CEC02 : Data Structures

• Linear Search (Time Complexity)

```
int LinearSearch (int A[], int n, int key)
// Returns index of the key element
// If not found returns -1
{
  int i;            // t1
  for (i=0; i<n; i++)
     if (A[i]== key)   // t2
       return i;
  return -1;        // t3
}
```

for best case, loop run one time, for ~~a~~ worst case it ~~...~~ runs n times.

Taking worst case,

$$t = t_1 + nt_2 + t_3$$

$$nt_2 >> t_1, t_3.$$

So $t = nt_2$ $\boxed{O(n)}$ Big-oh notation

hence loop controls the time taken by the code.

- Selection Sort

```
for(i=0; i<n-1; i++)
  for (j=i+1; j<n; j++)
    if (A[i] > A[j])
    { temp = A[i];
          A[i] = A[j];
          A[j] = temp;
    }.
```

here time taken is $n^2(t)$ $\left[O(n^2) \right.$

hence controlling factor is Quadratic power

- Matrix Multiplication

```
for (i=0; i<m; i++)
  for (j=0; j<p; j++)
  {
     C[i][j]=0;
     for (k=0; k<n; k++)
        C[i][j] += A[i][k] * B[k][j];
  }
```

Time is mpn

If square matrices,

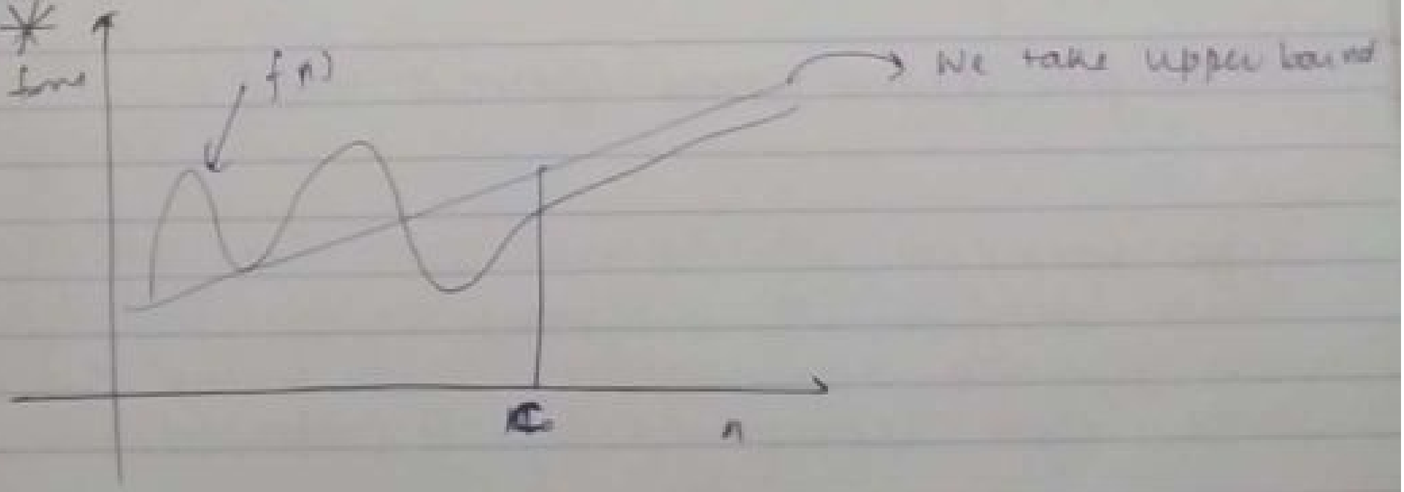
$t = n^3$ $\left[O(n^3) \right.$

hence, controlling factor is Cubic power

* Time taken is increased as nesting of loop increases.
* when two loops run, but are not nested then time taken is almost equal to the time taken by single loop.

* Time and storage are two factors which describe the program.
Like when we run fibonacci series program using recursion, storage is more. But as memory is cheap nowdays, main focus is on Time taken by the program.

*



If f(n) is same function than we can find another g(n)
Such that $n \geq n_0$ $|f(n)| \leq c|g(n)|$
$O f(n) = g(n)$.

* When time is constant,
Bigoh notation is O(1)

* $O(1) < O(n) < O(n^2) < O(n^3)$

- $a_n n^k + a_{n-1} n^{k-1} + a_{n-2} n^{k-2} + \cdots + a_1 n + a_0$

$\leq |a_n| n^k + |a_{n-1}| n^{k-1} + |a_{n-2}| n^{k-2} + \cdots + |a_1| n + |a_0|$

$\leq n^k \left(|a_n| + \frac{|a_{n-1}|}{n} + \frac{|a_{n-2}|}{n^2} + \cdots + \frac{|a_1|}{n^{k-1}} + \frac{|a_0|}{n^k} \right)$

$\leq n^k (|a_n| + |a_{n-1}| + |a_{n-2}| + \cdots + |a_1| + |a_0|)$

$\leq n^k (l) \leq l n^k$ [(taking upper bound)]

hence $O(\text{polynomial}) = O(n^k)$

[⇒ higher degree terms controls time]

Note:
upper bound ⇒ O (Bigoh)
lower bound ⇒ Ω (Sigma)
when function, upper bound, lower bound are same, then represented by Θ (theeta).

Polynomial Addition.

$\longrightarrow$ Descending Power

- $100x^{1000} + 2x^{500} - 10x^{50} - 50$

coef x^{exp} / cx^e $\boxed{c \mid e}$

	$100x^{1000}$		$2x^{500}$		$-10x^{50}$		-50	
4	100	1000	2	500	-10	50	-50	0
1	2	3	4	5	6	7	8	9

no of terms.

- $50x^{5000} + 10x^{600} - 2x^{500} + 100$

4	50	5000	10	600	-2	500	100	0
1	2	3	4	5	6	7	8	9

Adding above two polynomials

$50x^{5000} + 100x^{1000} + 10x^{600} - 10x^{50} + 50$

- compare the exponents
  if exp are equal, add the coefficient
  otherwise whichever is bigger, copy that to output and advance the pointer of that polynomial to next location.

- If the sum of coefficients are is zero, do not save it

- If one polynomial ends, copy the remaining second polynomial as it is

Function that can be performed:

Polynomial ( ) → Create a polynomial
Polynomial (zero) → Create a zero polynomial
Is zero (Polynomial) → True if polynomial is zero
polynomial attach (Polynomial, coef, exp).
attach new term in the polynomial.
poly remove (Poly, coef, exp) → delete the term
poly add (Poly1, Poly2) .
poly mult (poly, term) → cx^e
poly mult (Poly1, poly2) .

$a_{m-1} x^{e_{m-1}} + a_{m-2} x^{e_{m-2}} - - - - a_0$

$\rightarrow$ (m, a_{m-1}, e_{m-1}, e_{m-2}, -- a_0, 0)

- polyadd (A has m terms, B has n terms)

polyadd (A, B, C) // add poly A & B in poly C

// A (1 : 2m+1) , B (1 : 2n+1) , C (1 : 2(m+n)+1)
size of A   size of B   size of C

m ← A(1) // no of terms in A
n ← B(1) // no of terms in B

p = q = r = 2 : // p points to first term in A
// q points to first term in B
// r points to first term in C.

while (p <= 2m && q <= 2n) // while both have ten
{ If (A [p+1] == B[q+1]) .
  {
    C[r] = A[P] + B[q] ;
    If (C[r] != 0)
    { C [r+1] = A[p+1] ;
      r = r+2 ;
    }
    p = p+2 ; q = q+2 ;
  }
else
  if (A [p+1] > B[q+1])

```
{
   C[i] = A[p];
   C[i+1] = A[p+1];
   p = p+2;
   i = i+2;
}
else
{
   C[i] = B[q];
   C[i+1] = B[q+1];
   q = q+2;
   i = i+2;
}
} // end of while loop.

// copy remaining items in C.

while (p <= 2m)   // copy A
{
   C[i] = A[p];
   C[i+1] = A[p+1];
   i = i+2;   p = p+2;
}
while (q <= 2n) // copy B
{
   C[i] = B[q];
   C[i+1] = B[q+1];
   i = i+2;  q = q+2;
}
C[1] = ⌊s/2⌋  // floor s/2)
```

time complexity,
O(max(m,n))
worst case:
O(m+n)

• multiply by single term:

```
mult (A, c, e, C)    → output
{
  // A has m term)
  p=2; r=2;

  while ( p<=2m)
  {
    C[r] = A[p]*c;
    C[r+1] = A[p+1]+e;
    p=p+2; r=r+2;
  }
    C[1] = A[1];
}
```

For multiplication of 2 polynomial,

multiply one term of 1st polynomial, with 2nd polynomial, and add to some null polynomial,
then multiply 2nd term of 1st polynomial with 2nd polynomial, and add to (previously done added) null polynomial & so on.

Time Complexity : $O(m^2)$

$O[(m+n) \times (m)] = O(m^2+mn)$

Lower Triangular Matrix

$$\begin{array}{c|c}
 & \begin{array}{cccc} 1 & 2 & 3 & \end{array} \\
\begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \end{array} & \begin{bmatrix} x & & & \\ x & x & & \\ x & x & x & \\ x & x & x & x & \cdots \end{bmatrix}_{n \times n}
\end{array}$$

$$\Downarrow$$

	1,1	2,1	2,2	3,1	3,2	3,3
	Row 1	Row 2		Row 3		
	α	$\alpha+1$	$\alpha+2$	$\alpha+3$	$\alpha+4$	$\alpha+5$

$$A(i,j) = \alpha + \frac{(i-1)i}{2} + j - 1$$

$A[1,1] = \alpha + 0 + 0 = \alpha$

$A[2,1] = \alpha + 1$

$A[2,2] = \alpha + 2$

$A[3,1] = \alpha + 3$

and so on.

SPARSE MATRIX

Matrix with more than 50% zero

We store only non-zero elements, and not the zeroes.

$$
\begin{array}{c|cccccc}
 & 1 & 2 & 3 & 4 & 5 & 6 \\
1 & 15 & 0 & 0 & 22 & 0 & -15 \\
2 & 0 & 1 & 3 & 0 & 0 & 0 \\
3 & 0 & 0 & 0 & -6 & 0 & 0 \\
4 & 0 & 0 & 0 & 0 & 0 & 0 \\
5 & 91 & 0 & 0 & 0 & 0 & 0 \\
6 & 0 & 0 & 28 & 0 & 0 & 0
\end{array}
\quad 6 \times 6
$$

A	row	col	value
1	1	1	15
2	1	4	22
3	1	6	-15
4	2	2	1
5	2	3	3
6	3	4	-6
7	5	1	91
8	6	3	28
0	6	6	8

← non zero element

no of rows   no of coloums

- Transpose of Sparse Matrix

Transpose(A, B)
// Find Transpose of Sparse Matrix A in B.

$(m, n, t) \leftarrow (A(0,1), A(0,2), A(0,3))$

$B(0,1), B(0,2), B(0,3) \leftarrow (n, m, t)$

if $t=0$ then return.

$q \leftarrow 1$ // q is position of next term in B

for (col ← 1 to n) (n times)
{ (t times)
for (p ← 1 to t)
if (A(p,2) == col)

B(q,1), B(q,2), B(q,3) ← A(p,2), A(p,1), A(p,3)
q ← q + 1

}

B	row	col	value
1	1	1	15
2	1	5	91
3	2	2	1
4	3	2	3
5	5	6	28
6	4	1	22
7	4	5	-6
8	6	1	-15

(stored in increasing order of row

Time Complexity : $O(n.t)$

If we apply this algorithm to any other matrix,
then $t = mn$.

Time complexity = $O(mn^2)$

If we find transpose of non-sparse matrix, by the standard method (of swaping)

Time complexity = $O(mn)$.

```
for( i=1 ; i<=row ; i++ )
  for( j=1 , j <=col ; j++ )
     B[j][i] = A[i][j]
```

FAST_TRANSPOSE (A, B) {

```
declare   S(1:n) , T(1:n) of Integers
(m, n, t) <- (A(0,1) , A(0,2) , A(0,3))
(B(0,1) , B(0,2) , B(0,3)) <- (n, m, t)
   if (t = =0) return;
                                   //1   O(n)
for ( i=1 to n)
  -> S(i) <- 0
                                   //2   O(t)
for ( i=1 to t)
  -> S(A(i,2)) <- S(A(i,2)) + 1
 T(1) <- 1
                                   //3   O(n-1)
for ( i=2 to n)
  -> T(i) <- T(i-1) + S(i-1);
                                   //4   O(t)
for ( i=1 to t)
     j = A(i,2)
 (B(T(j),1), B(T(j),2) , B(T(j), 3)) <-
          (A(i,2), A(i,1) , A(i,3))
  -> T(j) <- T(j)+1
```

}

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

// by loop 2

T	1	2	3	4	5	6
	1	3	4	6	8	8

// by loop 3.
(S[1] + T[1] = T[2]
S[2] + T[2] = T[3]
)

Time complexity: $O(n+t+n-1+1) = O(2n+2t-1)$
$= O(n+t)$

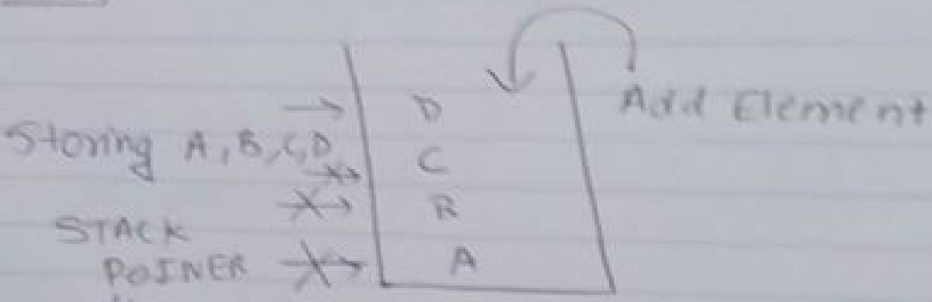
for normal matrix. $O(n+mn) = O(mn)$

DRY RUN OF LOOP 4

i	j	T(j)
1	1	1 → 2 (T(1))
2	4	6 → 7 [T(4)]
3	6	8 → 9 [T(6)]
4	2	3 → 4 [T(2)]
5	3	4 → 5 [T(3)]
6	4	7 → 8 [T(4)]
7	1	2 → 3 T(1)]
8	3	5 → 6 T(3)

B	row	col	value
1	1	1	15
2	1	5	91
3	2	2	1
4	3	2	3
5	3	6	28
6	4	1	22
7	4	3	-6
8	6	1	-15

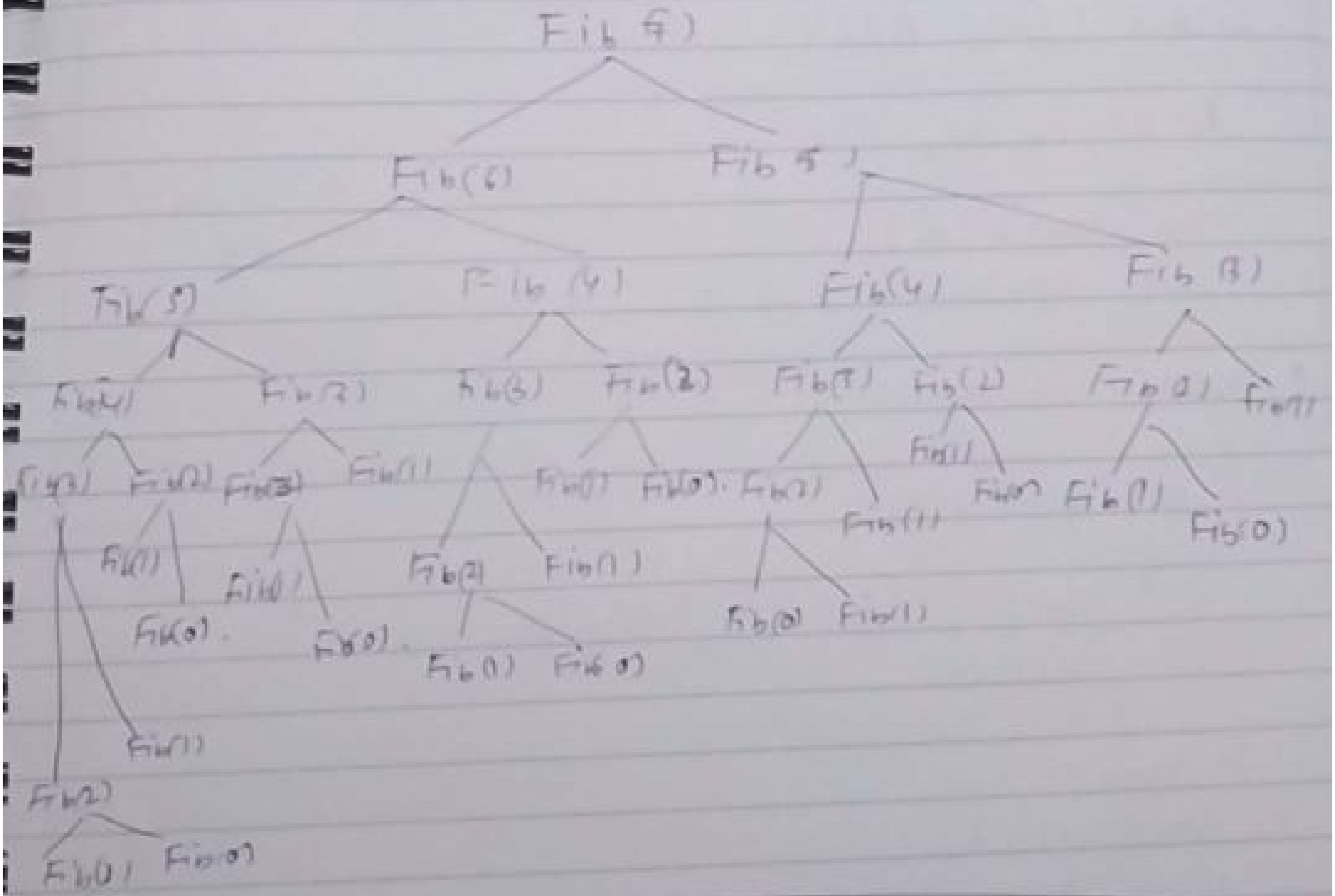
STACK



↳ Always point to top of Stack

Adding an element : Push (Stack pointer increment)
Deleting an element : Pop (Stack pointer decrement)
return the top element : Peep
True if Stack is empty : IsEmpty

-
```
Int fib(int n)
{if (n==0||n==1)
     return n;
return fib(n-1) + fin(n-2).
}
```



Memory stores variables in Stack in b/w function calls.

STACK USING ARRAY

```
# define N, 100

int Stack[N];
int sp=-1;

• Push (int x)
{
    if (sp==N-1)
    {
        printf ("stack is Full\n");
        return;
    }
    sp++;
    Stack[sp]=x;
    printf ("Push %d\n",x);
}

• int Pop()
{
    int x;
    if (sp==-1)
    {
        printf ("Stack is empty\n");
        printf (" Nothing to PoP \n");
        return -1;
    }
    x=Stack[sp];
    sp=sp-1;
    return x; }
```

• show ()

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

• int peep()

```
{
  if (!isempty())
     return stack[sp];
}
```

• int isempty()

```
{
return (sp==-1 ? 1: 0);
}
```

<u>Expression</u>

infix	A+B	
postfix	AB+	no () are required.
Prefix	+AB	

• Calculators internally use post fix expressions.

• Post fix expressions are called polish NOTATIONS

Infix: A+B*C+D+F/G
Postfix: ABC*+D+FG/+

Eval(E) {

// Evaluate the Postfix expression E, Assume the last character in E is $. NEXT_TOKEN(E) is used to extract from E next token which is either an operand, operator or $. S(1:n) for stack //

```
top=0
while (true) {

    x= next_token(E);
switch(x) {

    $: return; // answer is on the top of the stack//
x is operand: Push(x);
  Operator: item1= POP(); item2 = POP();
            push (item2 op item1)

    }
  }
}.
```

Q Evaluate the following postfix expression:
(using stack).

Scanned by CamScanner

• ABC * + D + FG / + $

A	Push	A
B	Push	A B
C	Push.	A B C.
*	Pop & push	A B*C
+	Pop & push.	A+B*C
D	Push	A+B*C • D.
+	Pop & push	A+B*C+D
F	push	A+B*C+D F
G	Push	A+B*C+D F G
/	pop & push	A+B*C+D F/G
+	pop & push	A+B*C+D+F/G
$	return	

Answer is: A+B*C+D+F/G

• 6 7 5 - * 36 6 / - $

6	push	6
7	push	6 7
5	push	6 7 5
-	pop & push	6 2
*	pop & push	12
36	push	12 36
6	push	12 36 6
/	pop & push	12 6
-	pop & push	6.
$	return	

Answer: 6.

Infix to Postfix

Assumptions — +, − have equal priority } (+, −) < (*, /)
*, / have equal priority

Initial symbol on the stack is '$' which has least priority.

1. Read the tokens left to right. It will return next token T
2. If T is operand, then print it in the output.
3. (i) If T is operator and has higher priority than operator on the top of stack, then Push T over the stack.
(ii) If T is operator and has less than or equal to priority than top symbol of the stack then pop the stack. Repeat popping stack until the priority of stack symbol is less than T. Push T over the stack.
(iii) If T is '(' [left paranthesis]. then simply push it over the stack
(iv) If T is ')' [right paranthesis], pop the stack

until '(' is found in stack. Delete '(' from the stack and discard ')' in this case.

4. If EOF (End of input) reached then POP until $ found.

- Convert to Post Fix:

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

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

$\boxed{ABC*+ \quad DE+GH+/+}$ Post Fix

$\boxed{++A*BC/+DE+GH}$ prefix. [Move Right to Left]

T	operation	Stack	output
A	Add to output	$	A
+	Push to stack	$+	A
B	Add to output	$+	AB
*	Push to stack	$+*	AB
C	Add to output	$+*	ABC
+	Pop & push	$+	ABC*+
(	Push to stack	$+(	ABC*+
D	Add to output	$+(	ABC*+D
+	Push to stack	$+(+	ABC*+D
E	Add to output	$+(+	ABC*+DE
)	Pop & push	$+	ABC*+DE+
/	Push to stack	$+/	ABC*+DE+
(	Push to stack	$+/(	ABC*+DE+
G	Add to output	$+/(	ABC*+DE+G

T	operation	Stack	output
+	Push to stack	$ + / (+	ABC ** + DE + G
H	Add to output	$ + / (+	ABC** + DE+GH
)	Pop & push	$ + /	ABC ** + DE+GH +
EOF	Pop	$	ABC ** + DE + GH + / +

ABC ** + DE + GH + / +

'(' (left paranthen) : When we push it in stack we assume it of highest priority, but when it is in the stack we assume is so lowest priority as we push any other operators above it.

Symbol	In stack priority (ISP)	In comping priority (ICP)
)	–	–
(exponential) **	3	4.
binary [*, /	2	2
binary [+, -	1	1
(	0	4.
$	Least priority.	

associativity matters here : ((9^2)^3^2)

```
Postfix(E) {
// Convert infix E into postfix . Last char in E is $
  Function NEXT_TOKEN returns operand, operator or $
  ISP(x) , ICP(x)
  S(1) = $ , top=1
while ( true )
  {
    x = next_token();
case {
  : x = '$' : while (top>1)
               {
                 print ( S(top))
                  top = top -1
               }
              return
 : x is operand : print x in output .
 : x is ')' : while ( S(top) ≠ '(' )
               {
                  print S(top)
                   top = top -1
               }
              top = top -1    // to eliminate '('
 : else : while ( ISP (top) >= ICP (x))
          { print S( top)
             top = top -1
          }.
          Push (x, S, top) // function call (3 parameter)
} // end of case
} // end of while
} // end of Postfix .
```

```
Prefix(E) {
// convert E to (infix) into Prefix //
reverse(E) // Reverse the infix exp E & append $ //
S(1) = $, top=1; // initial stack //
while (true)
{
  x = next token(E); // extract next token from E //
if (x == $)
{
  // when complete input has been processed //
  while (top > 1)
    {
      Add S(top) to prefix expression P
      top = top - 1;
    }
    reverse(p);
    return;
}
else if (x is operand)
    Add x to the end of P
else if (x == '(')
    {
      while ( S(top) != ')')
        {
          Add S(top) to P
          top = top - 1;
        }
      top = top - 1; // delete ')' //
    }
else
```

```
{
    while ( ICP(top) > ICP(x))
    {
        add S(top) to P
        top= top-1;
    }
    top = top+1;   S(top)= x ; // Push x //
}
}
}
```

• A+ B/C +D (Infix to prefix)

(By algorithm)

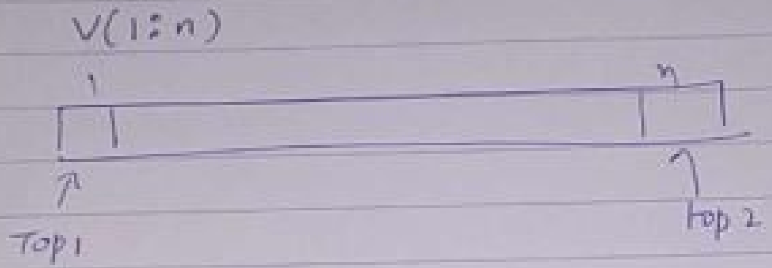
D+C/B+A $

Token	Operation	Stack	Output
D	add to o/p	$	D
+	add push	$ +	D
C	add to o/p	$ +	D C
/	Push	$+/	D C
B	add to o/p	$+/	D C B
+	Pop & push	$++	DCB/
A	add to o/p	$++	DCB/A
$	do pop & add	$	DCB/A++

P= reverse (output) = ++A/BCD.

$\boxed{\text{OUTPUT} = ++A/BCD}$

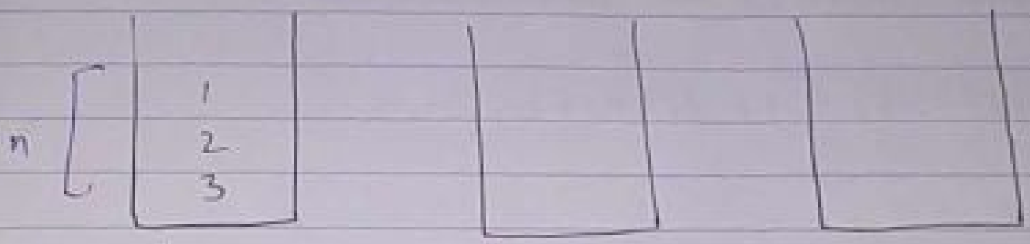
Implement two stacks in one array



(Initially)
top1 = 0,
top2 = n+1;

```
Push1(x)
{
  if (top+1 == top2)
  {
    printf("Element %d can't be pushed\n", x);
    return;
  }
  top1 = top1+1;
  v[top1] = x;
}

Push2(x)
{
  if (top+1 == top2)
  {
    printf("Element %d can't be pushed\n", x);
    return;
  }
  top2 = top2-1;
  v[top2] = x;
}.
```

TOWER OF HANOI



Source — A
auxiliary — B
Destination — C

Ruls: 1 picked at a time
bigger disk can't be placed over smaller disk

```
move (n, A, B, C)
{
    if (n==1)
    {
        printf (A → C);
        return;
    }
    else if (n>1)
    {
        move (n-1, A, C, B);
        printf (A→C);
        move (n-1, B, A, C);
    }
}
```

7 $\mathcal{U}(n)$ = no of moves required to transfer n disks from source to destination

$\mu(n) = \mu(n-1) + 1 + \mu(n-1)$

$\mu(0) = 0$

$\mu(1) = 1$

$\mu(n) = 2\mu(n-1) + 1$

$\mu_n = 2[2\mu(n-2)+1] + 1$

$= 2^2\mu(n-2) + 2 + 1$

$= 2^k\mu(n-k) + 2^{k-1} + \cdots + 2 + 1$

$k = n-1$ (for n rings)

$= 2^{n-1}\mu(n-(n-1)] + 2^{n-2} + \cdots 2 + 1$

(GP) $= 2^{n-1} + 2^{n-2} + \cdots 2 + 1$ as $\mu(1) = 1$

$= \dfrac{2^n - 1}{2-1} = \boxed{2^n - 1}$

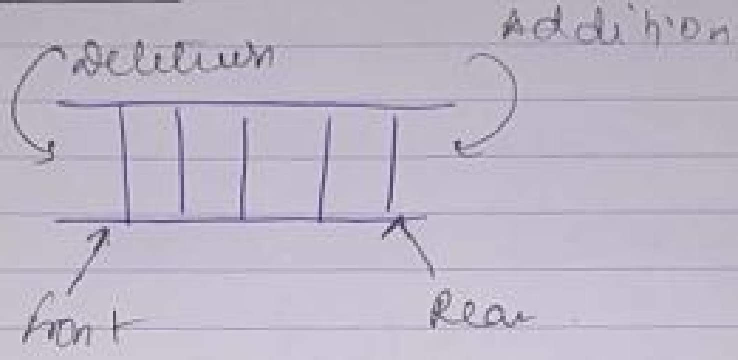
<u>Very large</u>

for 10 terms: $2^{10} - 1 = 1023$

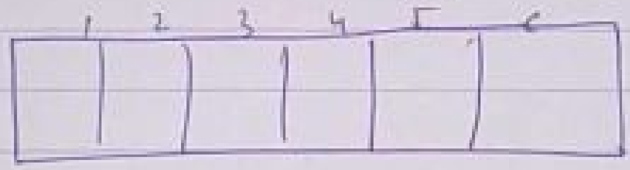
Iterative version

move (n, A, B, C)
  top = 0,
1. if (n==1)
   { print A → C; goto 5 }
2. (a) (i) top = top+1
   (ii) stackA[top] = n, stackA[top] = A
   stackB[top] = B, stack C[top] = C
   stack_ADD[top] = 3.
  (b) n = n-1
   interchange B & C
  (c) goto 1
3. Print A → C
4. (a) top = top + 1
   stack n [top] = n
   stackA[top] = A
   stack B [top] = B
   stack c [top] = C
   stack ADD [top] = 5
  (b) n = n-1
   Swap A and B
  (C) goto 1.
5 (a) if (top = = NULL) return
  (b) (a) n = stack n [top]
   A = Stack A [top]
   B = stack B [top]
   C = stack C [top]
   add = stack_Add [top]
  (b) top = top-1
  (c) goto address given in variable add.

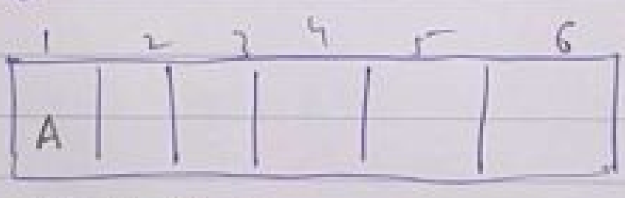
QUEUES



When Queue is empty F=0, R=0

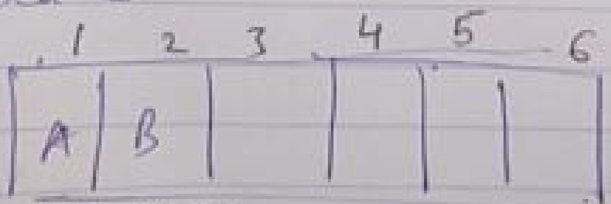


Add A



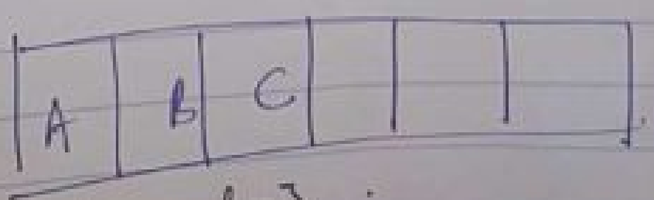
F=1, R=1

Add B



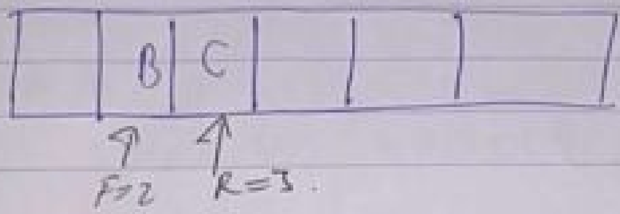
F=1 R=2

Add C

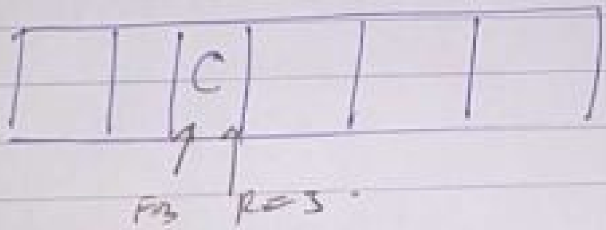


F=1 R=3

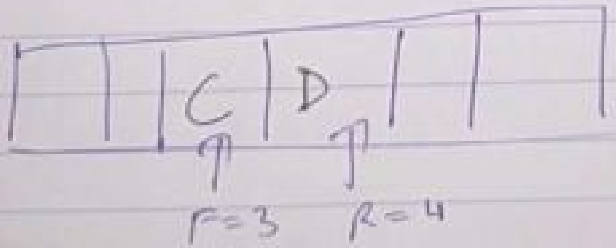
Delete A



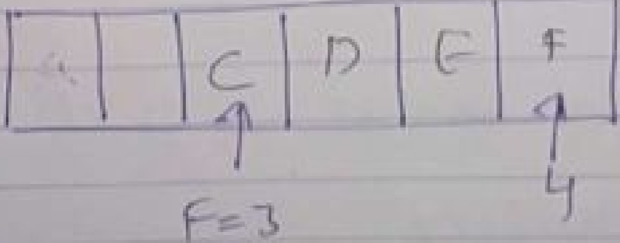
Delete B



Add D



Add E, F



To fix this

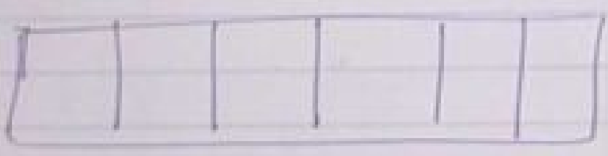
To fill these space, we can make Rear to point at start : Circular Queue

If we do not do anything, space is left but can't be used :, Skewed Queue

Delete C, D, E

F=6, R=6

Delete F



F=0, R=0.

In circular Queue, if F=R+1, Queue is full

If F=1, R=6, then Queue is full
that is last portion

(no item is deleted)

Operations on Queue's

1) Create a Queue
2) Insert / Add item
3) Delete item / element
4) isEmptyQ()
5) isQFull().

Q[1:N] → Array for Queue having capacity for n items.

```
F=0, R=0;

Add (Q, item)
{
  if((F==1 && R==N)||(F==R+1))
  {
    Queue_full();
    return;
  }
  if (F==0 && R==0)        //(Empty Queue).
      F=R=1;
  else
     if (R==N)             //(for circular)
        R=1;
  else
      R=R+1;               //(General case)
  Q[R]=item;
}.
```

```
Delete (Q,item)
{
  // delete front element from Q & store in
     item //

  if (F==0 && R==0)
   {
     QueueEmpty();
     return;
   }
  if (F==R)
  {
    item = Q[F];
    F=R=0;
  }
else
  if (F==N)
  {
    item= Q(F);
    F=1;
else
  {
    item = Q[F];
    F= F+1;
  }
}
```

```
isFull()
{
  if (((F==1)&&(R==N))||(F==R+1))
    return 1;
  return 0;
}.

isEmpty()
{
  if (F==0 && R==0)
    return 1;
  return 0;
}
```

Find the number of elements in a Q?

```
if ((F==1 && R==N)||(F==R+1))
    no=N;
else
    if (F<=R)
    no= R-F+1;
    else
      no= N-(F-R-1);
```

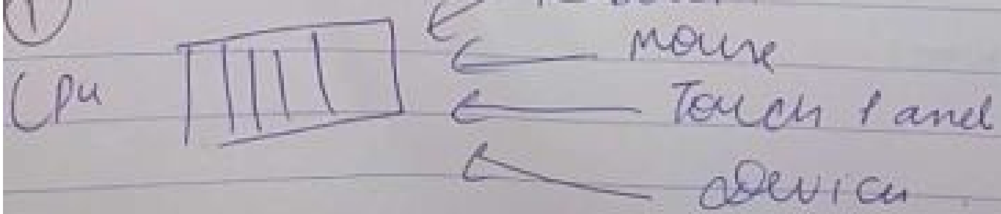
Dequeue (Deck)

Double Ended Queue

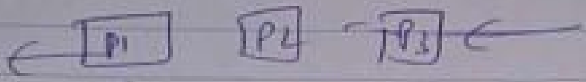
Input restricted	Output restricted
addition is from one end only (Deletion from any end).	Deletion only on one end. (addition from any end)

Applications

1.



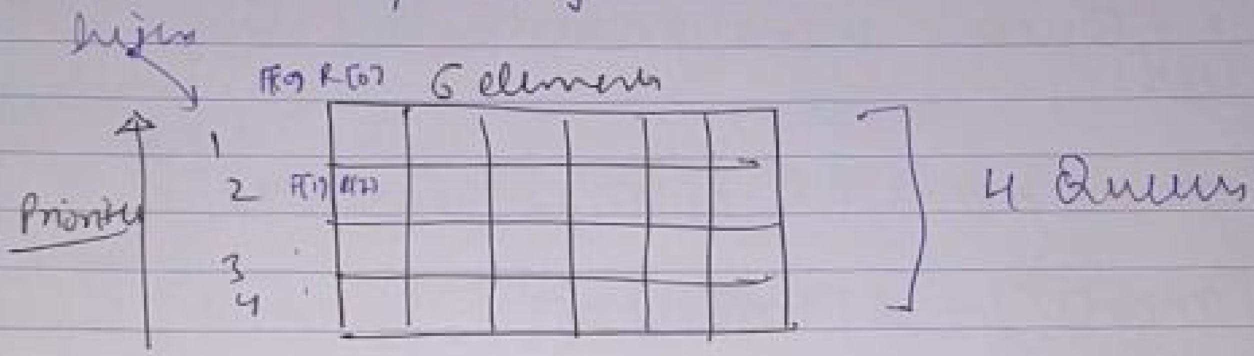
②



Programs.

Priority Queue

element → priority



F[4] R[4]

```
# define N 50
# define Priority 10.
int Q[Priority][N], F[Priority], R[Priority];

init() //initialize Queue as empty.
{
  int i;
  for (i=0; i<priority; i++)
       F[i]=R[i]=-1;
}
```

```
Add ( int priority , int item)
// add item in the Queue of priority precedence
{
  if(( F[Priority] = = 0 && R[Priority] = = N-1) ||
     ( F[Priority) = = R[Priority]+1)).
  {
    print ( Queue with priority is full);
    return ;
  }
  if (F[priority] = = -1 && R[priority] = = -1)
  {
      F[priorty] = R[Priority] = 0;
  }
else
  if (R[priorty = = N-1)
      R[Priority] = 0;
else
      R[Priority] = R[priority]+1;
 Q [priority][R[priority]] = item;
}.
```

```
Delete (int priority)
{

// delete element from the Q, having precedence
 as priority.
```

```
if ( F[priority]==-1 && R[priority==-1)
 {
  print(Queue with priority already empty);
  return;
 }
 item = F[priority];
if( F(priority] <= = R[priority])
   F[priority]= R[priority]==-1;
else
  if ( F[priority]= = N-1)
     F[priority]=0;
else
   F[priority]= F[priority]+1;
return item;
}
```

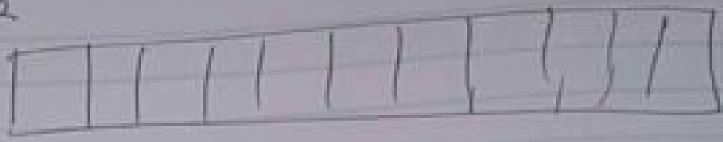
* Dequeue

- input restricted.
  addition will be on Rear
  Deletion will be on both sides.

- output restricted
  Deletion will be on ~~from~~ front.
  Addition will be on both sides.

Input Restricted

Q



int Q[N], F=-1, R=-1;

add (int item) {
// addition will be from rear //

Same as common code
}

Delete (int direction)
{
int item; [If Queue is not empty then =>]
if (direction ==0)
{
// delete from the front.

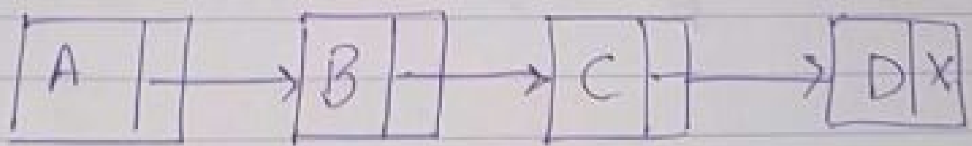
Same as common code.
}
else
if (direction ==1)
{
// delete from the rear.
item = Q[R];
if (F==R)
{
F=R=-1;
}
else

```
if (R==0)
  {
    R = N-1;
  }
else
    R=R-1;
return item;
}.
```

LINK LIST

In Arrays: Problem of insertion & deletion
(or) (storage problem)
complexity: O(n).

node:



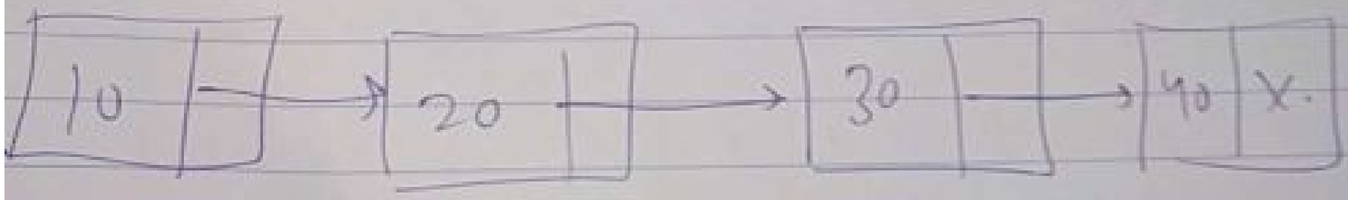
To swap two fields:

we have to take a the temp variable & swap in case of arrays. if A,B,C & nor variables but lot of information, then memory will used more. In case of link list, just swap pointers.

Stored in memory like this

	DATA	LINK
1	A	3
2	B	4
3	C	2
4	D	0

A → C → B → D



```
struct node {

  int data;
  struct node* link;

};

typedef struct node NODE;

NODE * getnode (int x)
 {

   NODE* t;
   t= (Node*) malloc ( size of(NODE));
    t -> data = x;
    t -> link = NULL;
   return t;

 }.

NODE * Create ()
 {
    NODE * head = NULL, *t;
    int n, item, i;
    printf(" How many nodes:");
    scanf ("%d", &n);
  for (i=0; i<n; i++)
   { printf (" Enter node data:");
      scanf ("%d", & item);
     if (i==0) {
     head = t = getnode (item);
      continue;
     }
```

```
t -> link = getnode (item) ;
 t = t -> link ;
}
return head ;
}.

Show (NODE * head)
{
  while (head != NULL)
  {
    printf ("%d" head -> data) ;
    head = head -> link ;
  }
}.
```

Reverse a singly linked list.

```
NODE* reverse (NODE * head) {
NODE* P = head , q = NULL , r = NULL ;
 while ( p != NULL)
 {
   q = p ;
   p = p -> link ;
   q -> link = r ;
   r = q ;
 }
 head = q ; .
 return q ;
}.
```

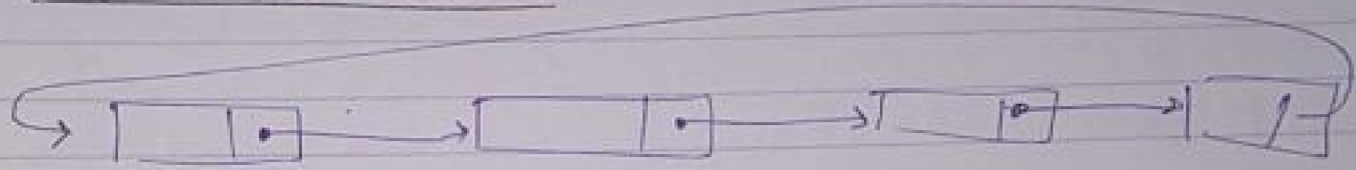
Concatenate (x, y, z) { // attach X and Y.

// X = x1, x2, x3 - - - - xn)   Y = (y1, y2, y3 - - ym)
// Program makes Z = (x1 x2 x3 - - xn, y1, y2, - - ym)

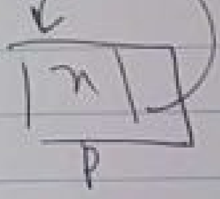
```
Z = X;

if ( Y == NULL)
  return;

if ( X == NULL)
  {
   Z = Y;
   return;
  }
P = X;
while ( P -> link != NULL)
      P = P -> link;
 P -> link = Y;
 return;
}
```

Scanned by CamScanner

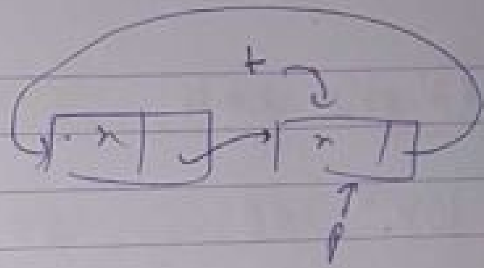
Circular linked list

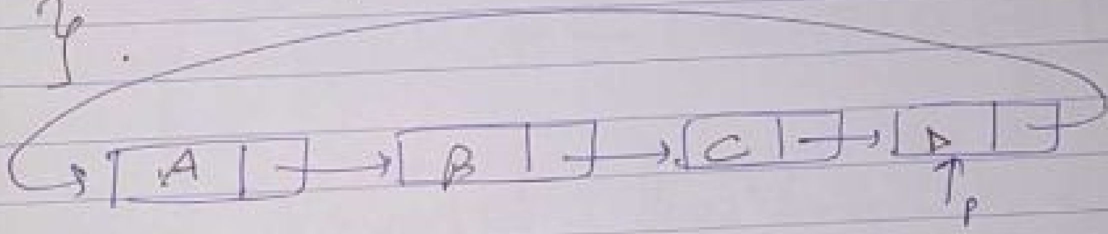


```
NODE *getnode(int x)
{
    NODE *t;
    t= (NODE*) malloc (sizeof (NODE)) .
    t->data =x;
    return t;
}

NODE * create(){

 NODE *p,*t;
 int n, i, x;
 printf("how many nodes:");
 scanf ("%d", &n);
 if (n>0)
 {
   printf("Enter node value:");
   scanf ("%d", &n);
    p= getnode (n); p->link =p;
   for( i=2; i<=n; i++) .
    {
      printf ("Enter the node value:");
      scanf ("%d", x);
```



```
t = getnode(x);
t -> link = p -> link;
p -> link = t;
p = t;
}
return p;
}.
```



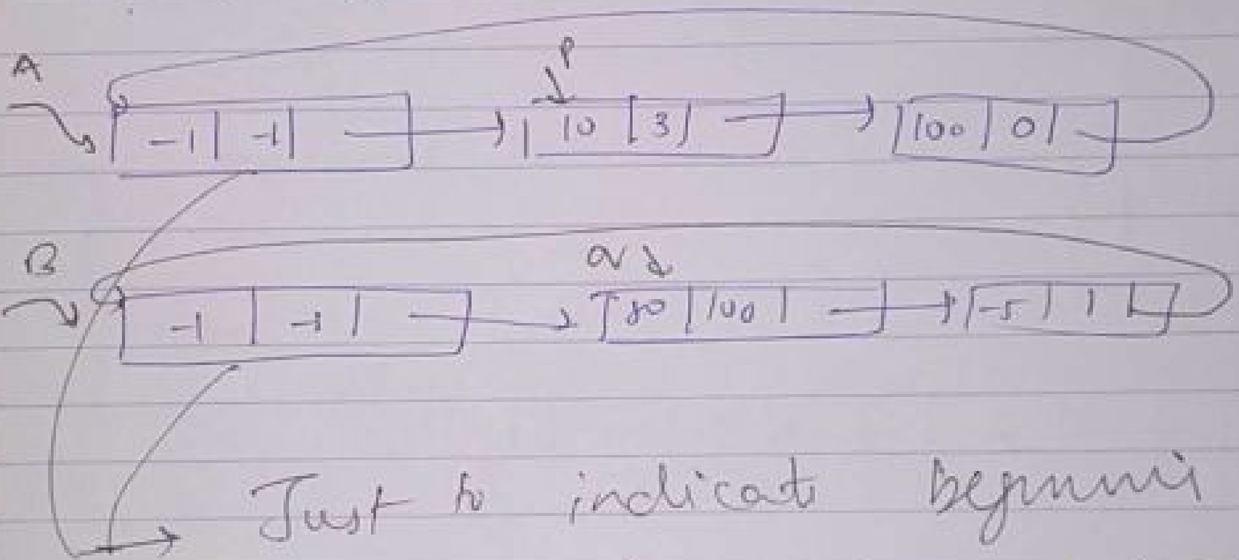



```
Show ( NODE *p)
{
NODE * head, *t;   if (p==NULL) return;
t = head = p -> link;
do
{
printf("%d", t->data);
t = t -> link;
}
while (t != head);
}.
```

Scanned by CamScanner

Polynomials.

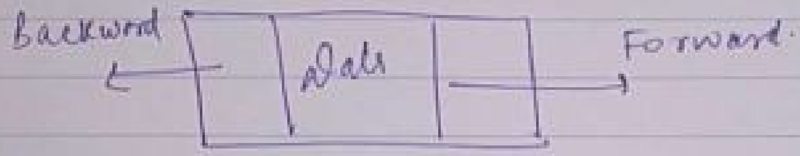
$10x^3+100$, $50x^{100}-5x$



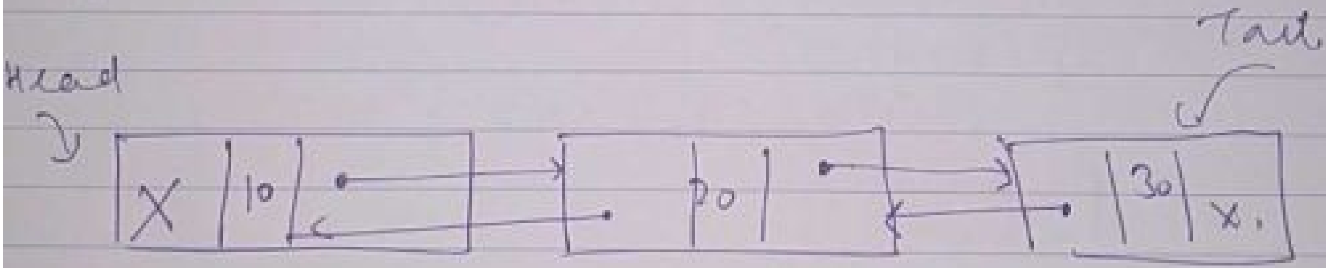

```
NODE *d;  // Points to last node in Z //.
attach (int c, int e)
{
    t = getnode();
    t -> coef = c;
    t -> exp = e;
    d -> link = t;
    d = t;
}.

add (x, y, Z)
{
    Z = getnode(-1,-1);
```

```
add (x, y, z)
{

  z= getnode();
   z->coeff = z->exp = -1
   d=z;
  p= x->link;
  q = y->link;
  while ((p->exp != -1)||(q->exp != -1))
   {
     if ( p->exp= q->exp)
      {
        m = p->coeff + q->coeff;
        if (m != 0)
          attach ( m, p->exp);
        p= p->link;
        q= q->link;
      }
    else if ( p->exp > q->exp)
       {
          attach ( p->coeff, p->exp);
         p=p->link;
       }
    else
       { attach (q-> coeff, q->exp);
         q=q->link;
       }
  } d->link=z; }
```

Doubly linked list.



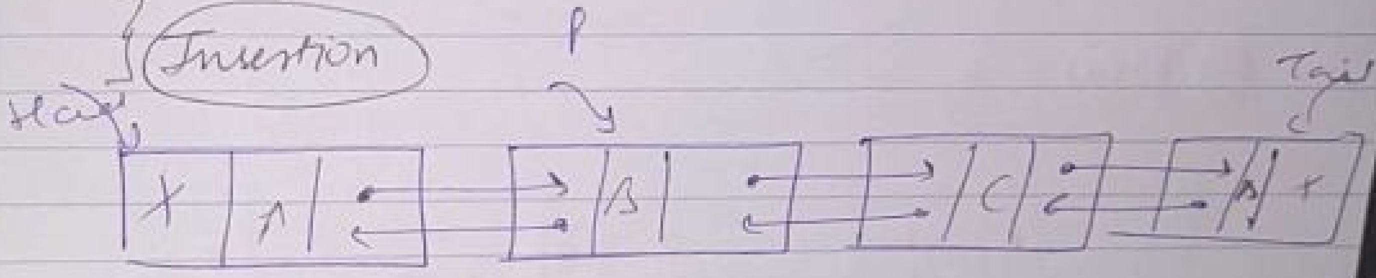

```
struct node
{
  int data;
  Struct node * forw;
  Struct node * back;
};
```

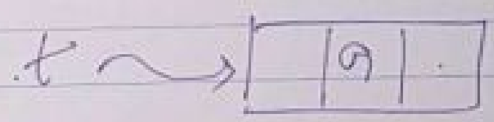



```
Show_forward ( NODE * head)
{
  NODE * p = head;
  while ( p != NULL)
  {
    printf (" %d", p→ data );
    p = p → forw;
  }
}
```

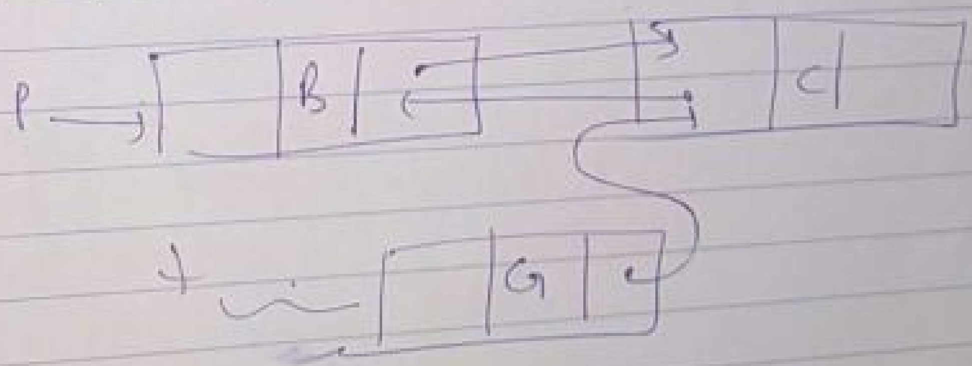
```
show_backward (NODE * tail)
{
    NODE * p = tail;
    while (p != NULL)
    {
        printf("%d", p->data);
        p = p->back;
    }
}
```

Insertion

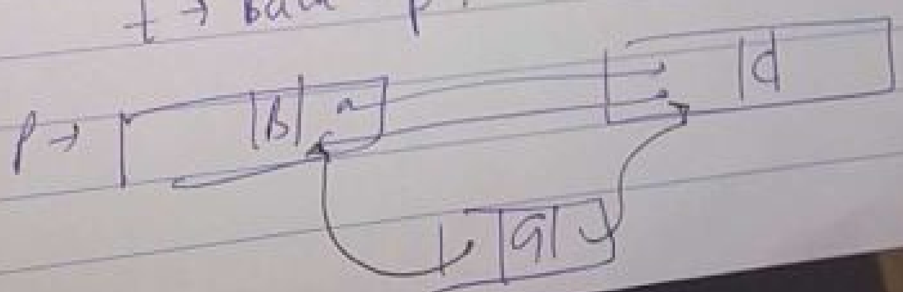




$t \to forw = p \to forw$

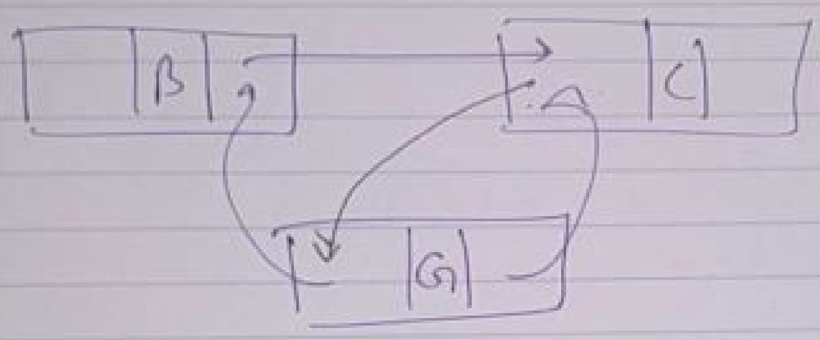


$t \to back = p$.

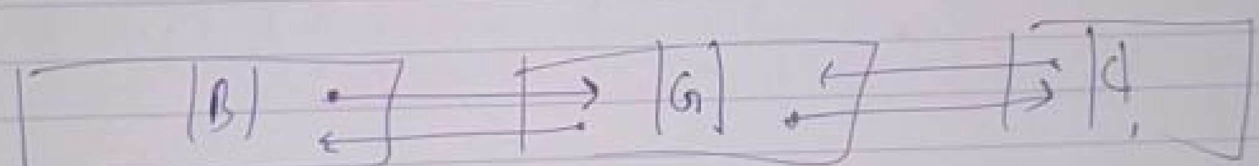


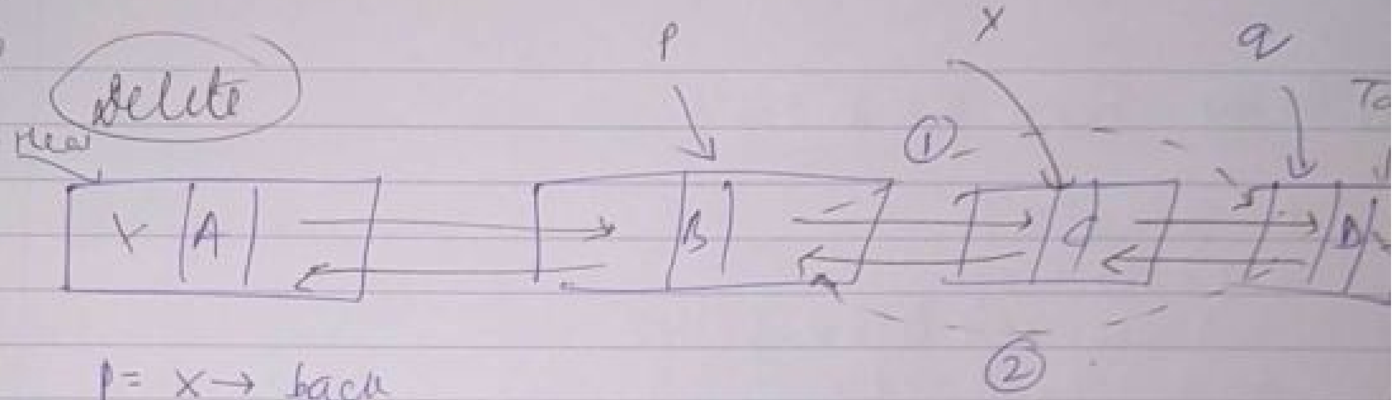
Scanned by CamScanner

p→ forw → back = t



p→ forw = t





p = x → back
q = x → forw

~~p → forward~~ ① p → forw = q

② q → back = p

free(x).

1) Delete head node

t = head;
head = head → forw;
head → back = NULL;
free(t);

2) Delete tail node

t = tail;
tail = tail → back;
tail → forw = NULL;
free(t);

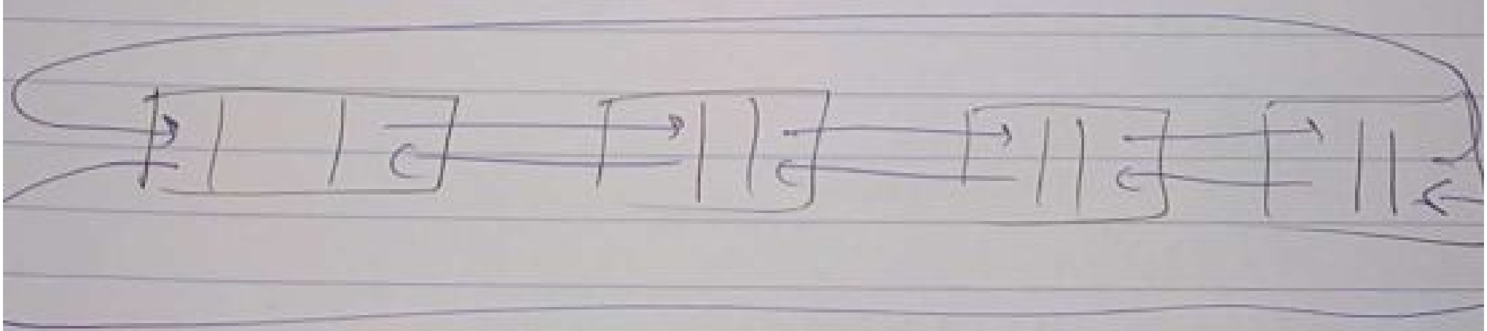
3) Insert before head node

t = getnode();
t → data = x;
t → forward = head;
head → back = t;
t → back = NULL;
head = t;

4) Insert after tail node

t = getnode();
t → data = x;
t → forw = NULL;
t → back = tail;
tail → forw = t;
tail = t;

Circular doubly link list



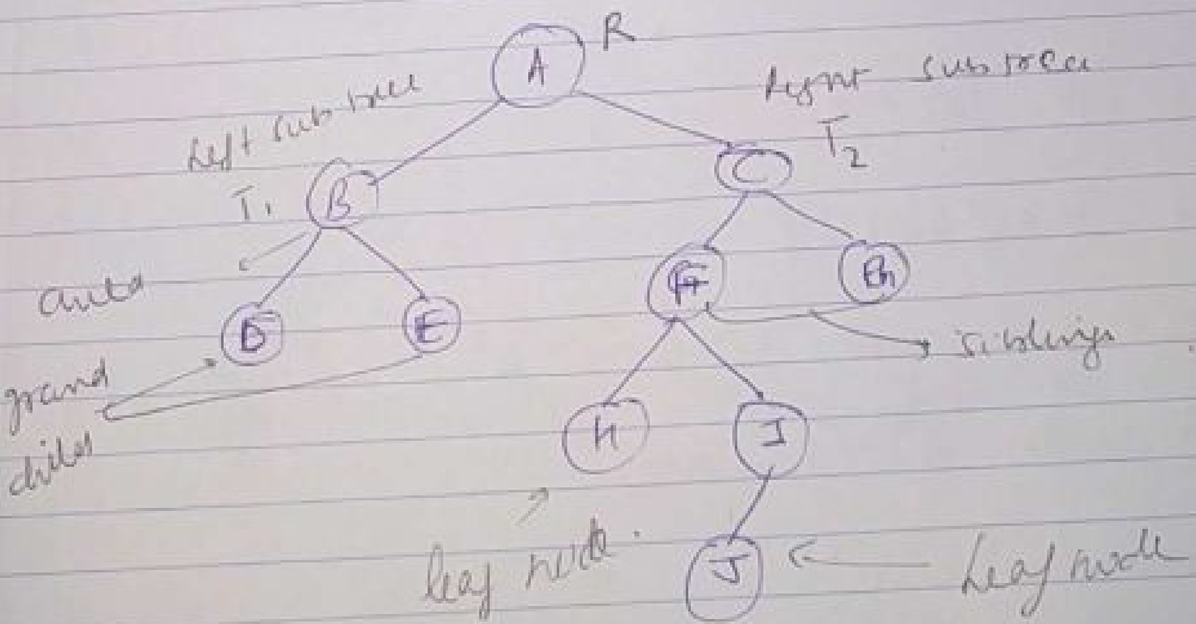
TREE

~~Binary Tree~~

$\boxed{\text{BINARY TREE}}$ T is defined to be a finite ordered set of elements (nodes) such that

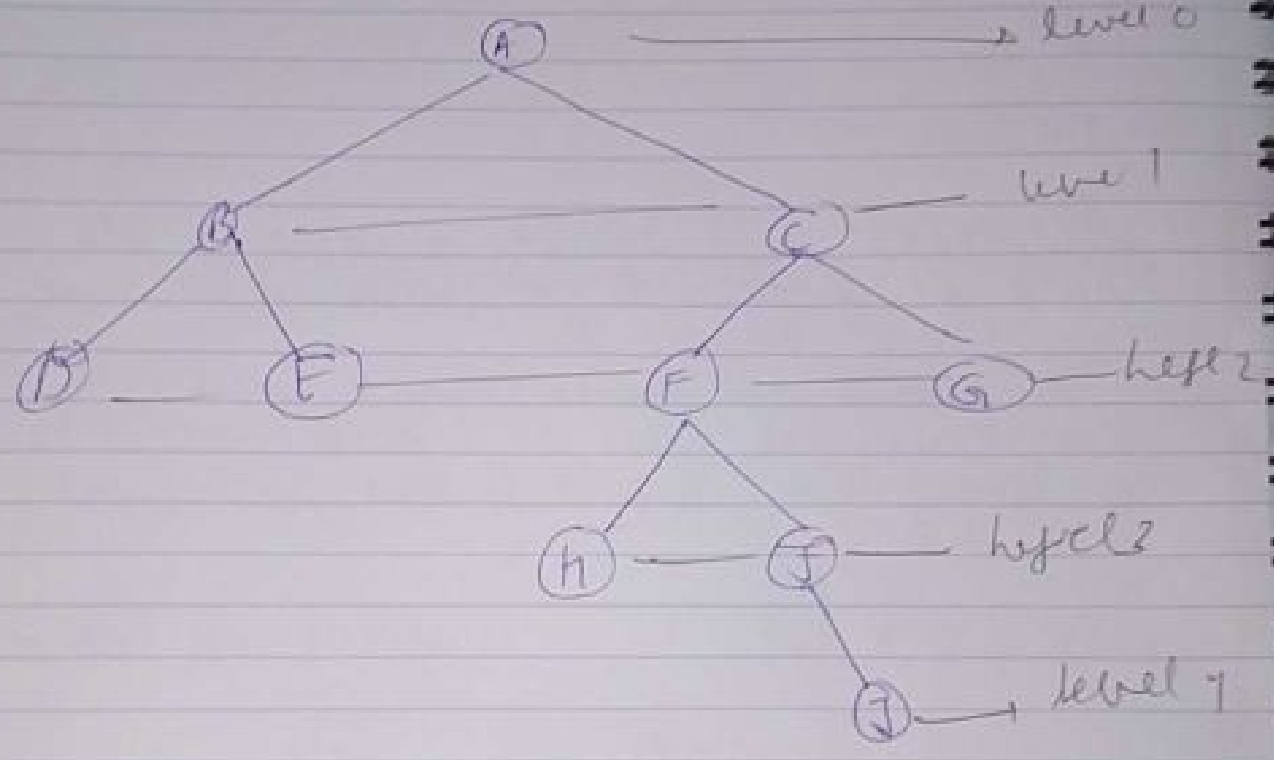
(i) T is either empty or
(ii) T contains a distinguish node R, called Root of T and remaining nodes of T form ordered pair of binary trees T_1 and T_2 such that $T_1 \cap T_2 \neq \phi$.

$T = \{T_1, R, T_2\}$ (recursive defination.



~~Character~~

If Nodes have same parent they are called Siblings
path is sequence of nodes
A node is a leaf node if it has no children

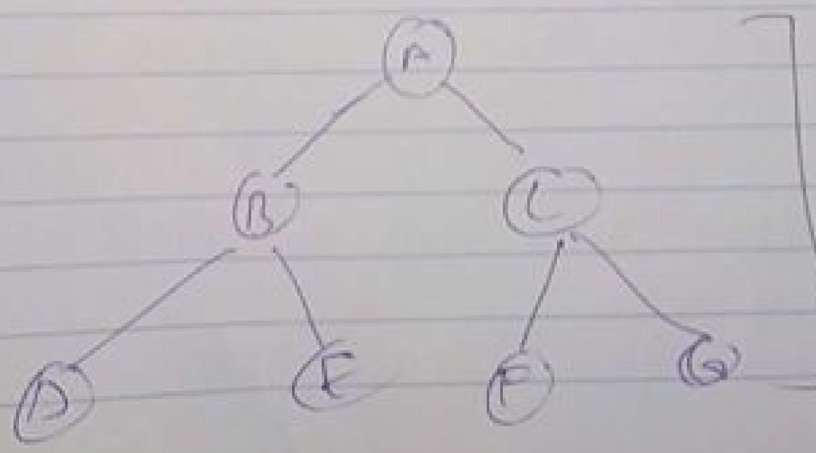


Height of Tree = 4.

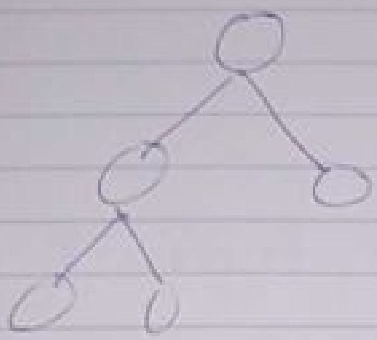
Height: max{level i}.

on level i, max nodes → 2^i $\quad i \geq 0$

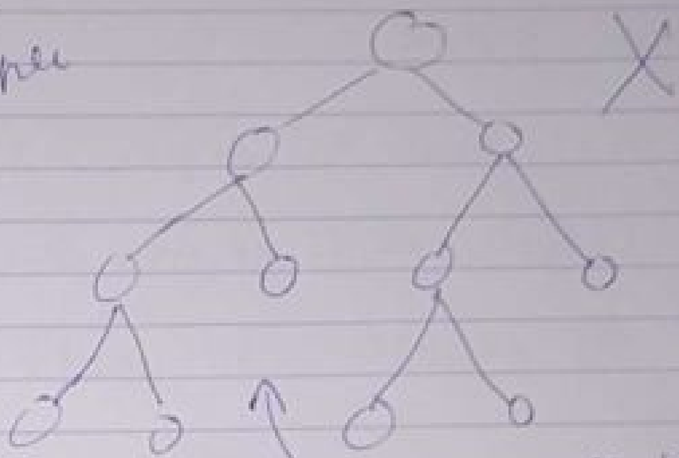
Total nodes: $2^{h+1} - 1$ (max)



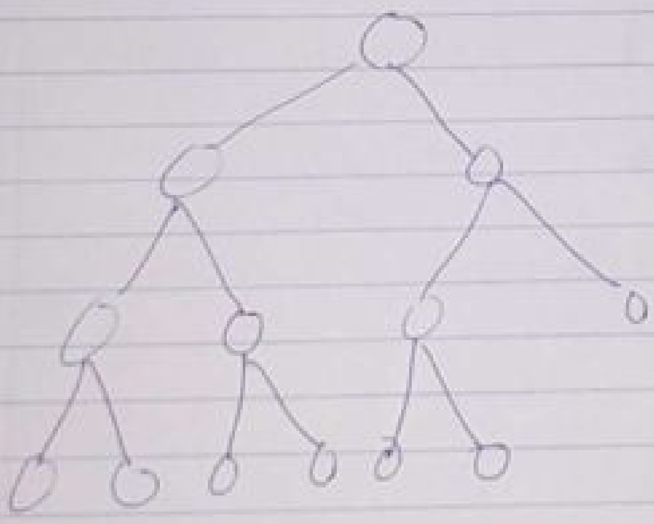
Full binary tree.
Every parent have 2 children, every leaf node should be on same level



✓ complete



There should be no gap

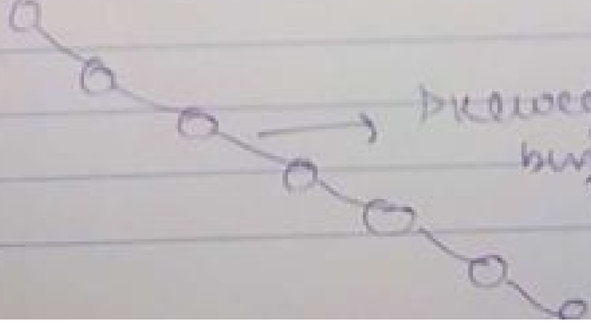


✓

{Complete Binary Tree will be used in Heaps}
sorting

If tree is full, height = $h = \log_2(n+1)$

↑ no of ~~trees~~ node



→ Skewed binary Tree

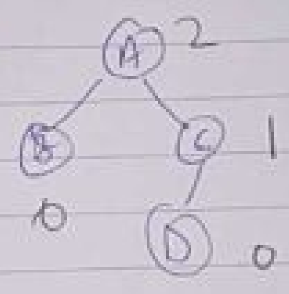
formula not applicable on this

Degree of node

leaf node ; degree = 0

exactly one child from any node, degree = 1.

If exactly 2 children, degree = 2.

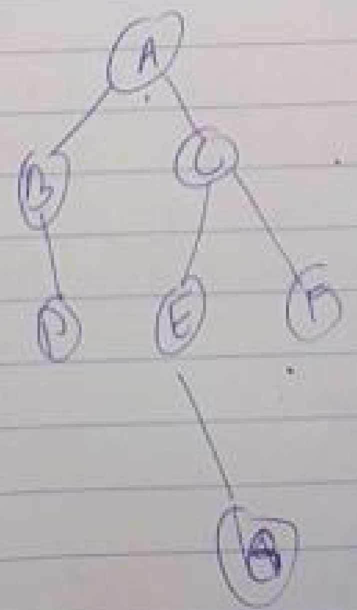


$n_0 \rightarrow$ no of nodes with degree 0

$n_1 \rightarrow$ no of nodes with degree 1

$n_2 \rightarrow$ no of nodes with degree 2

$$n = n_0 + n_1 + n_2$$

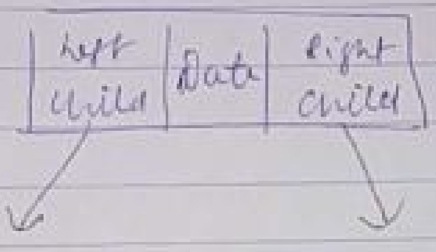


no of ~~branches~~ elements $\rightarrow 2n_2 + n_1 + 1$ for root node

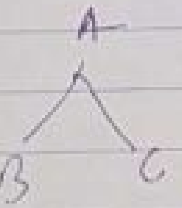
$n = 2n_2 + n_1 + 1$

$n_0 + n_1 + n_2 = 2n_2 + n_1 + 1$

$n_0 = n_2 + 1$

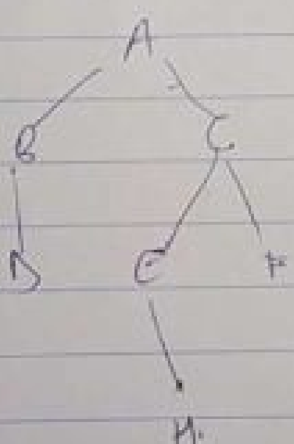



```
struct treenode {
 struct treenode *left;
       int data;
 struct treenode *right;
};
```



A(B,C)

left child   right child.



$\underbrace{A\left(B(,D),\ C\left(E(,H),\ F\right)\right)}$

Our input

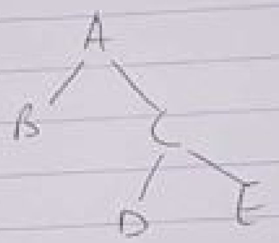
& program will

generate tree

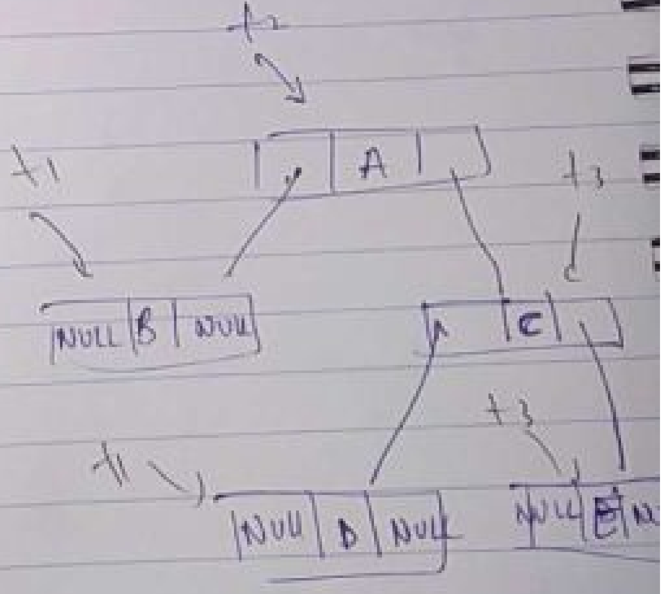
NODE * getnode()

```
NODE * getnode (char x)
{
  NODE * t
  t = (NODE *) malloc (size of (NODE));
  t -> left = t -> right = NULL; t -> data = x;
  return t;
}
```

To Create



```
t1 = getnode('D');
t2 = getnode('E');
t3 = getnode('C');
t3 -> left = t1;
t3 -> right = t2;
t1 = getnode('B');
t2 = getnode('A');
t2 -> left = t1;
t2 -> right = t3;
return t2
```

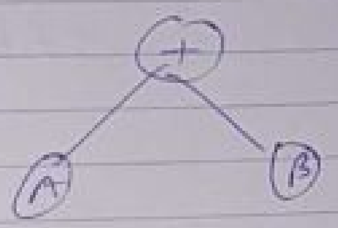


Attach can only be at end.

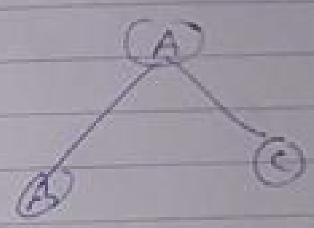
CEC02: DATA STRUCTURES

Traversal of Binary Tree

(1) Inorder: First to visit left child, visit Root, visit right child.

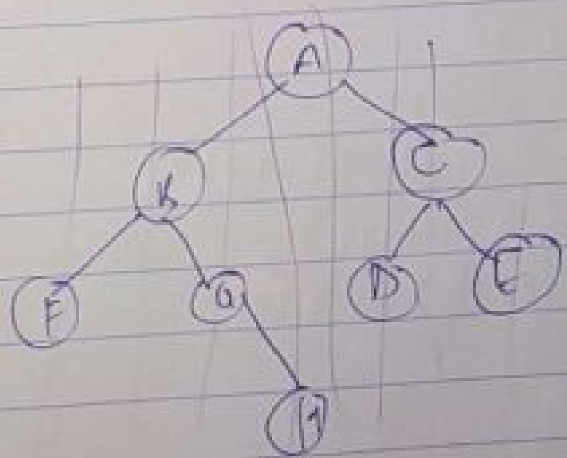


A+B



BAC

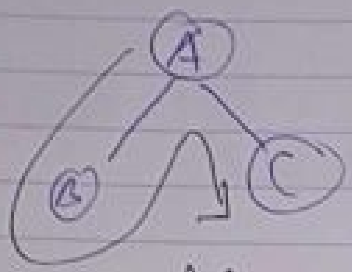
```
inorder(NODE * root)
{
   if (root != NULL)
   {
      inorder(root -> left);
      printf("%d", root -> data);
      inorder(root -> right);
   }
}.
```



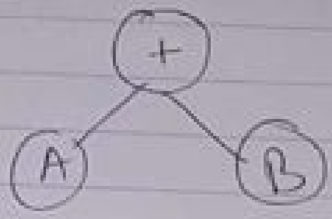
FFGHADCE

FBGHADCE

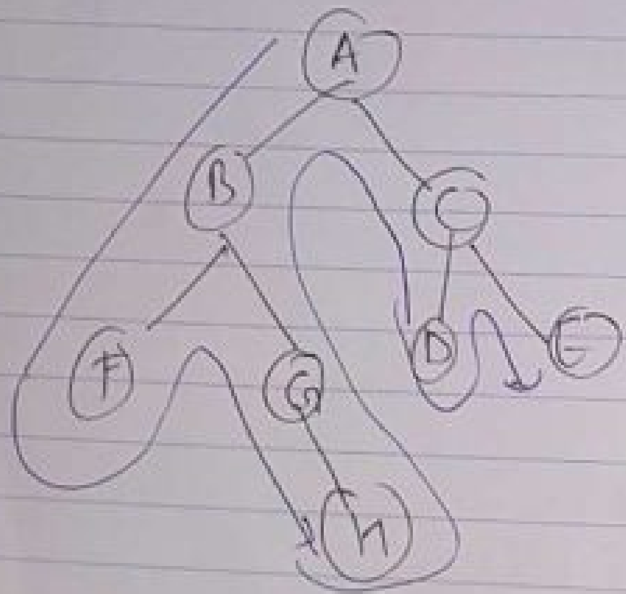
② PREORDER → visit the root then nodes on the left sub root, right. (To generate index of books)



ABC



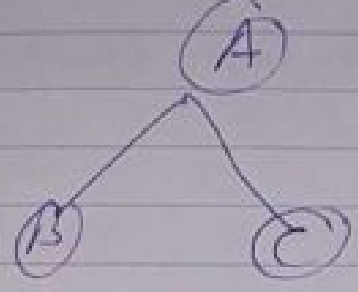
+AB

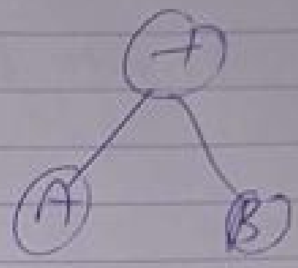


ABFGHCDE

```
preorder (NODE *root)
{
  if (root != NULL)
  {
    printf("%d", root->data);
    preorder(root->left);
    preorder(root->right);
  }
}
```

(3) Post order: Visit the left subtree, visit right subtree, visit root.



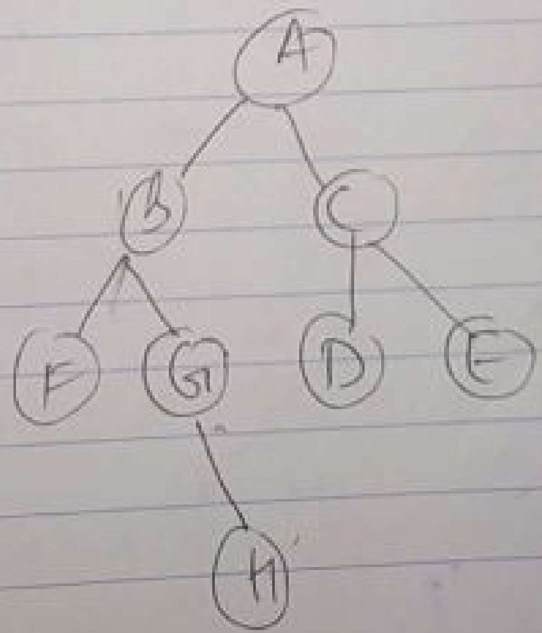


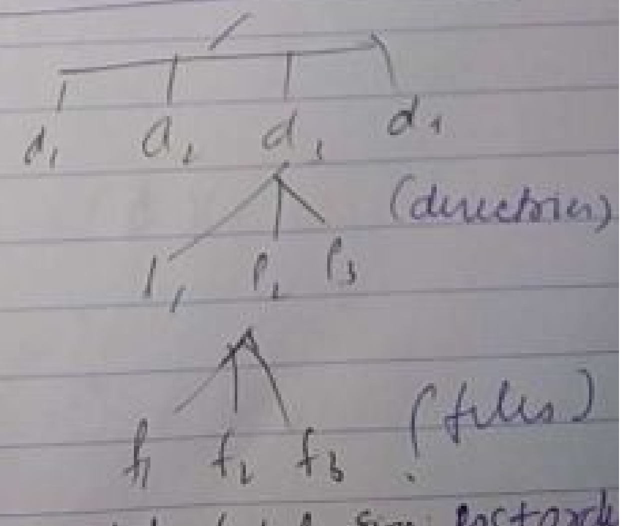
BCA

AB+.

```
Postorder ( NODE * root)
{
  if ( root != NULL)
  {
    Postorder (root → left);
    Postorder( root → right);
    printf ("%d", root → data);
  }
}
```

FHGBDECA





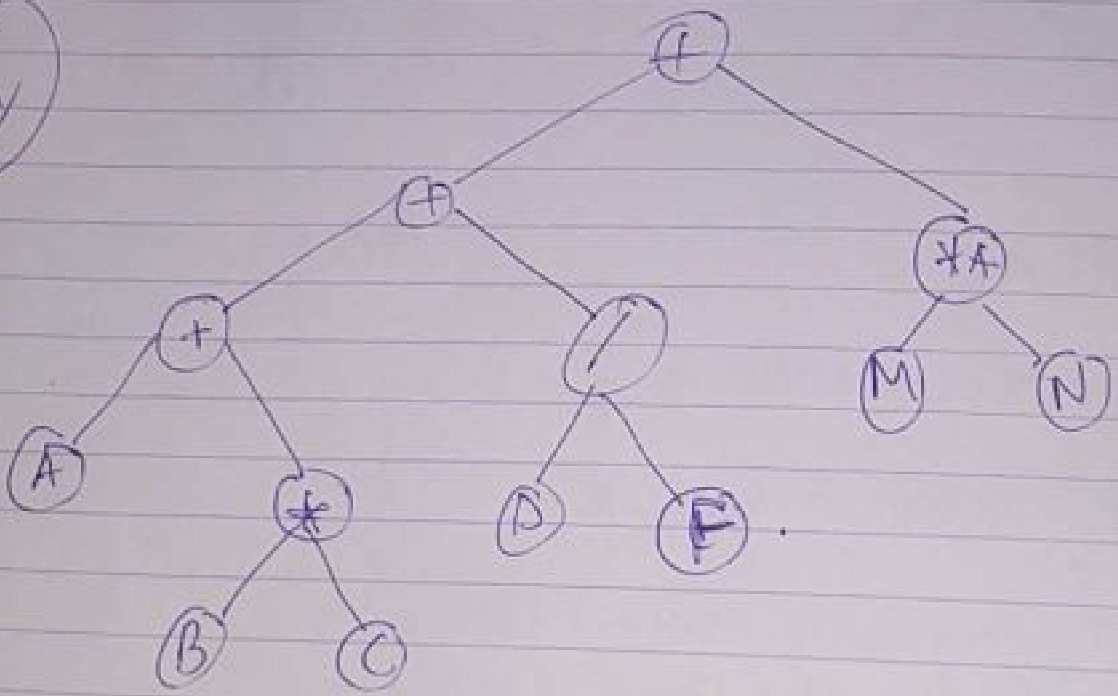
To calculate total size: Postorder

Expression Tree

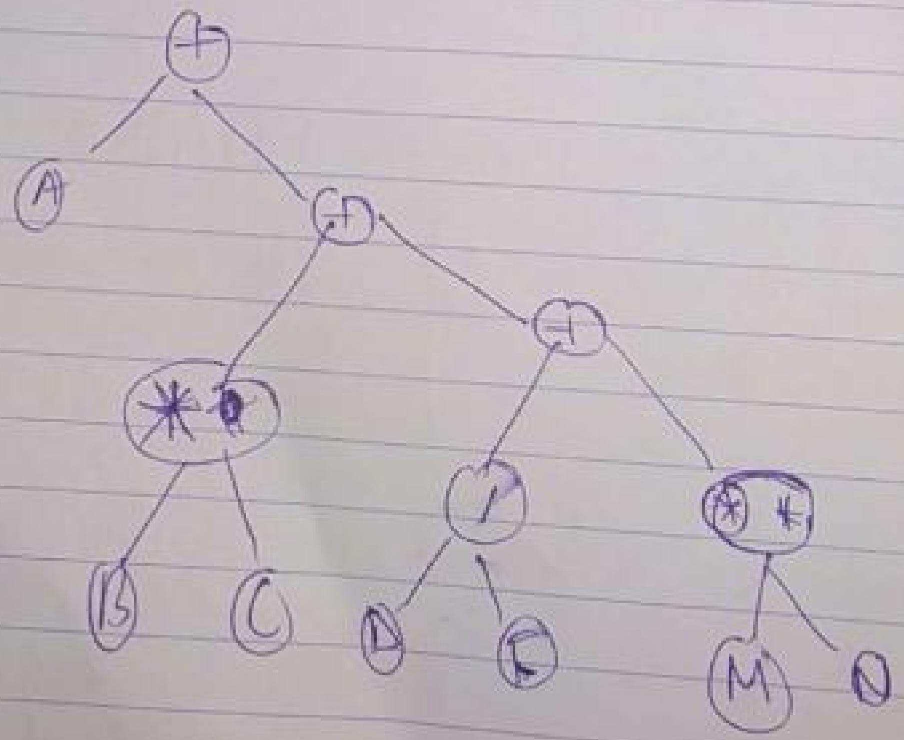
A + B * C + D / F + M^N (M ** N).

highest priority operator → lowest level of tree

(i)



or (ii)



① in order

A + B * C + D/E + M ** N } same

① In order

~~A (B)~~ A + B * C + D/F + M ** N

② prefix

+ + + A * BC / DF ** MN

post fix

ABC * + DF / + MN ** +

①

prefix.

+ ~~A + * BC~~ + A + * BC + / DF * + MN

post

~~A B C * + D F / + M N * + +~~

A BC + DF / M N * * + + +

Iterative versions of Traversals

INORDER

1) Start from the root node, pushing each node over stack on its left path.
2) Pop the node from the stack, process this node if NULL then stop the process. Otherwise set pointer to the right child and repeat the procedure from ①.

```
# define N 100
int top = -1;
NODE * stack [N];

Push (NODE *p)
{
   if (top==N-1)
   {
     printf("stack is full\n");
     return;
   }
   stack [++top]=p;
}

NODE* Pop( )
{
  if (top==-1)
   { printf ("stack is empty\n");
     return NULL;
   }
  return stack[top--];
}
```

PREORDER

```
PreOrder ( NODE * root )
{
   NODE * p=root;
 while(1)
  {
    while ( p!=NULL)
     {
        printf ("%d", p->data);
        if (p-> right != NULL)
          push(p->right);
       p=p->left;
     }
   p= pop();
    if (p==NULL)
      return;
  }
}
```

INORDER

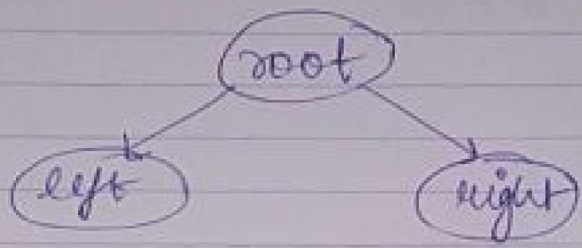
```
inorder ( NODE * root )
{
   NODE  *p = root;
  while(1)
   {
     while ( P != NULL)
       {
          Push (P);
          P = p -> left;
       }
     if ((P = POP()) == NULL)
          return;
     printf ("%d", p->data);
      p = p-> right;
   }
}
```

Post Order

```
Post Order (NODE *root)
{
  NODE *p= root;
   while (1)
   {
     while (p!=NULL)
      {
        Push (P);
        if (p->right != NULL)
          {  push (-p->right);
          }
        p= p-> left ;
     }
    If ((p= POP())==NULL)
     return;
    while ( p is positive)
    {
      print ("%d", p->data);
     If((p= pop ())==NULL)  return;
    }
     p= -p;
  }
}.
```

Scanned by CamScanner

Binary Search Tree

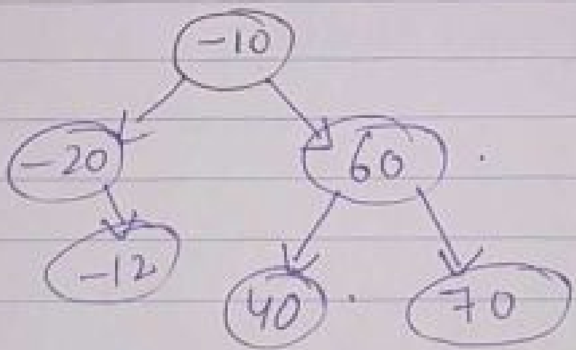


{all elements are unique}

left < root < right

any binary tree where these properties hold tree is binary search tree.

-10, -20, 60, 40, 70, -12 } Create BST



take first value as root, then take next no if it is > root attach to right, if < root attach to left.
and so on for every value

```
NODE *
search ( int x, NODE * root)
{
    NODE *p = root;
    while ( p != NULL )
    {
        if (x == p->data)
            return p;
        else if (x < p->data)
            p = p-> left;
        else
            p = p-> right;
    }
    return NULL;
}
```

returns NULL if x is not in BST else it returns pointer to the node containing x

If we produce inorder traversal we get ascending order of ~~no~~ numbers.

-20, -12, -10, 40, 60, 70.

for descending:

```
inorder (NODE * root)
{
    if (root){
        inorder (root -> right);
        printf ("%d", root->data);
        inorder ( root -> left);
    }
}
```

```
NODE * smallest (NODE *p)
{
  while (p→left != NULL)
    p = p→left;
  return p;
}.

NODE* largest (NODE *p)
{
   while (p→right != NULL)
    p = p→right;
   return p;
}
```

Scanned by CamScanner

```
NODE * createBST (NODE * root, int x)
{
    NODE *t
    if (root==NULL)
    {
        t=getnode();
        t->data=x;
        t->left=t->right=NULL;
        return t;
    }
  else if (x<root->data)
        root->left=CreateBST(root->left,x);
  else
      root->left = CreateBST(root->left,x);
  else
      root->right= Create BST(root->right,x);
 return root;
}
        NODE *root=NULL;
for (i=0; i<n; i++)
 {scanf("%d",&x);
  root = CreateBST(root,x);
 }.
```

for calling the func.

count no of leaf Nodes

```
int countleaf(NODE *p)
{
    if (p==NULL)
        return 0;
else if (p->left==NULL && p->right==NULL)
        return 1;
    return (countleaf( p->left )+
                countleaf ( p->right) );
}
```

all nodes except leaf nodes are called interior nodes.

```
int interior(NODE *p)
{
  if (p==NULL)
     return 0;
  else if (p->left != NULL || p->right != NULL)
    return interior (p->left) + interior(p->right)+1;
  else return 0;
}
```

↳ when at leaf Node

↑ for root

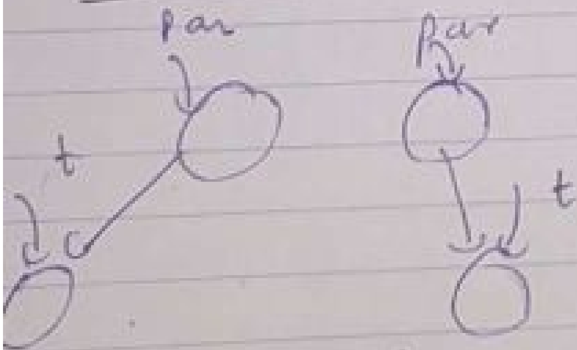
```
height(NODE *p)
{
  if (p==NULL)
     return 0;
  else
   return(max(height(p->left), height(p->right)+1
}
```

insert (int x, NODE **p) (in BST).

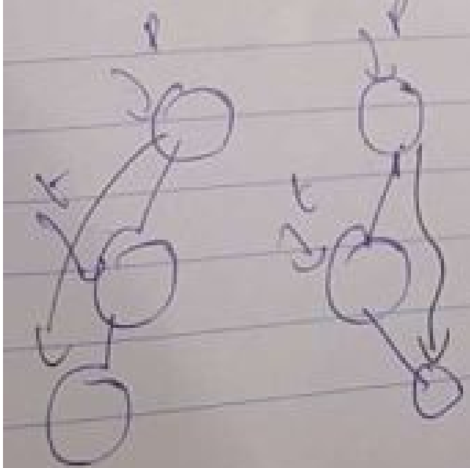
```
{
   NODE *t, *par.
   if (*p==NULL)
   {
      *p = getnode();
      *p->data = x;
     *p->left = *p->right = NULL;
     return;
   }
```

```
t = *root ; par = NULL ;
while ( t != NULL )
{
   par = t;
 if ( t->data == x )
 {
   printf ("%d already in BST\n", x);
   return;
 }
 else if ( t->data > x )
        t = t->left ;
 else
      t = t->right ;
}
```

Delete



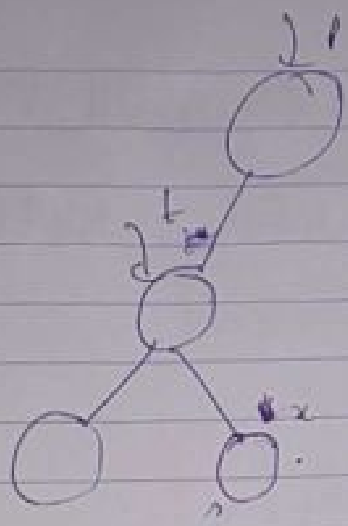
① When t is a leaf node.



② when t has one child / subtree → left / right

p->left = t->left ;

p->left = t->right ;



3) when t has both children.

```
void delete (NODE **p , int num)
{
    int found;
    Node * parent, *x, *xsucc;
    if (*p==NULL)
    {
        cout<<"\n Tree is empty";
        return;
    }
    parent = x = NULL;
    search ( p, num, &parent, &x , &found);

    if (found==0)
    {
        cout<<" Not found" ;
        return;
    }
```

```
// if the node to be deleted has two children
if (x→leftchild != NULL && x→rightchild != NULL)
{
  parent = x;
  xsucc = x→rightchild;
  while (xsucc→ leftchild != NULL)
      { parent = xsucc;
        xsucc = xsucc → leftchild;
      }
    x→data = xsucc→data
    x = xsucc;
}

// if node to be deleted has no child

if (x→ leftchild == NULL && x→ rightchild == NULL)
{
   if (parent → right_child == x)
       parent → rightchild = NULL;
   else
       parent → leftchild = NULL;

  delete x;
  return;
}.
```

// if the node to be delled has only
right child.

```
if (x→ left child == NULL && x→ rightchild != NULL)
{
    if (parent → leftchild == x)
        parent → leftchild = x → right child;
    else
        parent → right child = x → right child;
    delete x;
    return;
}

// if the node to be delete has only left child

if (x→ leftchild != NULL && x→ rightchild == NULL)
{
    if (parent → leftchild == x)
        parent → leftchild = x → leftchild;
    else
        parent → rightchild = x → left child,
    delete x;
    return;
}
}
```

To search for node.
// returns the address of the node to be deleted,
// address of its parent and whether the node
// is found or not.

```
void search( node **p ,int num, node **par,
                node **x, int *found)
{
    node *q;
    q = *p;

    *found = 0;
    *par = NULL;

    while ( q != NULL)
    {
        // is the node to be deleted is found

        if (q->data == num)
        {
            *found = 1
            *x = q
        }   return;

        *par = q;
        if ( q->data > num)
            q = q-> leftchild;
        else
            q = q-> rightchild;
    }
}
```