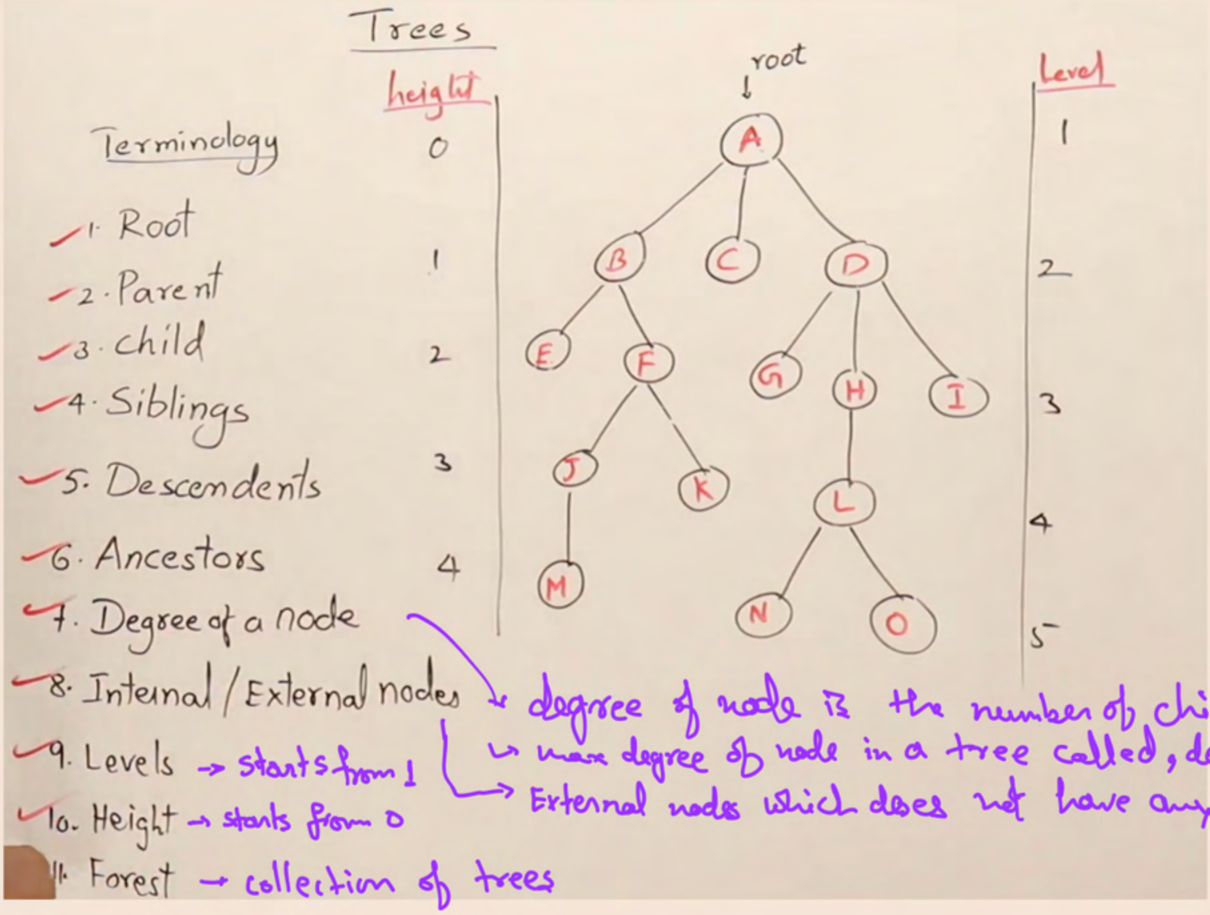
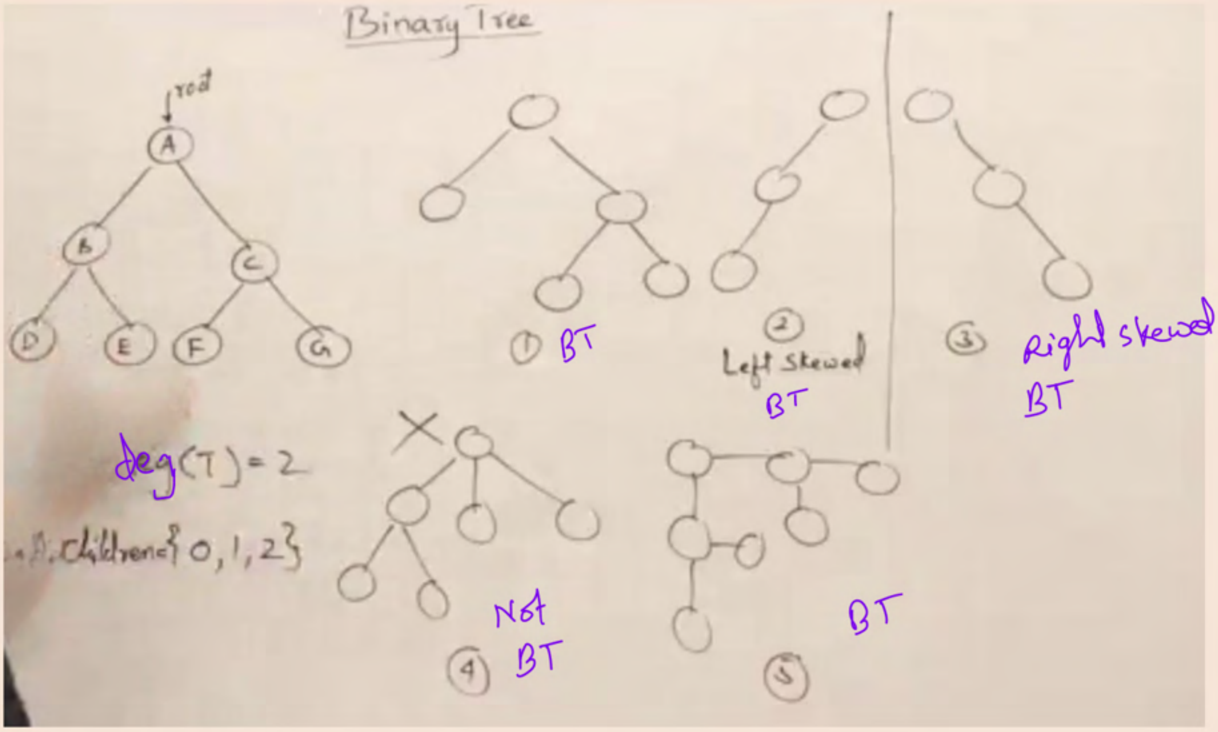


Trees

Terminology

1. Root
2. Parent
3. child
4. Siblings
5. Descendents
6. Ancestors
7. Degree of a node → degree of node is the number of child it is having
   ↳ max degree of node in a tree called, degree of tree
8. Internal / External nodes → External nodes which does not have any child node (eg. leaf node)
9. Levels → starts from 1
10. Height → starts from 0
11. Forest → collection of trees

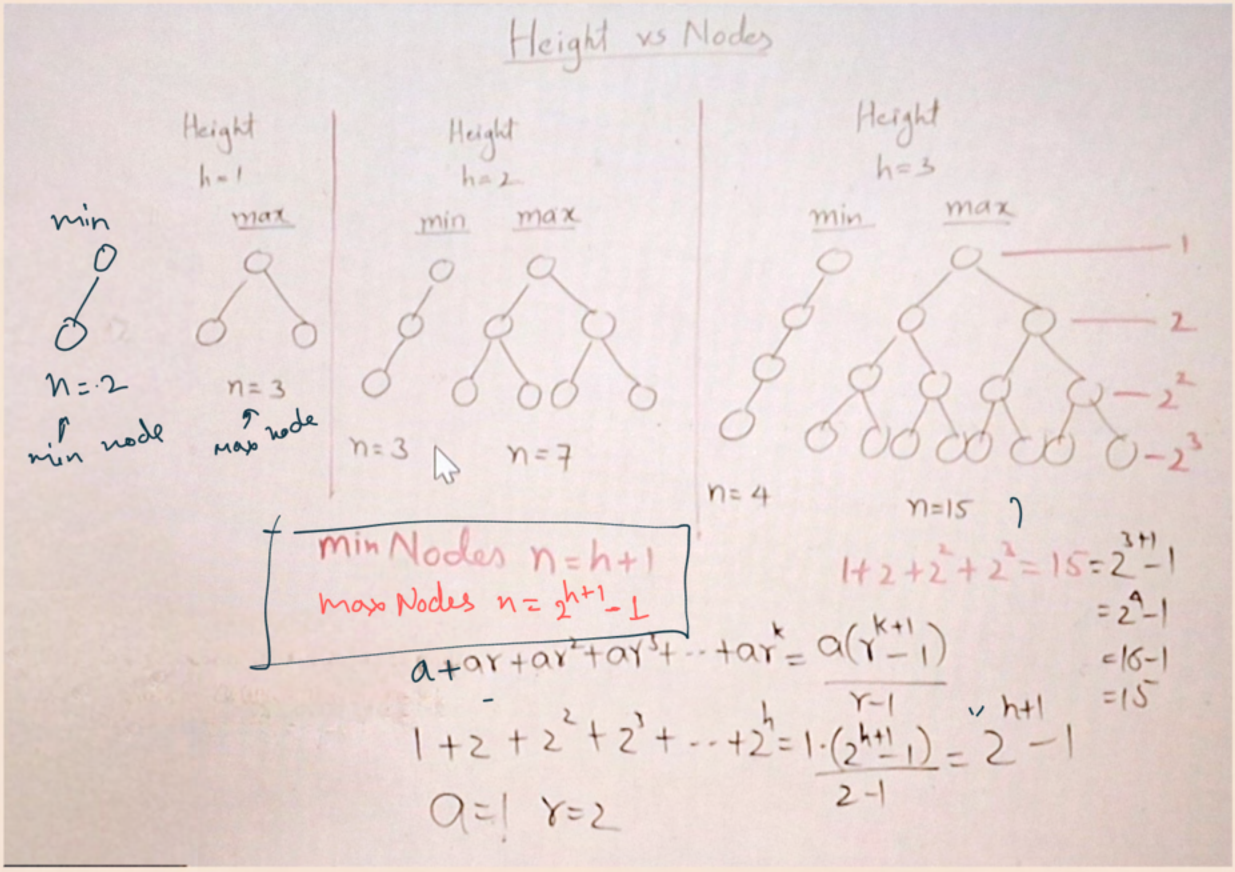


Binary Tree

deg(T) = 2

No. children = {0, 1, 2}

Height vs Nodes :



Min Nodes $n=h+1$

Max Nodes $n=2^{h+1}-1$

$1+2+2^2+2^3=15=2^{3+1}-1$

$=2^4-1$

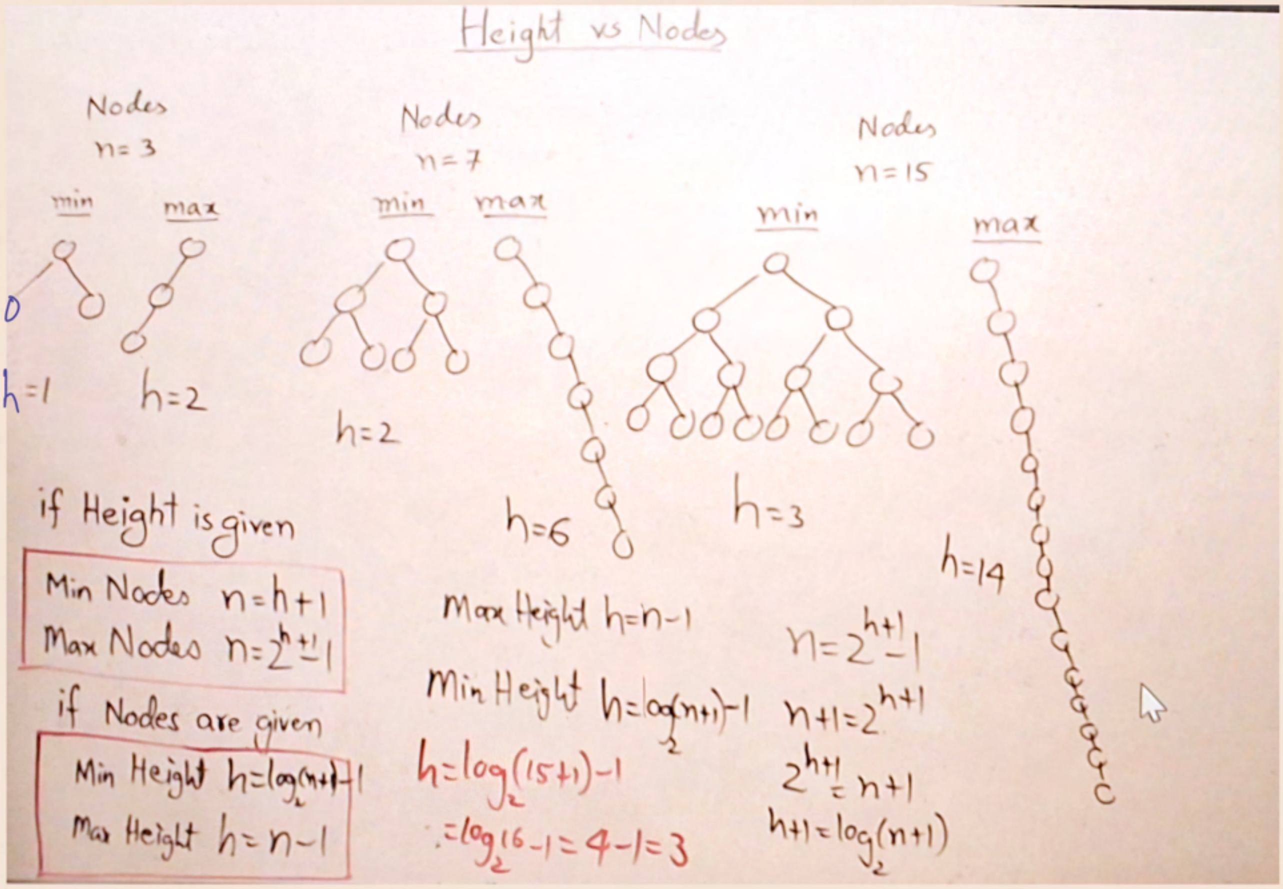
$=16-1$

$=15$

$$a+ar+ar^2+ar^3+\dots+ar^k=\frac{a(r^{k+1}-1)}{r-1}$$

$$1+2+2^2+2^3+\dots+2^h=1\cdot\frac{(2^{h+1}-1)}{2-1}=2^{h+1}-1$$

$a=1 \quad r=2$



if Height is given

Min Nodes $n=h+1$

Max Nodes $n=2^{h+1}-1$

if Nodes are given

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

Max Height $h=n-1$

Max Height $h=n-1$

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

$h=\log_2(15+1)-1$

$=\log_2 16-1=4-1=3$

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

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

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

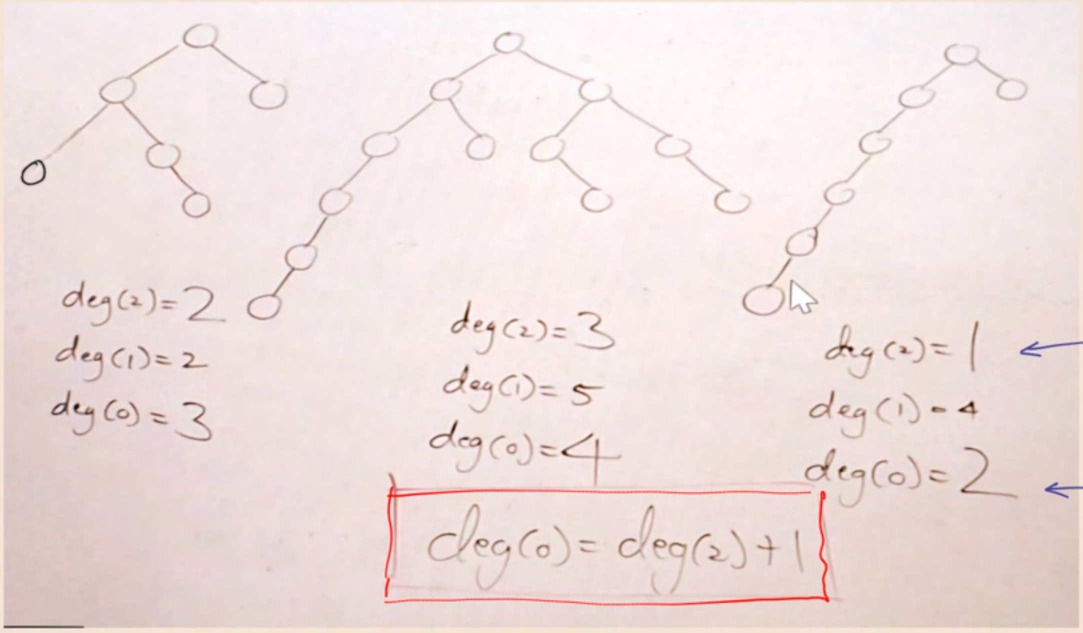
$h+1=\log_2(n+1)$

Height of binary tree :

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

Number of Nodes in a binary tree :

$$h+1\le n\le 2^{h+1}-1$$



deg(2) = 2
deg(1) = 2
deg(0) = 3

deg(2) = 3
deg(1) = 5
deg(0) = 4

deg(2) = 1
deg(1) = 4
deg(0) = 2

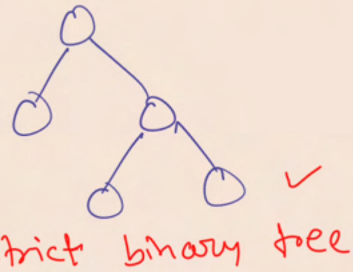
$$\boxed{deg(0) = deg(2) + 1}$$

↙ degree is How many Nodes are connected to a node

← Internal Node

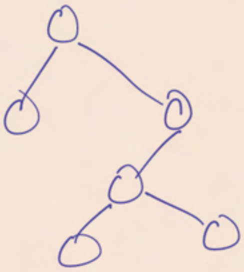
← external Node

⇒ Strict binary tree : A tree which has degree 2 & deg 0 Nodes only called as strict binary tree.



✓

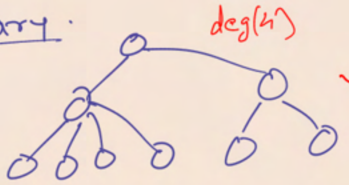
Strict binary tree



✗ Not a strict binary tree

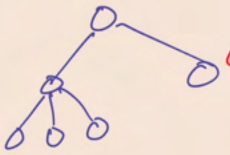
⇒ m-ary tree / n-ary tree : m-ary tree is a tree which has any of its node less than degree m.

4 - ary .

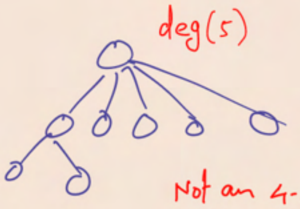


deg(4)

✓



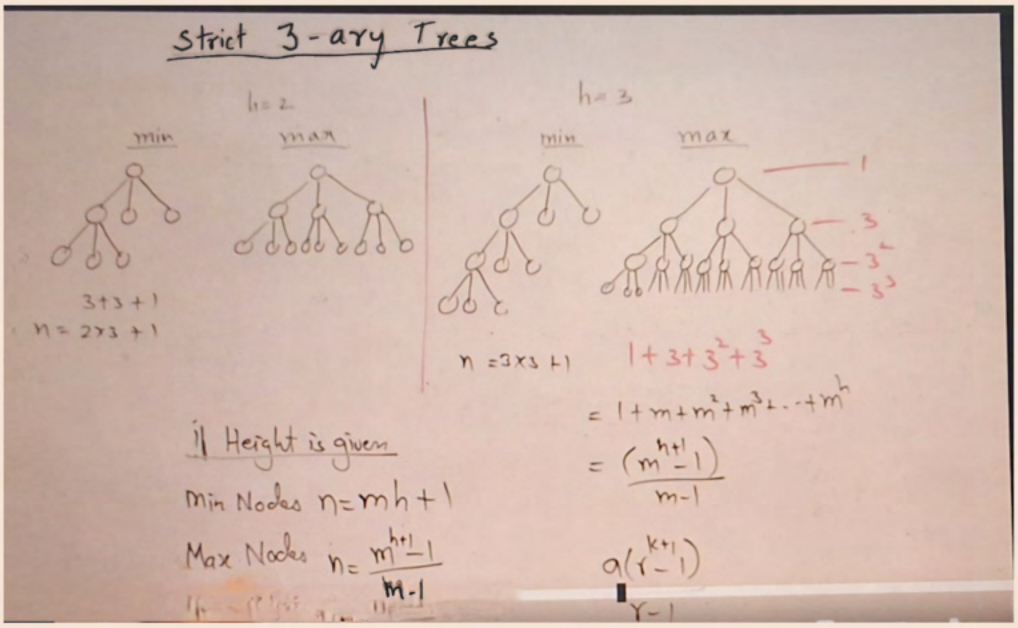
✓



deg(5)

✗

Not an 4-ary



Strict 3-ary Trees

h = 2

min

max

3+3+1
n = 2×3 + 1

h = 3

min

max

1
3
3^2
3^3

n = 3×3 + 1

$1 + 3 + 3^2 + 3^3$

$= 1 + m + m^2 + m^3 + \dots + m^h$

$= \frac{(m^{h+1} - 1)}{m-1}$

if Height is given

Min Nodes $n = mh + 1$

Max Nodes $n = \frac{m^{h+1} - 1}{m-1}$

$\frac{a(r^{k+1} - 1)}{r-1}$

if 'n' Nodes are given :

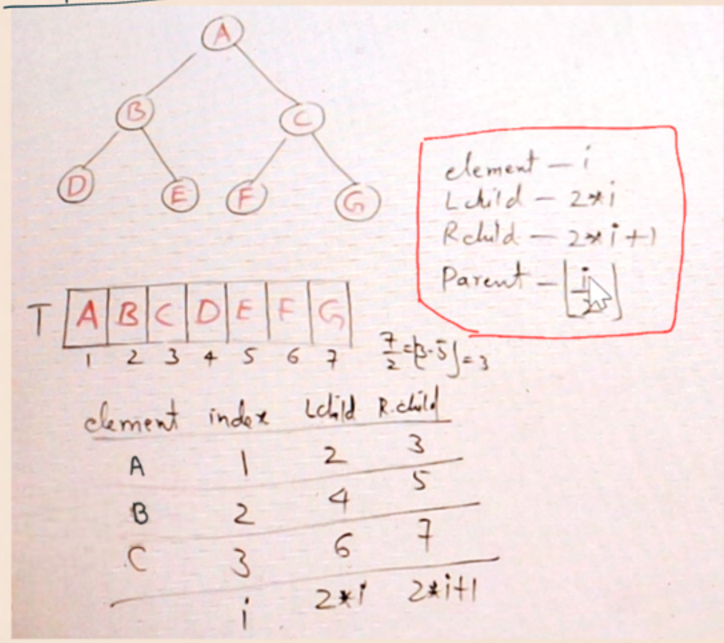
min Height $h = \log_m [n(m-1) + 1] - 1$

Max Height $h = \frac{n-1}{m}$

→ lets i are internal nodes in strict m-ary tree than external nodes will be

$$\boxed{e = (m-1)i + 1}$$

Representation of Binary tree :



element — i
Lchild — 2*i
Rchild — 2*i+1
Parent — $\lfloor \frac{i}{2} \rfloor$

A	B	C	D	E	F	G
1	2	3	4	5	6	7

$\frac{7}{2} = \lfloor 3.5 \rfloor = 3$

element	index	Lchild	Rchild
A	1	2	3
B	2	4	5
C	3	6	7
	i	2*i	2*i+1

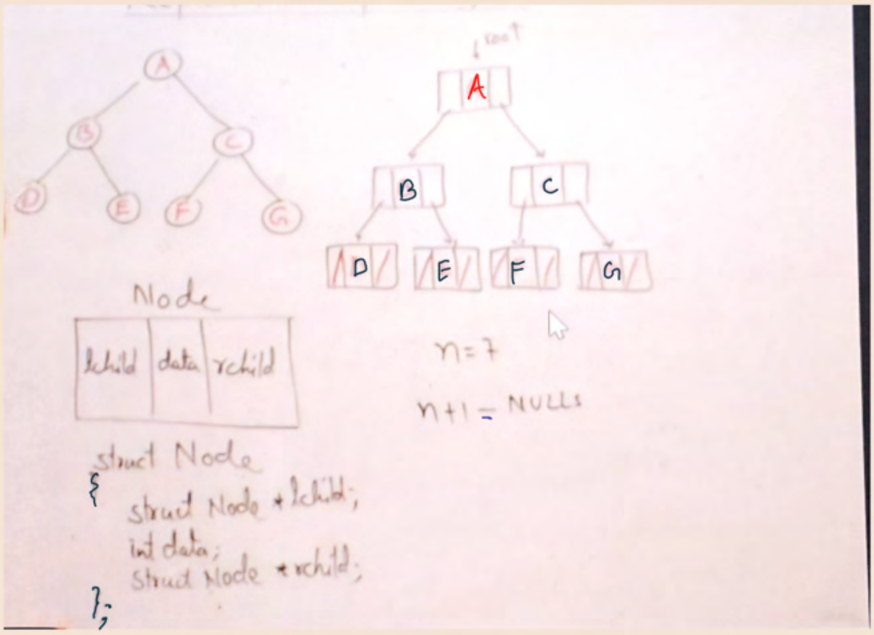
Array Representation of Binary tree

element – i

L child – 2×i

R child – 2×i +1

Parent – $\lfloor i/2 \rfloor$



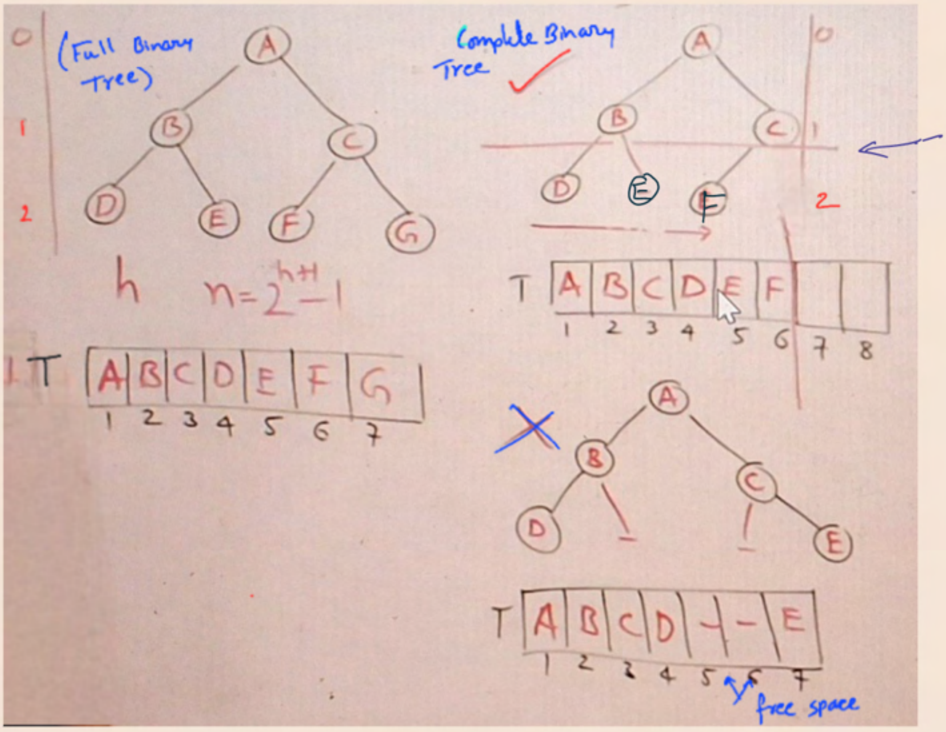
$n=7$

$n+1$ = NULLs

```
struct Node
{
  struct Node *lchild;
  int data;
  struct Node *rchild;
};
```

Linked Representation of a Tree.

=> full vs complete binary tree :-



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

T	A	B	C	D	E	F	G
	1	2	3	4	5	6	7

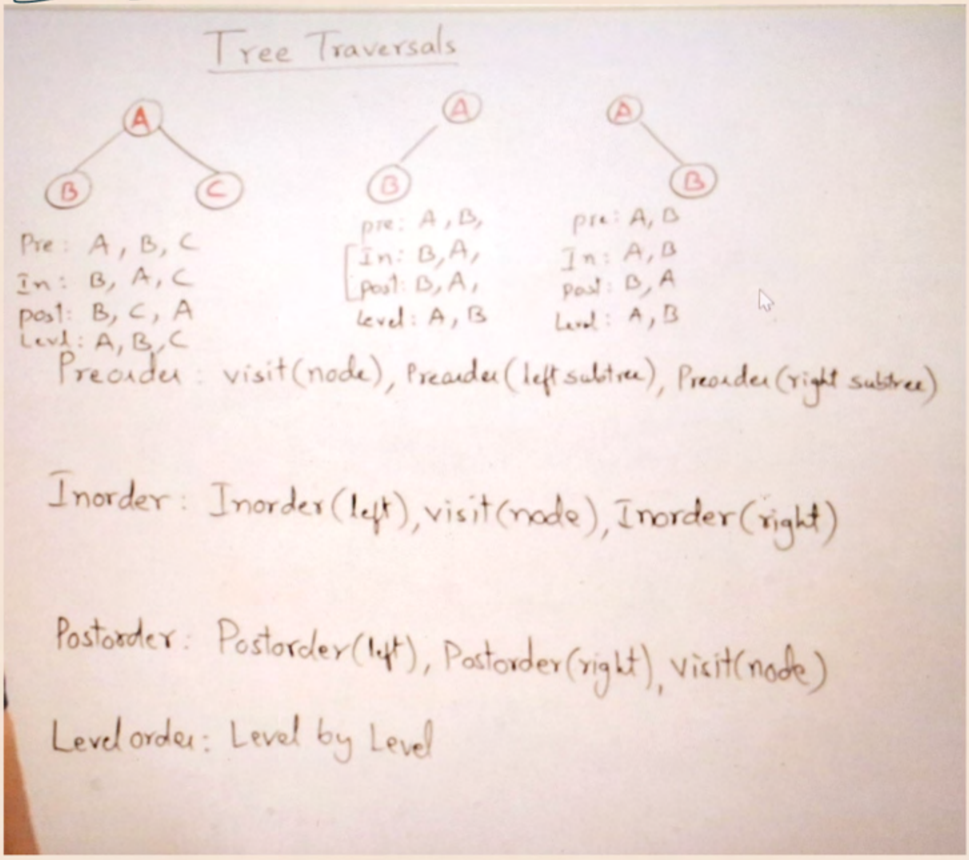
T	A	B	C	D	E	F		
	1	2	3	4	5	6	7	8

T	A	B	C	D	-	-	E
	1	2	3	4	5	6	7

A complete Binary tree is full binary tree till h-1 height and after that filled from left to Right

A full binary tree is a complete tree, but a complete binary tree need not to be a full tree

=> Tree Traversal :



Tree Traversals

Pre: A, B, C
In: B, A, C
post: B, C, A
Level: A, B, C

pre: A, B,
In: B, A,
post: B, A,
level: A, B

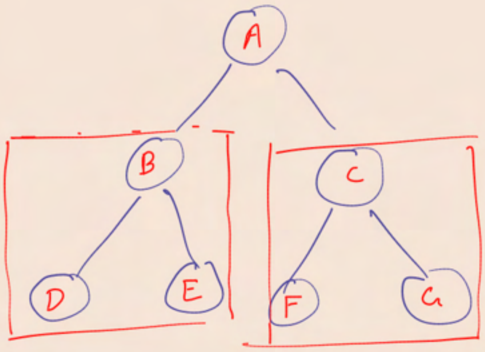
pre: A, B
In: A, B
Post: B, A
Level: A, B

Preorder: visit(node), Preorder(left subtree), Preorder(right subtree)

Inorder: Inorder(left), visit(node), Inorder(right)

Postorder: Postorder(left), Postorder(right), visit(node)

Levelorder: Level by Level



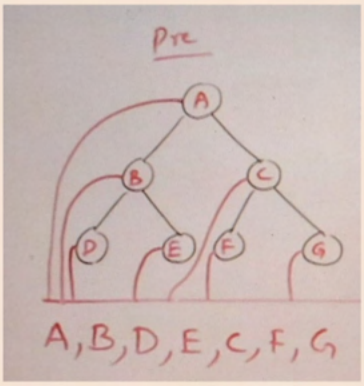
Pre: A, (B, D, E), (C, F, G)
A,B, D, E, C, F, G

In: (D, B, E), A, (F, C, G)
D, B, E, A, F, C, G

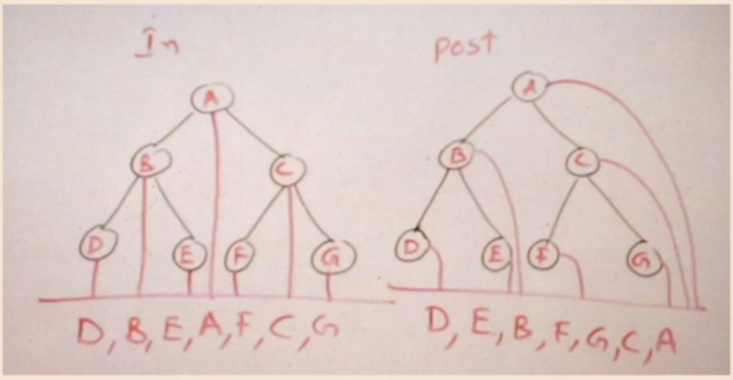
Post: (D, B, E), (F, G, C), A
D,B,E, F,G, C, A

Level: A,B, C, D, E, F, G

=> Easy methods to find tree Traversal :



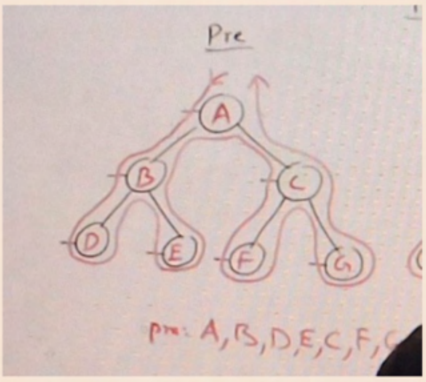
A,B,D,E,C,F,G



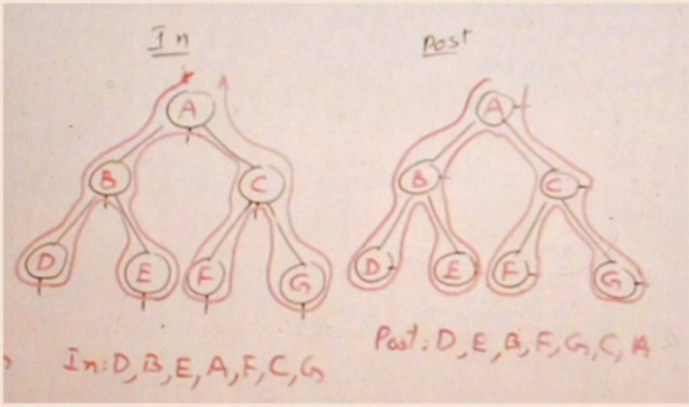
D,B,E,A,F,C,G

D,E,B,F,G,C,A

=> Second method :



pre: A,B,D,E,C,F,G



In: D,B,E,A,F,C,G

Post: D,E,B,F,G,C,A

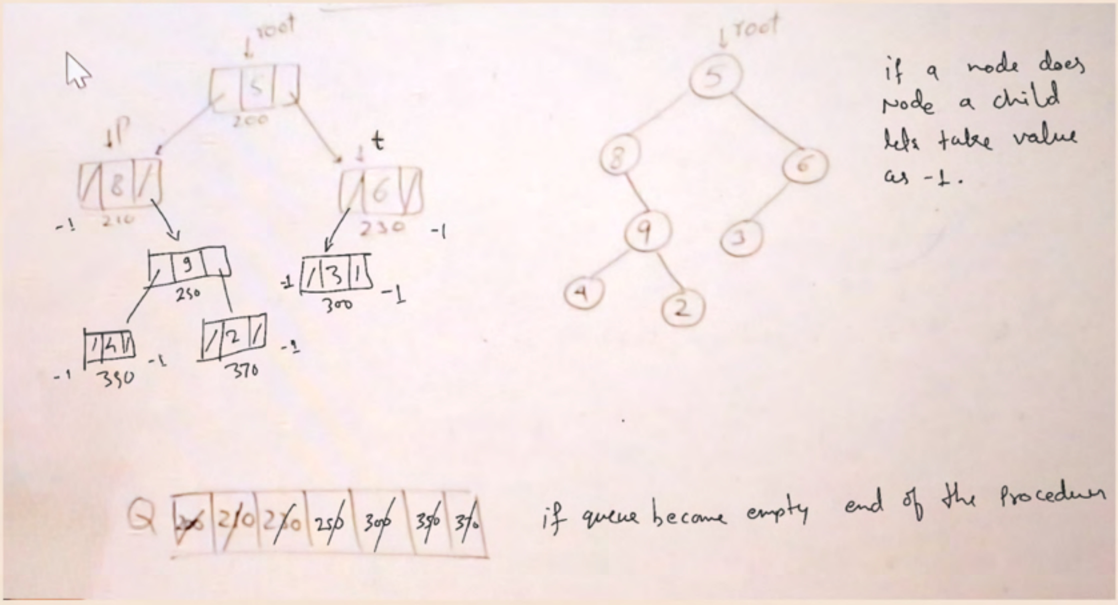
=> third method : use one finger and Point at node.

Preorder => Point from left to Right and go over all the nodes for whichever node its pointing consider that Node in sequence

Inorder => use finger direction from bottom to top

Post order => use finger direction from Right to left.

Creating Binary tree :-

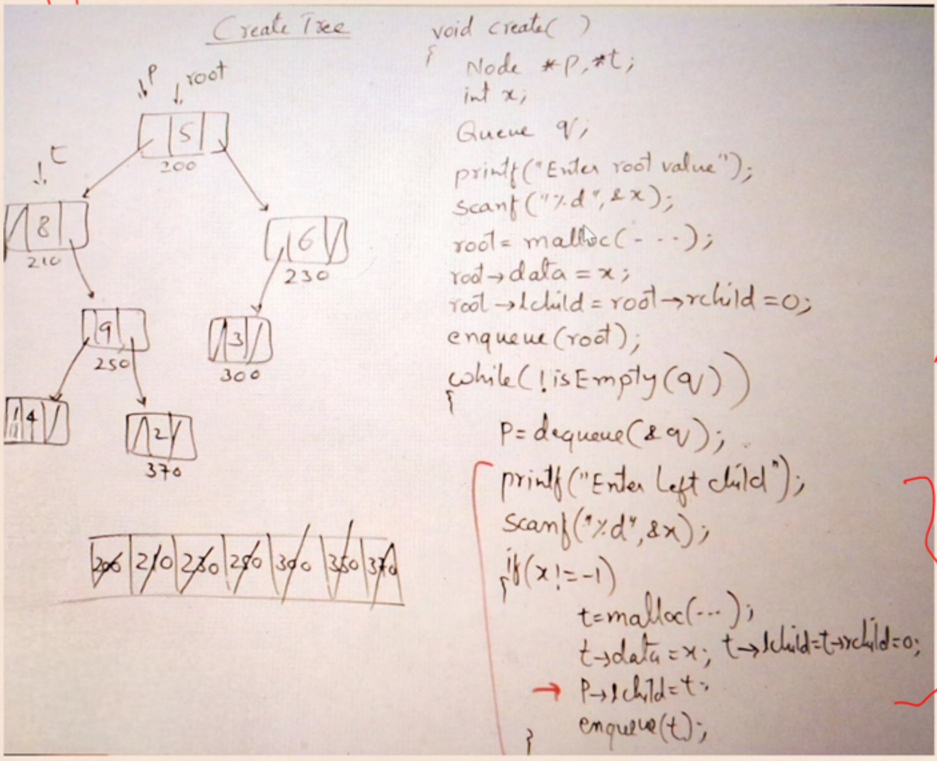


if a node does node a child lets take value as -1.

if queue become empty end of the procedure

285

Program to create binary tree :-

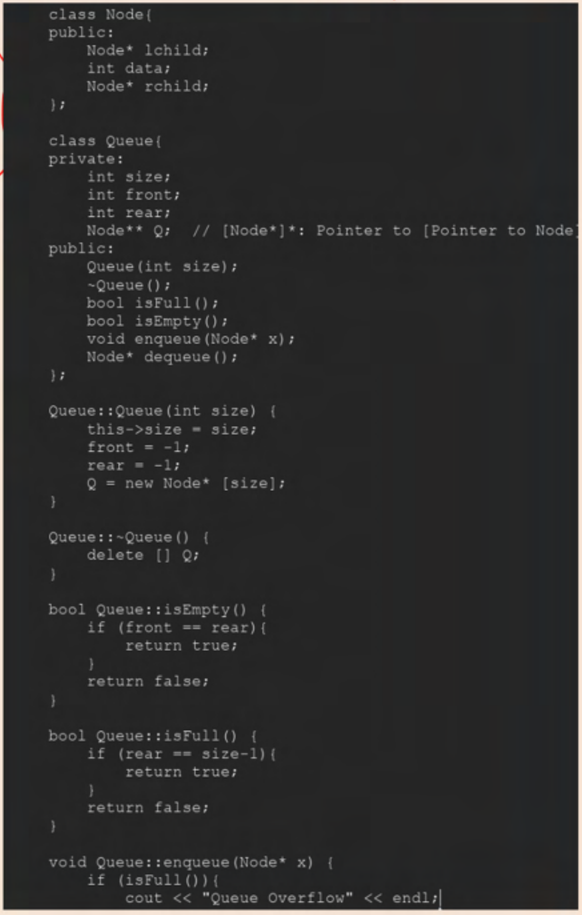


```
void create( )
{
  Node *p, *t;
  int x;
  Queue q;
  printf("Enter root value");
  scanf("%d", &x);
  root= malloc(- - -);
  root→data = x;
  root→lchild = root→rchild = 0;
  enqueue(root);
  while( ! isEmpty(q) )
  {
      p= dequeue(&q);
      printf("Enter Left child");
      scanf("%d", &x);
      if(x != -1)
      {
          t=malloc(- - -);
          t→data = x; t→lchild=t→rchild=0;
        → p→lchild = t;
          enqueue(t);
      }
```

Same code just some modification will be used for creating Right child.

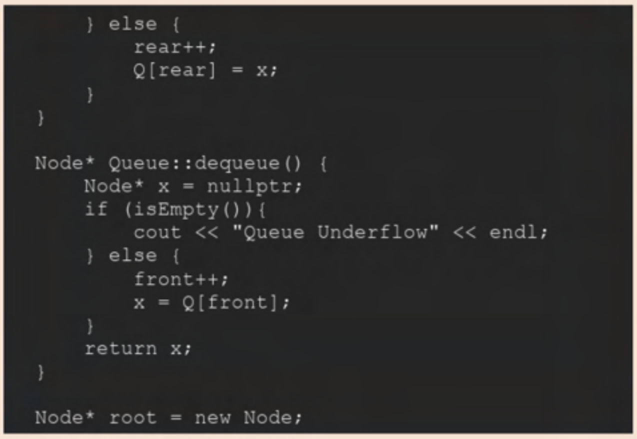
{for malloc function stdlib.h library need to import}

① ↓



```
class Node{
public:
    Node* lchild;
    int data;
    Node* rchild;
};

class Queue{
private:
    int size;
    int front;
    int rear;
    Node** Q;   // [Node*]*: Pointer to [Pointer to Node
public:
    Queue(int size);
    ~Queue();
    bool isFull();
    bool isEmpty();
    void enqueue(Node* x);
    Node* dequeue();
};

Queue::Queue(int size) {
    this->size = size;
    front = -1;
    rear = -1;
    Q = new Node* [size];
}

Queue::~Queue() {
    delete [] Q;
}

bool Queue::isEmpty() {
    if (front == rear){
        return true;
    }
    return false;
}

bool Queue::isFull() {
    if (rear == size-1){
        return true;
    }
    return false;
}

void Queue::enqueue(Node* x) {
    if (isFull()){
        cout << "Queue Overflow" << endl;
```

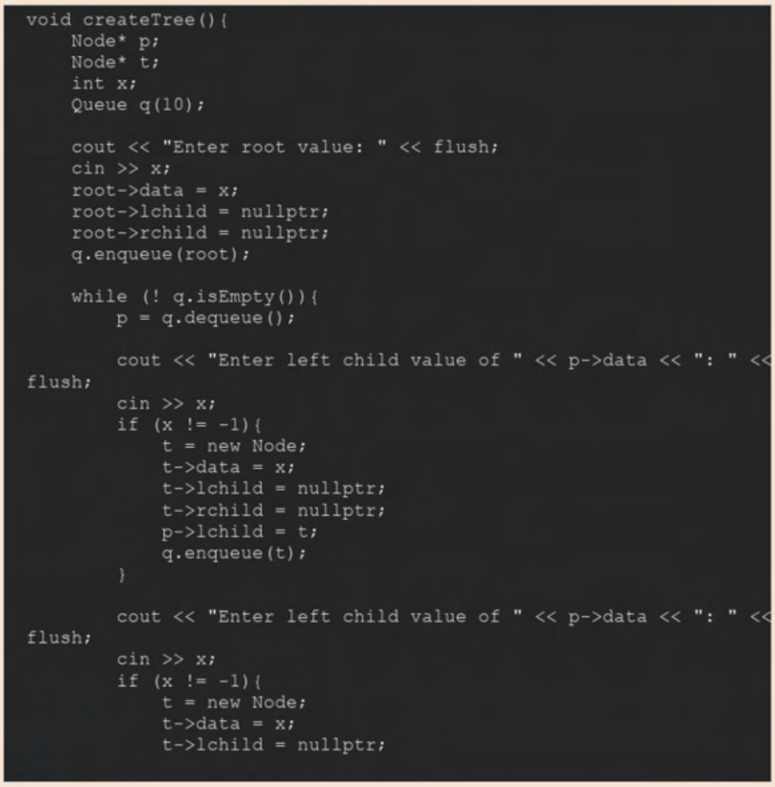
② ↓ (create queue)



```
    } else {
        rear++;
        Q[rear] = x;
    }
}

Node* Queue::dequeue() {
    Node* x = nullptr;
    if (isEmpty()){
        cout << "Queue Underflow" << endl;
    } else {
        front++;
        x = Q[front];
    }
    return x;
}

Node* root = new Node;
```

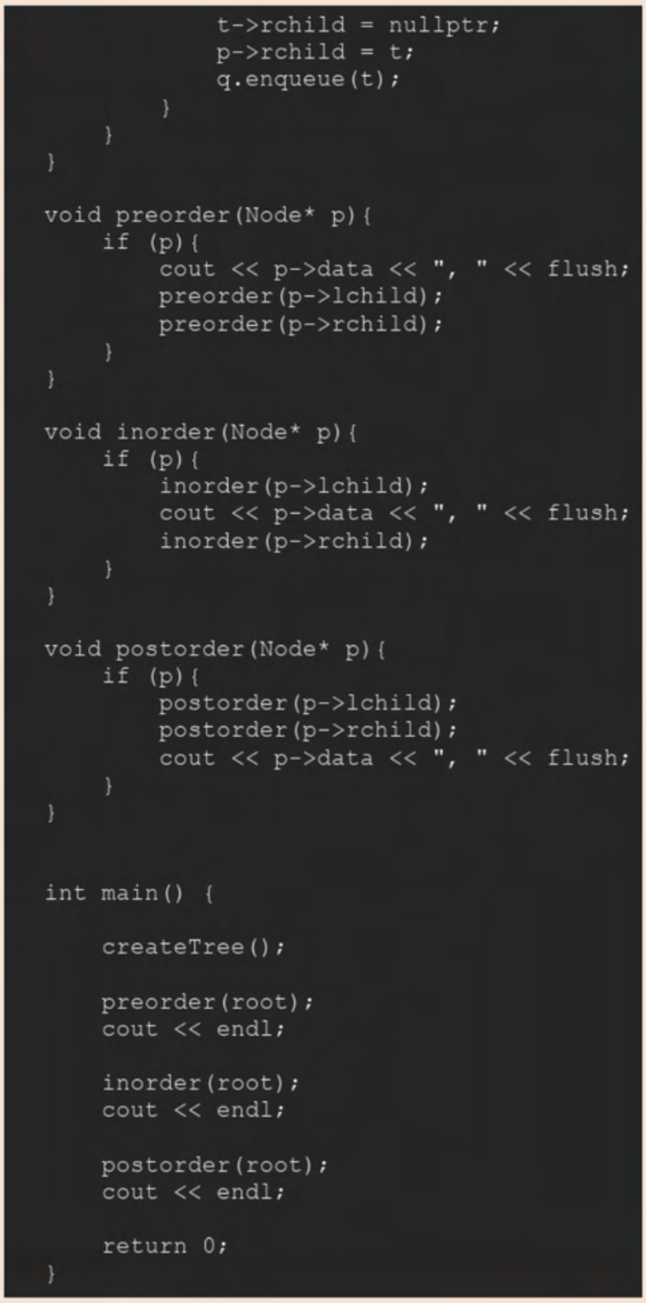
(create tree by inserting elements one by one.)

3



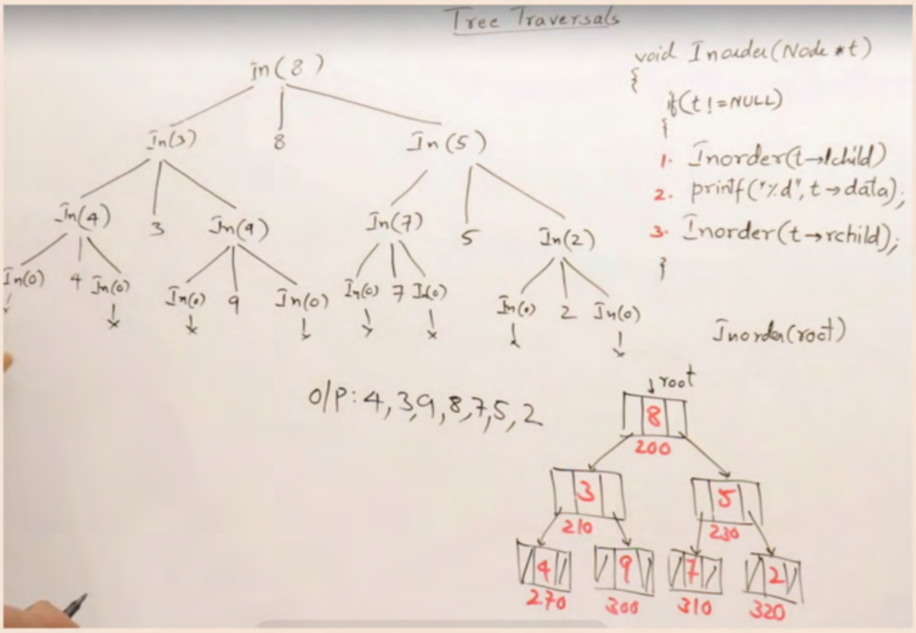
```
void createTree(){
    Node* p;
    Node* t;
    int x;
    Queue q(10);

    cout << "Enter root value: " << flush;
    cin >> x;
    root->data = x;
    root->lchild = nullptr;
    root->rchild = nullptr;
    q.enqueue(root);

    while (! q.isEmpty()){
        p = q.dequeue();

        cout << "Enter left child value of " << p->data << ": " <<
flush;
        cin >> x;
        if (x != -1){
            t = new Node;
            t->data = x;
            t->lchild = nullptr;
            t->rchild = nullptr;
            p->lchild = t;
            q.enqueue(t);
        }

        cout << "Enter left child value of " << p->data << ": " <<
flush;
        cin >> x;
        if (x != -1){
            t = new Node;
            t->data = x;
            t->lchild = nullptr;
```

4



```
                t->rchild = nullptr;
                p->rchild = t;
                q.enqueue(t);
            }
        }
}

void preorder(Node* p){
    if (p){
        cout << p->data << ", " << flush;
        preorder(p->lchild);
        preorder(p->rchild);
    }
}

void inorder(Node* p){
    if (p){
        inorder(p->lchild);
        cout << p->data << ", " << flush;
        inorder(p->rchild);
    }
}

void postorder(Node* p){
    if (p){
        postorder(p->lchild);
        postorder(p->rchild);
        cout << p->data << ", " << flush;
    }
}


int main() {

    createTree();

    preorder(root);
    cout << endl;

    inorder(root);
    cout << endl;

    postorder(root);
    cout << endl;

    return 0;
}
```

Tree Traversal : (In order tree traversal) :

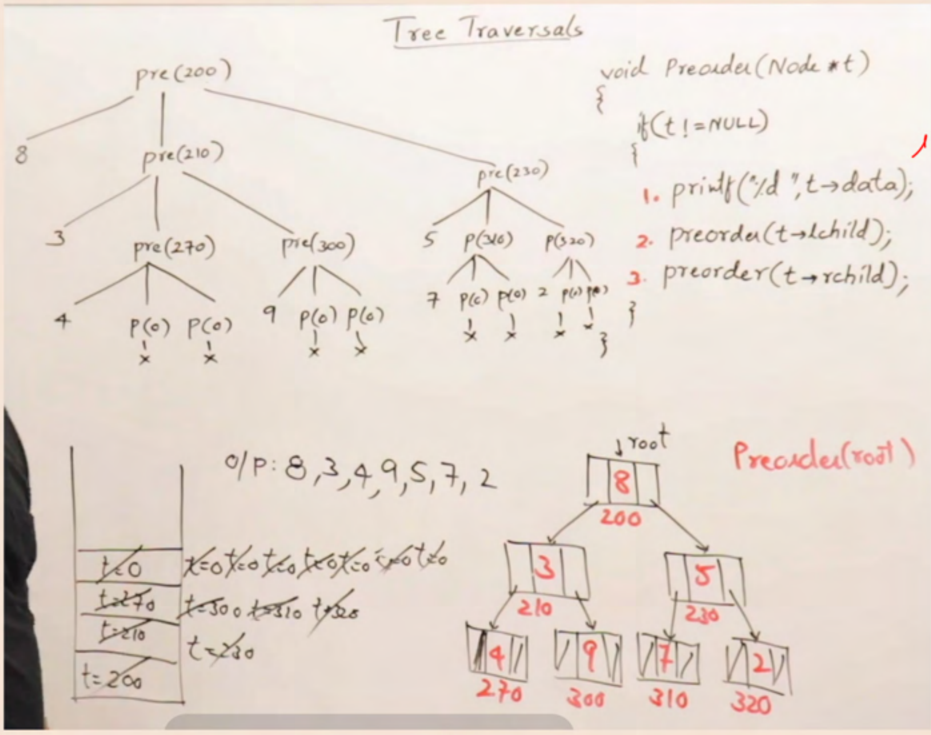


```
void Inorder(Node *t)
{
   if(t!=NULL)
   {
1.    Inorder(t->lchild)
2.    printf("%d", t->data);
3.    Inorder(t->rchild);
   }
}

Inorder(root)
```

O/P: 4,3,9,8,7,5,2

- This is a common tree traversal, it uses the recursion to traverse throgh three nodes.
- where once the left child is traversed then print the nodes value and after that travese through the right child of the current node.

Preorder tree traversal :-



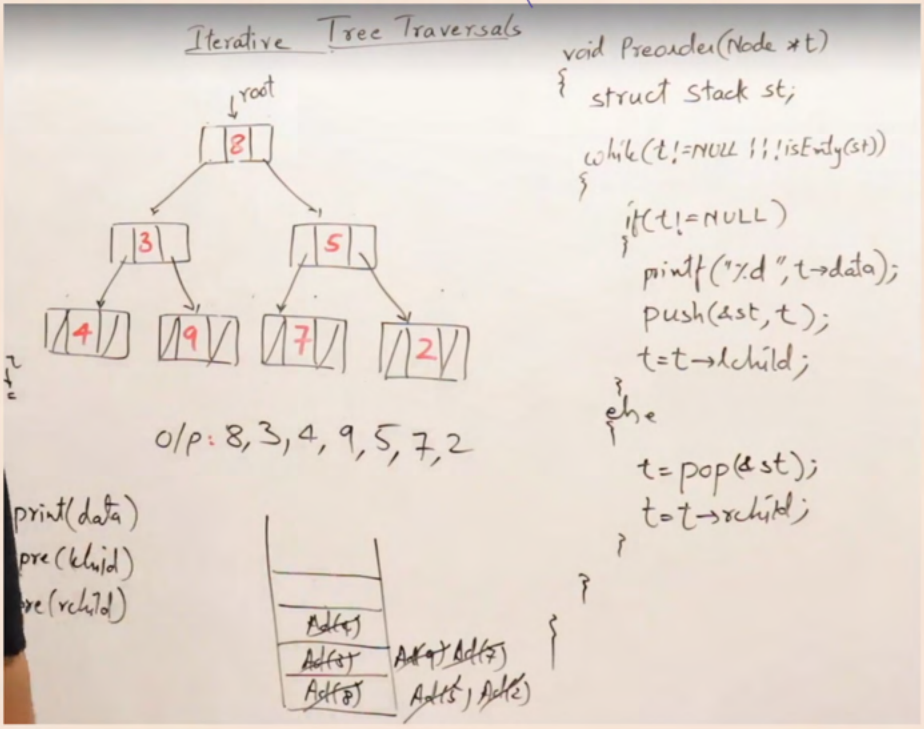
Tree Traversals

```
void Preorder(Node *t)
{
  if(t!=NULL)
  {
   1. printf("%d ",t->data);
   2. preorder(t->lchild);
   3. preorder(t->rchild);
  }
}
```

O/P: 8,3,4,9,5,7,2

Preorder(root)

Iterative Tree traversal :- (Pre order)



Iterative Tree Traversals

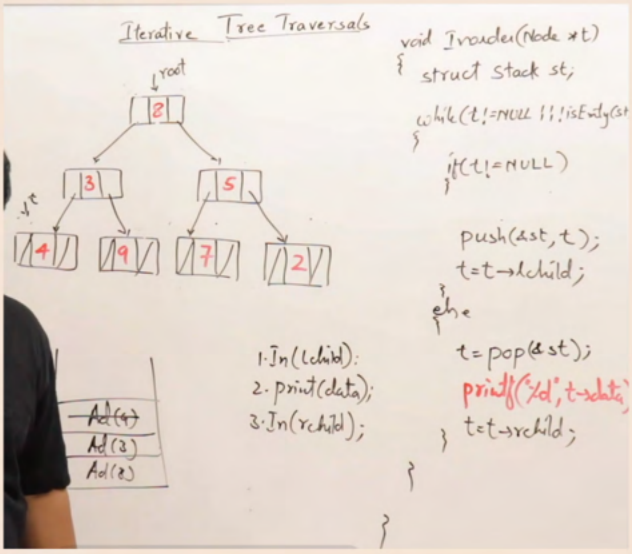
```
void Preorder(Node *t)
{
  struct Stack st;
  while(t!=NULL || !isEmpty(st))
  {
    if(t!=NULL)
    {
      printf("%d",t->data);
      push(&st,t);
      t=t->lchild;
    }
    else
    {
      t=pop(&st);
      t=t->rchild;
    }
  }
}
```

O/P: 8,3,4,9,5,7,2

print(data)
pre(lchild)
pre(rchild)

- Instead of the using recussion in iterative tree traversal we use stack to hold the information of the traversed nodes. Here stack should be of address type here.
- for Itereative tree traversal we have to use the stack which holds the address values. by using stack we can save the nodes which traversed.

Inorder :-

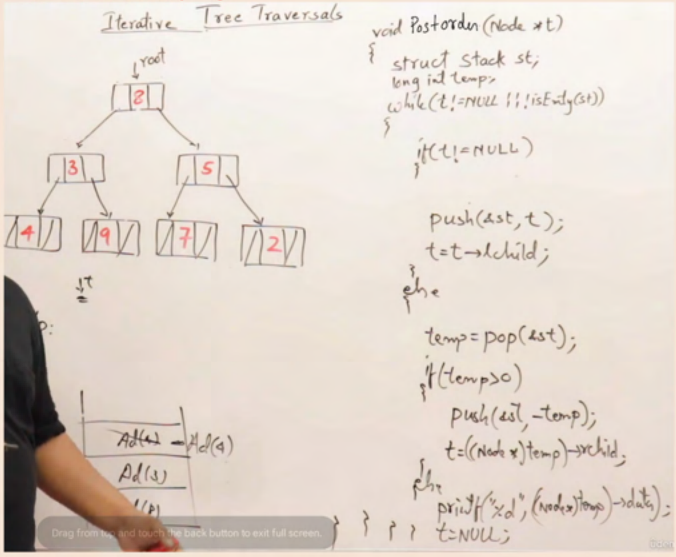


Iterative Tree Traversals

```
void Inorder(Node *t)
{
  struct Stack st;
  while(t!=NULL || !isEmpty(st
  {
    if(t!=NULL)
    {
      push(&st,t);
      t=t->lchild;
    }
    else
    {
      t=pop(&st);
      printf("%d",t->data);
      t=t->rchild;
    }
  }
}
```

1. In(lchild);
2. print(data);
3. In(rchild);

Post order :-



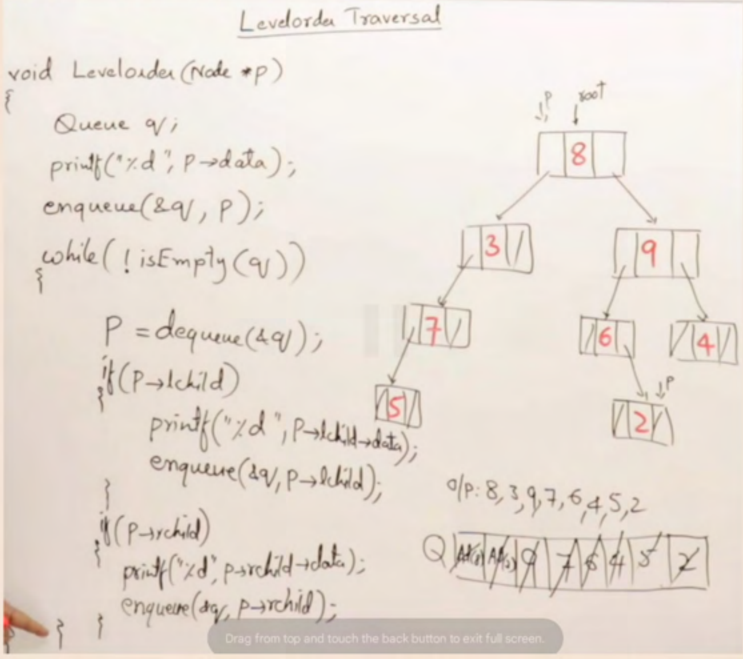
Iterative Tree Traversals

```
void Postorder(Node *t)
{
  struct Stack st;
  long int temp;
  while(t!=NULL || !isEmpty(st))
  {
    if(t!=NULL)
    {
      push(&st,t);
      t=t->lchild;
    }
    else
    {
      temp=pop(&st);
      if(temp>0)
      {
        push(&st,-temp);
        t=((Node *)temp)->rchild;
      }
      else
      {
        printf("%d",((Node *)temp)->data);
        t=NULL;
      }
    }
  }
}
```

root
8
3
5
4
9
7
2
~~Ad(4)~~ -Ad(4)
Ad(3)
Drag from top and touch the back button to exit full screen.

- the post order tree travesal is a little bit tricky, here we have to iterate through its nodes before printing its value.
- therfore when we iteate through left child we save normal address, but in next iteration when we got to the right child, we push the negative address to differentiate between the left and right child iterations. More in algo.

Level order :-



Levelorder Traversal

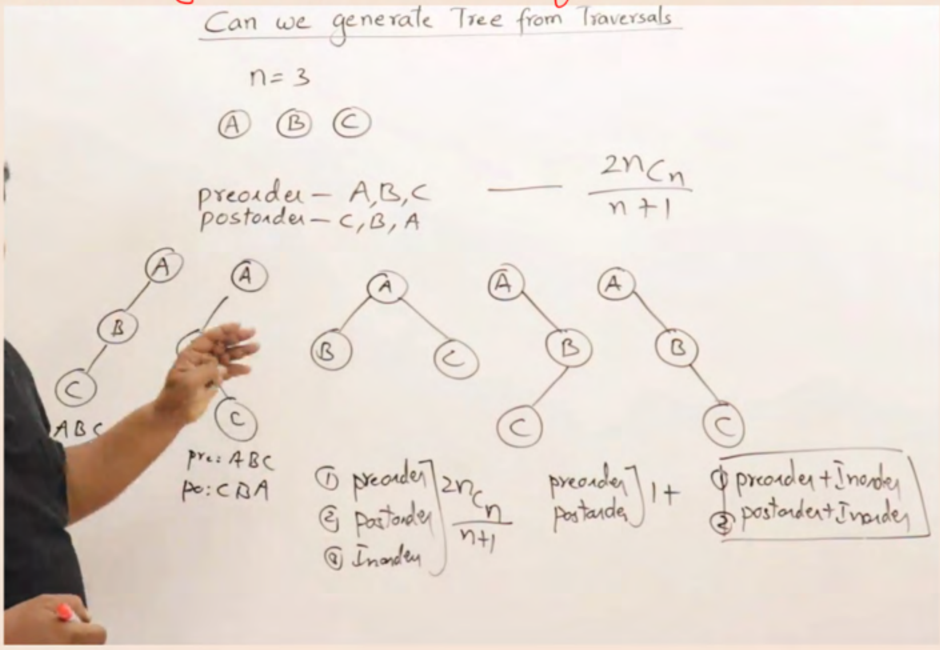
```
void Levelorder(Node *p)
{
    Queue q;
    printf("%d", p->data);
    enqueue(&q, p);
    while(!isEmpty(q))
    {
        p = dequeue(&q);
        if(p->lchild)
        {
            printf("%d", p->lchild->data);
            enqueue(&q, p->lchild);
        }
        if(p->rchild)
        {
            printf("%d", p->rchild->data);
            enqueue(&q, p->rchild);
        }
    }
}
```

o/p: 8, 3, 9, 7, 6, 4, 5, 2

p
root
8
3
9
7
6
4
5
2
p
Q
Drag from top and touch the back button to exit full screen.

Level order is similar as inserting nodes into tree

=> How to generate the Tree from Traversals:-



Can we generate Tree from Traversals

n = 3

Ⓐ Ⓑ Ⓒ

preorder – A, B, C
postorder – C, B, A
— $\frac{{}^{2n}C_n}{n+1}$

Pre: ABC
Po: CBA

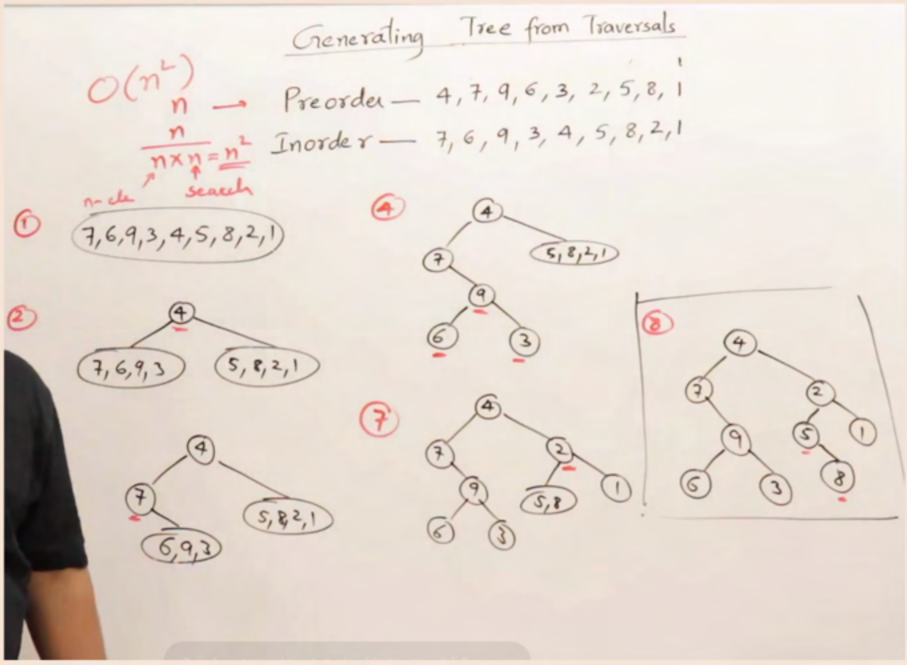
① preorder
② Postorder
③ Inorder
$\frac{{}^{2n}C_n}{n+1}$

preorder
Postorder
1+

① preorder + Inorder
② postorder + Inorder

- either using pre order + post order, we can able to generate the tree or post+ in order,
if we just use the pre +post order we can not construct the tree.

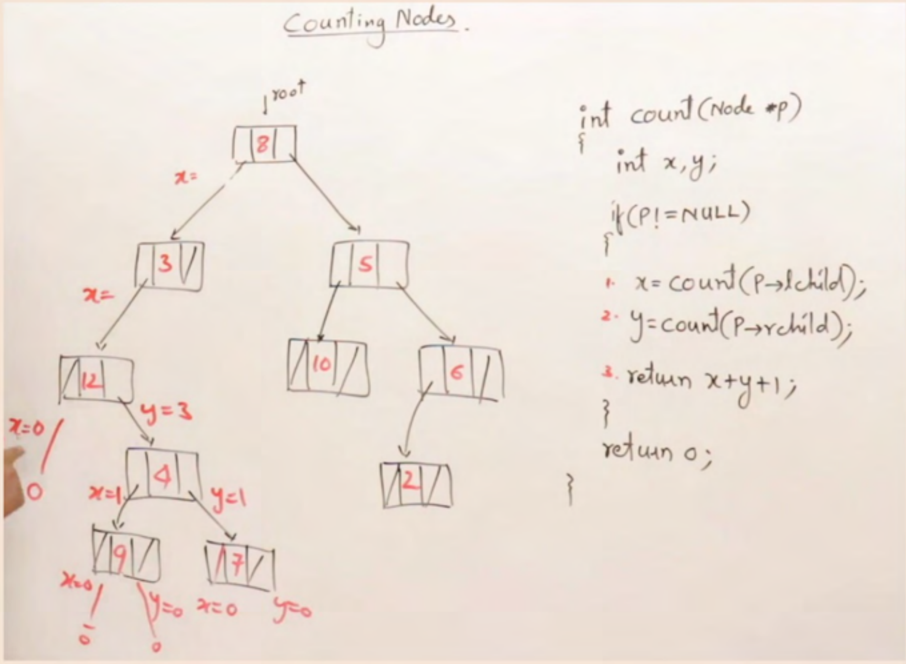
- We start iterating from the preorder elements, we find the same element in order sequence. once we finde that we split the left and right side of nodes for that node. we do this untill all the elemets are iterated. this procedure take the n^2 time complexity.



Generating Tree from Traversals

Preorder — 4, 7, 9, 6, 3, 2, 5, 8, 1

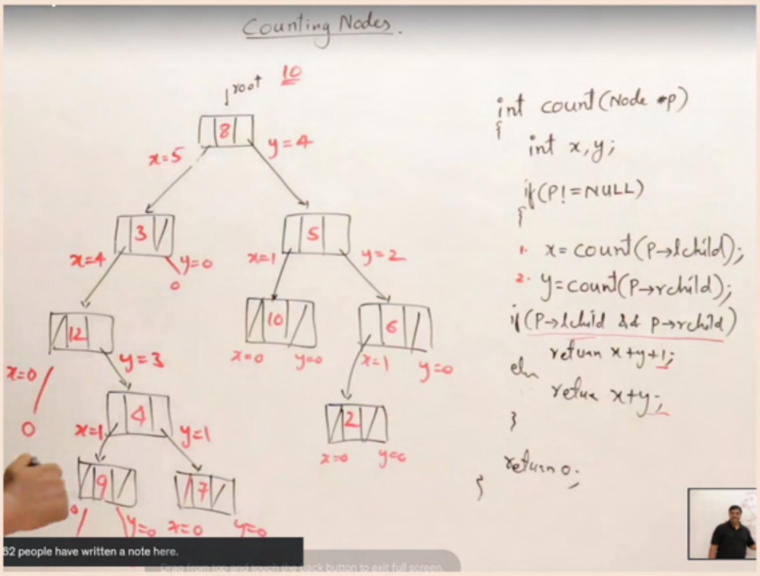
Inorder — 7, 6, 9, 3, 4, 5, 8, 2, 1



Counting Nodes.

```
int count(Node *p)
{
    int x,y;
    if(P!=NULL)
    {
        x=count(P->lchild);
        y=count(P->rchild);
        return x+y+1;
    }
    return 0;
}
```

count the nodes which are having degree 2 in a tree.

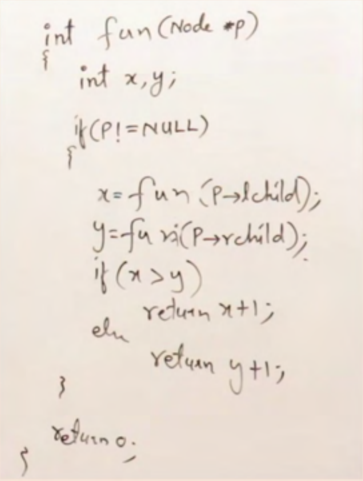


Counting Nodes.

```
int count(Node *p)
{
    int x,y;
    if(P!=NULL)
    {
        x=count(P->lchild);
        y=count(P->rchild);
        if(P->lchild && p->rchild)
            return x+y+1;
        else
            return x+y;
    }
    return 0;
}
```

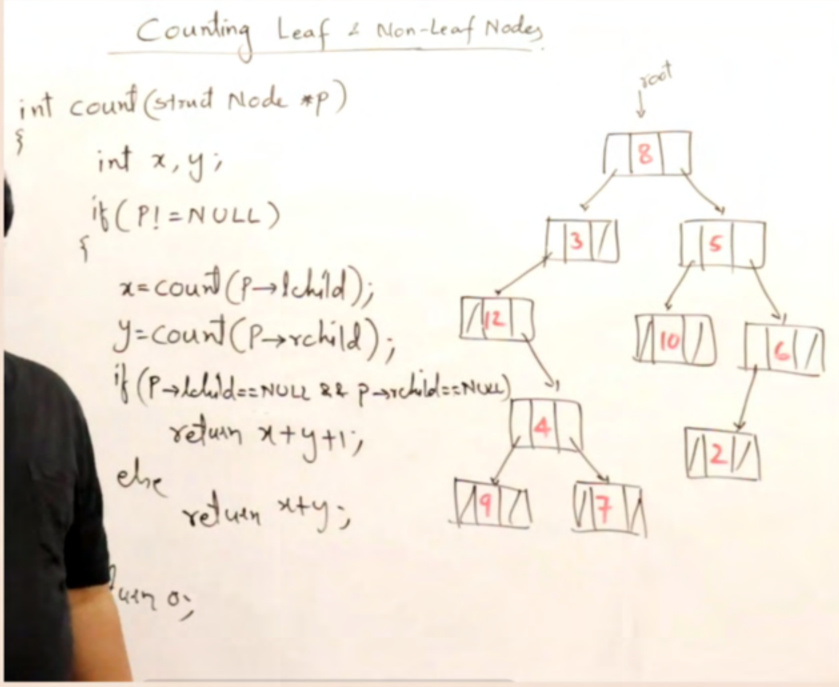
root
10
8
x=5
y=4
3
5
x=4
y=0
0
x=1
y=2
12
10
6
x=0
y=0
x=1
y=0
x=0
y=3
0
4
2
x=1
y=1
x=0
y=0
9
7
y=0
x=0
y=0
32 people have written a note here.
Swipe from top and touch the back button to exit full screen

Finding the height of the tree



```
int fun(Node *p)
{
    int x,y;
    if(P!=NULL)
    {
        x=fun(P->lchild);
        y=fun(P->rchild);
        if(x>y)
            return x+1;
        else
            return y+1;
    }
    return 0;
}
```

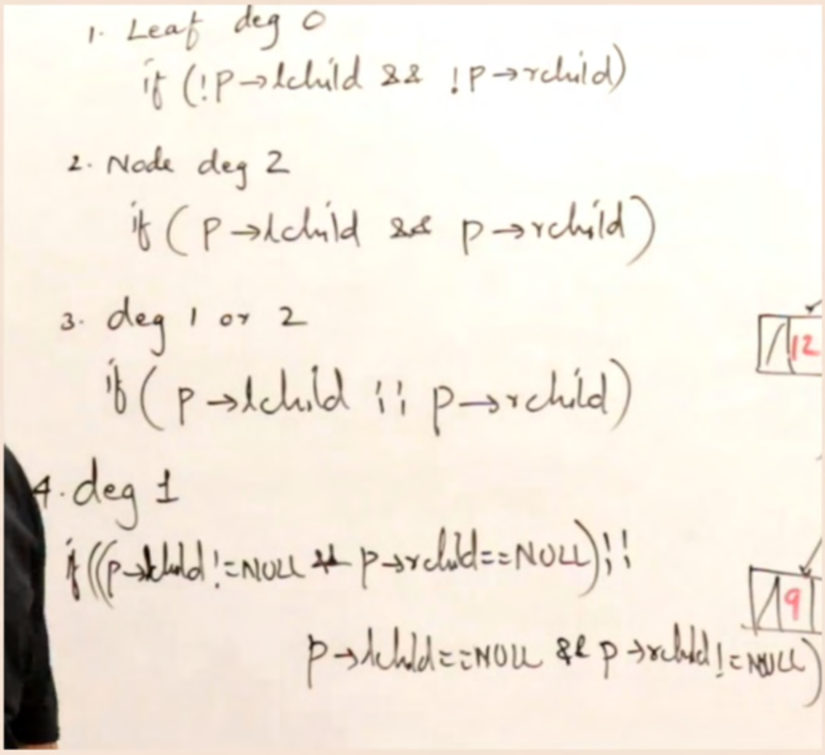
Counting leaf child.



Counting Leaf & Non-Leaf Nodes

```
int count(struct Node *p)
{
    int x,y;
    if(P!=NULL)
    {
      x=count(P→lchild);
      y=count(P→rchild);
      if(P→lchild==NULL && p→rchild==NULL)
          return x+y+1;
      else
          return x+y;

   urn 0;
```

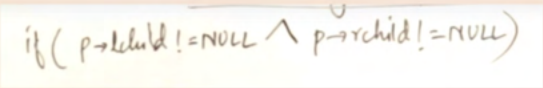
Counting the leaf nodes in a tree.



```
1. Leaf deg 0
     if (!P→lchild && !P→rchild)

2. Node deg 2
     if (P→lchild && p→rchild)

3. deg 1 or 2
     if (p→lchild || p→rchild)

4. deg 1
   if((p→lchild!=NULL && p→rchild==NULL)||
                 p→lchild==NULL && p→rchild!=NULL)
```

Different conditions to find the nodes with different degrees in a tree.

The short condition to find the degree 1 nodes in a tree is below.



```
if( p→lchild!=NULL ^ p→rchild!=NULL)
```