

Trees

Terminology

- ✓ 1. Root
- ✓ 2. Parent
- ✓ 3. child
- ✓ 4. Siblings
- ✓ 5. Descendants
- ✓ 6. Ancestors
- ✓ 7. Degree of a node
- ✓ 8. Internal / External nodes
- ✓ 9. Levels → starts from 1
- ✓ 10. Height → starts from 0
- ✓ 11. Forest → collection of trees

height

0

1

2

3

4

Level

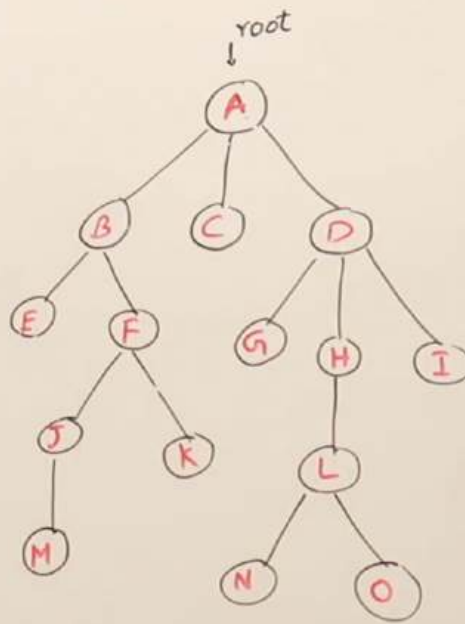
1

2

3

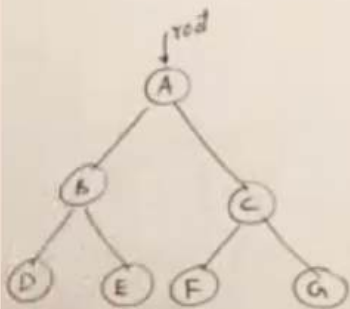
4

5



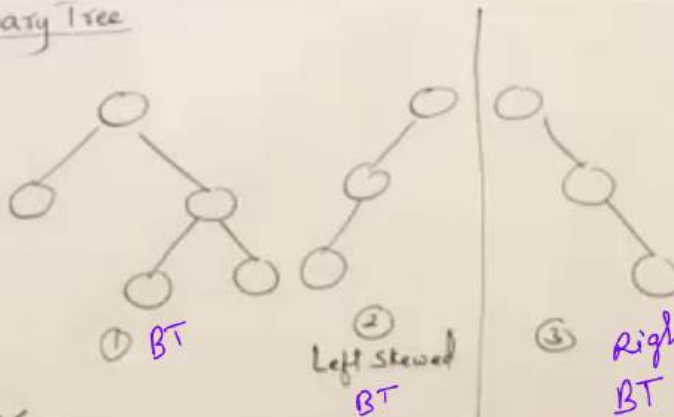
degree of node is the number of child it is having
 ↳ max degree of node in a tree called, degree of tree
 ↳ External nodes which does not have any child node → (eg. leaf node)

Binary Tree



$\deg(T) = 2$

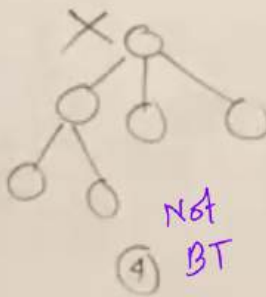
↳ children {0, 1, 2}



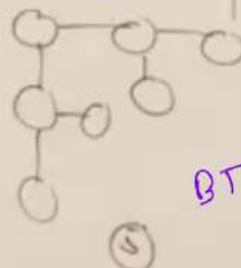
① BT

② Left skewed BT

③ right skewed BT



④ Not BT



⑤ BT

Height vs Nodes :

Height vs Nodes

Height $h=1$

min max

$n=1$ $n=1$

\uparrow \uparrow

min node max node

Height $h=2$

min max

$n=2$ $n=3$

Height $h=3$

min max

$n=3$ $n=7$

min Nodes $n = h + 1$

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

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

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

$Q=1, R=2$

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

$= 2^4 - 1$

$= 16 - 1$

$= 15$

Height vs Nodes

Nodes $n=3$

min max

$h=1$ $h=2$

Nodes $n=7$

min max

$h=2$ $h=6$

Nodes $n=15$

min max

$h=3$ $h=14$

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

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

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

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

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

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

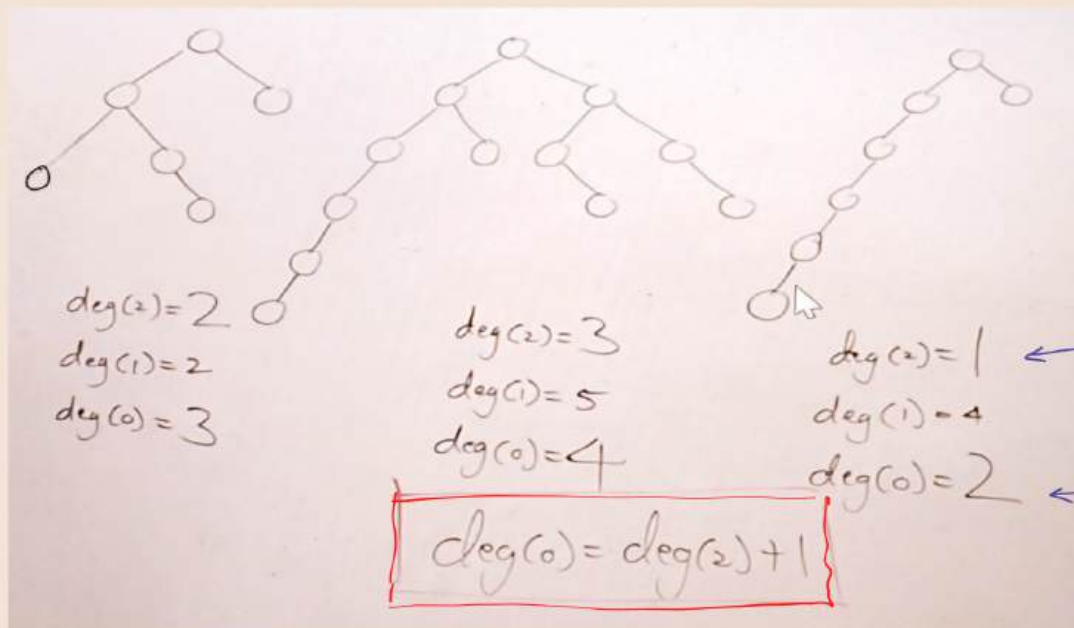
$h + 1 = \log_2(n + 1)$

Height of binary tree :

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

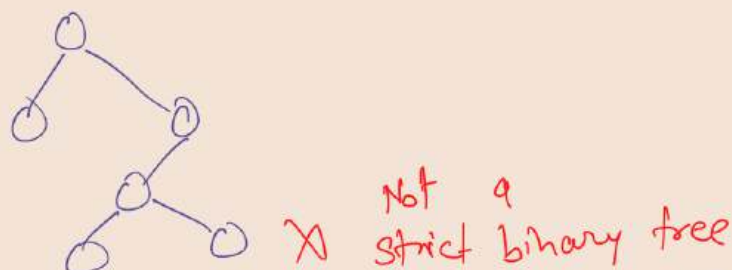
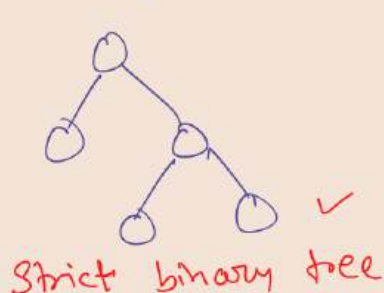
Number of Nodes in a binary tree :

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

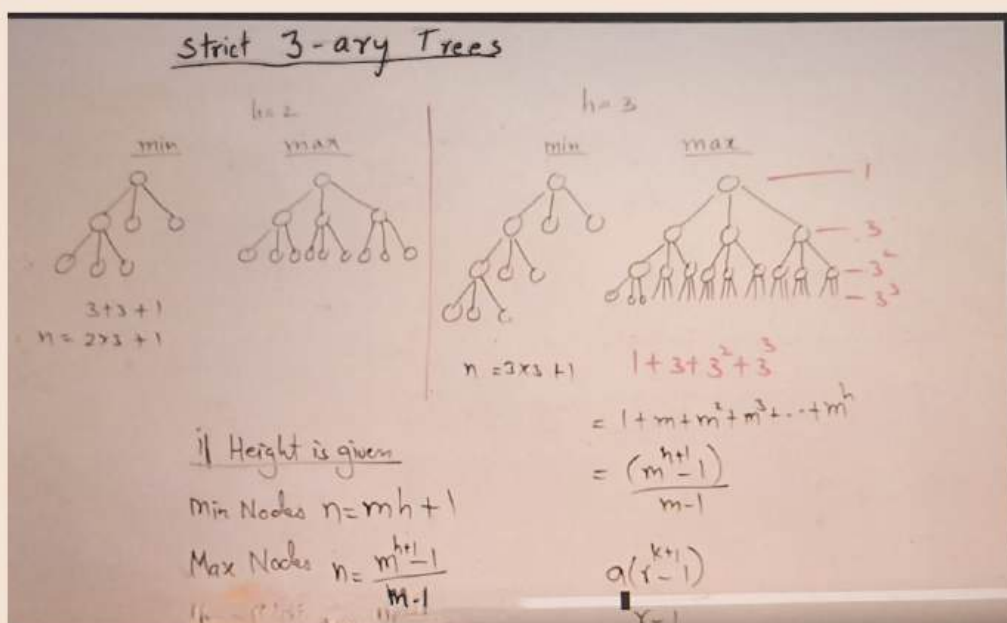
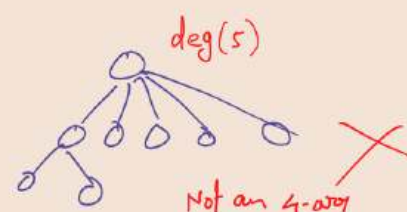
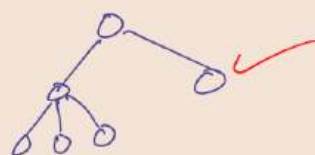
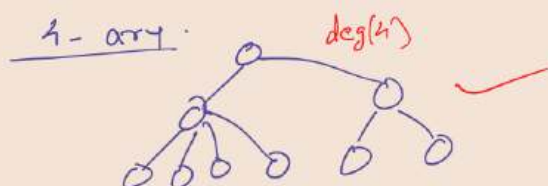


degree is
How many
nodes are connected
to a node

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



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



if 'n' Nodes are given:

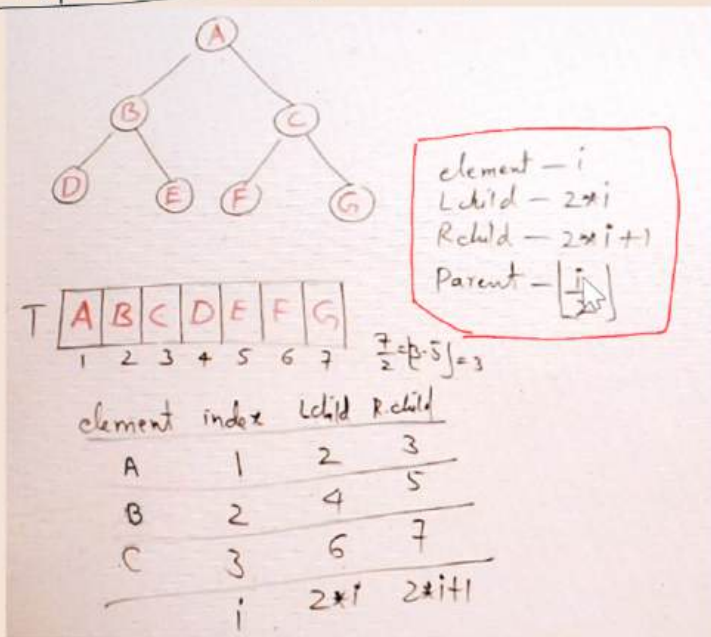
min Height $h = \log_m [n(m-1) + 1] - 1$

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

→ let i are internal nodes in
strict m-ary tree then
external nodes will be

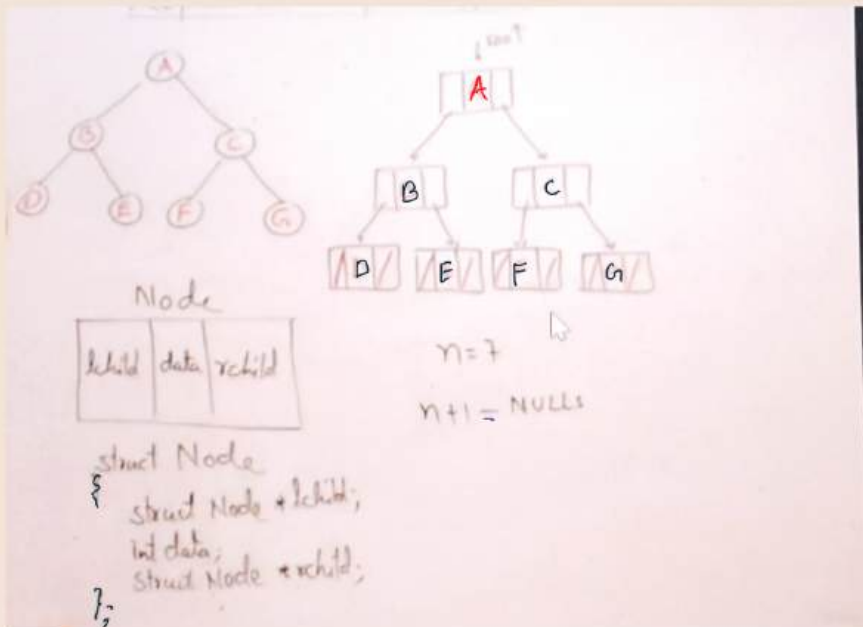
$e = (m-1)i + 1$

Representation of Binary tree :-



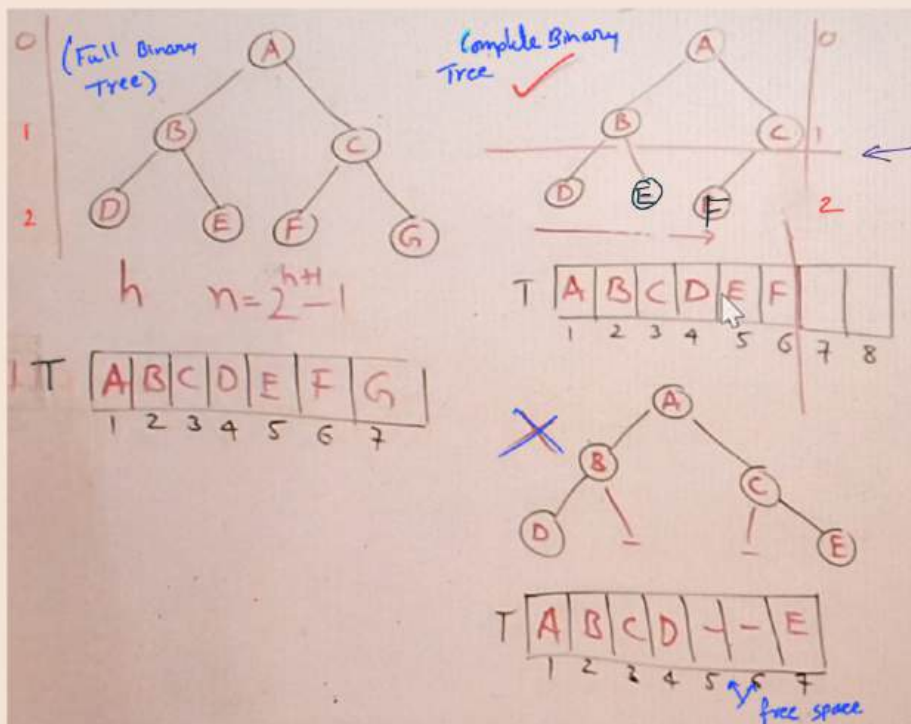
Array Representation of Binary tree

element - i
 L child - $2*i$
 R child - $2*i + 1$
 Parent - $\lfloor \frac{i}{2} \rfloor$



Linked Representation of a Tree

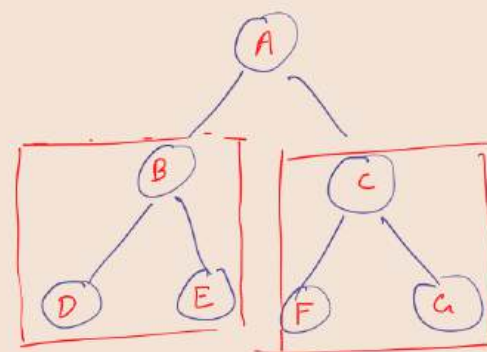
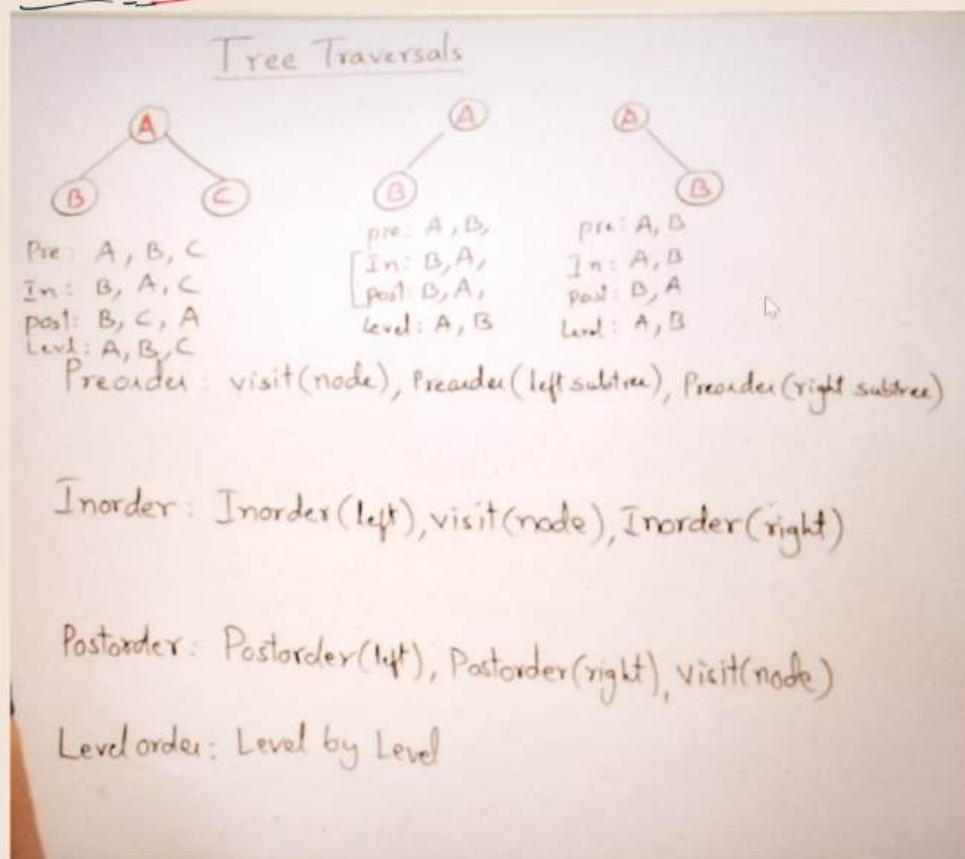
=> full vs complete binary tree :-



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

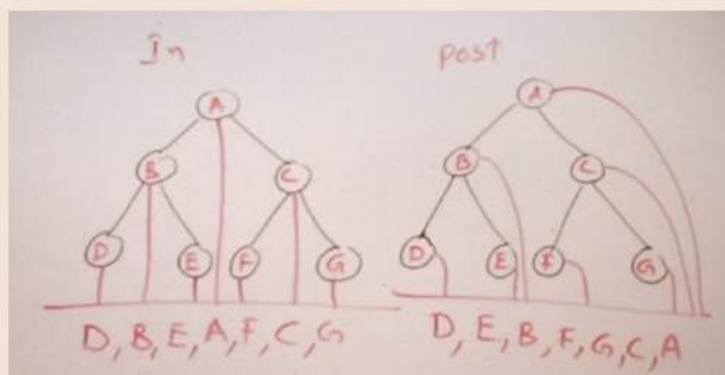
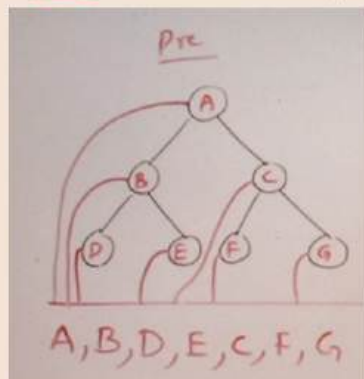
A full binary tree is a complete tree, but a complete binary tree need not to be a full tree

⇒ Tree Traversal :

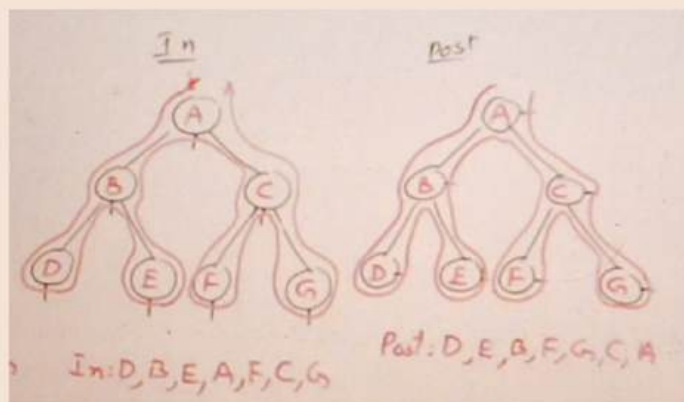
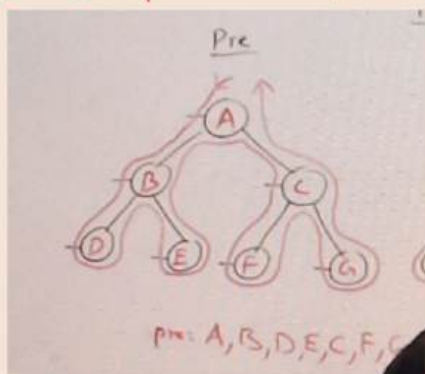


Pre: A, (B, D, E), (C, F, G)
 A, B, D, E, C, F, G
 In: (D, B, E), A, (F, C, G)
 D, B, E, A, F, C, G
 Post: (D, B, E), (F, G, C), A
 D, B, E, F, G, C, A
 Level: A, B, C, D, E, F, G

⇒ Easy methods to find tree Traversal :



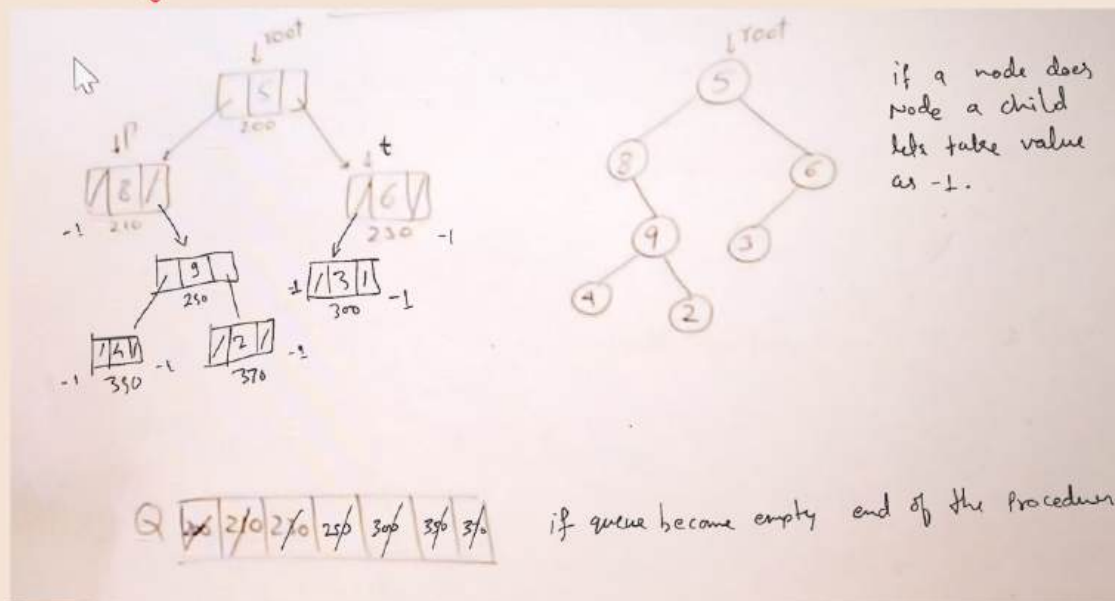
⇒ Second method :



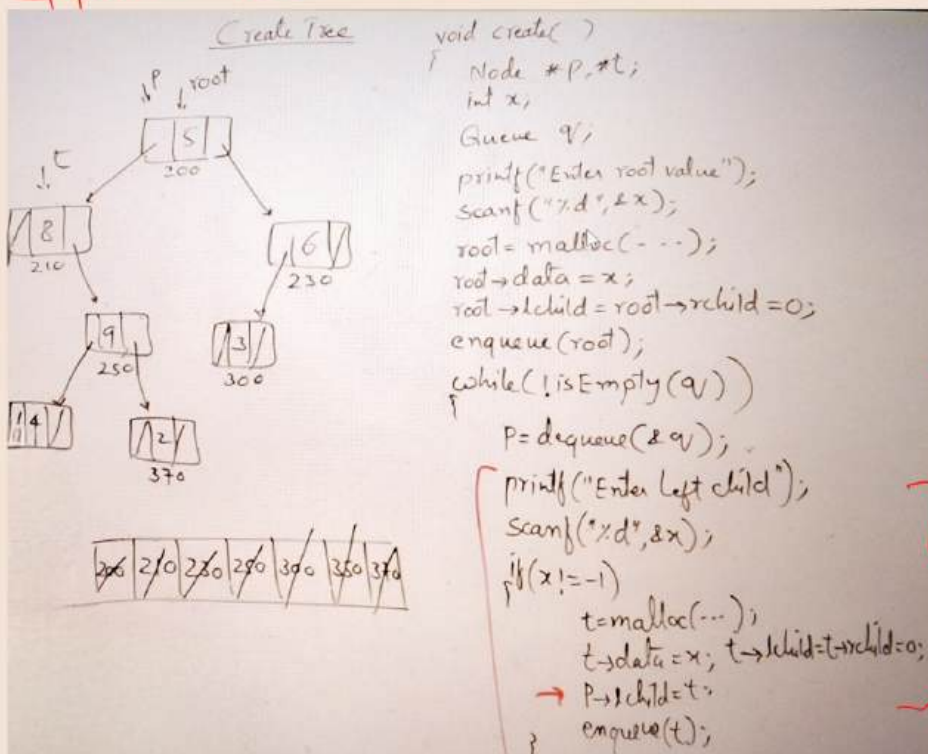
⇒ third method :

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 :-



285 Program to create binary tree :-



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

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)

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

```

} 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.)

4

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

```
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();

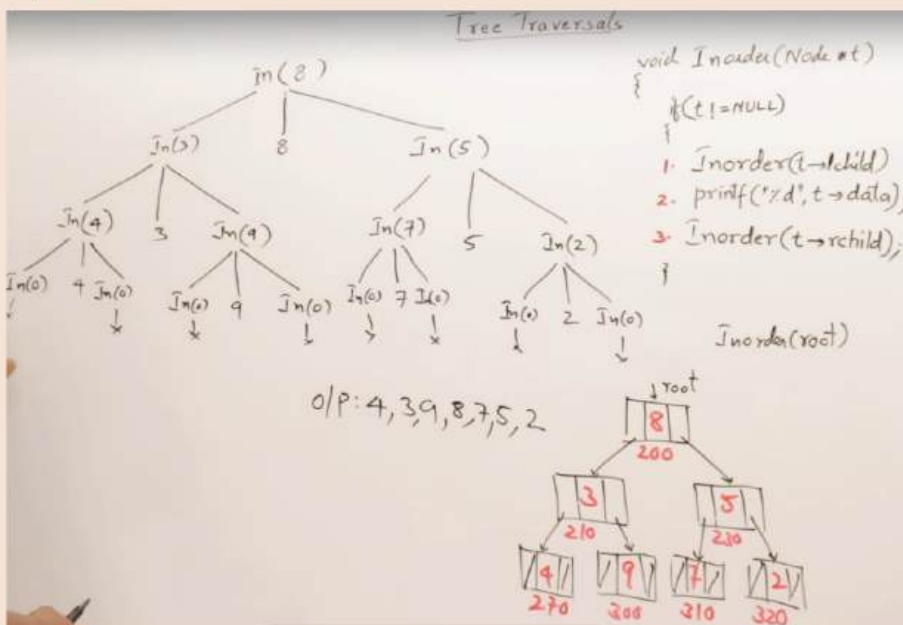
    preorder(root);
    cout << endl;

    inorder(root);
    cout << endl;

    postorder(root);
    cout << endl;

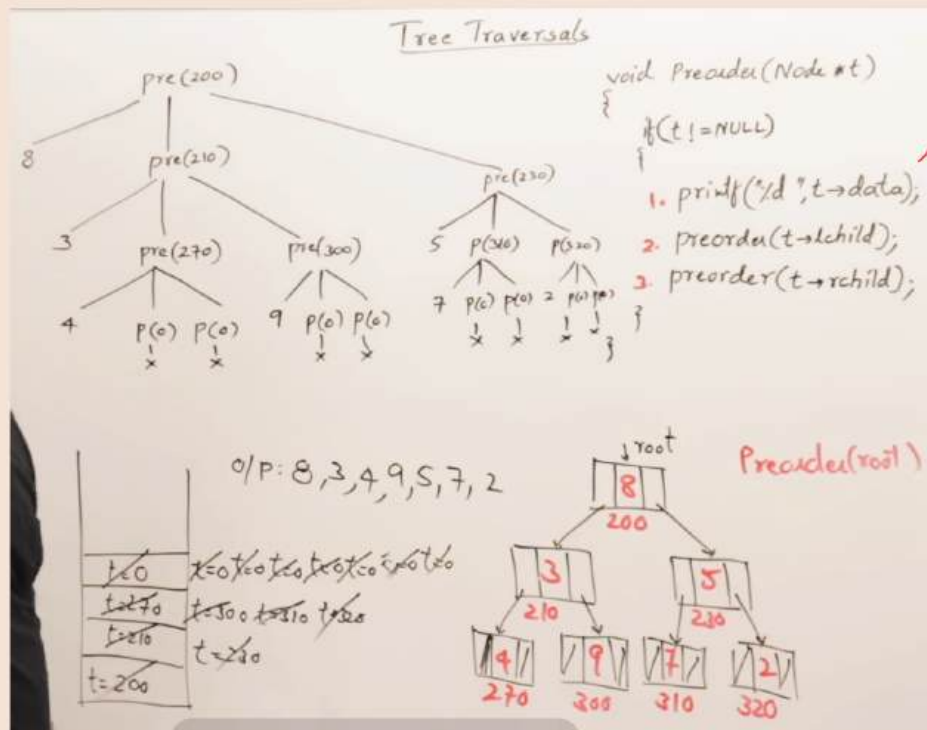
    return 0;
}
```

Tree Traversal : (In order tree traversal) :

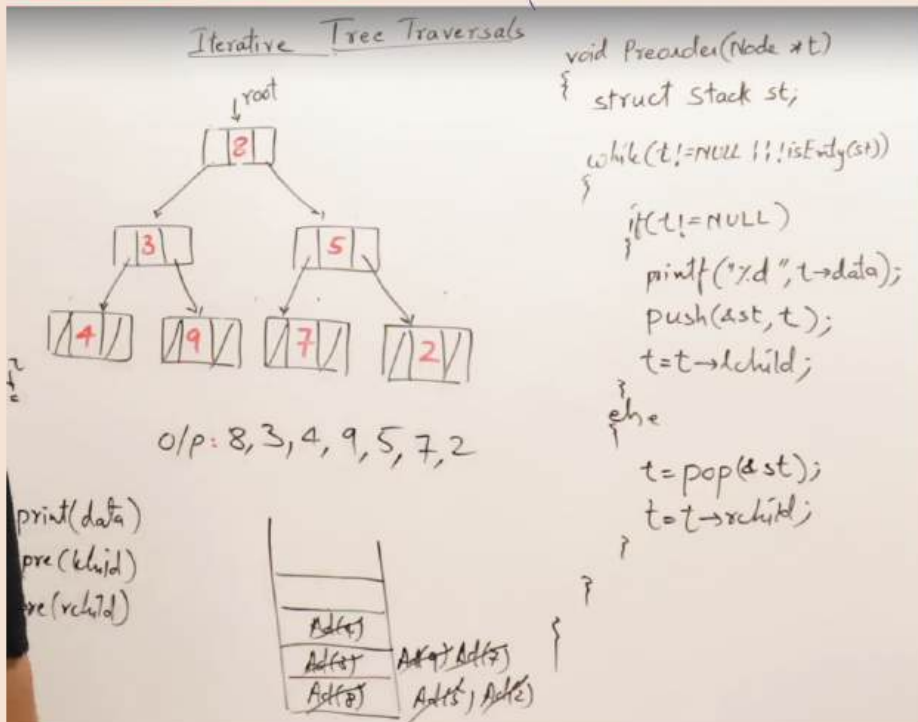


- This is a common tree traversal, it uses the recursion to traverse through three nodes.
 - where once the left child is traversed then print the nodes value and after that traverse through the right child of the current node.

Preorder tree traversal :-



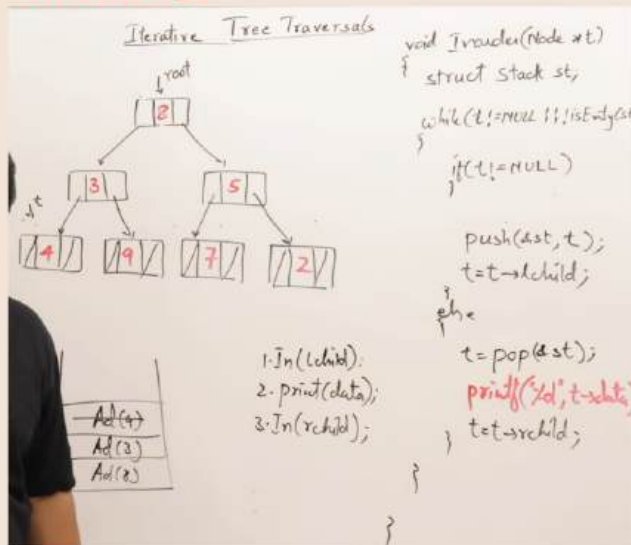
Iterative Tree traversal :- (Pre order)



- Instead of the using recursion in iterative tree traversal we use stack to hold the information of the traversed nodes. Here stack should be of address type here.

- for Iterative tree traversal we have to use the stack which holds the address values. by using stack we can save the nodes which traversed.

Inorder :-



Post order :-



- the post order tree traversal is a little bit tricky, here we have to iterate through its nodes before printing its value.
- therefore when we iterate 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);
        }
    }
}

```

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

Q: 8, 3, 9, 7, 6, 4, 5, 2

Level order is similar as inserting nodes into tree

=> How to generate the Tree from Traversals:-

Can we generate Tree from Traversals

$n = 3$

(A) (B) (C)

preorder - A, B, C
postorder - C, B, A

$\frac{2nC_n}{n+1}$

pre: ABC
po: CBA

① preorder
② postorder
③ inorder

$\frac{2nC_n}{n+1}$

preorder
postorder

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 find that we split the left and right side of nodes for that node. we do this until all the elements are iterated. this procedure takes the n^2 time complexity.

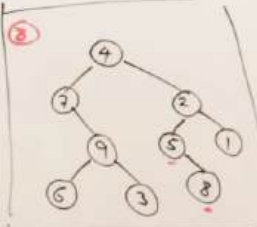
Generating Tree from Traversals

$O(n^2)$

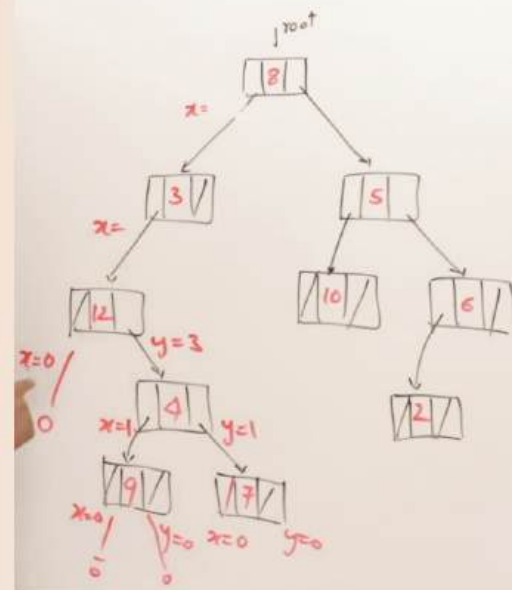
$n \rightarrow$ Preorder — 4, 7, 9, 6, 3, 2, 5, 8, 1

$\frac{n}{n \times n = n^2}$ Inorder — 7, 6, 9, 3, 4, 5, 8, 2, 1

n — 7, 6, 9, 3, 4, 5, 8, 2, 1



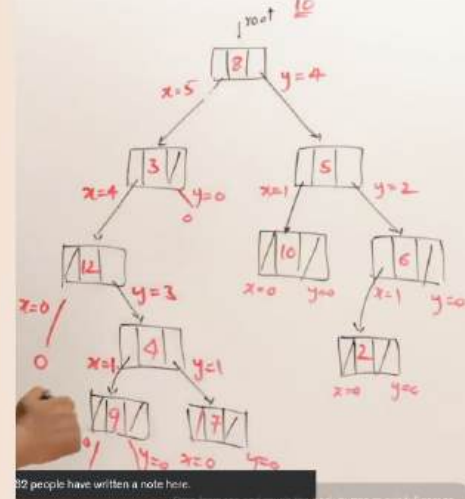
Counting Nodes



```
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;
}
```

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);
```

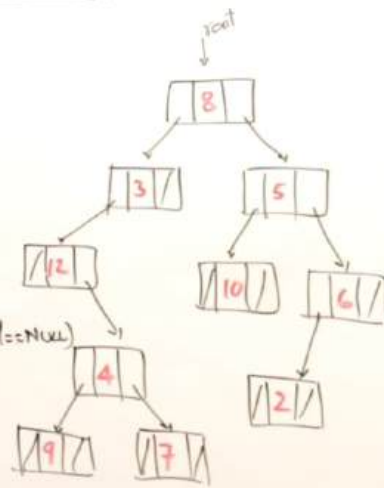
```
        if (p->lchild == NULL && p->rchild == NULL)
```

```
            return x + y + 1;
```

```
        else
```

```
            return x + y;
```

```
    }
    return 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)