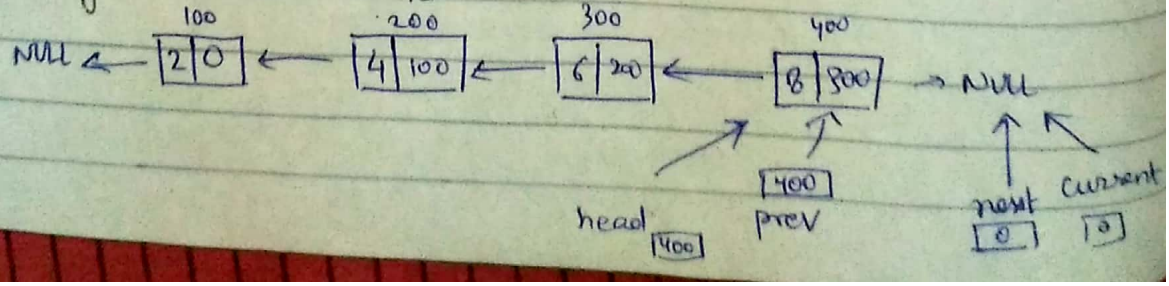
Step 2: Setting required links for 2nd node.



Step 3: Setting required links for 3rd node



Step 4: Setting links for 4th node.



Scanned by CamScanner

Code :

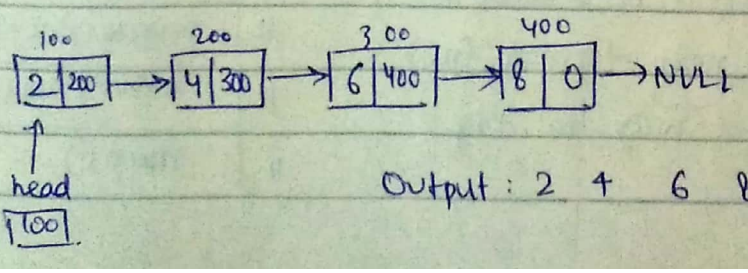
```
struct node* Reverse ( struct node* head )
{
        Struct node* current , * prev , * next ;
        current = head ;
        prev = NULL ;
        while (current != NULL)
        {
                next = current -> next ;
                current -> next = prev ;
                prev = current ;
                current = next ;
        }
        head = prev ;
        return head ;
}.
```

PROGRAM 19 : To traverse and print elements of a linked list in forward order. (using recursion)

Code :

```
void print (struct node* p)
{
        if (p == NULL)
            return ;
        printf (" %d ", p->data) ;
        print (p->next) ;
}.
```

main ( )
↓
print (100) (x)
↓
print (200) (x)
↓
print (300) (x)
↓
print (400) (x)
↓
print (NULL) (x)



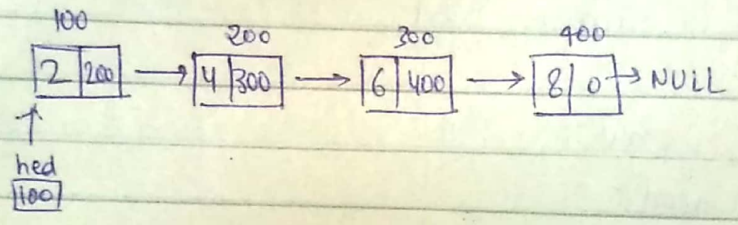
Output : 2 4 6 8

Scanned by CamScanner

PROGRAM 11 : Print elements of a linked list in reverse order. (using recursion))

```
void reversePrint (struct node *p)
{
        if (p == NULL)
            return ;
        ReversePrint (p->next);
        printf (" %d ", p->data) ;
}
```

logic :



In recursive traversal, firstly, main() will call reversePrint (100) which call reverseprint (200), then reverseprint (300), then reverseprint (400) and when reverseprint (NULL) will be called, condition (p == NULL) will be satisfied & this f^n reverseprint (NULL) will be stopped & control will return to the function which called reverseprint (NULL) i.e. reversePrint (400), then data element of node with address 400 will be printed and then execution of f^n. reverseprint (400) is completed & control returns to reversePrint (300), then its data element is printed & so on.

Recursive function's execution is similar to the insertion & deletion of function from stack memory.

Stack memory.

↳	reverse print (NULL)
‖	reverseprint(400)
‖	reverseprint(300)
‖	reverse print (200)
‖	reverse Print(100)
‖	main()

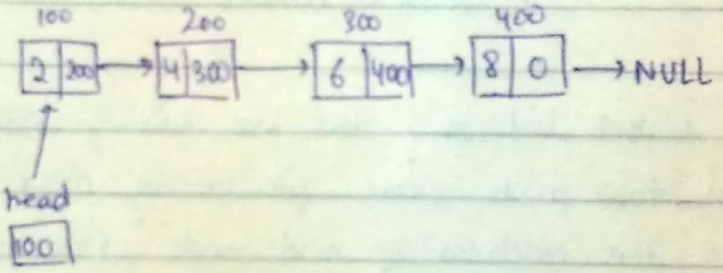
NOTE: Firstly, execution of topmost function in stack memory is finished, then it is deallocated from Stack Memory and then next topmost fn() is executed. This is the best way to dry run recursive functions

Scanned by CamScanner

PROGRAM 12 : Reverse a linked list using recursion.

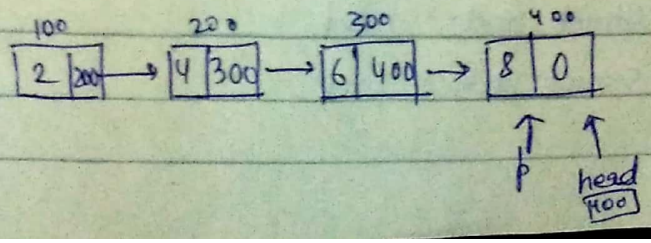
```
struct node * head ;
void Reverse ( struct node * p)
{
        if ( p->next == NULL)
        {
                head = p ;
                return;
        }
        Reverse (p->next) ;
        struct node * q ;
          q = p->next ;
          q->next = p ;
          p->next = NULL ;
}.
```

Initially :



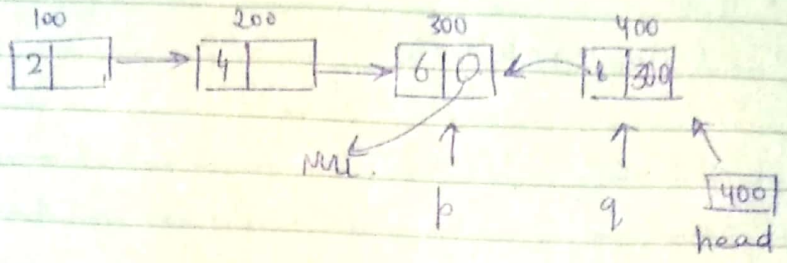
main() called reverse (head) i.e. reverse (100), then this f^n calls reverse (200) and so on, when reverse (400) calls rever is called then p->next == NULL is satisfied and head becomes p i.e. head becomes 400 & control is returned to reverse (300), then q points to 400 (address) node and q->next is changed to the previous node and link part of p-node is set to null. when whole recursion execution is finished, linked list gets reversed.

After calling reverse(400) :

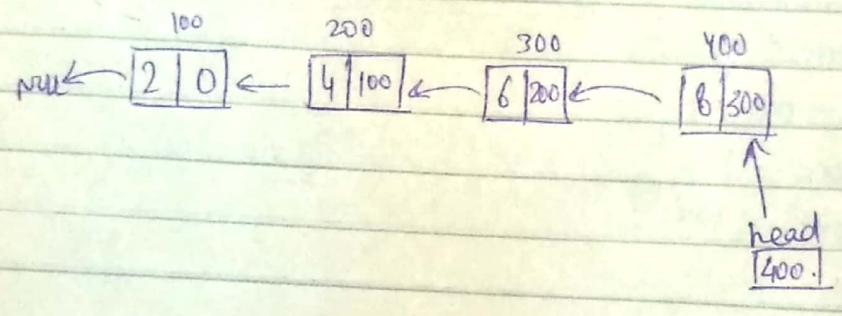


Scanned by CamScanner

After execution of reverse (300) is finished, status of list is :



After execution of reverse(100) is finished, status of list is :



• DOUBLY-LINKED LIST.

In a singly-linked list or what we simply call linked list, each node has two parts, one for storing data and other part is for storing the address of next node. Due to such a structure, reverse traversing is somewhat difficult to do. So, a concept of doubly-linked list is adopted, in which each node has three parts, one for storing data, one for storing address of next node and other for storing address of previous node. Using or keeping track of previous node, reverse traversing becomes easy to handle. Structure of doubly-linked list is :

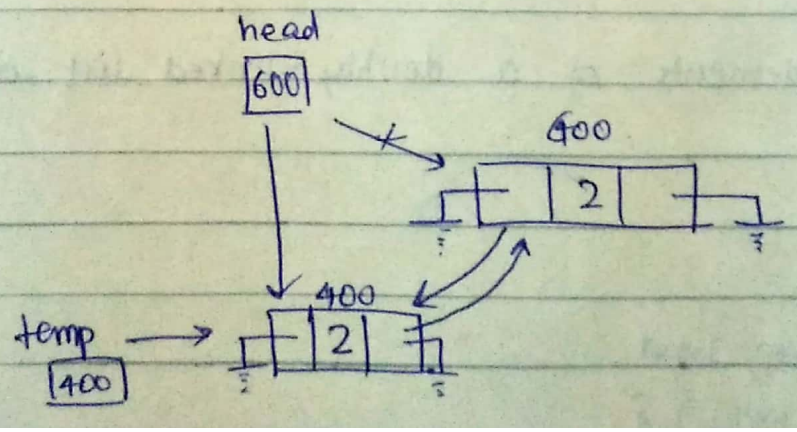
```
Struct node
  {     int data;
        Struct node * next;
        Struct node * prev;
  };
```

Scanned by CamScanner

PROGRAM 13: To insert a node in beginning of doubly-linked list.

```
struct node* head ;

struct node* getnewnode ( int x)
{       struct node* newnode = (struct node*) malloc(sizeof (struct node))
        newnode -> data = x ;
        newnode -> next = NULL ;
        newnode -> prev = NULL ;     return newnode ;
} // fn for allocating space to a node & initializing it.

void InsertAtHead (int x)
{
        Struct node* temp = getnewnode (x) ;
        if (head == NULL)
        {       head = temp;
                return;
        }
        head -> prev = temp ;
        temp -> next = head ;
        head = temp ;
}
```

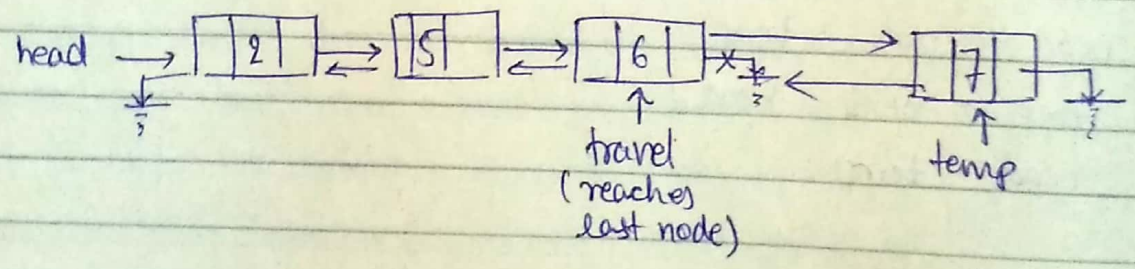
dry run:



Scanned by CamScanner

PROGRAM 14: To insert at node in end of doubly linked list.

```
void insertAtTail (int x)
 {
        struct node* temp = getnewnode (x);
        if (head == NULL)
          {     head = temp;
                return;
          }
        struct node* travel = head;
        while (travel -> next != NULL)   // to reach last node in list.
          {
                travel = travel -> next;
          }
        travel -> next = temp;          } to make links.
        temp -> prev = travel;
 }
```



PROGRAM 15: To print elements of a doubly-linked list in forward order.

```
void Print ()
 {      struct node* temp = head;
        while (temp != NULL) {
                printf(" %d ", temp->data);
                temp = temp -> next;
        }
        printf("\n");
 }
```

Scanned by CamScanner

PROGRAM 16 : To print elements of a doubly linked list in reverse or

```
void ReversePrint ()
{
        struct node* temp = head ;
        if (temp == NULL)
                return ;    // empty linked list.
        while (temp -> next != NULL)   // going to last node
        {
                temp = temp -> next;
        }
        while ( temp != NULL)
        {
                printf ("%d", temp -> data);
                temp = temp -> prev;
        }
        printf("\n");
}.
```

Scanned by CamScanner

<< INTRODUCTION TO STACKS >>

- Stack as an ADT

A list with the restriction that insertion and deletion can be performed only o from one end, called top. [LIFO]

- Operations

i) Push(x)
(ii) Pop()
iii) Top()
iv) Isempty()

} constant time or $O(1)$

- Applications
  - Function calls / recursion
  - undo in an editor
  - Balanced paranthesis

* ARRAY IMPLEMENTATION OF STACKS.

PROGRAM 17: Write push() and pop() to insert an element on to the stack and delete an element from stack resp.

```
#include <stdio.h>
# define maxs 20

struct stack
{       int a[maxs];
        int top;
};

void initialize (struct stack *s)     // to initialize top=-1 for empty stack
{       s->top = -1;
}
```

Scanned by CamScanner

```
void push (struct stack *s, int x)   // to insert element onto stack
{
    if(s->top == maxs-1)
    {
        printf("\n Stack overflow");
        return;
    }
    s->top++;
    s->a[s->top] = x;
}
```

passed by reference (by pointer)

```
int pop (struct stack *s)   // to delete topmost element.
{
    if(s->top == -1)
    {
        printf(" Stack is empty ");
        return;
    }
    int x = s->a[s->top];
    s->top--;
    return x;
}
```

```
void display ( struct stack(s))
{
    int i = s.top;
    if( s.top == -1)
    {
        printf("Stack Empty");
        return;
    }
    printf(" Status of Stack :\n");
```

passed by value because changes need not to be reflected in main().

Scanned by CamScanner

```
        while (i!=-1)
        {
            printf("%d", s.a[i]);
            i--;
        }
    }

int main()
{
    struct stack s;
    initialize(&s);
    pop(&s);
    display(s);
    push(&s, 2); display(s);
    push(&s, 4); display(s);
    push(&s, 6); display(s);
    push(&s, 8); display(s);
    push(&s, 10); display(s);
    int x = pop(&s);
    printf("\n Poped element : %d", x);
    display(s);
    initialize(&s);
    display(s);
}
```

10	← top
8	
6	
4	
2	

OUTPUT: Status is Empty Stack Empty
Status of Stack : 2
Status of Stack : 4 2
Status of Stack : 6 4 2
Status of Stack : 8 6 4 2
Status of Stack : 10 8 6 4 2
Poped element : 10
Status of Stack : 8 6 4 2
Stack Empty

Scanned by CamScanner

NOTE :

```
struct stack
{   (int) a[maxs];
    int top;
};
```

, it is the type of element which will be pushed onto the stack.

So, if we want to push characters onto the stack, then structure of stack will be modified to :

```
struct stack
{       char a[maxs];
        int top;
};
```

PROGRAM 18 : Write complete function to reverse a string using stack.

```
#define <stdio.h>
# define maxs 20

struct stack
{
    char a[maxs];
    int top;
};

void initialize (struct stack *s)
{
    s->top = -1;
}

void push ( struct stack *s, char &x )
{
    if (s->top == maxs-1)
    {       printf (" \n STACK OVERFLOW ");
            return;
    }
```

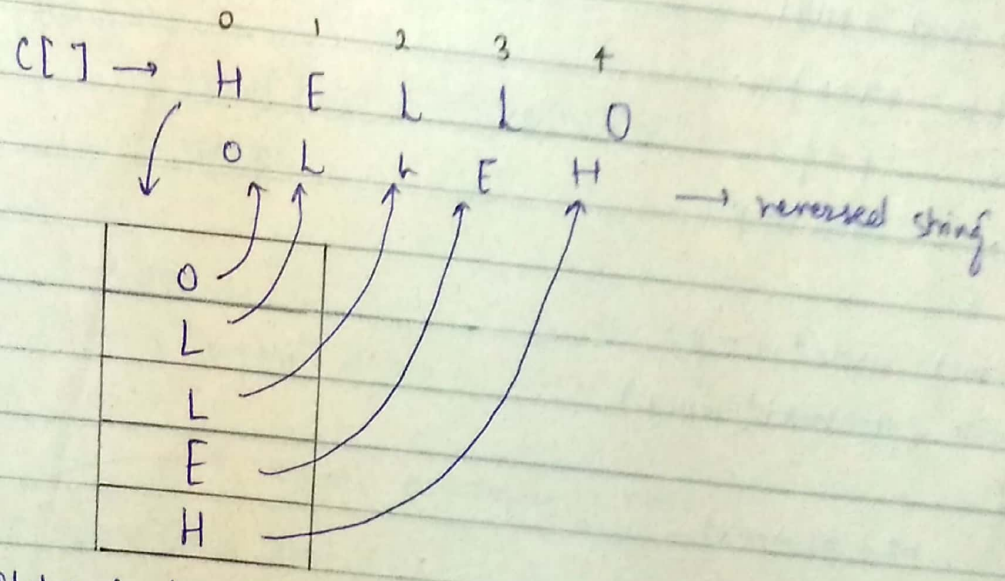
Scanned by CamScanner

```
        s->top++;
        s->a[s->top] = x;
}

char pop(struct stack *s)
{
        if (s->top==-1)
        {
                printf("\n Stack is Empty");
                return;
        }
        char x = s->a[s->top];
        s->top--;
        return x;
}

void reverse (char c[])
{
        struct stack s1;
        initialize (&s1);
        int i=0;
        while (c[i]!='\0');
        {
                push(&s1, c[i]);
                i++;
        }
        i=0;
        while (c[i]!='\0');
        {
                c[i] = pop(&s1);
                i++;
        }
}
```

Scanned by CamScanner

```
int main()
{
    char c[51];
    printf("Enter a string");
    gets(c);
    reverse(c);
    printf("Reversed string": %s", c);
}.
```

Logic:



(Status of stack after each character of string is pushed onto it)

PROGRAM 19: To reverse a linked list using stack.

```
#include <stdio.h>
# define max s 20

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

Scanned by CamScanner

∵ we have to push address of a node in each push operation of stack ∴ type of array is struct node *.

```
struct stack
{       struct node * a[maxS];
        int top;
}


struct node* insert (struct node *p, int x) // to insert node at end of
{                                           // linked list.
    struct node * q = (struct node *) malloc( sizeof (struct node));
    q -> data = x ;
    q -> next = NULL ;
    if ( p == NULL)
    {       p = q ;
            return p ;
    }.
    Struct node* p1 = p;
    while (p1 -> next != NULL)
    {
        p1 = p1 -> next ;
    }
    p1 -> next = q ;
    return p;
}


void print (struct node * p) // print linked list in forward order
{                            // recursive.
    if (p == NULL)
    {   return;
    }
    printf(" %d ", p -> data);
    print (p -> next);
}
```

Scanned by CamScanner

// Stack functions → push(), pop() & initialize().

```
void initialize (struct stack *s)
{
        s→top = -1;
}

void push (struct stack *s, struct node* x)
{
        if(s→top == maxS-1)
        {
                printf("\n STACK OVERFLOW");
                return;
        }
        s→top++;
        s→a[s→top] = x;
}
```

x is of type pointer to node ie stores address of node and this is what we need to push onto the stack.

returns address of a node.

```
struct node * pop (struct stack *s)
{
        if (s→top==-1)
        {       printf(" STACK EMPTY");
                return;
        }
        struct node *q x = s→a[s→top];
        s→top--;
        return x;
}
```

address of head node of linked list to be reverse.

```
struct node* reverse (struct node *head)
{       struct stack S;
        initialize (&s);
```

Scanned by CamScanner

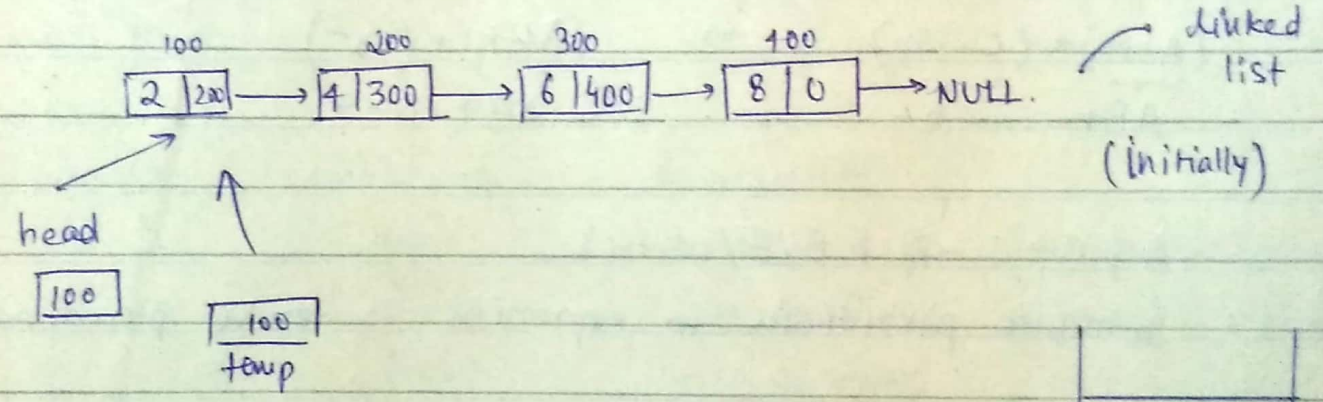
```
    struct node* temp = head;
    while (temp != NULL)
    {       push(&s, temp);
            temp = temp->next;
    }
    temp = pop(&s);
    head = temp;
    while (s.top != -1)
    {
        temp->next = pop(&s);
        temp = temp->next;
    }
    temp->next = NULL;
    return head;
}

int main()
{       struct node * head = NULL;
        head = insert (head, 2);
        head = insert (head, 4);
        head = insert (head, 6);
        head = insert (head, 8);
        head = insert (head, 10);
        print(head);
        print("\n");
        head = reverse(head);
        print(head);
}.
```

OUTPUT:

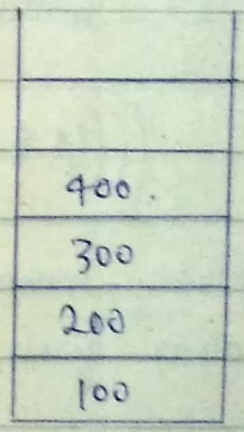
```
2   4   6   8   10    // given list
10  8   6   4   2     // reverse list
```

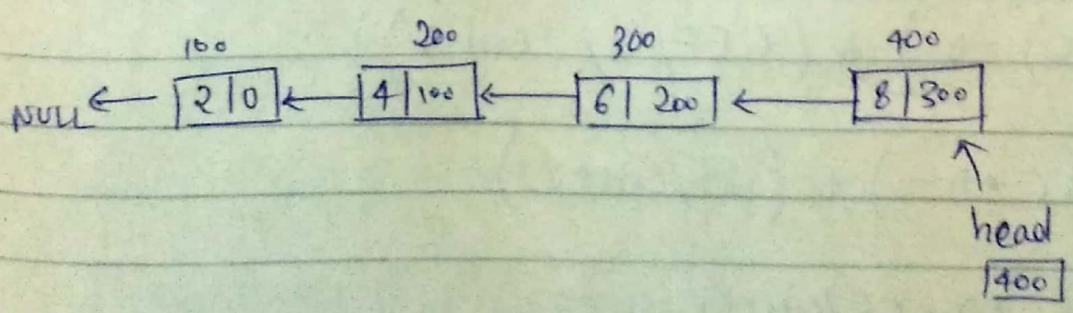
Scanned by CamScanner

LOGIC: first, we will create a linked list. Then, we will traverse through the list and will store address of each node in stack. After doing so, our temp will be at NULL but we need it at location of last node. So, we will pop the topmost element from stack and store it in temp. Now, temp points to last node in given list. But, we have to make it first node in reversed list ∴ we will shift head to temp. Then, we will pop one element from stack & will store it in next part of temp. Then, we will shift temp to temp→next and will do same process & untill our whole stack becomes empty. At last, we will make temp→next part point to NULL.



After, all operations of pop() has been executed, reversed list will be:





Scanned by CamScanner

<< INFIX, POSTFIX & PREFIX >>

- **Infix :** The expression of form a op b. when an operator is in - between every pair of operands. eg 1+2.

- **Prefix :** The expression of form op ab when operator comes before the operands. eg +12.

- **Postfix :** The expression of form a b op. when an operator is followed by every pair of operands. eg. 1 2 +.

- Conversion of Infix expression into postfix expression.

eg : $\underbrace{(A+B)}_{AB+} * \underbrace{(C-D)}_{CD-} \Rightarrow (AB+) * (CD-)$

$= AB+CD-*$

eg : $A \$ B * C - D + E / F / (G+H)$

Step 1 : Introduce paranthesis in expression to resolve precedence

$$((((A \$ B) * C) - D) + ((E/F) / (G+H)))$$

Firstly, innermost paranthesis are solved i.e. innermost operations occur first.

$$= ((((AB\$) * C) - D) + ((EF/) / (GH+)))$$

$$= (AB\$C*D-) + (EF/GH+/)$$

$$= AB\$C*D-EF/GH+/+.$$

Scanned by CamScanner

eg 2: $A - B / (C * D \$ E)$

first, introduce paranthesis. $(A - (B / ((C * (D \$ E)))))$

$= ABCDE\$*/-$

NOTE: Paranthesis are introduced according to precedence & associativity. operator with highest precedence is given paranthesis first. If precedence of operators is same, then associativity comes into solve this issue.

In above eg, \$ has highest precedence ∴ paranthesis are introduced first for \$ operator. Then, there are operators with same precedence ie. / & * ∵ associativity of / & * is from left to right and / comes on left ∴ / is paranthesized prior to *.

Operators	precedence	Associativity	
\$	top	right to left	[exponentiation operator]
*, /	middle	left to Right	{ eg. $2\$3 = 2^3 = 8$ }
+, −	lowest.	left to right.	

• Conversion of Infix to Prefix

eg: $(A+B) * (C-D)$

$(+AB) * (-CD) \Rightarrow *+AB-CD$

eg: $((((A \$ B) * C) - D) + ((E / F) / (G+H)))$

$((((\$AB) * C) - D) + ((/EF) / (+GH)))$

$= ((*\$ABC - D) + (/EF+GH))$

$= ((-*\$ABCD) + (/EF+GH))$

$= +-*\$ABCD/EF+GH$

Scanned by CamScanner

eg> A - B / (C * D $ E)

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

$DE

= - A / B * C $ DE

* Evaluation of Postfix Expression.

Given, a postfix expression :

6 2 3 + - 3 8 2 / + * 2 $ 3 +

Evaluate it.

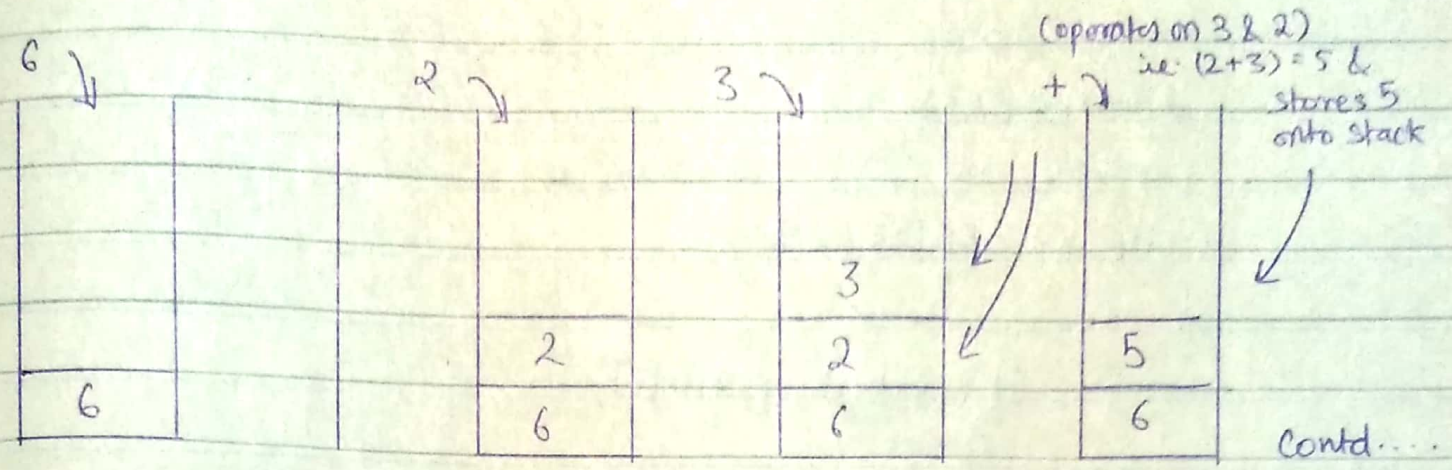
Symb.	operand 1	operand 2	value	status of stack
6				6.
2				6 2
3				6 2 3
+	2	3	5	6 5
-	6	5	1	1
3				1 3
8				1 3 8
2				1 3 8 2
/	8	2	4	1 3 4
+	3	4	7	1 7
*	1	7	7	7
2	2			7 2
$	7	2	49	49
3				49 3
+	49	3	52	52.

Logic: We push operands onto the stack and whenever an operator comes, it operates on the top two values on the stack, the top value acts as operand 2 & other one acts as operator/ operand 1. The result of (operand 1) operator (operand 2) is pushed onto the stack. This process continues and at last, only

Scanned by CamScanner

one element remains in the stack which is the required result.

- few steps for evaluation of postfix expression.



Contd....

PROGRAM 20 : Code for Evaluation of Valid postfix expression.

Sol :

```
// push(), pop(), initialize() of Program 17 has been used.
// assuming maxS = 20.

#include <stdio.h>
#include <ctype.h>      // for isdigit().

int optr (char ch)      // to find whether scanned character is operator.
{       switch (ch)
        {       case '+' :
                case '-' :
                case '*' :
                case '/' : return 1;
                default : return 0;
        }
}
```

Scanned by CamScanner

equivalent to char* postfix.

```
void PostFixEvaluation (char postfix[])
{
        struct stack s;
        int op1, op2;
        initialize(&s);
        int i=0;
        while (postfix[i] != '\0')
        {
                if (isdigit (postfix[i]))
                {
                        push(&s, postfix[i]-'0');
                }
                if (optr(postfix[i]))
                {
                        op2 = pop(&s);
                        op1 = pop(&s);
                        switch (postfix[i])
                        {
                                case '+': push(&s, op1+op2); break;
                                case '-': push(&s, op1-op2); break;
                                case '*': push(&s, op1*op2); break;
                                case '/': push(&s, op1/op2); break;
                        }
                }
                i++;
        }
        printf("\n RESUT: %d ", s.a[s.top]);
}
```

we can also write s.a[0] in place of it.

```
int main()
{
        char c[80];
        gets(c);
        PostFixEvaluation(c);
}
```

Scanned by CamScanner

NOTE: {postfix[i]-'0'} → what does this statement mean?

∵ postfix[i] is a character and we have to push integer in stack ∴ we write postfix[i]-'0' which means that subtract ascii value of postfix[i] & ascii value of zero & result will be an integer, which can now be passed onto stack.

eg. Postfix[i] = 2.

Postfix[i] = 50 − 48 = 2
(ASCII of 2) (Ascii of 0) (integer).
(∵ subtraction operator gives integer as its result)

- Conversion of Infix to Postfix. (logic).

Let infix operation expression be given.

i) Scan the given infix expression (in form of string) from left to right.

ii) If scanned character is an operand, directly assign it to postfix string (the result string).

iii) If scanned character is an operator

a) having higher precedence than the operator present on top of stack, then simply, push it onto the stack (if stack is empty, do the same).

b) having lower precedence than operator present on top of stack, then pop the operator from stack & assign it to postfix[] string untill an operator with lower precedence than scanned operator is found on the stack or till opening parantheses is encountered on top of stack. At last, push scanned operator on stack.

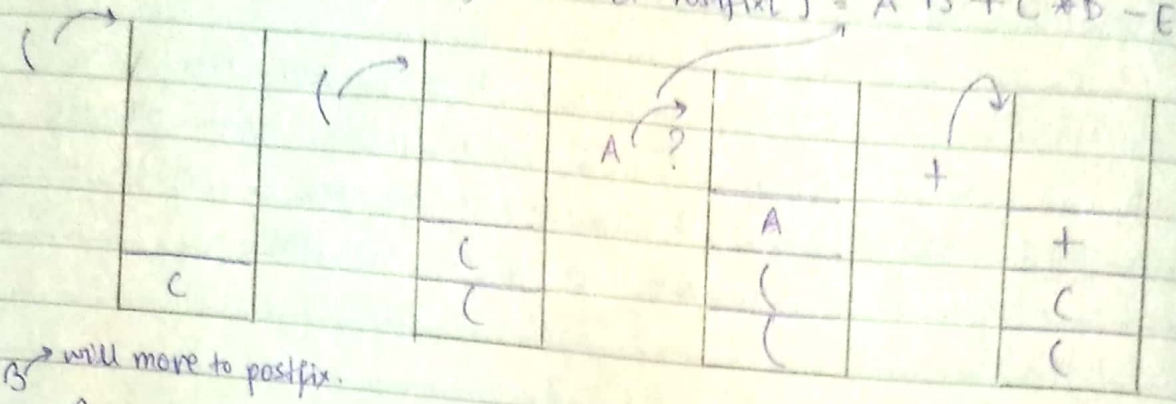
iv) If scanned character is '(', push it onto stack.

v) If scanned character is ')', then, pop out all elements from stack and put in string postfix[], till an opening parantheses '(' is encountered. Then, pop-out this opening parantheses as well.

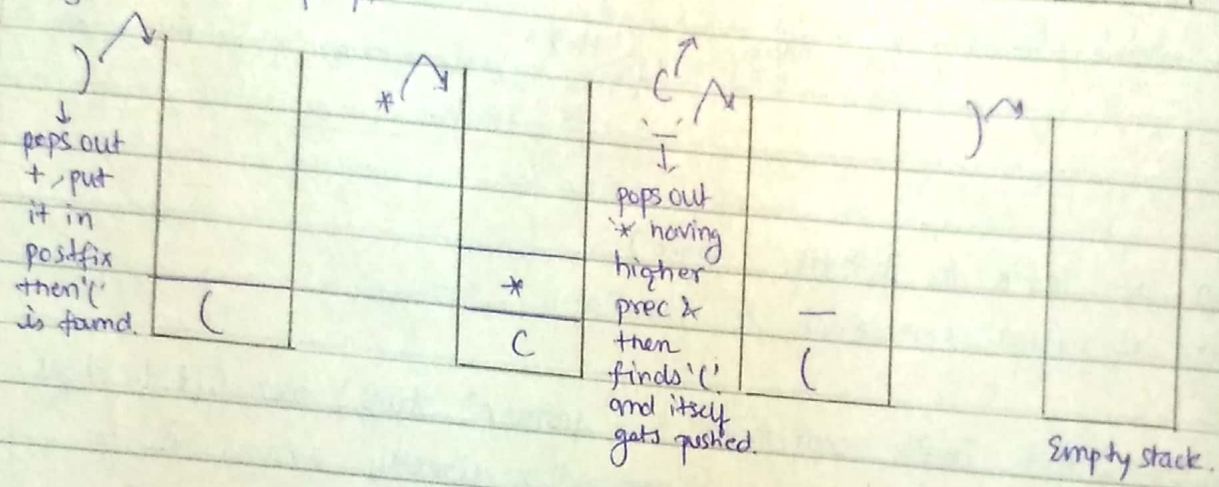
vi) Repeat above steps till whole string (infix) given is scanned.

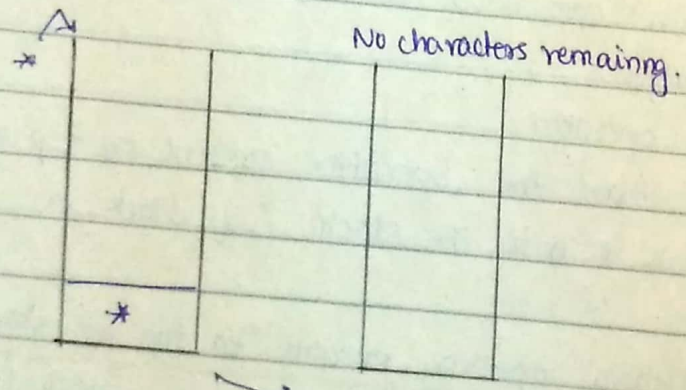
Scanned by CamScanner

eg: ((A+B)*C-D)*E & Postfix[] = A B + C * D - E *



B → will move to postfix.





PROGRAM 21: Code to convert an infix to postfix expression.

Sol: // we have used pop(), push(), initialized() of program 18.

```
#define <Stdio.h>

int operand (char ch)
{
        if (ch>='A'&& ch<='Z')
                return 1;
}               return 0;
```

Scanned by CamScanner

```
int optr (char ch)
{
        switch (ch)
        {
            case '+':
            case '-':
            case '*':
            case '/':   return 1 ;
            case '$':   return 1;     (x)
            default :  return 0;
        }
}


int weight (char op)        // for precedence
{
        switch (op)
        {   case '+':
            case '-':  return 1;
            case '*':
            case '/':  return 2;
            case '$':  return 3;   (x)
        }
}


void InfixToPostfix ( char infix[], char postfix[] )
{           struct stack S;
            initialized (&S);
            int i=0 , j=0;
            while ( infix[i] != '\0')
            {
                if( operand(infix[i]))
                {     postfix[j++] = infix[i];
                }
```

Scanned by CamScanner

```
else if ( optr(infix[i]) )
 {
     while (s.top != -1 && s.a[s.top] != '(' && weight(infix[i]) <=
                                                weight (s.a[s.top]))
      {
          postfix[j++] = pop(&s);
      }
      push (&s, infix[i]);
 }

else if ( infix[i] == ')')
 {
     while ( s.a[s.top] != '(')
      {
          postfix[j++] = pop(&s) ;
      }
      pop(&s); // to delete '(' from top of stack.
 }
else if ( infix[i] == '(')
 {
      push (&s, infix[i]);
 }
 i++;
}
while (s.top != -1)
 {
      postfix[j++] = pop(&s);
 }
postfix[j] = '\0'
}.
```

Scanned by CamScanner

Program 22: Implement two stacks in single array.

There can be two approaches :

a) We can give both stacks same size of Array. ie if array has size n, then elements can be filled in stack 1 from 0 to n/2 and these of Stack to can be filled from index n/2+1 to (n-1).

- But this method is inefficient as space is wasted.
- A stack push opⁿ may result in stack overflow even if there is space in array.

  eg: array size is 6. We push 3 elements to Stack 1 without inserting anything to stack 2. When we push 4th element, there will be overflow even we have space for 3 more elements in array.

b) Efficient approach: Start two stacks from two extreme corners of arr[]. Both stacks will grow & shrink in opposite direction. To check overflow, we just need to check for space b/w top element of Both stacks.

Code :

```
#define maxs 20
# include <stdio.h>

struct two_stacks
  {
        int arr[maxs];
        int top1, top2;
  };

void initialize ( struct two-stacks * s)
  {         s → top1 = -1;
            s → top2 = -21;
  }.
```

Scanned by CamScanner

```
void push1 ( struct two.stacks *s, int x) // to push element in stack 1
{
        if (s->top1 < s->top2 - 1)
        {
                s-> top1++;
            s->arr [s->top1] = x;
        }
        else
        {   cout<<"Stack 1 overflow";
            exit(1);
        }.
}


void push2 ( struct two-stacks *s, int x) // to push element in Stack 2
{
        if ( s-> top1 < s->top2 - 1
        {       s-> top2--;
                s-> arr [s->top2] = x;
        }
        else
        {       cout << "Stack 2 overflow";
        }
}.

int pop1 ( struct two-stacks *s) // to pop element from Stack 1.
{
        if ( s-> top1 >= 0)
        {   int x = s-> arr [s->top1];
            (s-> top1)--;
            return x;
        }
        else
        {       cout<<"Stack Overflow"; exit(1);
        }
}
```

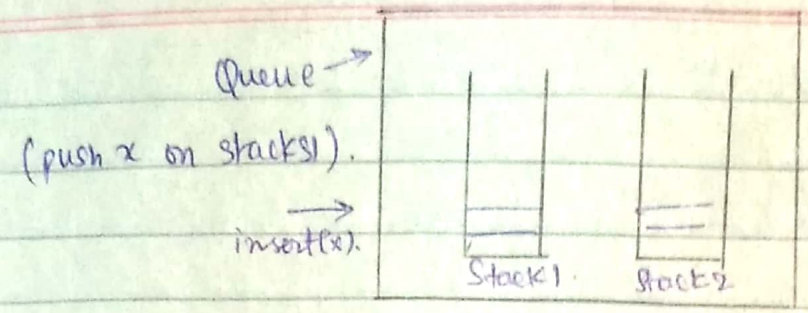
Scanned by CamScanner

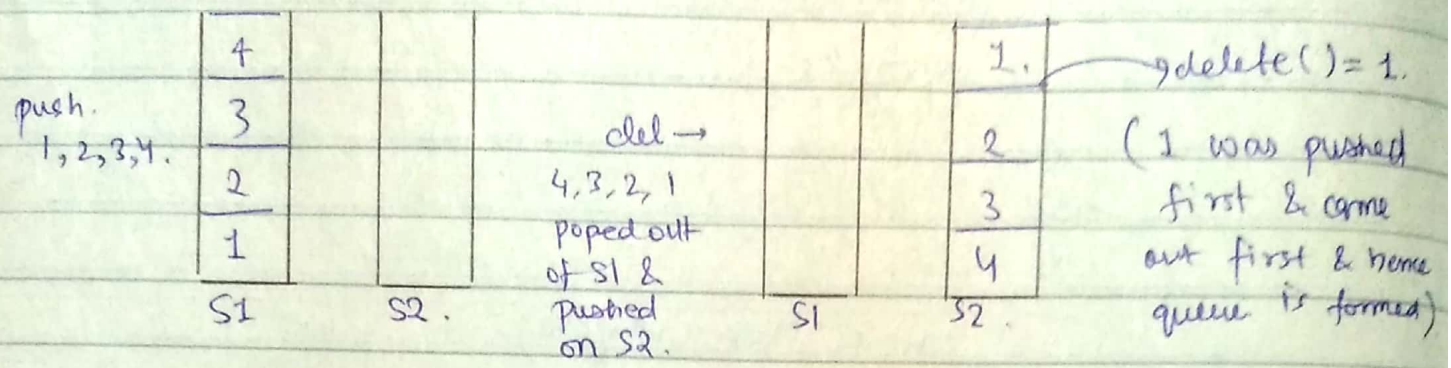
```
int pop2 ( struct two_stacks *s )
{
        if ( s-> top2 < size)
        {
            int x = s->arr [s->top2] ;
            (s-> top2) ++ ;
            return x;
        }
        else
        {
            cout << "Stack Underflow";
            exit(1);
        }
}

int main()
{
struct two_stacks s ;
    push1 ( &s, 2 ) ;
    push 2 (&s, 4) ;
   printf(" Poped element from Stack 1 : %d ", pop1(&s));
   printf (" Poped element from Stack 2 : %d", pop2 (&s));
}.
```

PROGRAM 23: Implement operations of Queue using 2 stacks.

We will create a structure queue having 2 stacks. Insertion will always be performed on Stack 1 and deletion will always be performed from Stack 2. If elements is not present in Stack 2, then, we will pop-out all elements of Stack 1 & will push them in Stack 2. Then, perform, pop opn on Stack 2.

Scanned by CamScanner






```
# include <stdio.h>
# define maxs 20

Struct stack
{   int a[maxs];
    int top;
};


Struct queue
{   Struct stack S1;
    Struct stack S2;
};


void initialize (struct queue *q)
  {
        q→S1.top = -1;
        q→S2.top = -1;
  }
```

Scanned by CamScanner

```
void insert ( struct queue *q , int x)
{
        if (q->s1.top == maxs)
        {
                printf (" \n Queue is full ");
        }
        else
        {       q->s1.top++;
                q->s1.a[q->s1.top] = x;
        }
}


int delete ( struct queue *q)
{
        if ( q->s2.top == -1)
        {
                if (q->s1.top == -1)
                {
                        printf ("Queue is Empty ");
                        return;
                }
                while ( q->s1.top != -1)
                {
                        int x = q->s1.a [q->s1.top];
                        q->s1.top -- ;
                        q->s2.top++;
                        q->s2.a[q->s2.top] = x;
                }
        }
        int y = q->s2.a[q->s2.top];
        q->s2.top -- ;
        return y;
}
```

Scanned by CamScanner

```
void display (struct queue *q)
{
        int t=0;
        int l; int p=q->S2.top;
        while (p!=-1)
        {
                int y = q->S2.a[p];
                printf ("%d ", y);
                t=1;
                p--;
        }
        for (l=0; l<=q->S1.top; l++)
        {
                int x=q->S1.a[l];
                printf ("%d", x);
                t=1;
        }
        if(t==0)
        {
                printf ("\n Queue is Empty");
        }
        printf("\n");
}


int main()
{       Struct queue q;
        initialize (&q);
        display(&q);    // Queue is Empty.
        insert(&q,5);   // Display(&q);    //5
        insert(&q,1),   display(&q);    //5 1
        insert(&q,2);   display(&q);    // 5 1 2.
        delete(&q);     display(&q);    // 5 1.
}
```

Scanned by CamScanner

PROGRAM 24 : Check the string for Balanced paranthesis. ((), {}, [])

```
int Are pair ( char opening , char closing )
{
        if (opening == '(' && closing == ')' )  return ~~tru~~ 1 ;
    else if ( opening == '{' && closing = '}' )  return 1 ;
    else if (opening == '[' && closing == ']')  return 1 ;
    return 0 ;
}


int check Balanced Paranthesis ( s char exp[] )
{
    Struct stack s ; int i = 0; initialise(&s);
    For ( i=0; exp[i] != '\0' ; i++)
    {
        if ( exp[i] == '(' || exp[i] == '{' || exp[i] == '[' )
                push(&s, exp[i]);
        else if ( exp[i] == ')' || exp[i] == '}' || exp[i] == ']' )
        {
            if ( s.top == -1 || ! Arepair ( s.~~top~~a[s.top] , exp[i] ))
                return 0;
            else
                pop(&s);
        }
    }
    return s.top == -1 ? 1 : 0 ;
}.
```

Scanned by CamScanner

<< HEAP DATA STRUCTURE >>

A heap is a tree-based data structure in which nodes of tree are in a specific order.

* Binary heap: A binary heap is a binary tree with following properties:
  i> It's a complete Binary tree. (All levels are completely filled except possibly last level & last level has all keys (or nodes) as left as possible.
  ii> Nodes must have a specific ordering. i.e. it should be either a max-heap or a min-heap.

- Max-heap: If X is a parent node of Y, then value of X must be greater than or equal to value of Y.

- Min-heap: If X is a parent node of Y, then value of X must be smaller than or equal to value of Y.

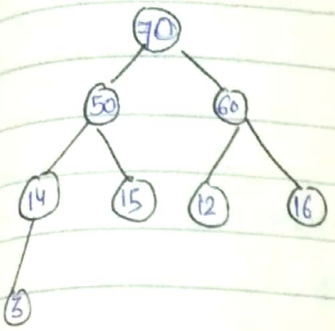
- A binary heap is typically represented as Array.
  For key at i^{th} index in array, its parent key will be at index $\left(\frac{i-1}{2}\right)$ and left child and right child will be at index $(2*i+1)$ & $(2*i+2)$ respectively.
  {Indexing starts from zero}.

- Applications of Heaps:
  i) heap sort: Complexity with $O(n\log n)$ time.
  ii) Priority Queue.
  iii) Graph algorithms like Dijkstra's and Prim's algorithm.
  iv) K^{th} largest element in Array
  v) Merge K sorted Arrays

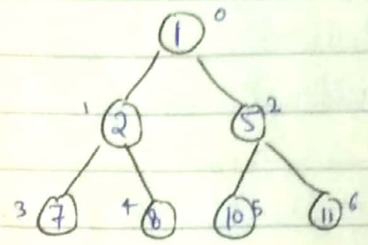
Scanned by CamScanner

Examples of Binary Heap.

i) Max-Binary Heap.



ii) Min-Binary Heap.



*

0	1	2	3	4	5	6	7
70	50	60	14	15	12	16	5

0	1	2	3	4	5	6
1	2	5	7	8	10	11

{ Array representation }

* Operations performed on heaps (for max-heap)

i) getMax() : It returns root element of max-heap.
ii) insert() : It inserts a key into the heap while retaining its heap property.
iii) deleteMax() : Delete root node while retaining heap property.
iv) deleteAtIndex() : Delete a node (key) at particular index.
v) BuildHeap() : To build a heap from any random array.
vi) heapsort() : Sorts any given array.

* Insert() for max-heap.

→ Algorithm :

i) It checks if size of Array is equal to capacity of Array, if so, then it cannot insert the element.
ii) otherwise, it will add the element to the last of Array and then compare the inserted element with its parent's value. If parent's value < inserted element, then swapping occurs & this process goes on till a parent node is encountered whose value > inserted element.

Scanned by CamScanner

* Some HELPER FUNCTIONS

```
#include <stdio.h>
#include <limits.h>
# define capacity 20.

void swap (int *a ,int x, int y)  // it swaps two elements of Array present
{                                 // at indices x & y.
        int t=a[x] ;
         a[x]=a[y];
         a[y]=t ;
}

int parent(int i){          // returns index of parent node of node with index 'i'.
        return (i-1)/2 ;
}
```

* Insert() for max-heap → code.

```
int Insert (int a[],int x, int size) {
    if(size== capacity)
     {
         printf("\n Heap is Full \n");
         return size;
     }
    a[size]=x ;
    int i=size;
    size ++ ;
    while (i!=0 && a[i] >a[parent(i)] )
     {
         swap(a,i,parent(i));
         i= parent(i);
     }.
    return size;
}.
```

Scanned by CamScanner

heapify() for max heap.

It will regain heap property in a heap (or subheap) where root node is present at index 'i'.

Let a node (root) of a heap be present at index 'i'. Then, its left child will be at index 2*i+1 & right child at index 2*i+2.
It will compare key at 'i' index with its left child & right child key. If any of the child key is greater than node key, then, it will swap the root key with the larger key. This process goes on till both child keys has smaller values than root node key.

- heapify() can be implemented either iteratively or recursively.

* Iterative heapify for max-heap.

```
void heapify (int *a , int size , int i)
{
    while (1)
    {   int l = 2*i+1;
        int r = 2*i+2;
        int largest = i;
        if ( l<size && a[l] > a[largest])
                largest = l;
        if ( r<size && a[r] > a[largest])
                largest = r;
        if (largest != i)
        {
            swap( a, i, largest);
            i = largest;
        }
        else break;
    }
}
```

Scanned by CamScanner

* heapify - recursively

```
void heapifyrec (int *a, int size, int i)
{
        int l = 2*i+1;
        int r = 2*i+2;
        int largest = i;
        if (l < size, && a[l] > a[largest])
                largest = l;
        if (r < size && a[r] > a[largest])
                largest = r;
        if (largest != i)
        {
                swap(a, i, largest);
                heapifyrec (a, size, largest);
        }
}
```

* deleteMax - for max-heap.

This function will delete max key ie. root node while maintaining heap property.

Algorithm:

i) Swap root node with last element in Array.
ii) Decrease size by one.
iii) Now, heapify the whole heap.

CODE:

```
int deleteMax (int *a, int n)
{       swap(a, 0, n-1);
        n--;
        heapify(a, n, 0);
        return n;
}.
```

Scanned by CamScanner

* deleteAtIndex () – for max heap.

This will delete the key at given index & will retain heap property.

Algorithm:

i) Replace value at that index with INT_MAX.

ii) Compare that key with its parent key and swap the two untill given key is smaller than or equal to the parent key.

iii) So, modified root key will be INT_MAX.

iv) Now, call deleteMax().

CODE:

```
int deleteAtIndex (int *a, int size, int i)
{
    a[i]= INT_MAX;
    while (i!=0 && a[i] > a[parent(i)])
    {       swap (a, i, parent(i));
            i = parent(i);
    }
    return deleteMax (a, size);
}.
```

PROGRAM 25: To build a max-heap from a random given Array.

Logic: All leaves nodes forms one element heap in itself. So, all other internal nodes (acting as a root node of given subheap) will be heapified.

CODE: void

```
build heap (int *a, int n)
{   int i;
    for (i=(n/2)-1 ; i>=0 ; i--)        // n-1 is index of last
    {       heapify(a, n, i);           // internal node
    }
}.
```

Scanned by CamScanner

* heapsort (Ascending order using max-heap)

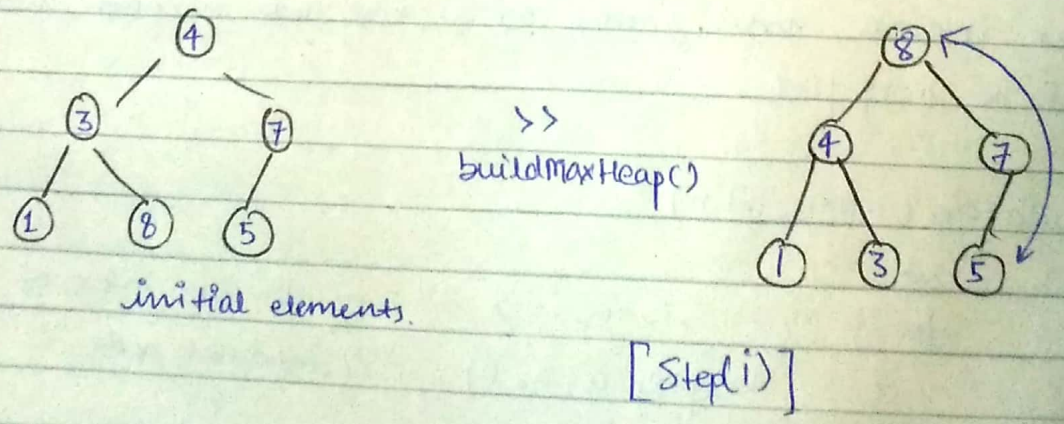
Algorithm:

i) build a heap using given unsorted Array by invoking buildheap().
ii) Swap the topmost or root element with last element in Array
iii) Decrease size of Array by one.
iv) Heapify whole heap.
v) Repeat steps ii) to iv) (n-1) times, since one element gets to its correct position in one iteration.
vi) sorted array is obtained.
{ Steps ii) to iv) constitute deletemax() }.

CODE:

```
void heapsort (int *a, int n)
 {
      buildHeap(a,n);
      int i, x=n;
      for(i=1 ; i<= x-1 ; i++)
      {
           n=deletemax(a,n);
      }
 }
```

* Dry-Run: given array elements: 4 3 7 1 8 5

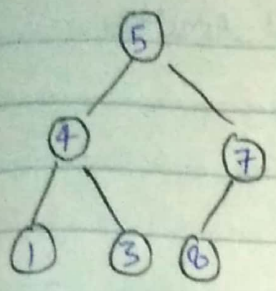


initial elements.

[Step(i)]

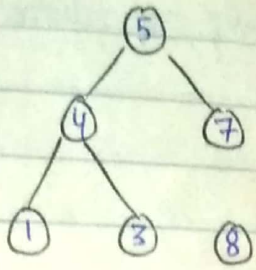
Scanned by CamScanner

Step 2.



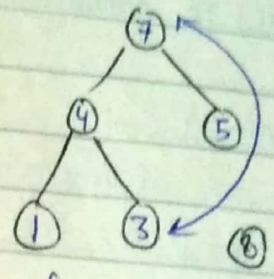
{ After swapping }.

Step 3.



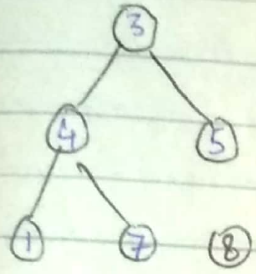
{ decreasing size by one }.

Step 4



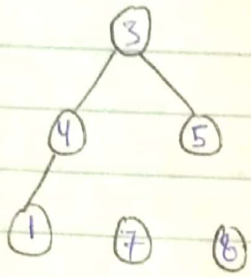
{ heapifying }.

Step 2 (repeated)



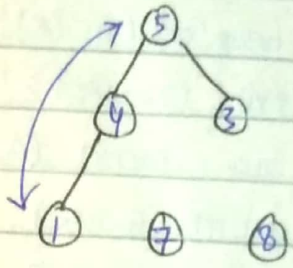
{ After swapping }.

Step 3 (repeated)



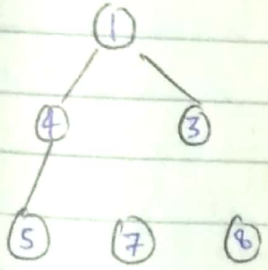
{ decreasing size by one }.

Step 4 (repeated)



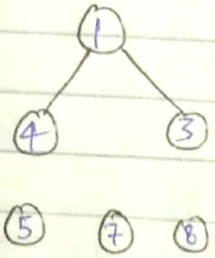
{ heapifying }.

Step 2 (repeated)



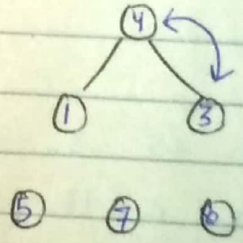
{ After swapping }.

Step 3 (repeated)



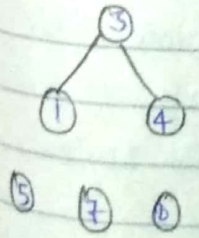
{ decreasing size by one }.

Step 4 (repeated)

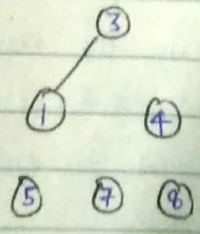


{ heapifying }.

Step 2 (repeated)



Step 3 (repeated)



Step 4 (repeated)

1 3 4 5 7 8

{ directly repeating steps 4, step 2, step 3 }.

Scanned by CamScanner

* Main() to demonstrate the working of above discussed function.

```
int main()
{     int a[capacity], size=0, i;
      for (i=0; i<0; i++)
       {
            scanf ("%d", &a[i]);
            size++;
       }
      heapSort (a, size);
      print (a, size);
      size = insert (a, 65, size);
      print (a, size);
      size = deleteMax (a, size);
      print (a, size);
      size = deleteatIndex (a, size, 5);
      print (a, size);
}
```

OUTPUT :

70 50 45 13 53 6 3 5

3 5 6 13 45 50 53 70

65 3 6 5 45 50 53 70 13

13 3 6 5 45 50 53 70

70 13 6 5 3 50 53.

* PRIORITY QUEUE :

It is also a queue but different in the sense that logical order of elements in priority queue depends on priority of elements. Element with highest priority will moved to front of queue & lowest priority element to back of Queue.

Eg: Array elements: {4, 8, 1, 7, 3}.

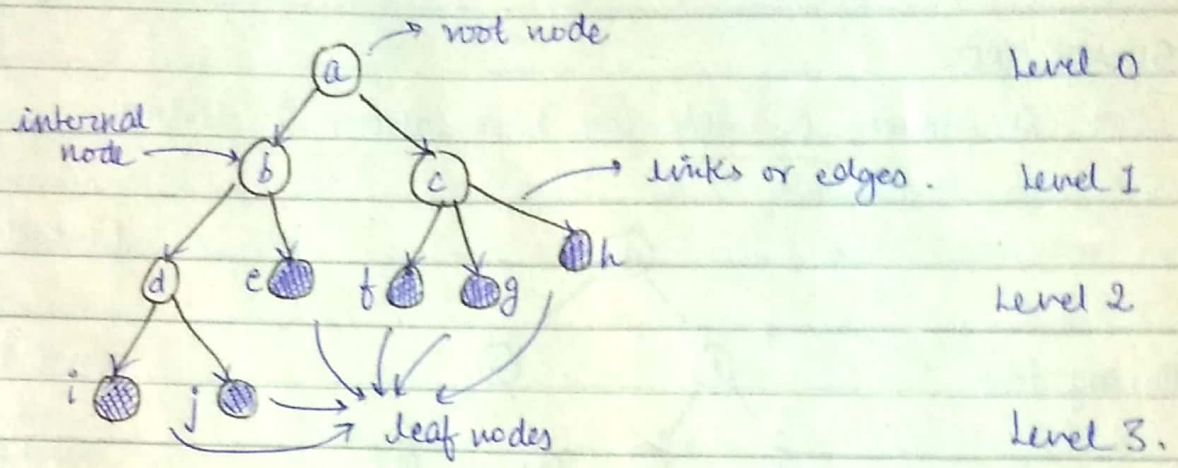
8 7 4 1 3 1 → priority queue.

Scanned by CamScanner

DATA STRUCTURE : TREES.

Unlike arrays, linked lists, stacks and queues, Trees are non-linear data structures. They are used to store data which involves hierarchical relationship.

- Tree vocabulary



PARENT: Node which is directly above some nodes is called parent.
LEAVES NODE: Nodes with no children. {i, j, e, f, g, h}
INTERNAL NODES: Nodes which are not leaf nodes.
ANCESTOR: A is ancestor of b.
DESENDENT: b is descendant of a.
SIBLINGS: Nodes with same parents.

If there are N nodes in a tree, then there will be N-1 linkes or edges.

Depth of a node x: No. of edges in path from root to x.
eg. depth of node e is 2.

Height of a node x: No. of edges in longest path from x to leaf node.
for eg. Height of node b = 2. (path b to i).

Height of leaf nodes = 0
Height of a tree = height of root node. (= 3 in above tree)

Scanned by CamScanner

Application of Trees:

i) To store and manipulate hierarical data.

ii) Trees provide moderate search operations. (BST)

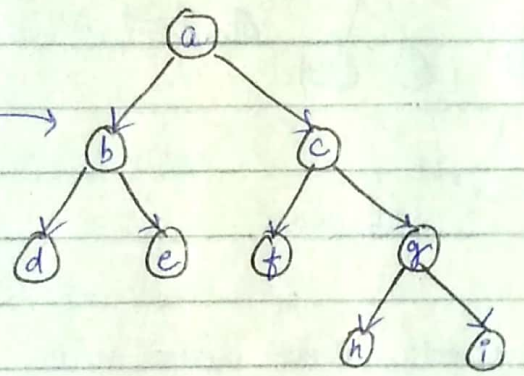
iii) To provide moderate insertion/deletion.

- Degree of a Tree: The maximum no. of children any node have in a tree.
  Degree of above tree is 3.

* BINARY TREE

A tree in which a node can have atmost 2 children.

Binary tree



level 0

level 1

level 2

level 3.

Let n be total no. of nodes in a tree with height h.

Max. no. of nodes at level i : 2^i

Max. no. of nodes in a binary tree with height h,

$$= 2^0 + 2^1 + \cdots + 2^h.$$

$$= 2^{h+1} - 1$$

$$= 2^{(\text{no. of levels})} - 1.$$

$$\therefore \quad n = 2^{h+1} - 1$$

$$2^{h+1} = n+1$$

$$h = \log_2(n+1) - 1$$

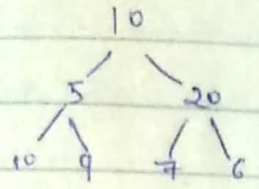
- No. of leaf nodes is always one more than nodes with two children.

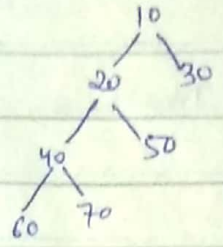
Scanned by CamScanner

TYPES OF BINARY TREES.

i) Full Binary Tree: A binary tree in which every node has either zero or two children.

eg





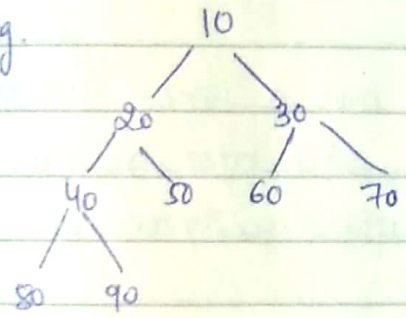
$L = I + 1$

I → no. of internal nodes

L → no. of leaf nodes.

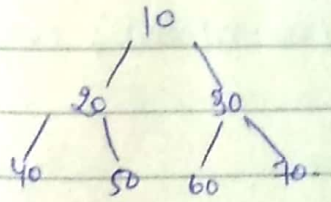
ii) Complete Binary Tree: A binary tree in which every level is completely occupied except possibly the last level in which all keys (nodes) are as left as possible.

eg.



iii) Perfect Binary Tree: A binary tree in which every internal node has two children & last level has all leaf nodes i.e. all leaves are at same level.

eg.

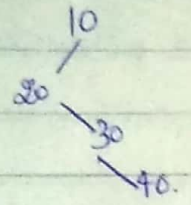


iv) Balanced Binary Tree: A binary tree in which for each nodes, the diff. in heights of its left subtree & right subtree is not greater than k (k=1 mostly).

Scanned by CamScanner

• degenerate Binary Tree : Every internal node has one child.

eg.



• Structure for node of a Binary Tree.

```
struct node
  {   int data;
      struct node * left;
      struct node * right;
  };
```

* TREE TRAVERSAL.

In Arrays, linked lists etc, there is a logical start and a logical end & hence, there is only one way to traverse it. But, since trees are non-linear data structure, therefore, they can be traversed in more than one ways. These are :

i) Level order Traversal } Breadth first Traversal.

ii) Pre-Order Traversal
iii) In order Traversal
iv) Post-Order Traversal
} Depth first Traversal

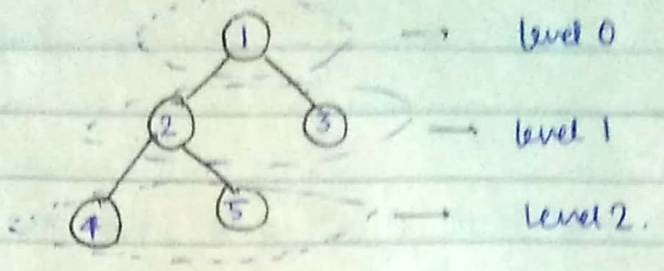
'Tree Traversal' basically means to visit (process / access / read) every node once in a tree.

i) Level Order Traversal

It means to read the nodes level by level fashion i.e. we will read the data of nodes from left to right in a particular level, only after that, we will move to next level.

Scanned by CamScanner

eg :



level order traversal : 1 2 3 4 5

Algorithm for printing nodes of a tree in level order Traversal.

i) Create an empty Queue Q.
ii) set current-node = root
iii) Loop while current-node is not null
   a) print root->data // print current-node -> data ;
   b) Enqueue current-node's children to Q (first left then right).
   c) Dequeue a node from Q & assign it to current node.

CODE :

```
void PrintLevelOrder ( struct queue *q ) :
                       struct node * root
 {
   struct queue q ;
   initialize (&q) ;
   struct node * temp_node = root ;
   while ( temp_node != NULL)
    {
        printf ("%d ", temp_node ->data) ;
        if ( temp_node ->left != NULL )
                insert ( &q, temp_node ->left) ;  //enqueue operation
        if ( temp_node -> right != NULL )
                insert ( &q, temp_node -> right) ; // enqueue opn.

    temp node = deleteq (&q) ; // dequeue opn.
    }
 }
```

Scanned by CamScanner

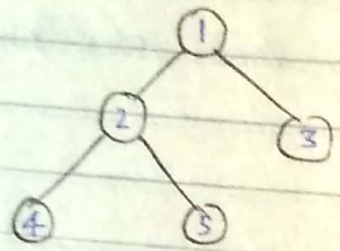
B) Depth-First traversal.

i) Pre-Order Traversal.

<root> < left > < right >

In this, we will first read data of root node, then we will move to left subtree and after that, we will move to right sub-tree.

eg.



Pre-order Traversal : 1 2 4 5 3

CODE: (Recursive implementation)

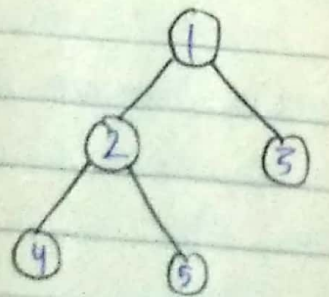
```
void PrintPreOrder (struct node * root)
{
        if ( root == NULL)
                return;
        printf ("%d", root -> data);
        PrintPreOrder ( root -> left);
        PrintPreOrder ( root -> right);
}
```

ii) InOrder Traversal

< left > < root > < right >

In this, we will first travel left subtree, then read data of root node & after that, we will travel right subtree.

eg.



Inorder Traversal : 4 2 5 1 3

Scanned by CamScanner

Code : (recursive implementation)

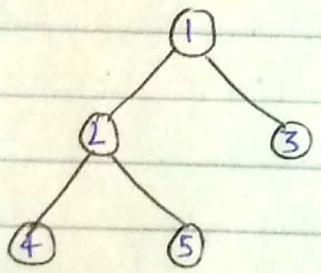
```
void Inorder ( struct node* root)
{
        if(root == NULL) return;

        Inorder ( root -> left) ;
        printf ("%d ", root -> data) ;
        Inorder( root -> right) ;
}
```

iii) PostOrder Traversal

<left> <right> < root >

In this, we will first travel left subtree, then right sub-tree & at last, we will read data of root node.

eg:



Postorder Traversal : 4 5 2 3 1

Code : (recursive implementation)

```
void PrintPostorder (struct node* root)
{
        if (root == NULL) return ;
        PrintPostorder ( root -> left) ;
        PrintPostorder (root -> right) ;
        printf ("%d ", root -> data) ;
}
```

Scanned by CamScanner

* INORDER TREE TRAVERSEL WITHOUT RECURSION
(iterative approach)

→ Algorithm

i) Create an empty stack.

ii) Initialize current node = root.

iii) If current != NULL, push current to Stack & do current = current → left untill current = NULL.

iv) If current == NULL & stack is empty
   a) Pop from stack & assign it to current.
   b) Print current → data.
   c) current = current → right
   d) goto step iii)

v) If current is null & stack is Empty, we are done.

→ CODE : (iterative implementation)

```
void PrintInorder ( struct node * root)
 { struct Stack S ;
   initialize(&S) ;
   struct node* current = root ;
   while (1)
   {
       if (current != NULL)
       {       push(&S, current) ;
               current = current → left ;
       }
       else
       {
           if ( ! isEmpty (&S) )
           {
               current = pop(&S) ;
               printf ("%d ", current → data) ;
```

Scanned by CamScanner

```
            current = current -> right;
        }
        else
            break;
    }
}
}.
```

* RECONSTRUCT A BINARY TREE FROM GIVEN VALID PRE-ORDER and IN-ORDER. -TRAVERSAL.

Suppose that Preorder sequence : A B D E C F
and the Inorder sequence : D B E A F C. } valid (left)

Now, Preorder sequence tells us that A is the data of root node. So, search 'A' in inorder traversal. So, DBE are elements of left sub-tree and FC are elements of right subtree. So, new left & right sub-trees can be built up using same function.

ALGORITHM :

i) Pick element of Pre-order. Increment Pre Index so that next element is taken in next recursive call.
ii) Create new tree node and assign picked element to its data.
iii) Find picked element's index in Inorder. (Say InOrder).
iv) Call buildTree for creating left subtree.
v) Call buildTree for creating right-subtree.
vi) Return address of tree node made above.

Scanned by CamScanner

CODE :

```
struct node * buildTree ( char in[], char pre[], int inStrt , int inEnd )
{       static int preIndex = 0 ; // initialized only once.
        if ( inStrt > inEnd )
                return NULL ;
        struct node * tNode = newNode ( pre[preIndex ++] ) ;

        if ( inStrt == inEnd )
                return tNode ;

        int inIndex = search ( in , strt , InEnd , tNode -> data ) ;

        tNode -> left = buildTree ( in , inStrt , InIndex - 1 ) ;
        tNode -> right = buildTree ( in , pre , InIndex + 1 , inEnd ) ;
        return tNode ;
}

int Search ( char in[], int strt , int End , char value )
{       int i ;
        for ( i = strt ; i <= end ; i ++ )
          { if ( in[i] == val )
                return i ;
          }
}
```

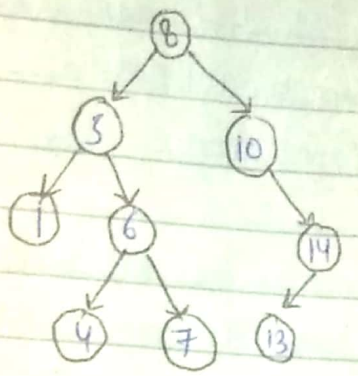
Scanned by CamScanner

* BINARY SEARCH TREES (BST)

Binary Search trees are binary trees which has following properties:

a) left subtree of a node contains only nodes with Keys less than node's key
b) right subtree of a node contains only nodes with Keys > node's key.
c) The left and right subtree are also BST.
d) There are no duplicate nodes.

eg.



due to such a special ordering of nodes, operations on trees becomes less costly (meaning time complexity ↓) Searching becomes too easy.

PROGRAM 25: To search an element in BST.

Logic: Compare element to be searched with root→data.

b) if root→data < key, then search it in right subtree.
c) else if root→data > key, then search it in ~~ri~~ left subtree.
d) if root→data = Key, then return root.

CODE:

```
struct node * searchinBST ( struct node * root , int Key)
 {
        if ( root→data == key || root == NULL)
              return root ;
        if ( root→data < Key)
           return searchinBST ( root→right , key) ;

        if ( root→data > key )
           return searchinBST ( root→left , key) ;
 }
```

Scanned by CamScanner

Time-complexity : It depends on height of BST. = O(h)

If tree is like degenerate tree, O(h) = O(n)

for a balanced BST, O(h) = O(logn)

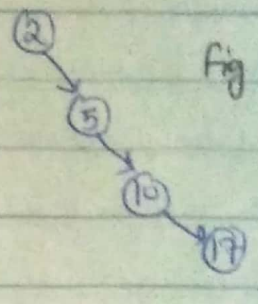


Fig 2.

PROGRAM 26 : Insertion in BST

logic: Insertion in BST is always done at leaf node.

Compare root→data with element to be inserted.

a) if root = NULL, then make created node as root.

b) Set current_node = root.

c) If current_node → data > x, then move to left subtree.

d) If current_node → data < x, then move to right subtree.

e) Proceed in same manner & at last, insert node.

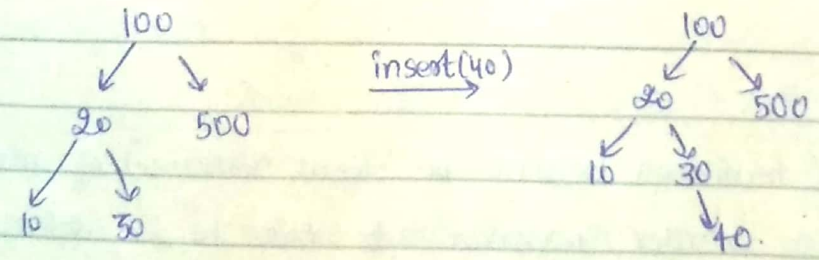
CODE :

```
struct node* Insert (struct node* root) int x)
{
    struct node* q = (struct node*) malloc (size_of (struct node));
    q→data = x ;
    q→left = NULL;
    q→right = NULL;
    if (root == NULL)
        root = q ;
    else
    {   struct node* current_node = root ;
while(1)  if ( x > current_node → data)
  {       {       if(current_node → right != NULL)
                      current_node = current_node → right;
                  else
                      break;
        }
```

Scanned by CamScanner

```
else if (current-node -> data > x)
  {
        if(current-node -> left != NULL)
          current-node = current-node -> left;
        else
          break;
  }
} (while loop ends).

if ( current-node -> left == NULL

if( x < current-node -> data)
      current-node -> left = q;
if (x > current-node -> data);
      current-node -> right = q;

}

return root;

}.
```

DRY-RUN:



i) Compare 100 with 40. ∵ 40 ≤ 100, current-node = 20.

ii) Compare 20 with 40 ∵ 20 < 40 ; current-node = 30

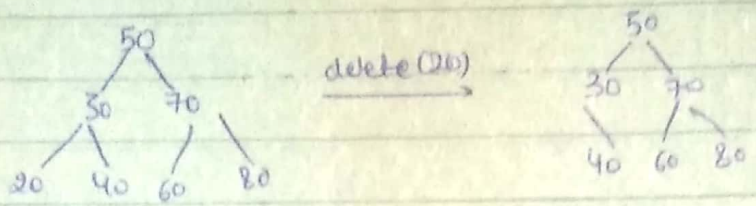
iii) Compare 30 with 40 ∵ 40 > 30 & 30 -> right = NULL. ∴ 30 -> right = 40

Scanned by CamScanner

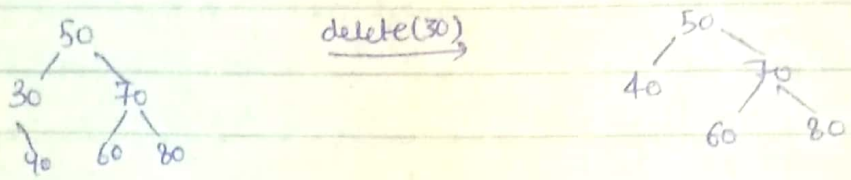
PROGRAM 27: Deletion From BST.

Logic: Three types of nodes exists in BST.

i) Node to be deleted is leaf node: Simply remove leaf node.

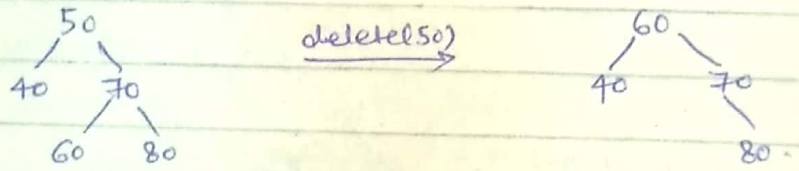


ii) Node to be deleted is a node with one child.



The child of node to be deleted is made child of parent of node to be deleted.

iii) Node to be deleted has two children:



Find inorder successor (minimum element in right subtree) of node to be deleted. Copy data of inorder successor to node to be deleted & delete inorder successor.

```
// utility function to find minimum value node in a tree (BST)

struct node* minvalue node (struct node *root)
{
        struct node * current = root;
        while (current -> left != NULL)
                current = current -> left;
        return current;
}
```

Scanned by CamScanner

// function to search & delete a key in BST.

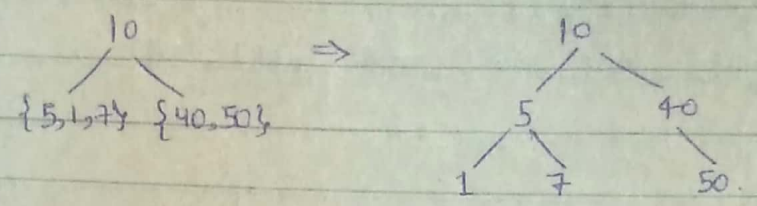
```
struct node * deleteNode ( struct node *root , int key)
{
        if (root ==NULL)
            return NULL ;
        if ( root -> data >key)
            root->left = deleteNode ( root->left , key)

    else if (root -> data < key )
        root->right = deleteNode (root -> right , key)

    else
    {
        // if node to be deleted has one or no child.
        if ( root -> left ==NULL)
          { struct node *temp=root -> right ;
                free (root) ;
                return temp ;
          }.
    else if ( root -> right ==NULL)
          {     struct node * temp = root -> left ;
                free(root) ;
                return temp ;
          }.
        // for node having 2 child.
        struct node *temp = minvalueNode (root->right) ;
        root -> data = temp -> data ;  // copying contents.
        root->right = deleteNode (root->right , key) ;
                                        temp->data
    }
    return root ;
}
```

Scanned by CamScanner

• Construct Binary Search Tree from Given pre-order Traversal.

Let Given pre-order be {10, 5, 1, 7, 40, 50}.
So, root node of tree will be 10.
Now Find next element > 10 in preorder. (ie. 40).
So, from 5 to 7 ⇒ {5, 1, 7} will form left subtree. and {40, 50} will form right - subtree.
Now, again using Preorder {5, 1, 7} built left subtree. & {40, 50} build right - subtree.

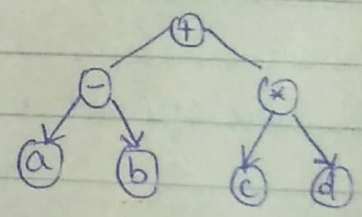
∴



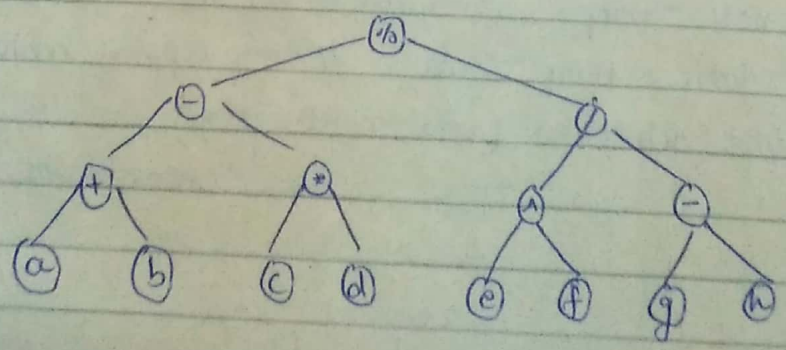
NOTE: Ascending order of elements of BST is inorder. Traversal.
eg. In above constructed tree, 1 5 7 10 40 50 is ascending order, which is also pre-in-order.

Expression tree: Binary trees are used to store algebric expressions. Such Binary trees are called expression trees.

Let expression be : (a-b) + (c*d)
(always see precedence).



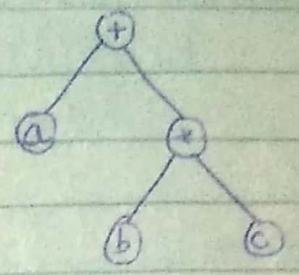
Let expression be : ((a+b)-(c*d))%((e^f)/(g-h))



Scanned by CamScanner

let expression tree be : $a + b * c$

$\equiv (a+(b*c))$



PROGRAM 27 : To count nodes in a tree.

```
int countNodes (struct node* root)
{
    if(root==NULL)
        return 0;
    return (1+ countNodes(root->right)+ countNodes(root->left));

}.
```

PROGRAM 28 : To calculate height of Binary tree.

```
int height ( struct node* root)
{
    if (root ==NULL)
        return 0;
    return ( 1+ max( height(root->left), height(root->right)));
}.
```

PROGRAM 29 : Check two Binary Trees for equality.

```
int similar (struct node *t1, struct node *t2)
{
    if (t1 ==NULL && t2 ==NULL)
        return 1;
```

Scanned by CamScanner

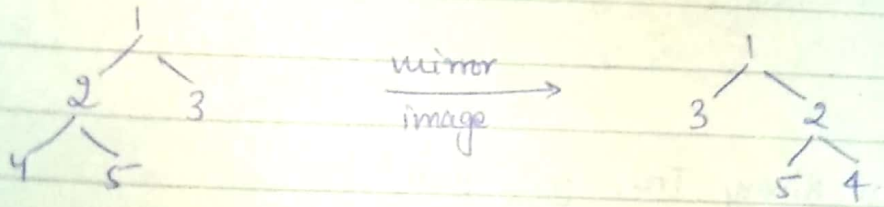
```
if ( t1!=NULL && t2!=NULL && t1->data = t2->data && similar(t1->left,
    t2->left) && similar (t1->right , t2->right) )
        return (1);

return (0);
}.
```

PROGRAM 30 : To copy a Binary Tree.

```
struct node * copy (struct node *root)
{
        if (root == NULL) return NULL
        Struct node * p = (Struct node*) malloc(sizeof(struct node) ) ;
        p->key = root->key ; // p->data = root->data.
        p->left = copy (root->left)
        p->right = copy (root->right);
        return p;
}
```

PROGRAM 31 : To find mirror image of a Binary Tree.

Mirror image of Binary tree means a binary tree in which childrens of each node are swapped or interchanged.




```
void mirror ( struct node * node) // Converting same tree into its Binary Tree Mirror
{
        if (node == NULL) return ;
        mirror (node->left) ;
        mirror (node->right) ;
        Struct node *temp;
```

Scanned by CamScanner

```
        temp = node->right;
        node->right = node->left;
        node->left = temp;
    }
```

PROGRAM 32: To check whether given binary Tree is a Full Binary Tree.

```
bool isfullTree (struct node* root)
{
        if (root==NULL)
            return 1;
        if (root->left ==NULL && root->right ==NULL)
            return 1;
        if ( root->left !=NULL && root->right !=NULL && isfullTree(root->left)
            && isfullTree(root->right) )
            return 1;
        return 0;
}
```

* THREADED BINARY TREE:

In a binary tree, there are a large no. of null left & null right pointers which proves to be a waste of memory. So, these pointers can be utilized to make inorder traversal of tree faster and do it without stack & without recursion.

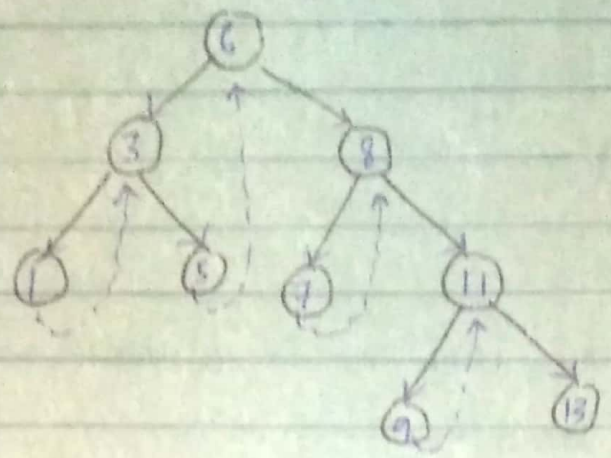
Two types of threaded trees exists :-

i) Single threaded : where a null right pointer is made to point to its inorder successor (if exists).

NOTE: Threads are useful for fast accessing of ancestor nodes.

Scanned by CamScanner

eg



Binary Tree Inorder :
1 3 5 6 7 8 9 11 13

Dotted lines are threads.

C representation of Single Threaded node.

```
struct node {
    int data;
    struct node * left;
    struct node * right;
    bool rightthread; // this is used to tell that whether node has
}                     // a right thread or a right child.
```

PROGRAM 33: Inorder Traversal using single threaded Binary Tree.

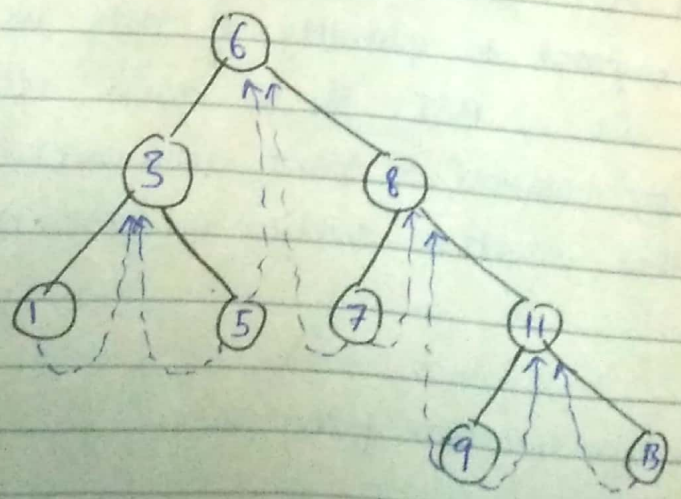
```
//utility function to access (reach) leftmost node in a binary tree.

struct node * goleft (struct node * root)
{
    if(root == NULL) return NULL;

    while (root -> left != NULL)
        root = root -> left;

    return root;
}
```

Scanned by CamScanner

```
void inOrder (struct node* root)
{

    struct node * temp = goleft(root);
    while (temp != NULL)
    {

        printf("%d ", temp->data);

        if (temp -> rightThread == 1)
            temp = temp->right;

        else

            temp = goleft (temp->right);
    }
}.
```

Double-Thereaded: where both left and right null pointers are made to point to inorder predecessor & inorder successor respectively.

E: Predecessor threads are useful for reverse inorder traversal.



threads: dotted lines

norder: 1 3 5 6 7 8 9 11 13.

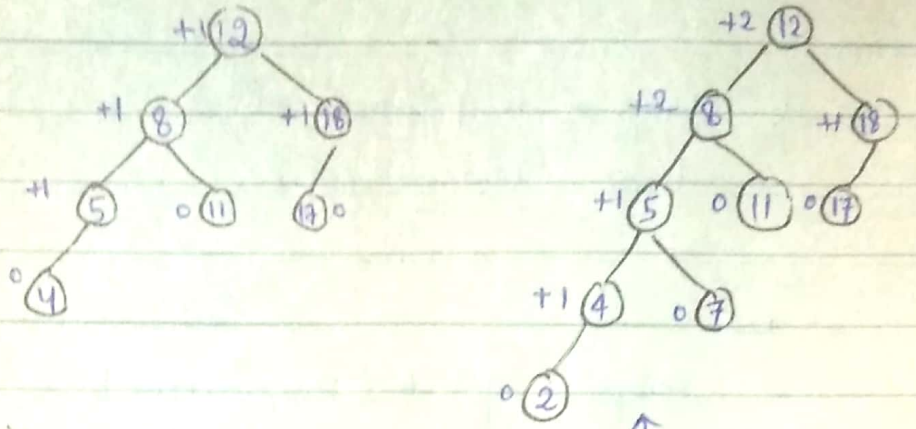
Scanned by CamScanner

* AVL TREES (self-balancing binary Search trees)

AVL trees are special types of Binary Search trees which are balanced i.e. in which balance factor of each node lies in range {-1, 0, +1} where balance factor is calculated as:

⇒ Balance factor of node = height of left subtree − height (right ST)

eg of AVL tree:



eg of BST which is not Binary AVL:

{ see node 8 and 12 has B.F = +2 ∴ Not AVL }.

- Why to use AVL trees: ?

Binary Search Trees (BST) are used to store data which involves quick and frequent searching, insertion & deletion operations. Complexity of all these operations is $O(h)$ where h is height of Binary search tree. If BST is a skewed BST, then complexity will become $O(n)$ which is quite large when compared to quantity of data we store in BST. where n = no. of nodes in a BST. So, to reduce $O(h)$ to nearly $O(\log n)$, tree should be balanced or nearly balanced. That's why we use AVL trees to ensure that insertion, deletion & searching operations takes $O(\log n)$ time.

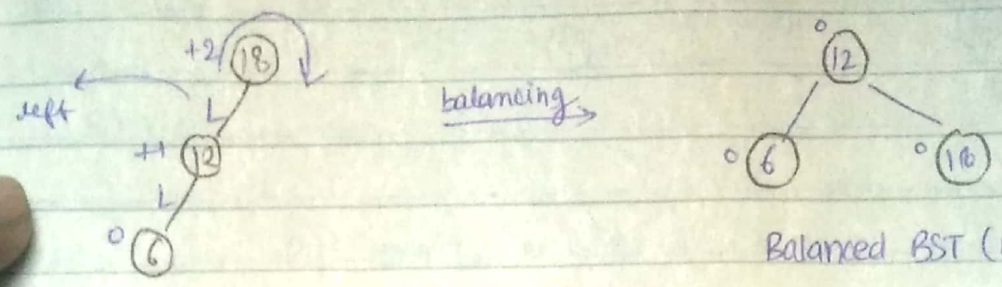
NOTE: All leaf nodes has balance factor zero.

Scanned by CamScanner

How to balance a BST to form AVL tree ?

Any unbalanced BST can be balanced using four types of rotations :

i) left-left rotation.



Balanced BST (AVL).

Procedure for doing left-left rotation :

a) Notice the leaf node. (6)

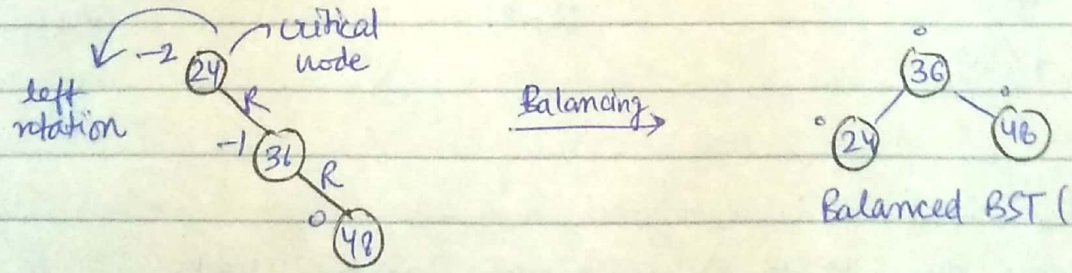
b) using a bottom-up-approach, move from leaf node to root-node (6 to 18) and find the first node whose B.F does not lie in desired range. (18).

c) Now, notice the first two route through which you can reach leaf node from critical node. (A critical node is a node which is unbalanced).

d) From 18, we have to reach 6, so movement will be left & then again left. That's why, it's a left-left problem. (LL)

d) Since, its a LL-problem, so, do right rotation about critical node to make tree balance.

ii) Right-Right Case :



Balanced BST (AVL).

Procedure for doing right-right problem solving :

a) Follow points a) → c) of above left-left case. These are general steps to be followed to find the type of problem.

b) from 24, we have to reach 48 which can be done using right-right movement

c) So, its RR problem. So, do left rotation about critical node to balance tree.

Scanned by CamScanner

c) Left-Right Case :

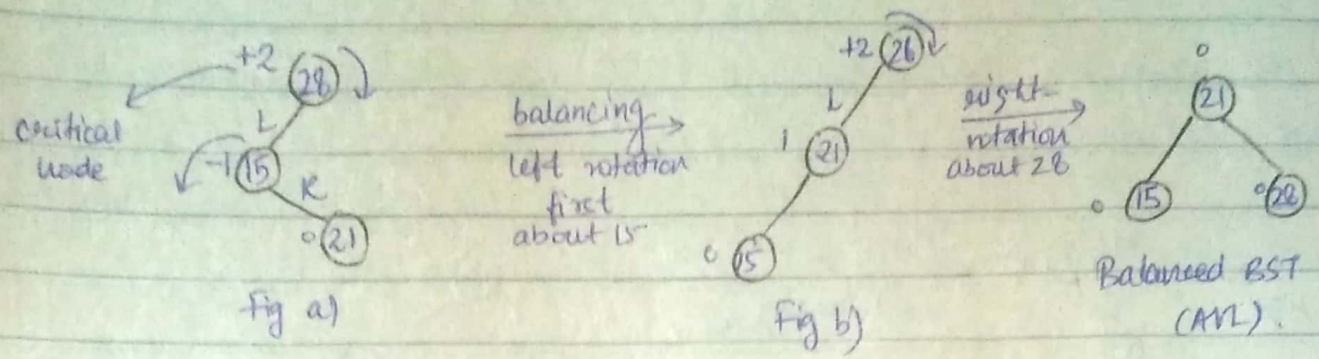


Fig a) Fig b)

Procedure :

a) Follow general steps a) to c) of previous problems first.

b) Now, we can reach 21 from 28 by moving first left then right, so, its a left-right case.

c) This problem is solved using two rotations.

d) Since, in fig a), node 15 suffers from R ∴ do left-rotation about 15 to get fig b).

e) Since, in fig b), node 28 suffers from L ∴ do right-rotation about 28 to get Balanced tree (AVL).

d) Right-Left Case :

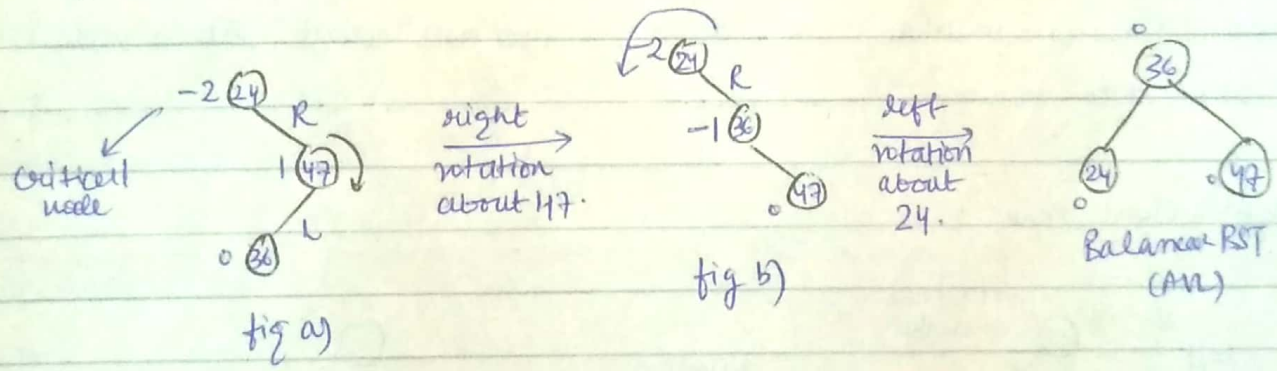


fig a) fig b)

Procedure :

a) Follow general steps a) to c) of previous problems first.

b) Now, we have to reach 36 from 24 by moving Right & left. So, its a right-left case.

c) Since, in fig a), node 47 suffers from L, do right-rotation about 47 to get fig b).

d) Since, in fig b), node 24 suffers from R, do left-rotation about 24 to get AVL.

Scanned by CamScanner

OPERATIONS ON AVL

i) Searching

Searching operation is same as done in case of BST.

ii) Insertion

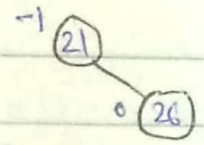
We will first insert the node in the same way as done in case of BST. Since, after insertion, tree may get some nodes having B.f out of desired range, so, we will balance it using above four discussed techniques.

Ques → Insert nodes : 21 26 30 9 4 14 28 18 15 10 2 3 7 in AVL tree.

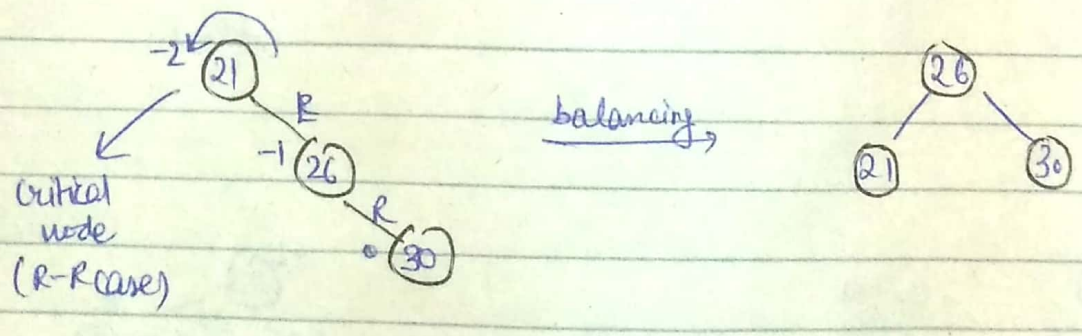
{ In insertion case, notice inserted node instead of leaf node }.

a) insert (21)

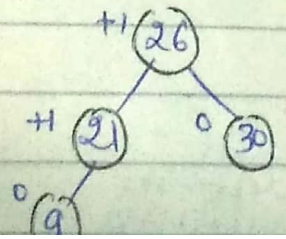
b) insert (26)



c) insert (30)



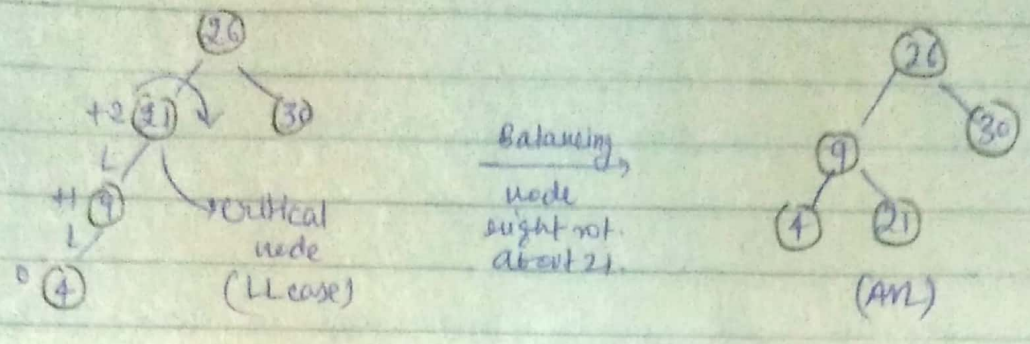
d) Insert (9)



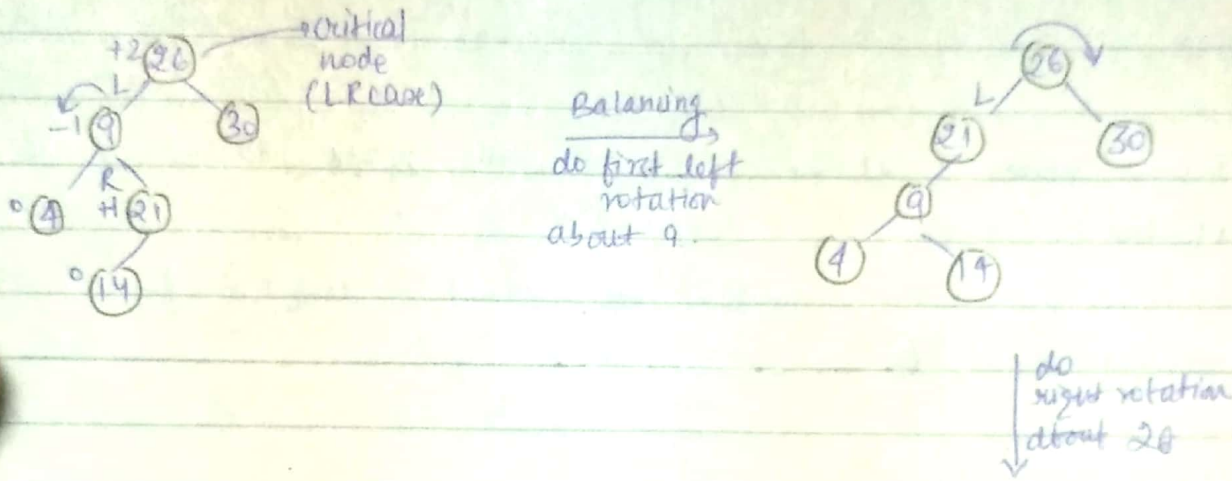
(Balanced) automatically

Scanned by CamScanner

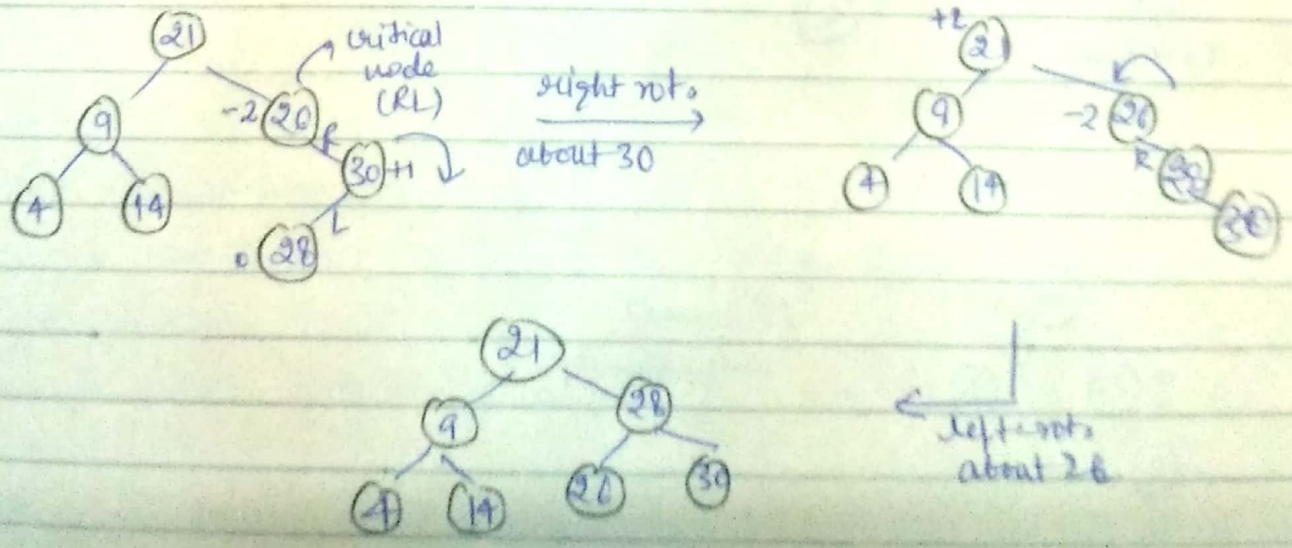
e) insert(4)



f) insert(14)

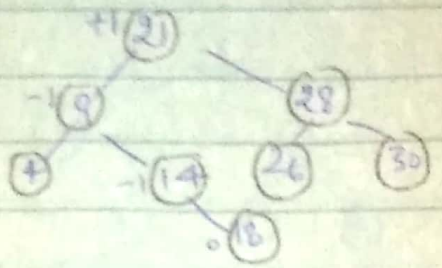


g) insert(28)



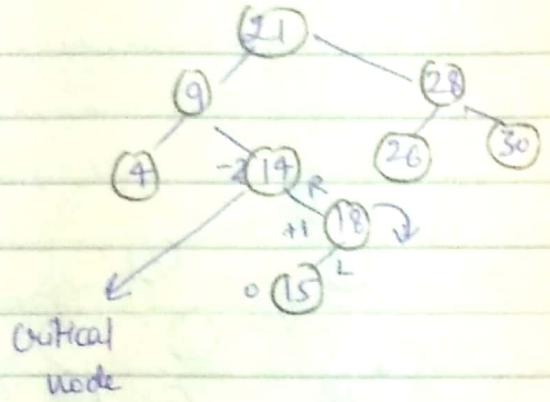
Scanned by CamScanner

h) insert (18)

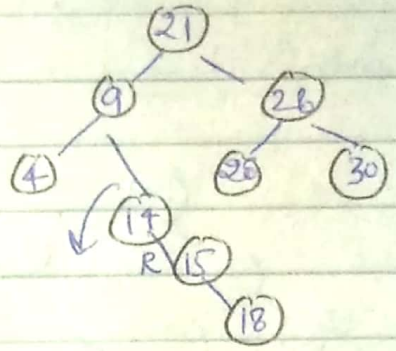


(balanced)

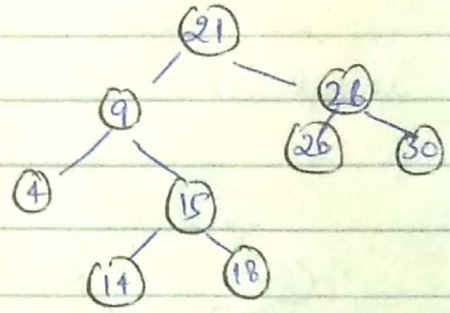
i) insert (15)



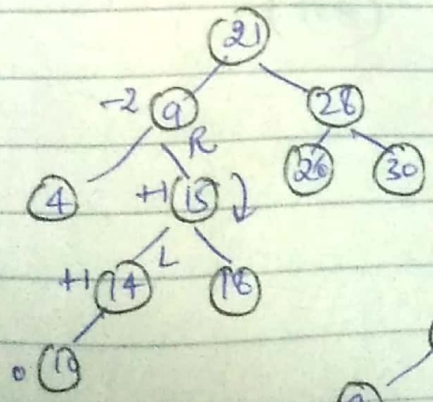
right rot
About 18.



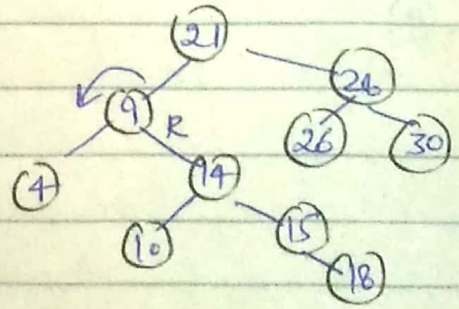
do left rotation about 14.

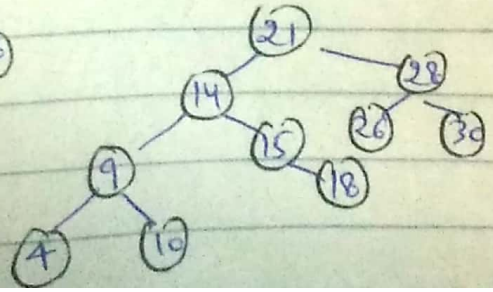


j) insert (10)



do rot.
rot.
about 15

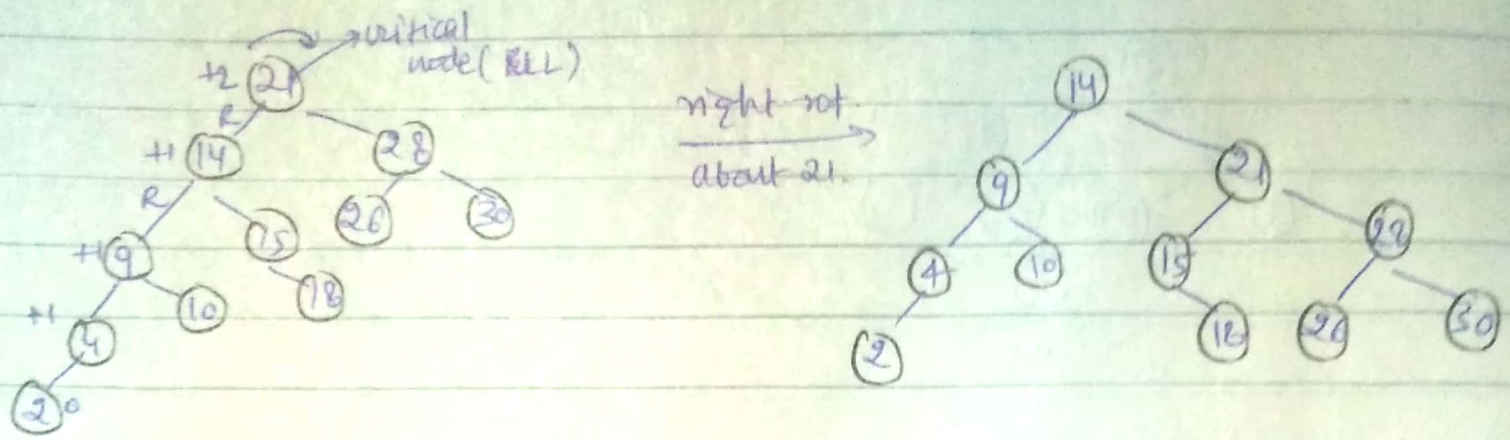




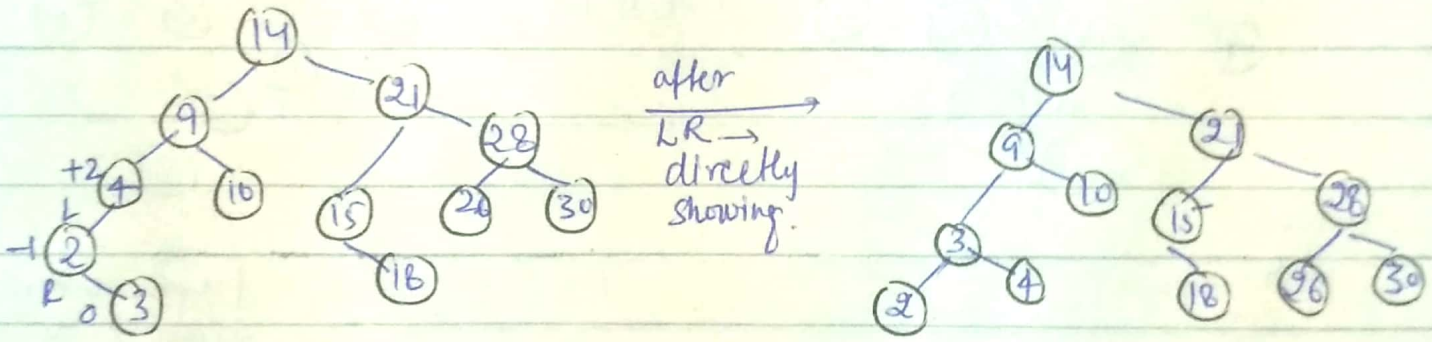
do left
rotation
about 9.

Scanned by CamScanner

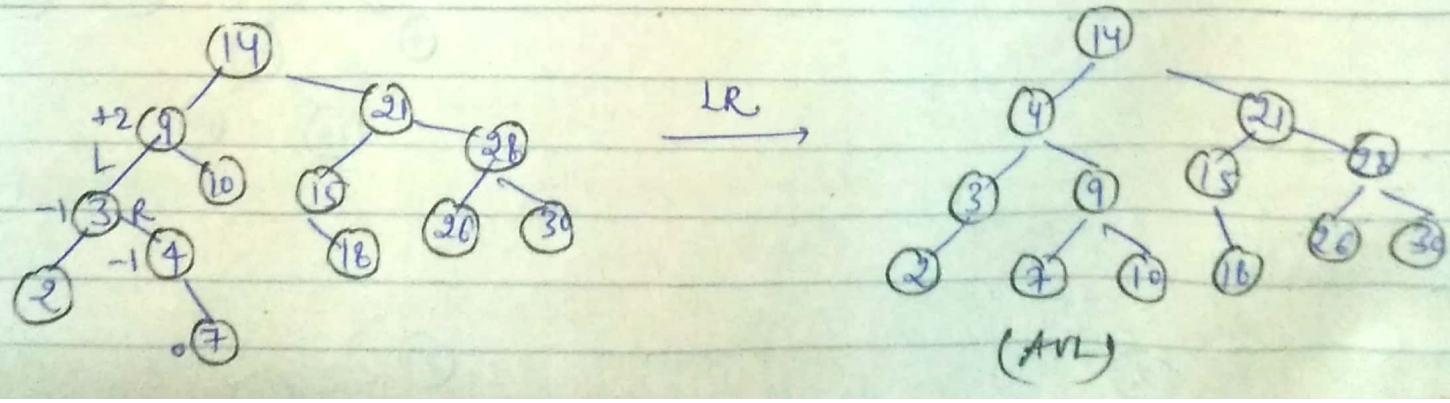
k) insert(2)



l) insert(3).



m) insert (7).



Scanned by CamScanner

* SORTING TECHNIQUES.

i) Bubble sort.

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

unsorted array

2	4	1	5	3	7

2	1	4	3	5	7

1	2	3	4	5	7.

For each pass through array, largest element in unsorted array is shifted to highest index of unsorted array.

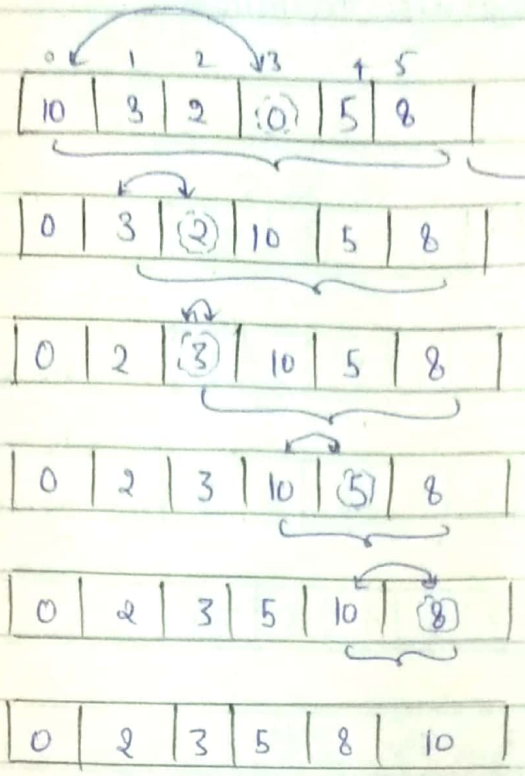
PSEUDOCODE:

```
Bubble_Sort ( A , n)
{
        for k←1 to n-1 do
        {
                flag←0 ;
                for i←1 to n-k-1 do
                {
                        if ( A[i] > A[i+1])
                        {
                                swap ( A[i] , A[i+1]) ;
                                flag←1;
                        }
                i←i+1 ;
                }
        if ( flag ==0) break ;   k ← k+1 ;
        }
}
```

- Best Case Time Complexity = $O(n)$
- Worst case Time Complexity = $O(n^2)$

Scanned by CamScanner

ii) Selection Sort.



→ unsorted array.

For each pass through the unsorted array, the smallest element in the unsorted array is swapped with first element in unsorted array.

* PSEUDOCODE :

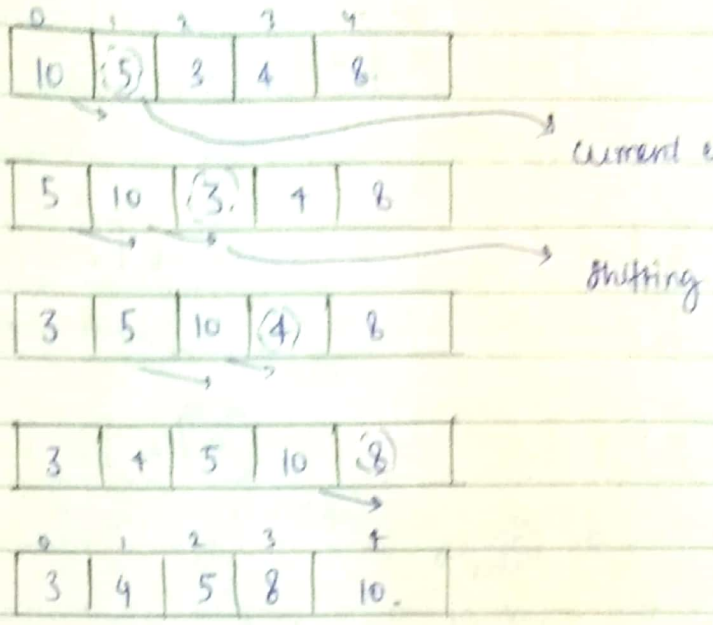
```
Selection-Sort (A, n)
{
        for i ← 1 to i ← n-1 do
        {
                Smallest ← A[i] ;
                pos ← i ;
                for j ← i to n-1 do
                {
                        if ( A[j] < smallest)
                        {
                                smallest ← A[j];
                                pos ← j;
                        }
                } j ← j+1 ;

                if ( pos != i )
                swap ( A[pos], A[i]) ;
        i ← i+1;
}
```

• Best Case Time Complexity : $O(n^2)$
• Best Case Time Complexity : $O(n^2)$
↳ worst

Scanned by CamScanner

iii) Insertion sort :

(A particular element in array is inserted at correct position in the array present before that element).



* PSEUDOCODE :

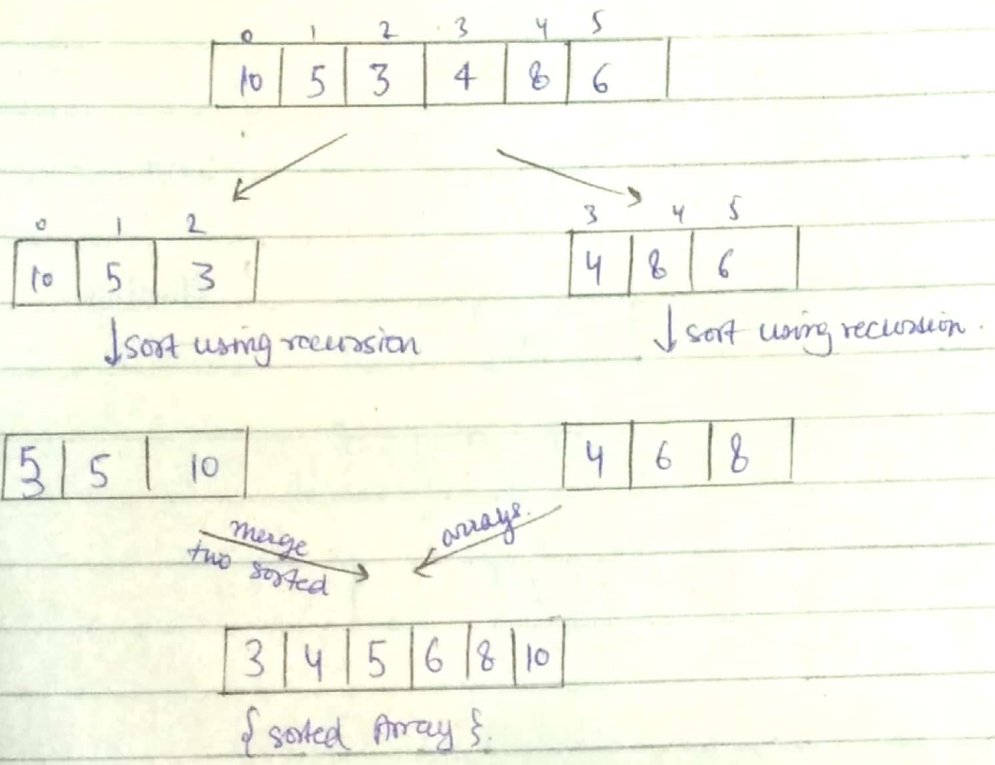
```
InsertionSort (A,n)
{
        for i ← 1 to n-1 do
        {
                x ← A[i];
                j ← i-1 ;
                while ( j>=0 & A[j]>x)
                {
                        A[j+1] ← A[j];
                        j ← j-1 ;
                }
                A[j+1] ← x ;
                i ← i+1;
        }
}.
```

- Best Case Time Complexity : $O(n)$
- Worst case Time Complexity : $O(n^2)$

Scanned by CamScanner

iv) Merge sort :

(Break given array into two halves, then sort each half using mergesort() and merge two sorted arrays).



* Pseudocode :

```
Mergesort (A, start, end)
 {
        if (start == end)
            return;

        mid ← (start + end)/2 ;
            Merge Sort (A, start, mid) ;
            Mergesort ( A, mid+1, end) ;
            Merge (A, start, mid, end) ;

 }
```

• Time Complexity = o(nlogn)
  ↳ (in all cases)

Scanned by CamScanner

* Quick Sort.

{ Find the correct position of first element in Array. Then place that element, and rearrange the array elements, such that all elements smaller than the placed element lies on left side of it and all elements greater than placed element lies on right side of it. }. Now, sort the left and right part using same QuickSort() and whole array will get sorted }.

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

↓ placing 0th index element.

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

since, there are 2 elements greater than 5 at previous stage in Array (ie 3 & 4).
∴ 5 is swapped with the 2th indexed element.

↓ rearranging

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

↓ Sorting left & right parts

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

{ Sorted Array }.

* PSEUDOCODE:

```
QuickSort ( A, start, end)
{
        if( start >= end)
            return;
        mid ← partition (A, start, end);
        QuickSort (A, start, mid-1);
        QuickSort (A, mid+1, end);
}
```

* Avg-case Time Complexity : $O(n \log n)$. (Best)
* Worst case Time Complexity : $O(n^2)$ {Array is already sorted}.

Scanned by CamScanner