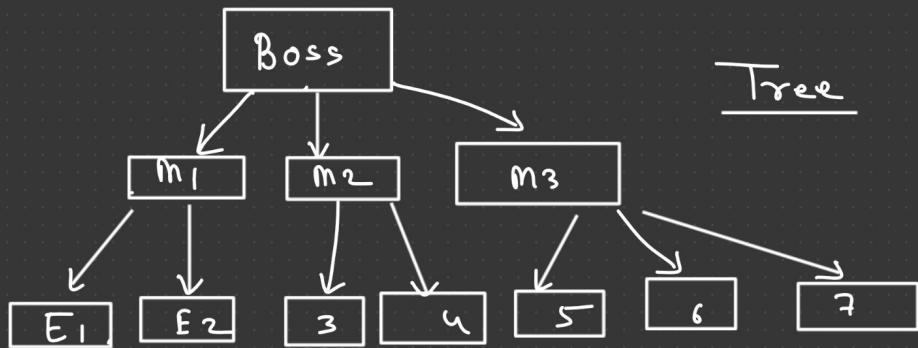


Tree

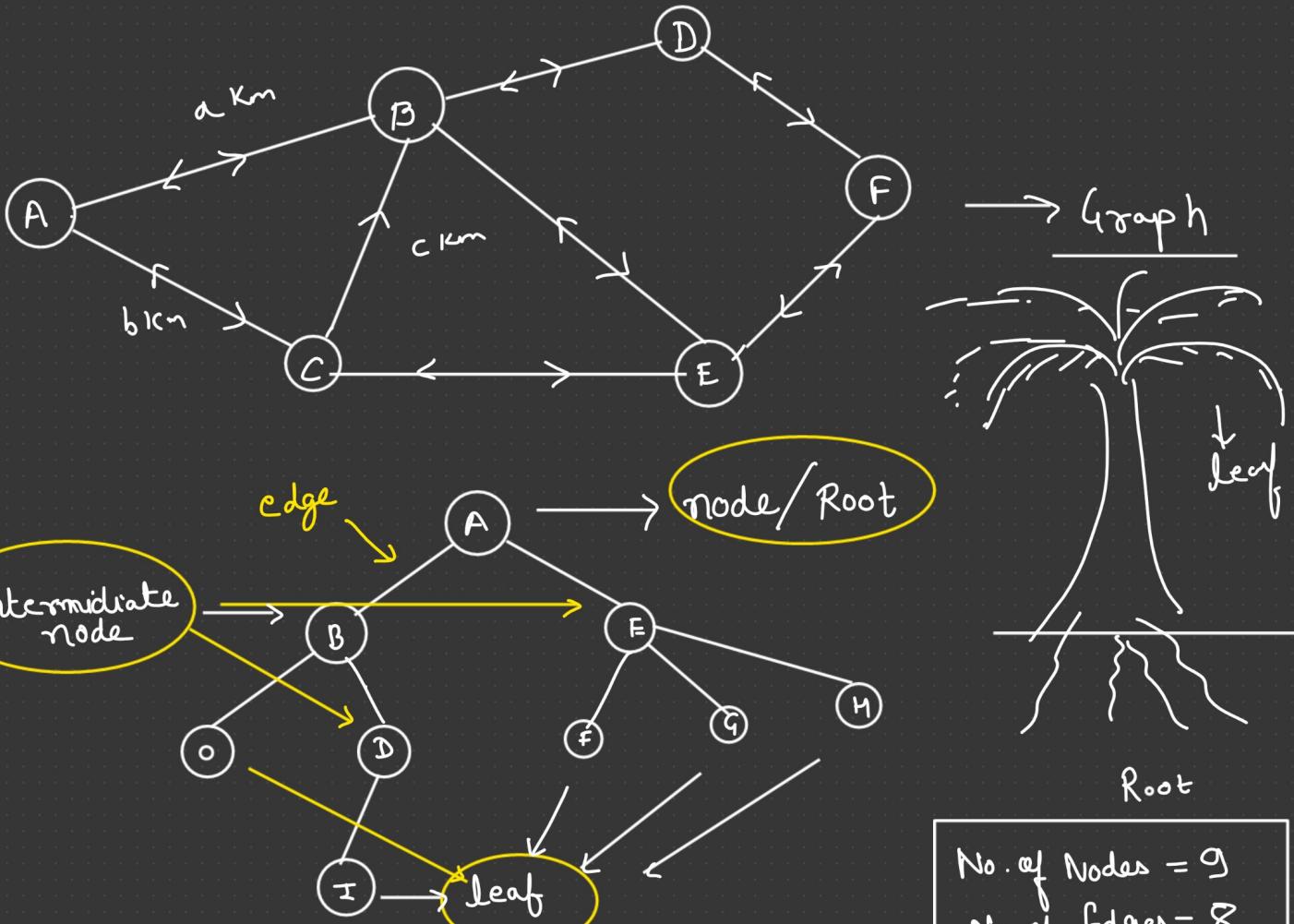
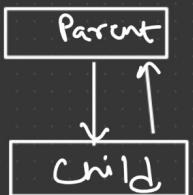
Prateek Jain

YouTube channel - Prateek Jain Academy
@iNeuron.ai

DS \leftrightarrow 1) Linear \rightarrow (Array, DA, Stack, Queue)
 DS \leftrightarrow 2) Non-Linear \rightarrow (Tree, Graph)



Tree



No. of Nodes = 9
No. of Edges = 8

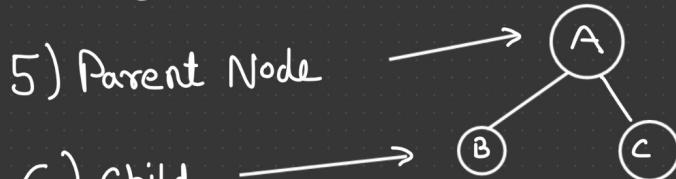
Leaf Node \Rightarrow which do not have any child node.

1) Root \rightarrow first node.

2) Leaf

3) Intermediate Node

4) Edges

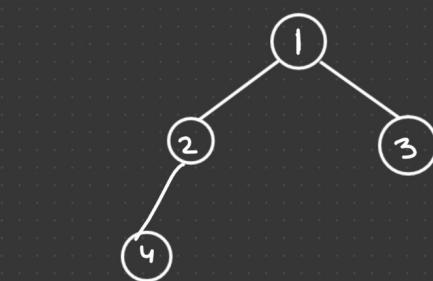


5) Parent Node

6) Child

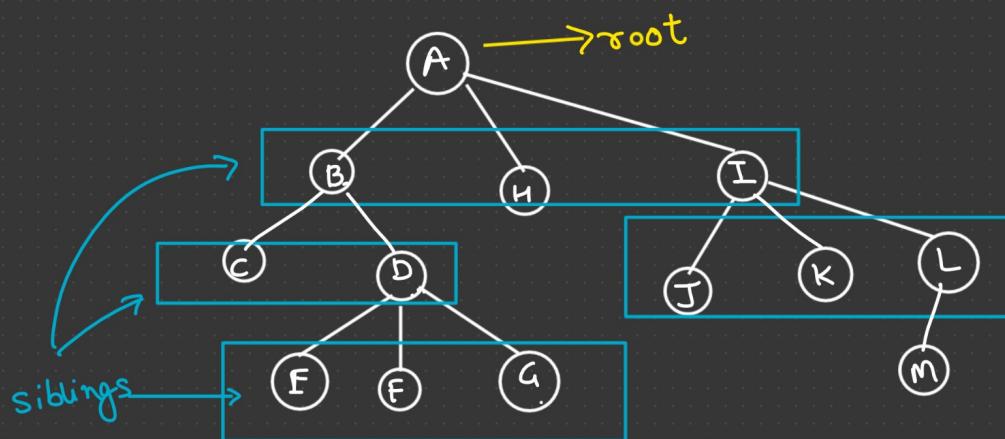
$$\begin{aligned} \# \text{ of Nodes} &= 3 \\ \# \text{ of Edges} &= 2 \end{aligned}$$

7) Siblings



If we have "n" Nodes in our tree then it must have " $n-1$ " edges.

$$\text{no. of edges} + 1 = \text{No. of Nodes}$$

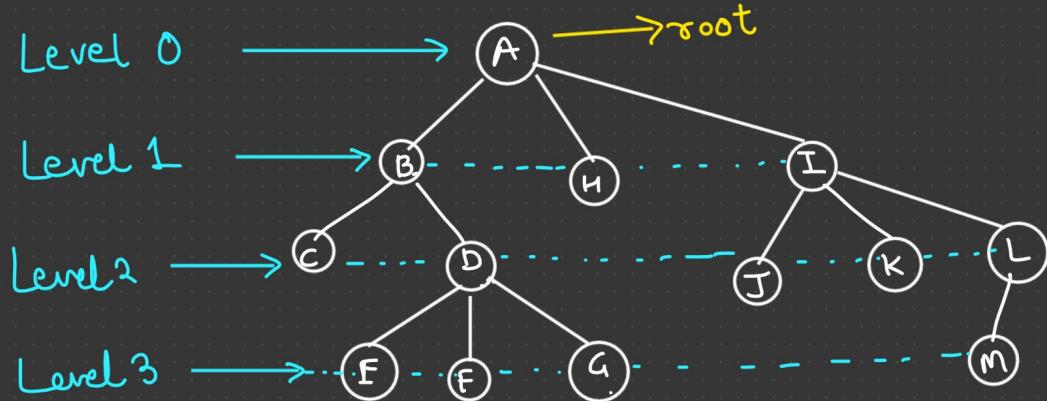


Root = A

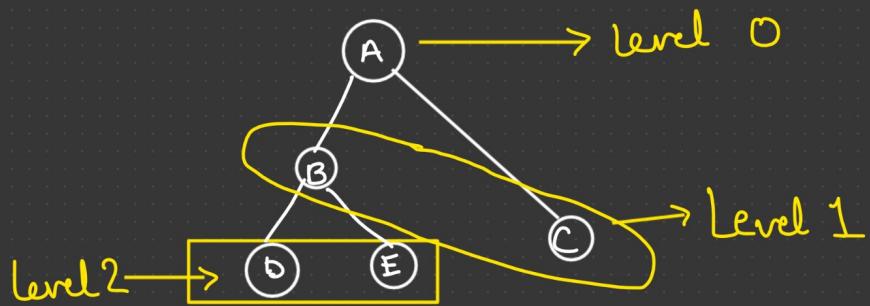
Leaf = C, E, F, G, H, J, K, M

Intermediate = B, D, I, L

Level :-

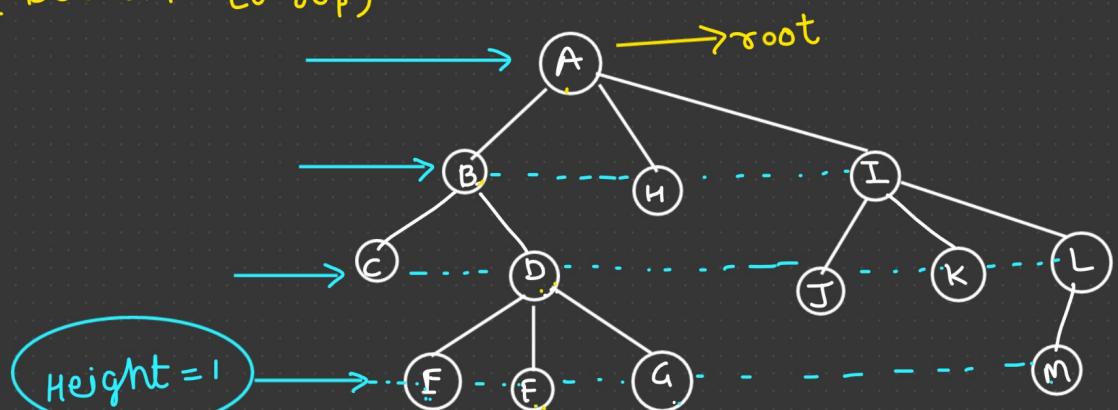


Total 4 Levels → [0 - 3]

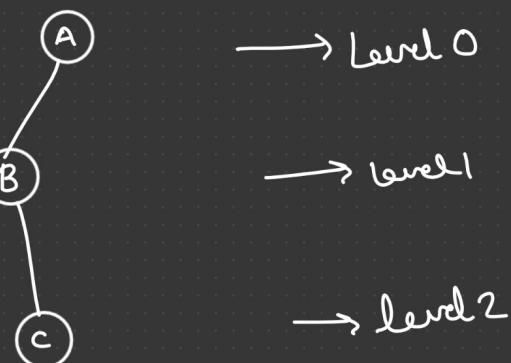
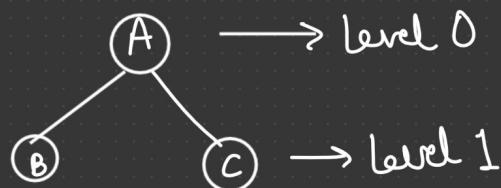


Height :- (Bottom → top)

edge / node
Height = 3 / 4



Depth :- (Top to Bottom)



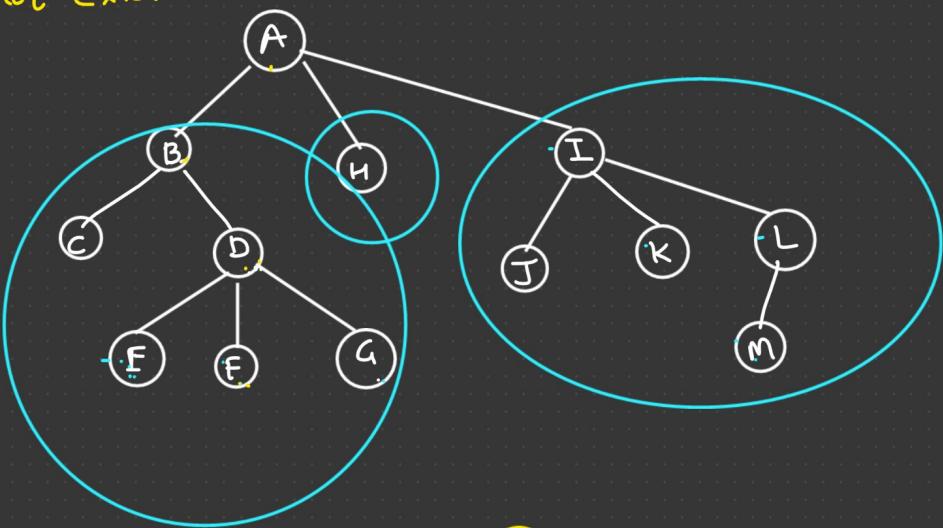
Path :-

(B - K) → Does not exist

(A - G) → A - B - D - G

(A - M) → A - I - L - M

(B - E) → B - D - E



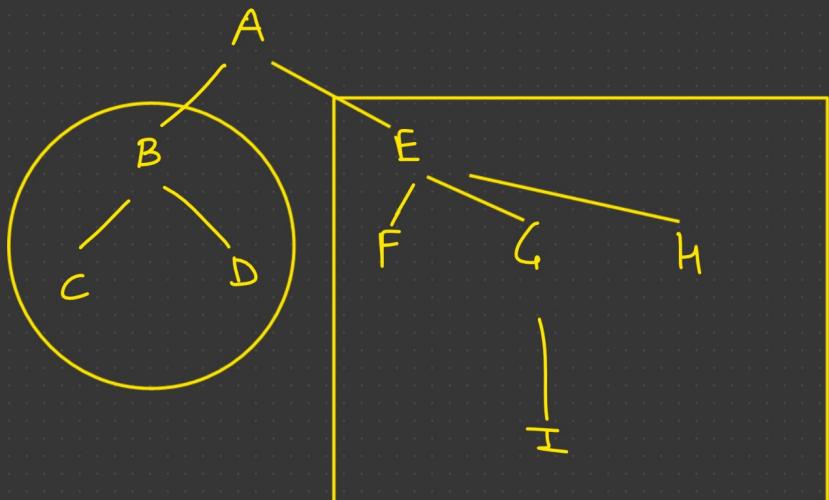
Subtree :-

○ → single node is also a tree.

$$A = \{1, 2, 3, 4, 5, 6\}$$

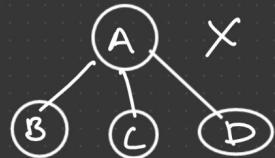
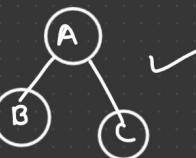
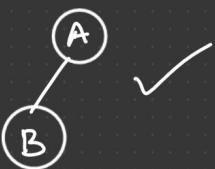
$$B = \{2, 3, 6\}$$

$$B \subseteq A$$

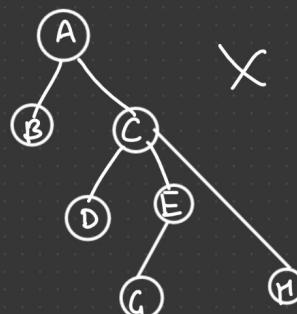
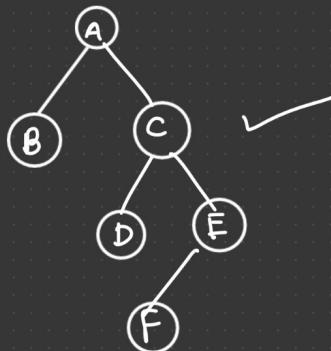


Types of Trees :-

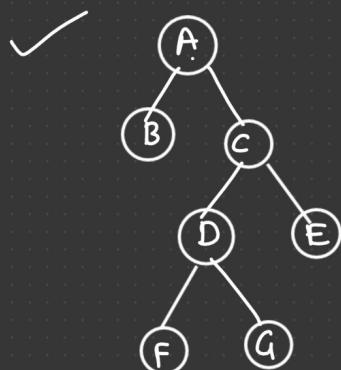
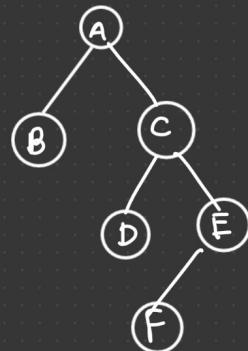
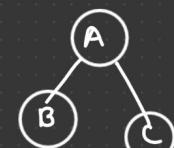
1) Binary Tree :- A tree is called BT if every node in it has maximum of two child nodes.



(Because it 0 child)



Strict Binary Tree :- Every node has exactly 0 or 2 child.

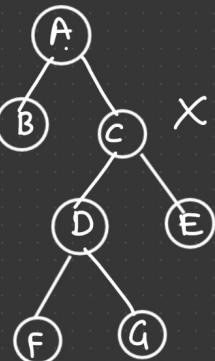
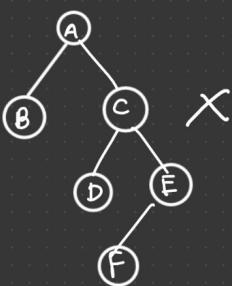


$\text{Leaf} = 4$
 $\text{non Leaf} = 3$

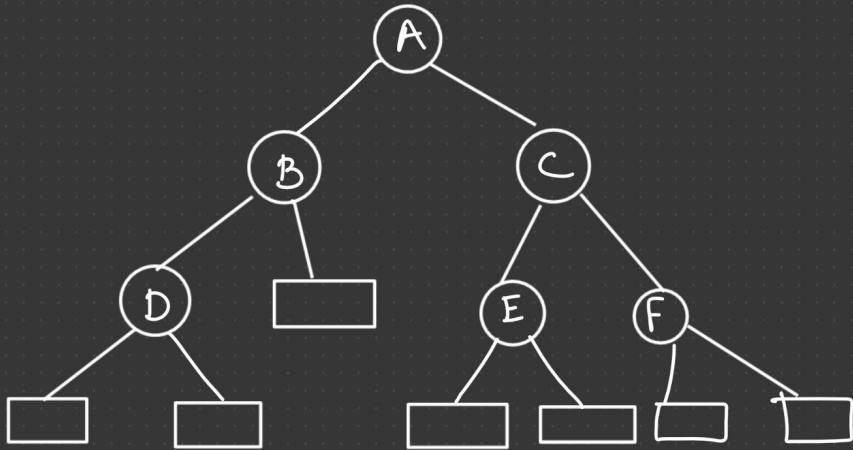


if it has "n" non leaf node then, it must have "n+1" leaf nodes.

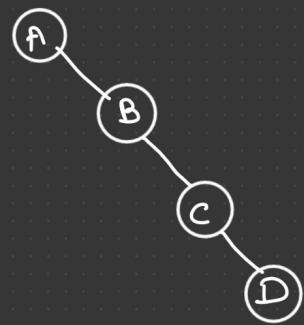
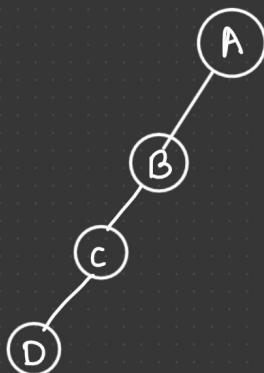
Complete BT :- Every internal node must have exactly 2 children. And all the leaf nodes are at the same level.



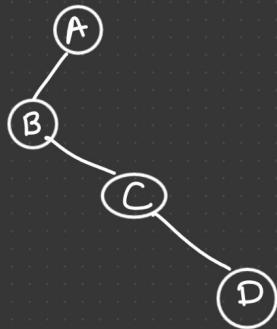
Extended BT:-



Skewed BT:-



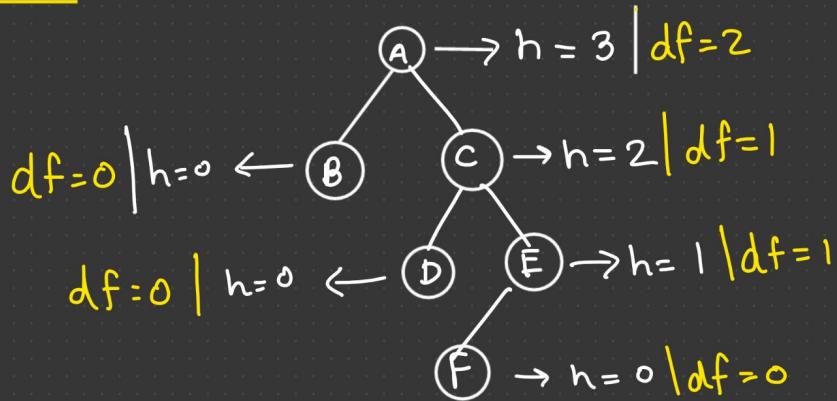
Degenerate Tree :-



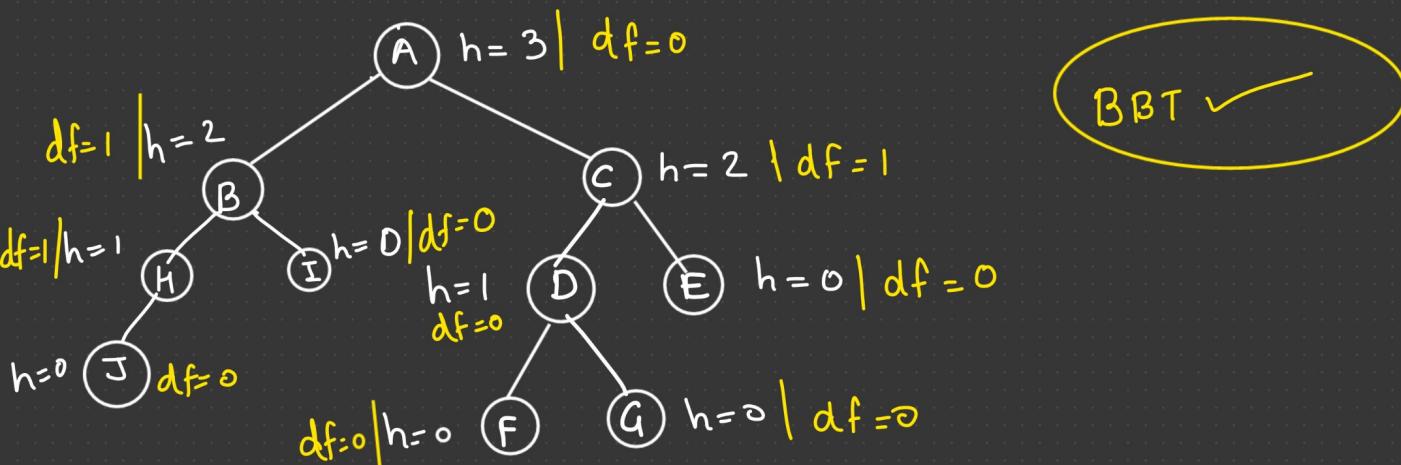
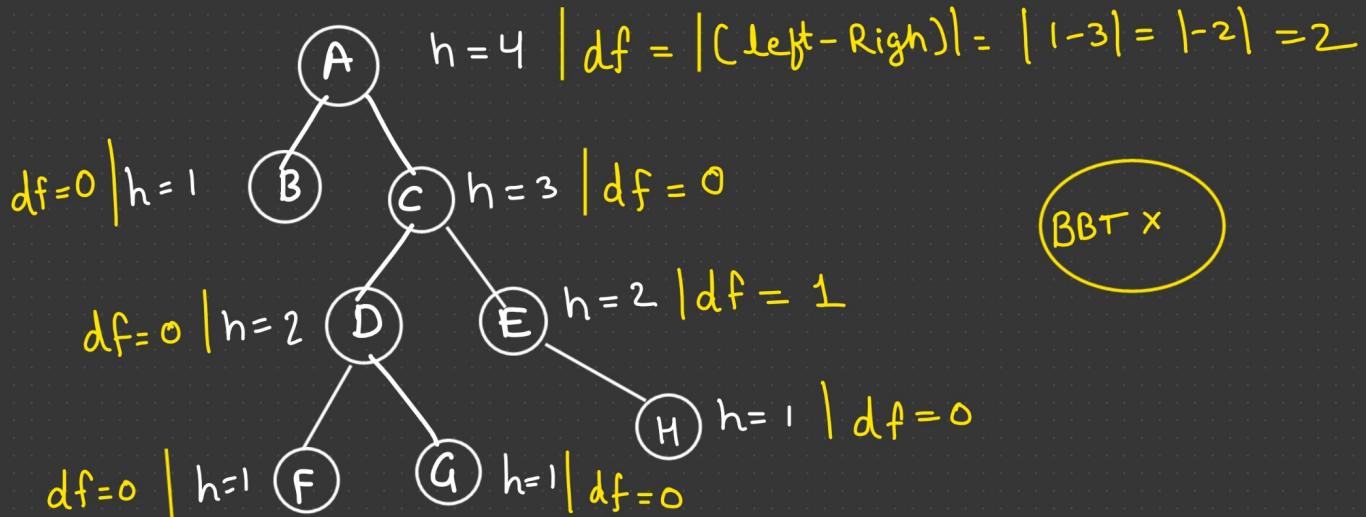
Left - skewed

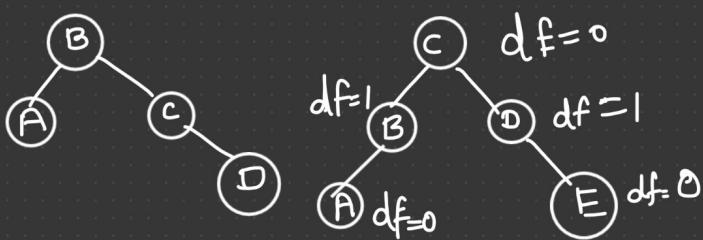
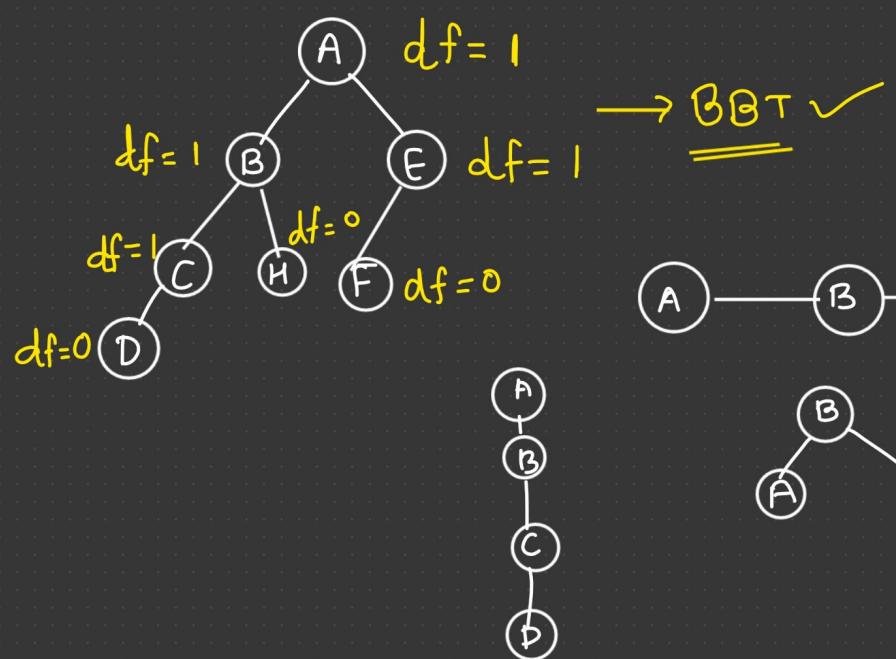
Right - skewed

Balanced BT :- if ($df = 0$ "or" 1) \rightarrow BBT



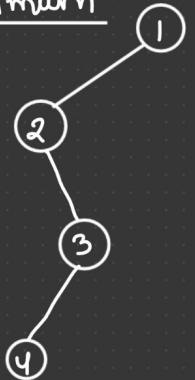
$$\text{difference } (df) = \left| (\text{height of Left Subtree}) - (\text{height of Right Subtree}) \right|$$



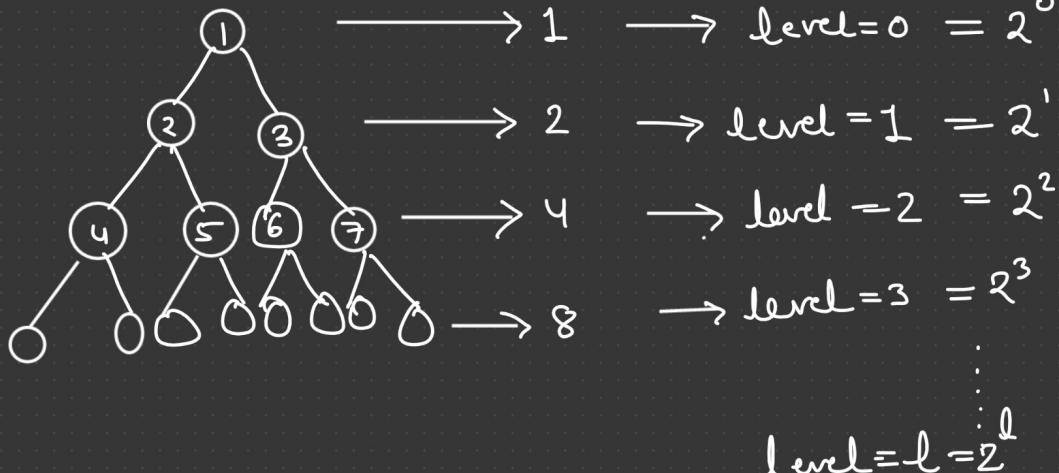


Properties of BT :-

1) The maximum number of nodes in a Binary Tree at level "l" is 2^l .

minimum  * minimum no. of nodes in BT at any level is 1.

degenerated BT



2) Minimum no. of nodes in a BT. of level L.

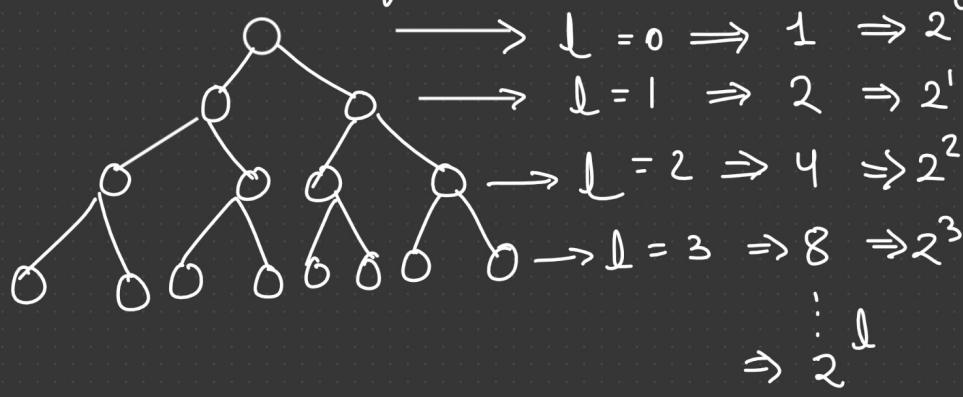
① \Rightarrow Level = 0
 No. of nodes = 1
 height = 1

Level = 1
 no. of nodes = 2
 height = 2

Level = 2
 no. of nodes = 3
 height = 3

Minimum no. of nodes $(L+1)$ or (h) .

3) Maximum no. of nodes in a BT.



$l=2$
 $h=3$
 $n=7$

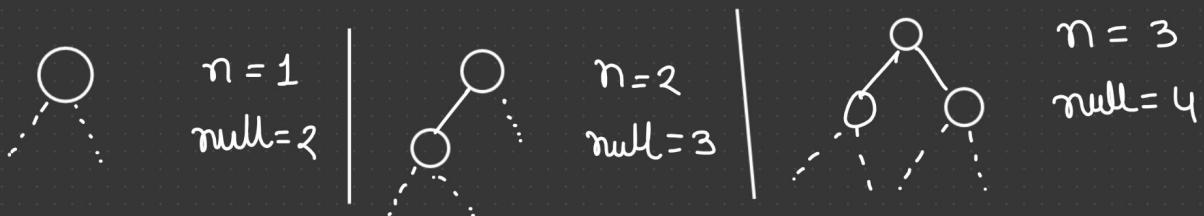
$\text{Max no. of nodes} = 2^h - 1 = 2^{l+1} - 1$
 $= 8 - 1 = 7$

Total no. of nodes = $1 + 2 + 4 + 8 + \dots + 2^l$

$$\text{Sum of G.P} = \frac{a(r^n - 1)}{(r - 1)} = 1 \cdot \frac{(2^{l+1} - 1)}{(2 - 1)} = 2^{l+1} - 1$$

4) $e = n - 1$

5) No. of Null Reference = no. of nodes + 1.



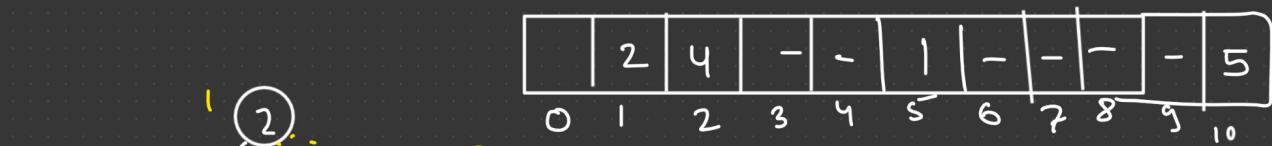
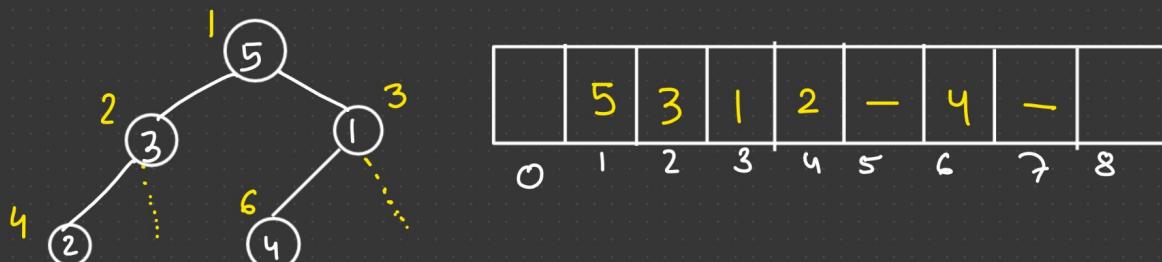
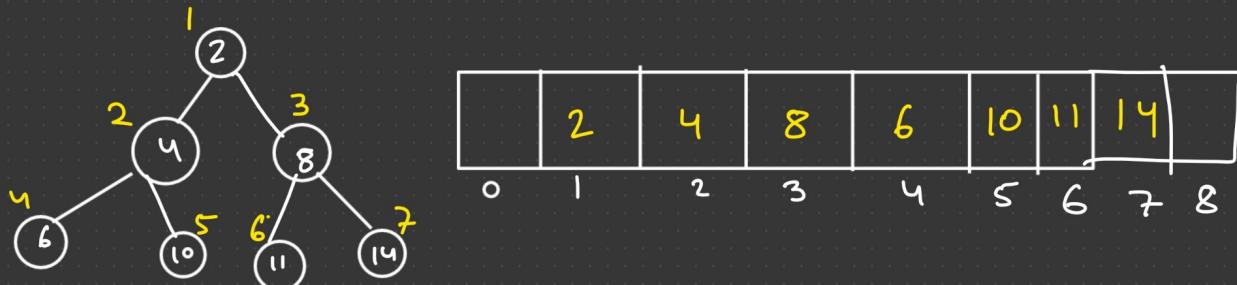
Binary Tree Representation :-

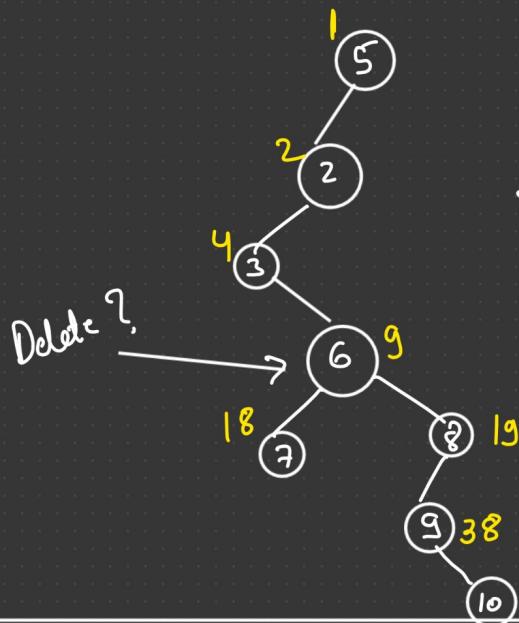
- 1) Array Representation
- 2) Linked list Representation

1 (Parent) = LC = 2 , RC = 3

2 (Parent) = LC = 4 , RC = 5

1) Array Representation :-





7 elements \Rightarrow 38 nodes

if (K is parent node index)
then
 $2K = \text{left child index}$
and
 $2K+1 = \text{right child index}$

* K is index of child node then its parent node is at $\lfloor K/2 \rfloor$.

$$\lfloor 3 \cdot 5 \rfloor = 3, \lfloor 3 \cdot 1 \rfloor = 3, \lfloor 4 \cdot 9 \rfloor = 4, \lfloor 5 \rfloor = 5$$

```

class Tree
{
    int a[100];

    void insert(int v, int n)
    {
        a[n] = v;
    }

    void insertLeftChild(int v, int pi)
    {
        a[2*pi] = v;
    }
}

```

```

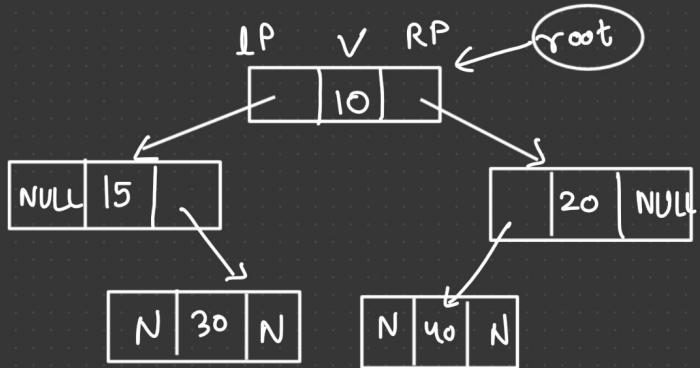
Void insertRightChild(int v, int pi)
{
    a[2*pi+1] = v;
}

```

Drawbacks of Array implementation :-

- 1) we can't change the size of Tree dynamically.
- 2) Wastage of memory.

2) Linked list Representation :-



```

class node
{
public:
    int data;
    node *left;
    node *right;
    node(int v)
    {
        data = v;
        left = NULL;
        right = NULL;
    }
};

```

```

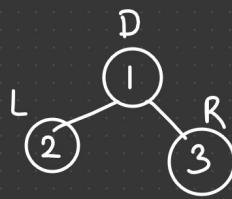
int main()
{
    node *root = new node(10);
    root->left = new node(15);
    root->right = new node(20);
    root->left->right = new node(30);
    root->right->left = new node(40);
}

```

This code can be used for testing any function.

Tree Traversal :-

L, R, D



1, 2, 3

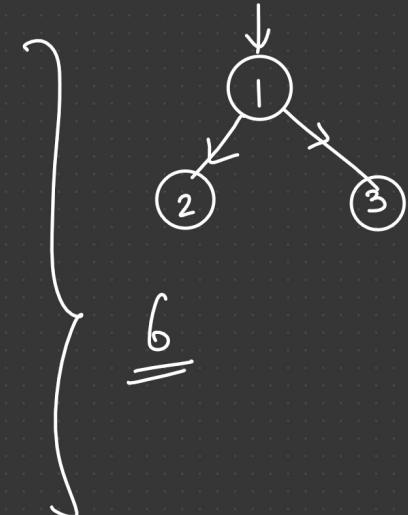
1, 3, 2

2, 1, 3

2, 3, 1

3, 1, 2

3, 2, 1



D, L, R
D, R, L

→ 1 [Root] → Preorder

L, D, R
R, D, L

→ 2 [Root] → Inorder

L, R, D
R, L, D

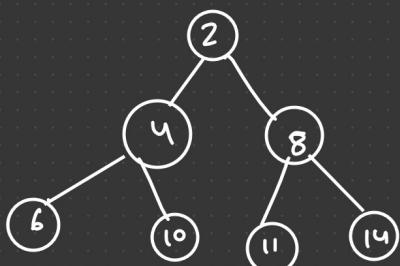
→ 3 [Root] → Postorder

→ LevelOrder

Preorder Traversal :-

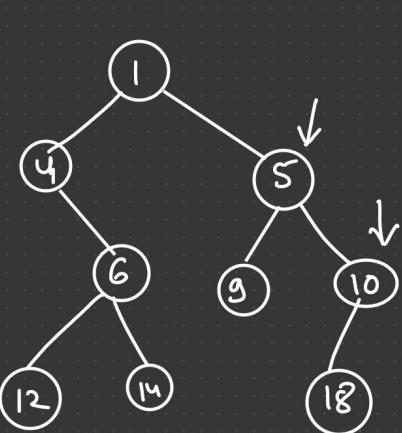
(Root → _{Subtree} left → _{Subtree} Right) (It is same as DFS)

(we can use stack)

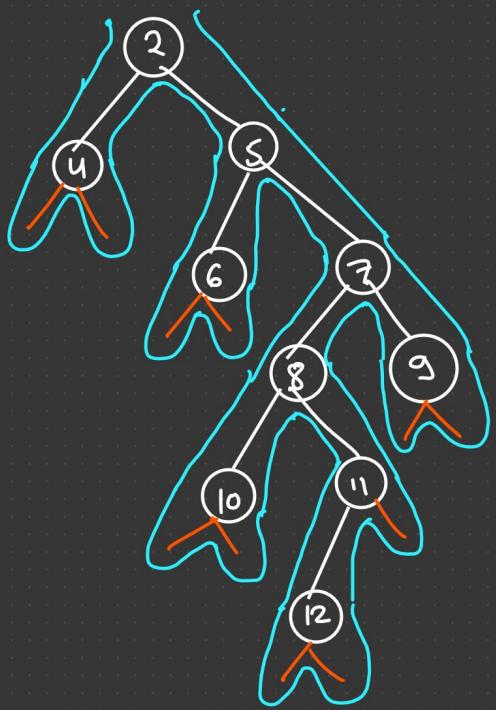


2, 4, 6, 10, 8, 11, 14

1, 4, 6, 12, 14, 5, 9, 10, 18 → Preorder



4, 12, 6, 14, 1, 9, 5, 18, 10 → Inorder.



2, 4, 5, 6, 7, 8, 10, 11, 12, 9 → Preorder

1st (Preorder) [D - L - R]

2, 4, 5, 6, 7, 8, 10, 11, 12, 9

2nd (Inorder) [L - D - R]

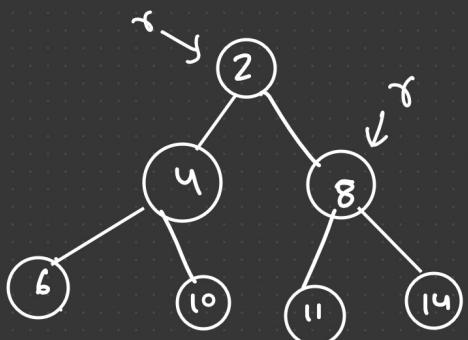
4, 2, 6, 5, 10, 8, 12, 11, 7, 9

3rd (Postorder) [L - R - D]

4, 6, 10, 12, 11, 8, 9, 7, 5, 2

Inorder Traversal :-

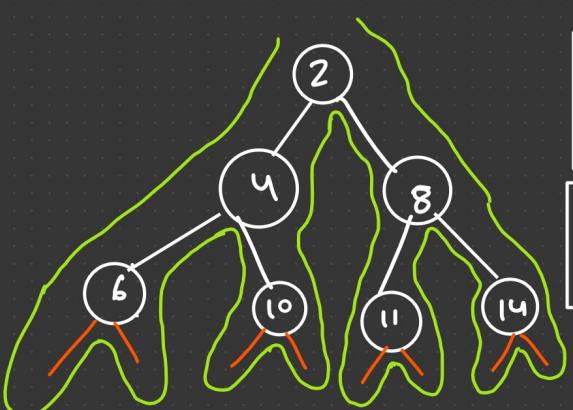
(Left → Root → Right)



6, 4, 10, 2, 11, 8, 14

Postorder Traversal :-

(Left → Right → Data)
Subtree Subtree



6, 10, 4, 11, 14, 8, 2

(3rd time)

← post order traversal.

6, 4, 10, 2, 11, 8, 14

← Inorder
(2nd time)

6, 4, 10, 2, 11, 8, 14

Preorder Traversal :- [D → L → R]

Void Preorder(Node *root)

{

if (root != NULL)

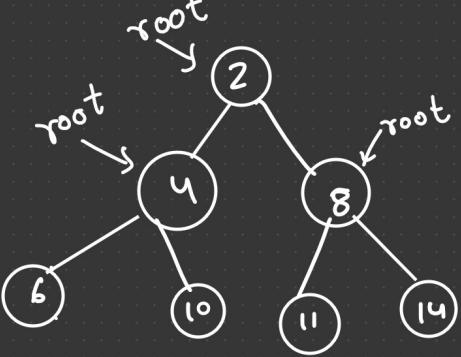
{

cout << root → data << " " ;

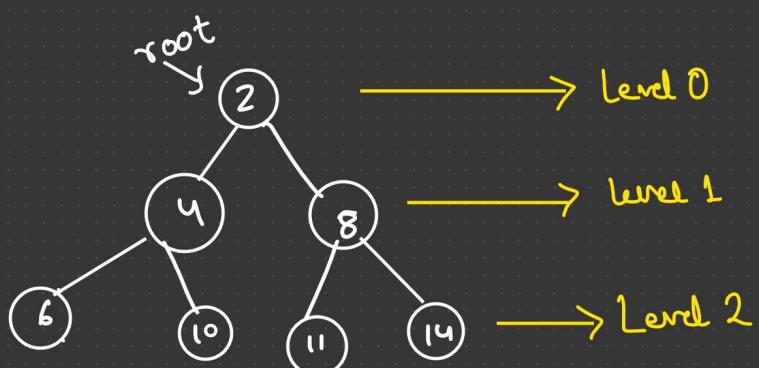
preorder (root → left) ;

} preorder (root → right) ;

}



Level order Traversal :- (It is same as BFS)

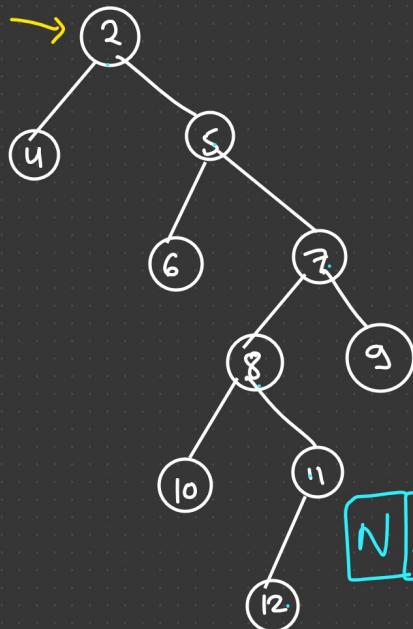


2, 4, 8, 6, 10, 11, 14



* we will use Queue DS to implement Level order Traversal.

2, 4, 8, 6, 10, 11, 14



↓ rear

↓ front

2, 4, 5, 6, 7, 8, 9, 10, 11, 12

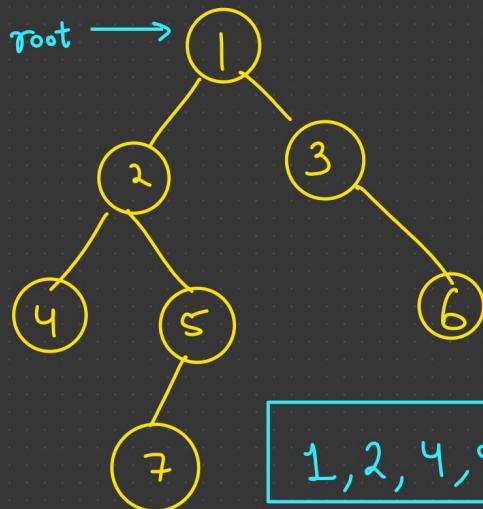
count = 1
2
3
4
5
6



```

void preorder( Node *root )
{
    if( root != NULL )
    {
        cout << root->data;
        preorder( root->left );
        preorder( root->right );
    }
}
int main()
{
    preorder( root );
}

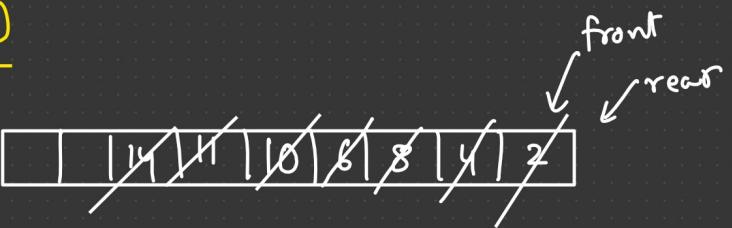
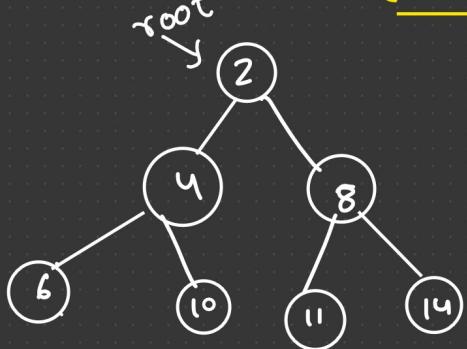
```



1, 2, 4, 5, 7, 3, 6

Level order Traversal :- (It is same as BFS)

(Non- Recursive)



2, 4, 8, 6, 10, 11, 14,

```
void printLevelOrder( Node *root)
```

```
{
```

```
    if (root == NULL)
```

```
        return;
```

```
    queue<Node*> q;
```

```
    q.push(root);
```

```
    while ( q.empty() == false )
```

```
{
```

```
    Node *temp = q.front();
```

```
    cout << temp->data;
```

```
    q.pop();
```

```
    if ( temp->left != NULL )
```

```
        q.push(temp->left);
```

```
    if ( temp->right != NULL )
```

```
        q.push(temp->right);
```

```
}
```

```
}
```

Height of Tree :- (Recursive).

Height of Tree \Rightarrow Height (1) = 5

$$\text{Height}(4) = 3$$

$$\text{Height}(1) = \left[\begin{array}{l} \text{lTree} = \text{Height}(2) = 4 \\ \text{rTree} = \text{Height}(3) = 3 \end{array} \right]$$

```

if(lTree > rTree)
    return lTree + 1;
else
    rTree + 1;
}

```

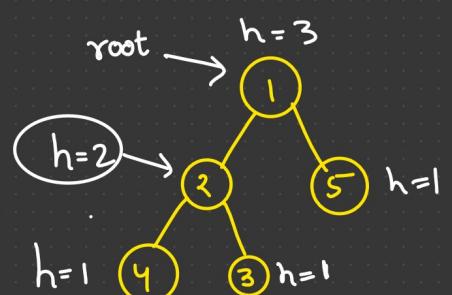
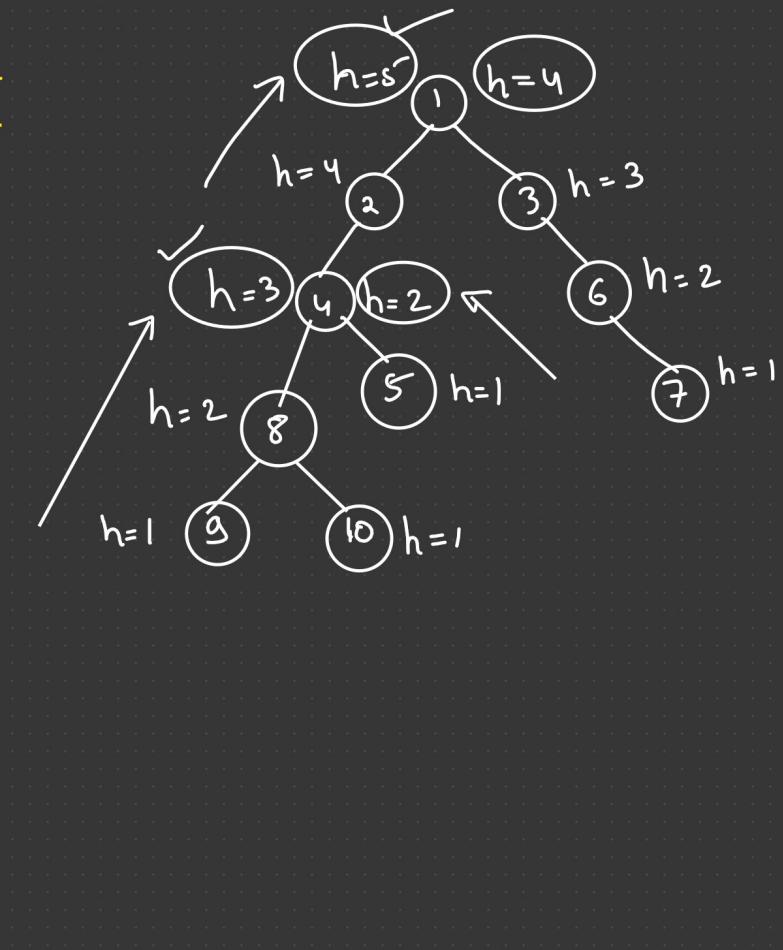
```

int Height( Node *root)
{
    if (root == NULL)
        return 0;

    else {
        int lHeight = Height( root->left);
        int rHeight = Height( root->right);

        if ( lHeight > rHeight)
            return lHeight+1;
        else
            return rHeight +1;
    }
}

```



```

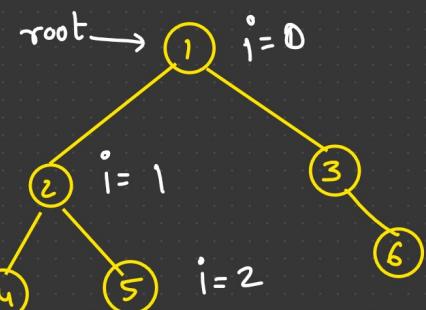
H(1) = 3
L = H(2) => 2
R = H(5) = 1
return 3

```

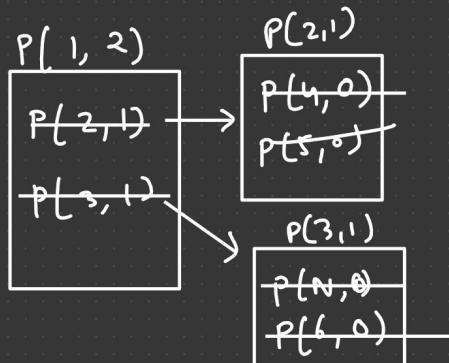
Level order Traversal (Recursive) :-

```
void printLevelOrder ( Node *root )
{
    int h = Height ( root );      = 3
    for ( int i = 0; i < h; i++ )
        printCurrentLevel ( root, i );
}

void printCurrentLevel ( Node *root, int level )
{
    if ( root == NULL )
        return;
    if ( level == 0 )
        cout << root->data << " ";
    else if ( level > 0 )
        printCurrentLevel ( root->left, level - 1 );
        printCurrentLevel ( root->right, level - 1 );
}
```

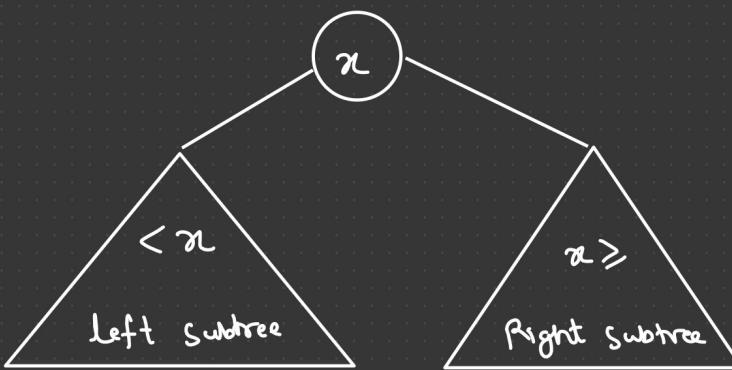
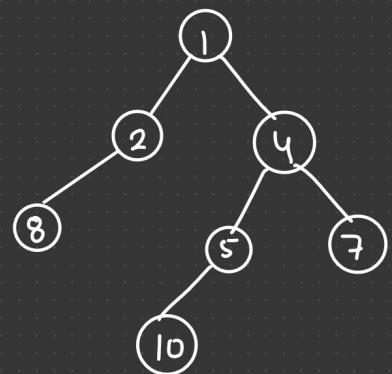


$\frac{1, 2, 3}{\checkmark}, \frac{4, 5, 6}{\checkmark}$

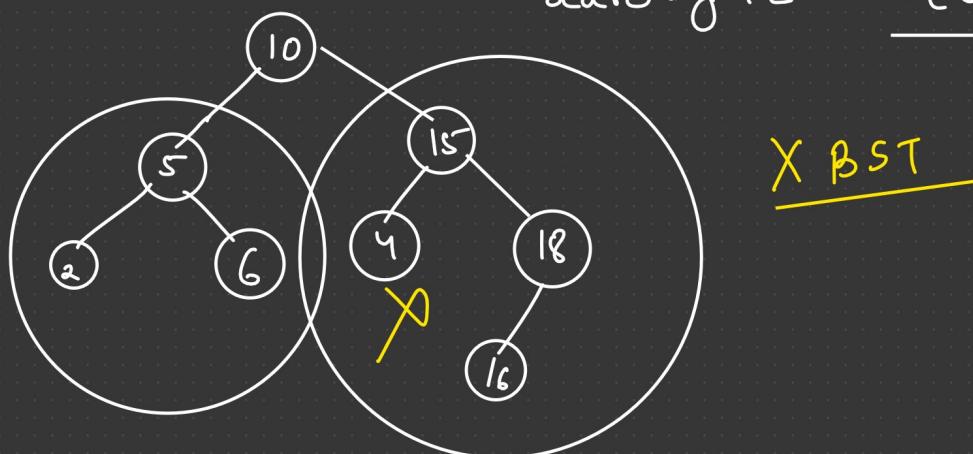


Binary Search Tree

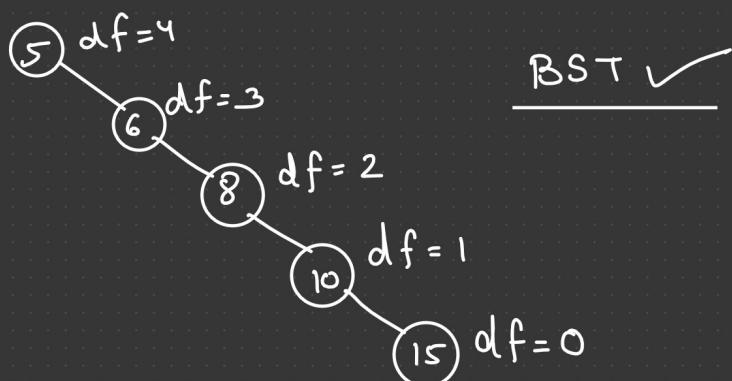
Time Complexity :- $O(n)$



Searching TC = $O(\log n) < O(n)$



$$n + \frac{n}{2} + \frac{n}{4} + \dots + 1 = \log n$$

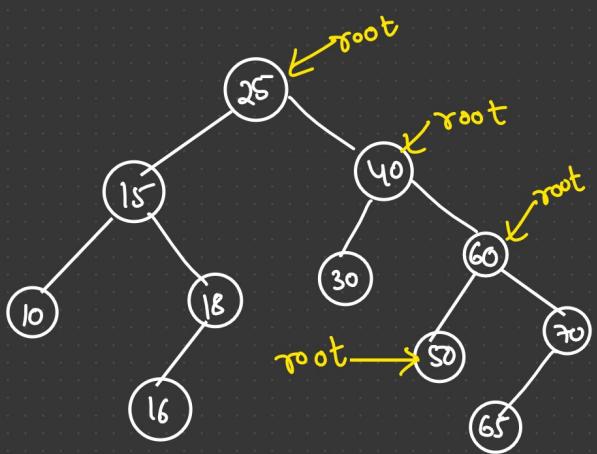


Problem:-

So, if BST is a skew tree then it will take $O(n)$ in searching.

Searching :-

Key = 50



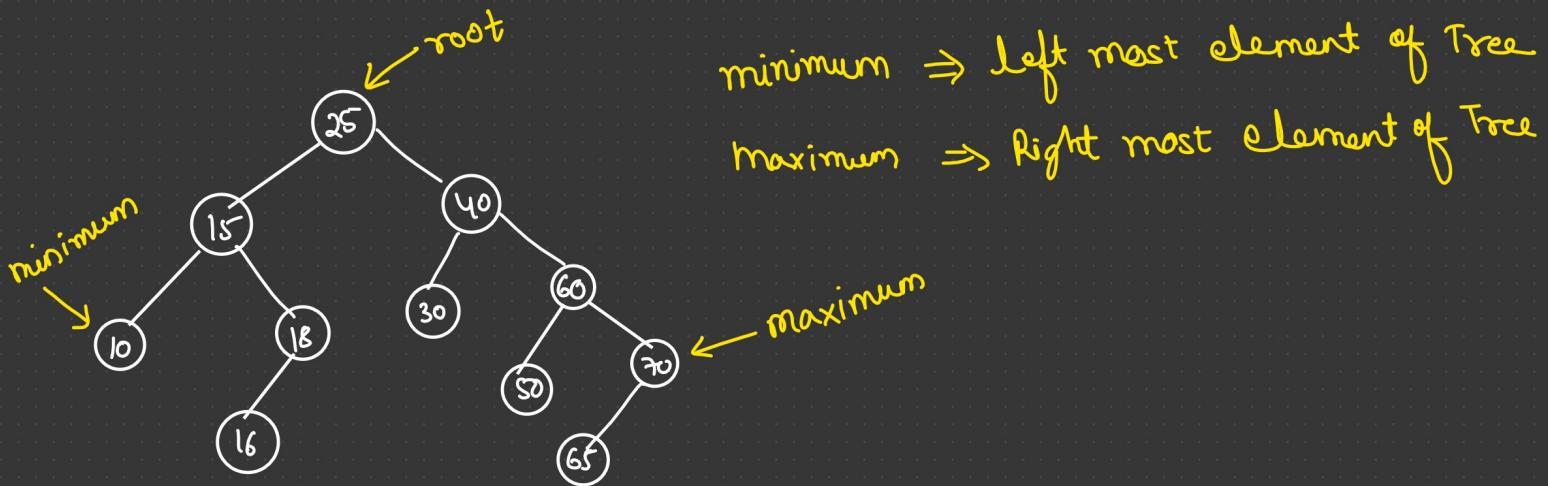
```

Node *Search( Node *root, int key)
{
    if( root == NULL )
        return NULL;
    if( root->data == key )
        return root;
    else if( root->data > key )
        return search( root->left, Key );
    else
        return search( root->right, Key );
}
  
```

```

class Node
{
    int data;
    Node *left;
    Node *right;
};
  
```

Minimum & Maximum value in a BST :-



minimum \Rightarrow Left most element of Tree
 maximum \Rightarrow Right most element of Tree

```

Node * MinElement( Node *root )
{
    if (root == NULL)
        return NULL;
    else if (root->left == NULL)
        return root;
    else
        minElement( root->left );
}
  
```

```

Node * temp = root;
if (temp == NULL)
    return NULL;
while (temp->left != NULL)
{
    temp = temp->left;
}
return temp;
  
```

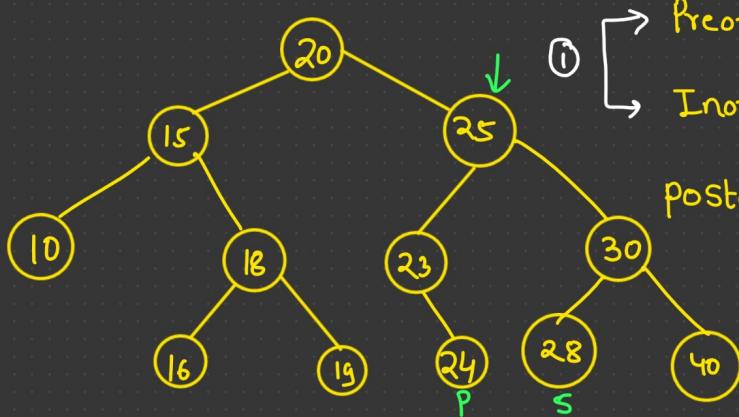
```

Node * MaxElement( Node *root )
{
    if (root == NULL)
        return NULL;
    else if (root->right == NULL)
        return root;
    else
        maxElement( root->right );
}
  
```

```

Node * temp = root;
if (temp == NULL)
    return NULL;
while (temp->right != NULL)
{
    temp = temp->right;
}
return temp;
  
```

Inorder Successor & Inorder predecessor of BST :-



Preorder = 20, 15, 10, 18, 16, 19, 25, 23, 24, 30, 28, 40

Inorder = 10, 15, 16, 18, 19, 20, 23, 24, 25, 28, 30, 40

postorder = 10, 16, 19, 18, 15, 24, 23, 28, 40, 30, 25, 20

Inorder = 10, 15, 16, 18, 19, 20, 23, 24, 25, 28, 30, 40

Inorder Successor of (E) = Minimum Element of its Right Subtree.

Inorder Predecessor of (E) = Maximum Element of its Left Subtree.

(1 step)

[Node → Right → Left most]

[Node → Left → Right most]

(1 step)

Node * InorderSuccessor (Node * node)

{

return MinElement (node → right);

{

Node * InorderPredecessor (Node * node)

{

return MaxElement (node → left);

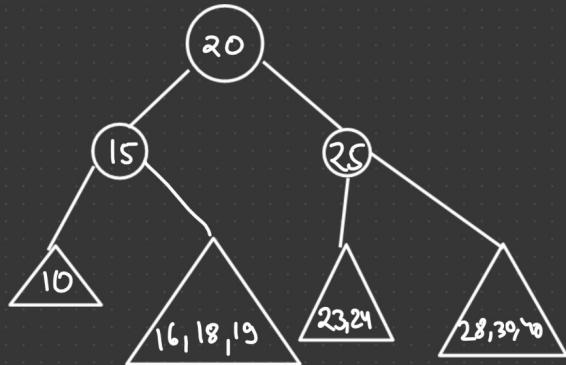
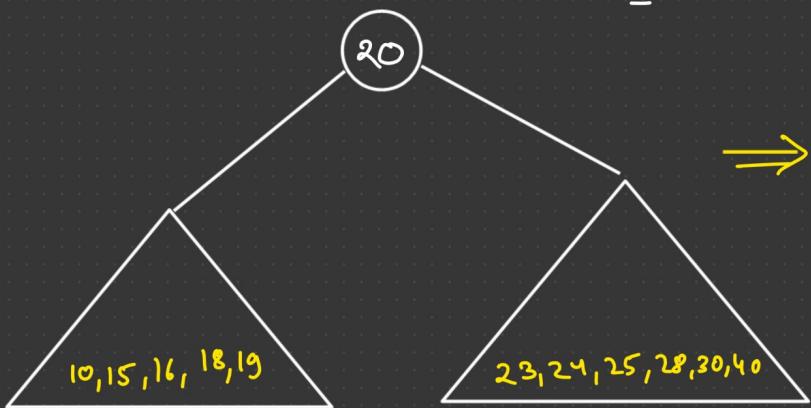
b

Create BST using Inorder & preorder Traversal :-

Preorder = $[20, \underline{15}, \underline{10}, 18, 16, 19, \underline{25}, \underline{23}, 24, 30, 28, 40]$

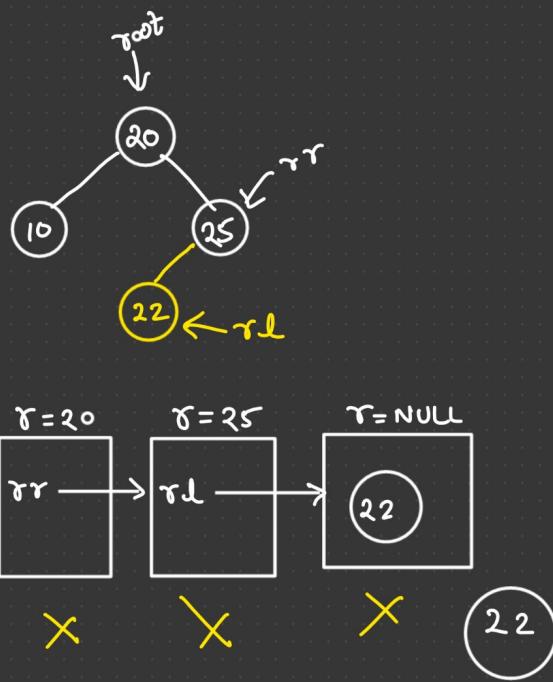
Postorder & Inorder

Inorder = $\underline{\underline{10, 15, 16, 18, 19}}, \boxed{20}, \underline{23, 24, 25, 28, 30, 40}$



Preorder & Postorder = ? (Not possible)

Creation / Insertion in BST:-



Node * Insert (Node *root, int value)
{

if ($\text{root} == \text{NULL}$)

return new Node (value);

if ($\text{root} \rightarrow \text{data} < \text{value}$)

$\text{root} \rightarrow \text{right} = \text{Insert} (\text{root} \rightarrow \text{right}, \text{value});$

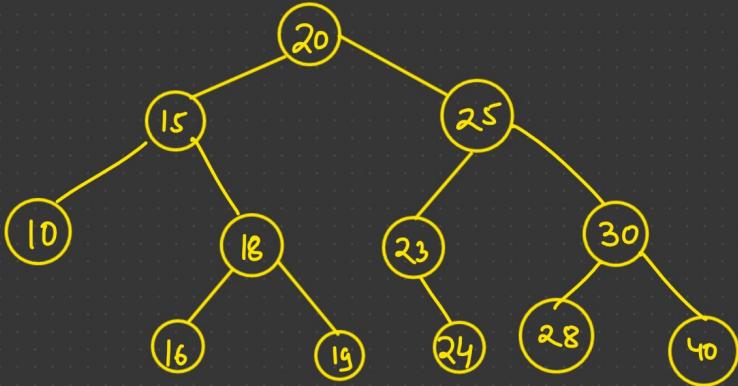
if ($\text{root} \rightarrow \text{data} \geq \text{value}$)

$\text{root} \rightarrow \text{left} = \text{Insert} (\text{root} \rightarrow \text{left}, \text{value});$

1) Time complexity of Inserting a node = $O(\log n)$

2) Time complexity to create BST = $O(n \log n)$

Deletion in BST :-



1) Leaf Node (zero child)

2) Non-Leaf Node (one child)

3) ————— (two child)

Node* delete(Node* root, int value)

{

if(root == NULL)

return root;

if(root->data < value)

root->right = delete(root->right, value);

else if(root->data > value)

root->left = delete(root->left, value);

else { if((root->left == NULL) && (root->right == NULL)) //zero child

{ delete root;

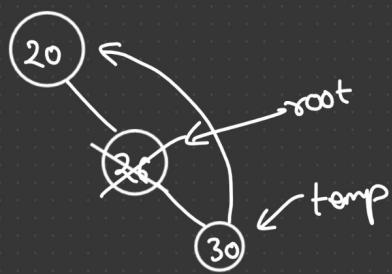
{ return NULL;

else if(root->left == NULL)

{ Node * temp = root->right;

delete root;

{ return temp;



else if ($\text{root} \rightarrow \text{right} == \text{NULL}$)

{
 Node * temp = $\text{root} \rightarrow \text{left}$;
 delete root ;
 return temp;
}

else { // two child

 Node * temp = minElement($\text{root} \rightarrow \text{right}$);

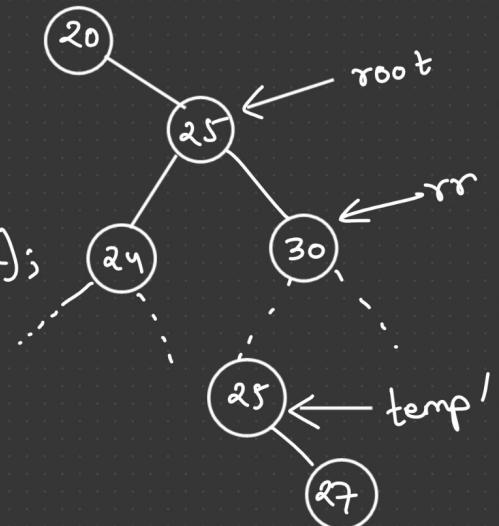
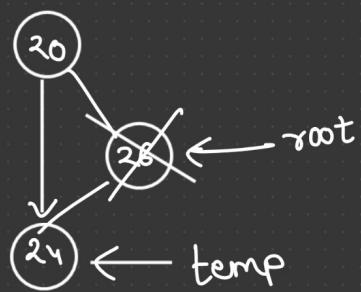
$\text{root} \rightarrow \text{data} = \text{temp} \rightarrow \text{data}$;

$\text{root} \rightarrow \text{right} = \text{delete}(\text{root} \rightarrow \text{right}, \text{temp} \rightarrow \text{data})$;

}

return root ;

}



AVL Tree :- It is same as BST + Balanced Tree



→



df=1

✓

AVL

✓

AVL



df=1

✓

AVL

✓

✓

1

2

3

4

5

6

1) 1 2 3



-2



0

1

2) 1 3 2

X

3) 2 1 3

RR

4) 2 3 1

2

5) 3 1 2

if(C.N->R < Key)

6) 3 2 1

if(C.N->R > Key)

	1	2	3	4	5	6
1) 1 2 3						
2) 1 3 2	X					
3) 2 1 3	RR	X				
4) 2 3 1	2	RL				
5) 3 1 2		4				
6) 3 2 1	if(C.N->R < Key)	if(C.N->R > Key)				

✓

AVL

✓

AVL

X

LR

3

if(C.N->L < Key)

2

1

LL

1

if(C.N->L > Key)

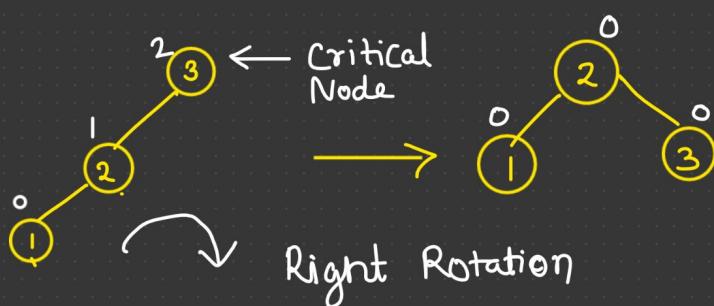
Balance factor = Height of left Tree - Height of Right Tree

AVL ✓ \Rightarrow BF = 0, +1, -1

* AVL Tree is self Balancing Tree in nature.

* AVL = Adelson Velskii & Landis

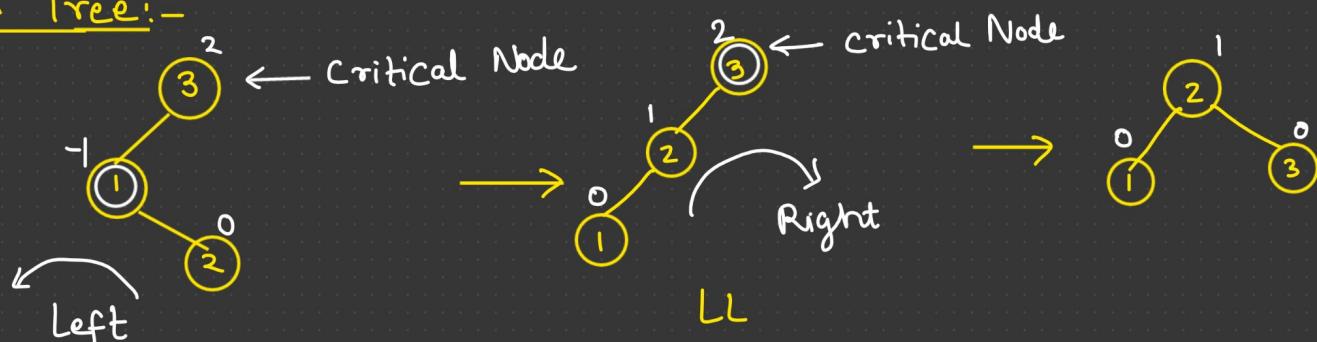
1) LL Tree :-



2) RR Tree :-



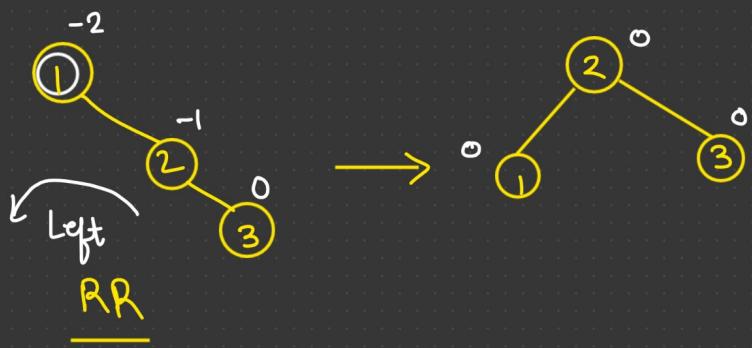
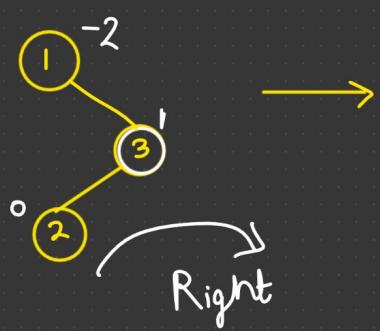
3) LR Tree :-



Reason :- Left - Right insertion

4) RL Tree :-

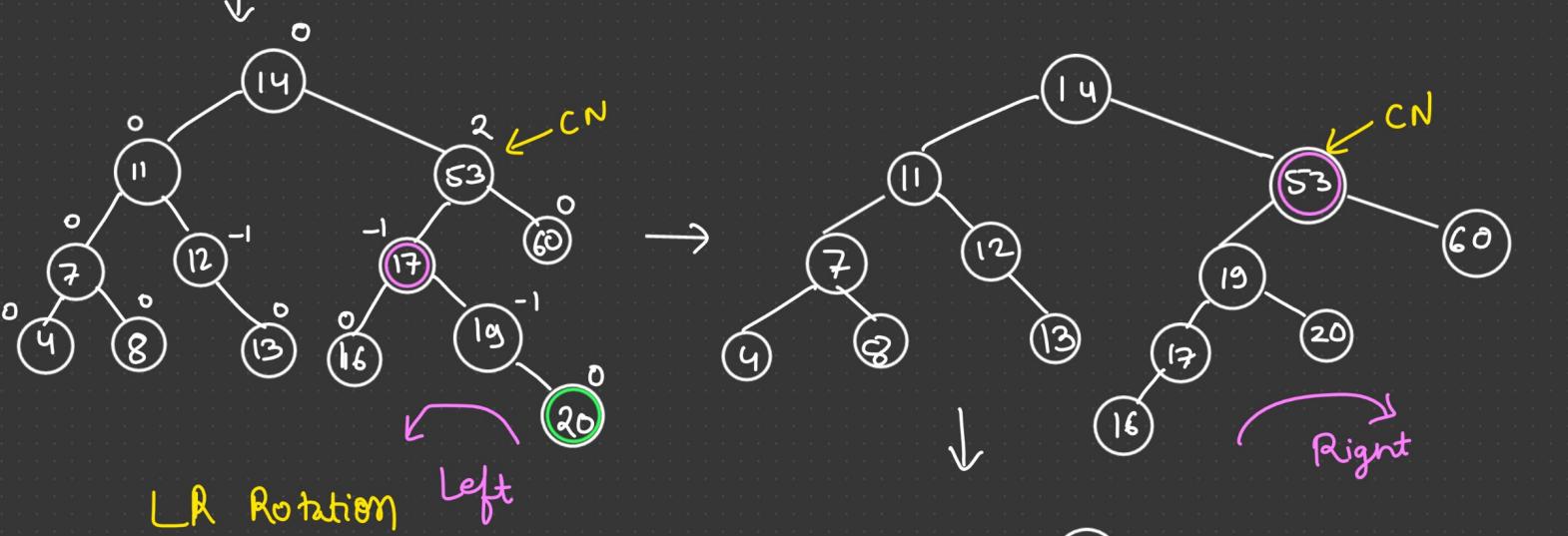
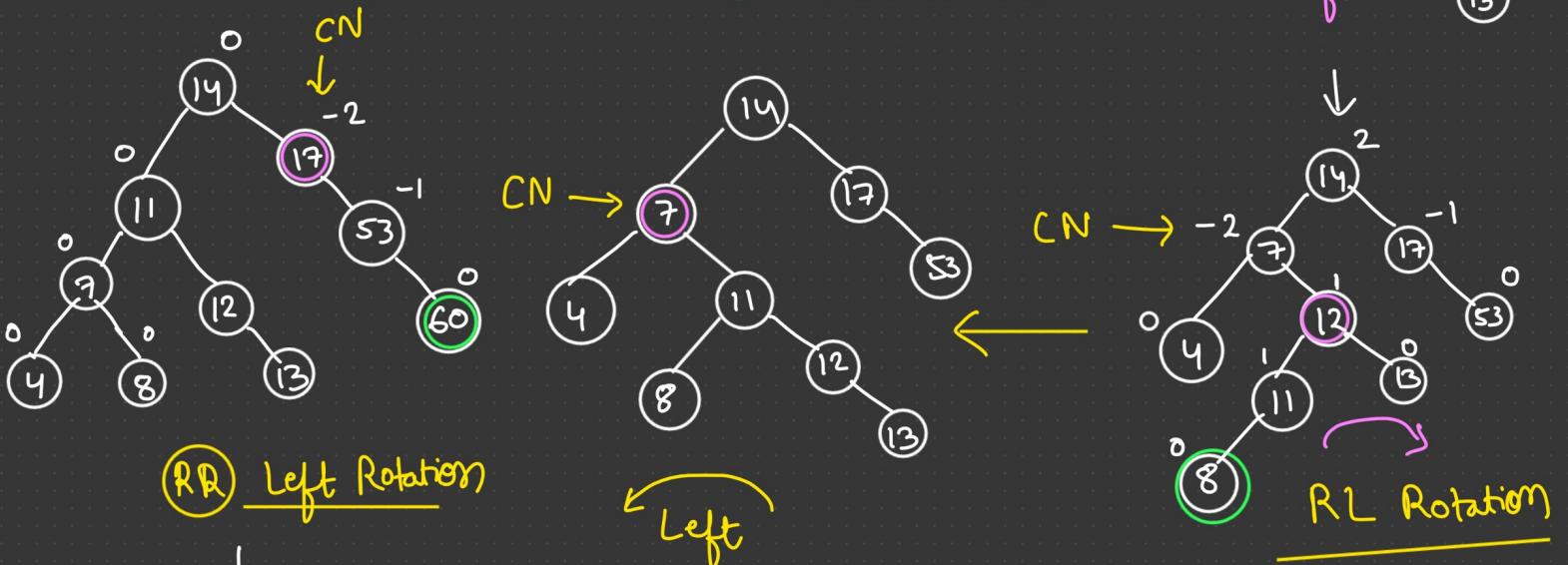
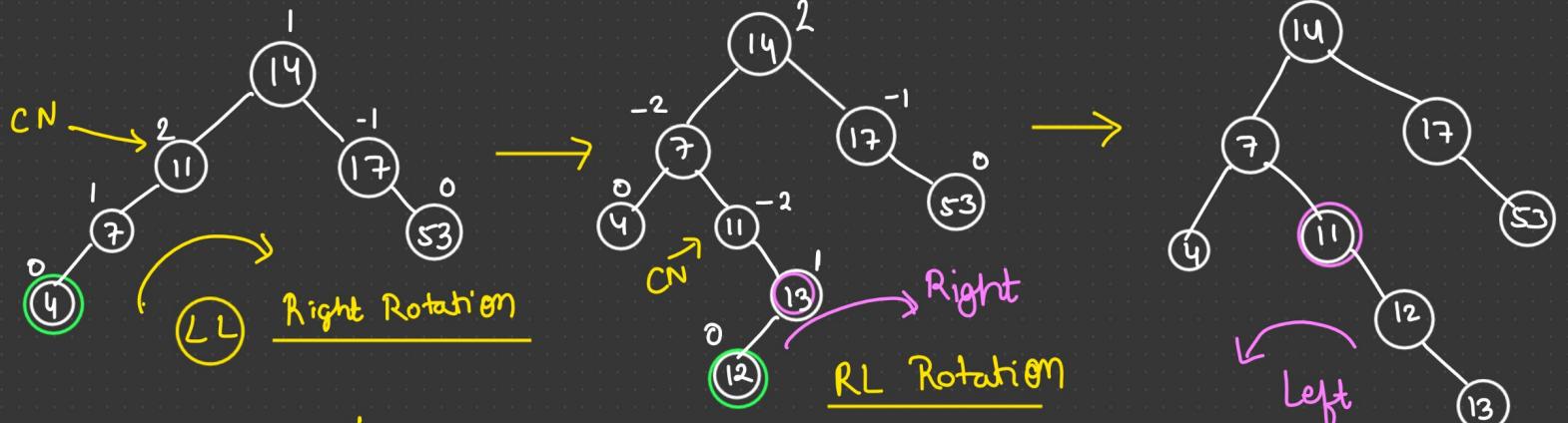
Critical
Node



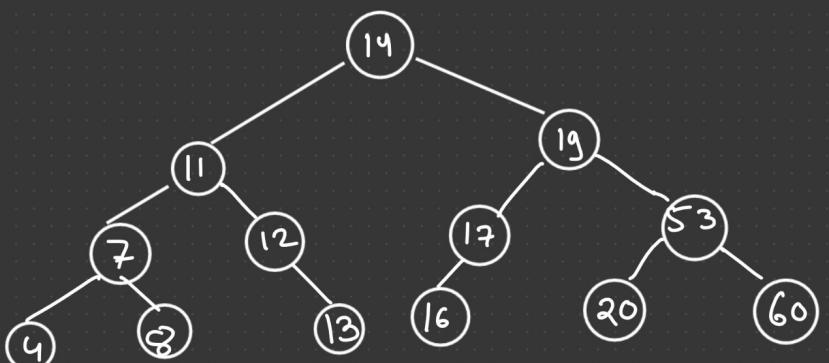
Reason :- Right - Left insertion.

Ex:- 14, 17, 11, 7, 53, 4, 13, 12, 8, 60, 19, 16, 20

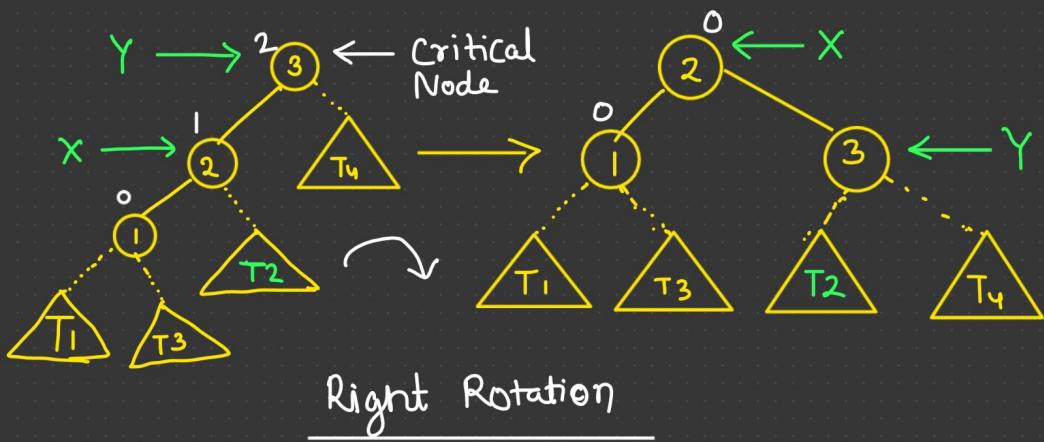
○ → Rotation About
which Node
○ → New Node



Final AVL Tree



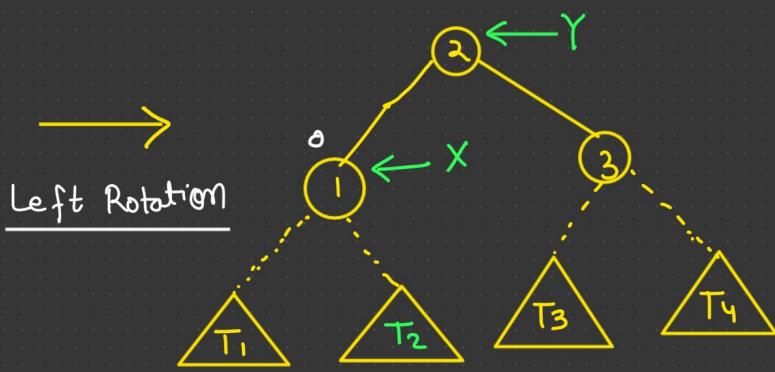
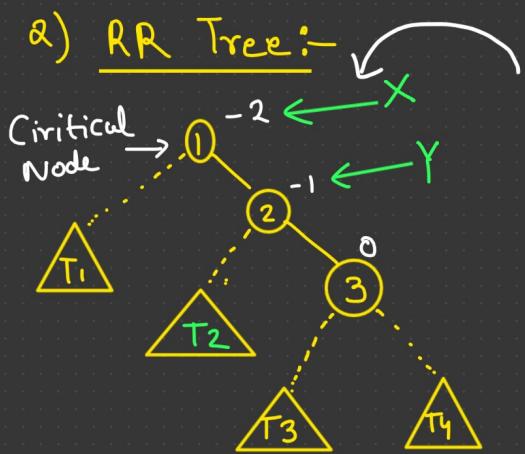
1) LL Tree :-



```
class Node  
{  
public :  
    int key ;  
    Node* left, *right;  
    int height ;  
};
```

```
Node(int v)  
{  
    Key = v;  
    Left = NULL;  
    right = NULL;  
    height = 1;  
};  
};
```

```
Node* RightRotation(Node* Y)  
{  
    Node * X = Y->left ;  
    Node * T2 = X->right ;  
    X->right = Y ;  
    Y->left = T2 ;  
    Y->height = max(H(Y->left), H(Y->right)) + 1 ;  
    X->height = max(H(X->left), H(X->right)) + 1 ;  
    return X ;  
}
```



Node * LeftRotation(Node * X)

{
 Node * Y = X->right;

 Node * T2 = Y->left;

 Y->left = X;

 X->right = T2;

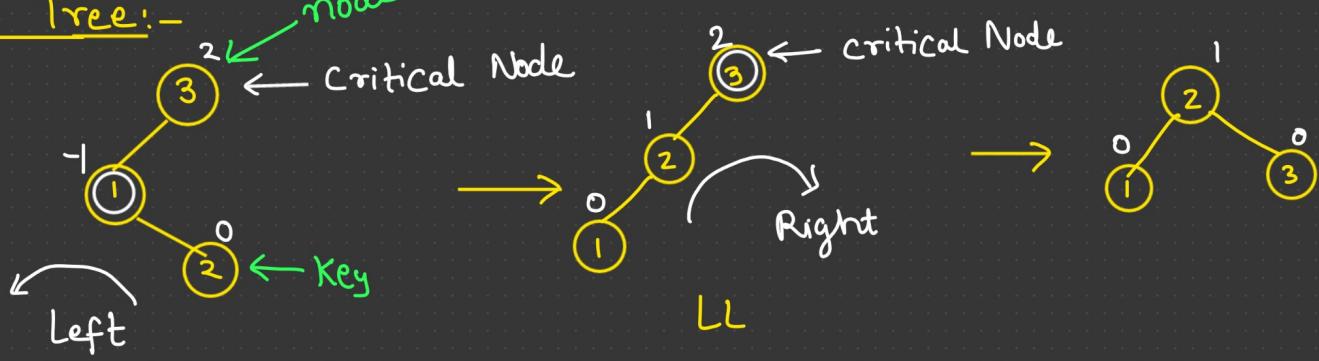
 X->height = max (Height(X->left), Height(X->right)) + 1;

 Y->height = max (Height(Y->left), Height(Y->right)) + 1;

 return Y;

}

3) LR Tree:-



```
int Balancefactor = getBalancefactor(Node);
```

```
if (Balancefactor > 1)
```

```
{
    if (Node->left->value > key)
    {
        return RightRotation(Node);
    }
    else
    {
        node->left = LeftRotation(Node->left);
        return RightRotation(Node);
    }
}
```

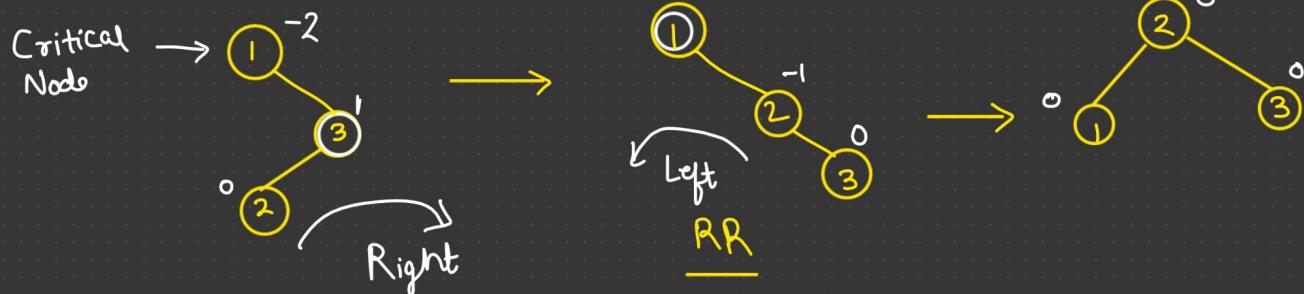
1) Check BF

2) Node = C.N

3) C.N → left < > key

LL ✓
LR ✓

4) RL Tree:-



```
if (Balancefactor < -1)
```

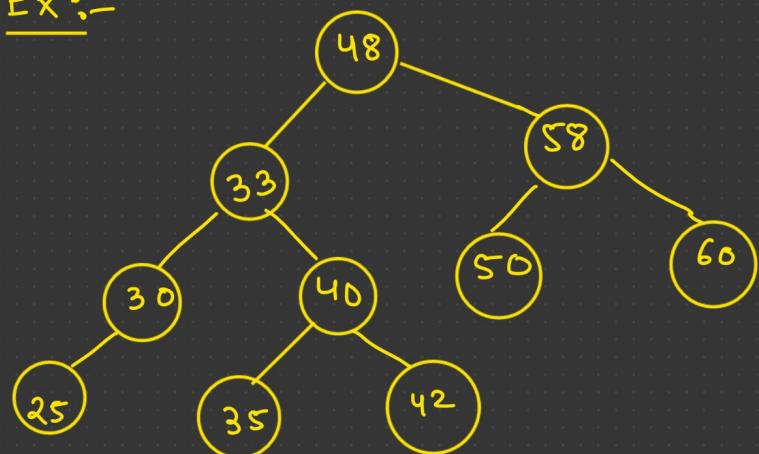
```
{
    if (Node->right->value < key)
        return LeftRotation(Node);
    else
        node->right = RightRotation(Node->right);
        return LeftRotation(Node);
}
```

Deletion in AVL Tree :-

You must know 2 things:-

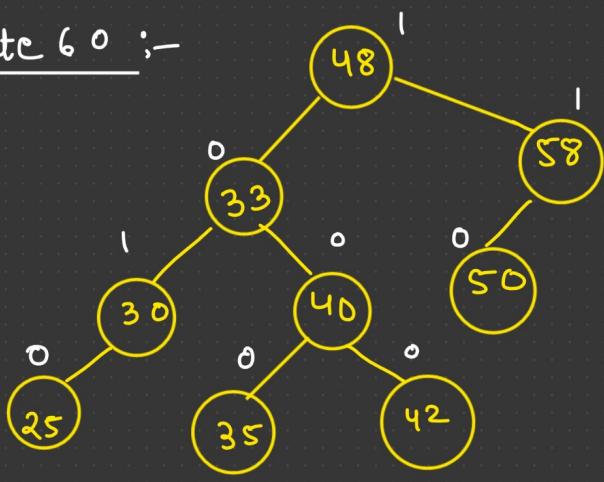
- 1) Deletion BST.
- 2) All types of Rotation.

Ex :-



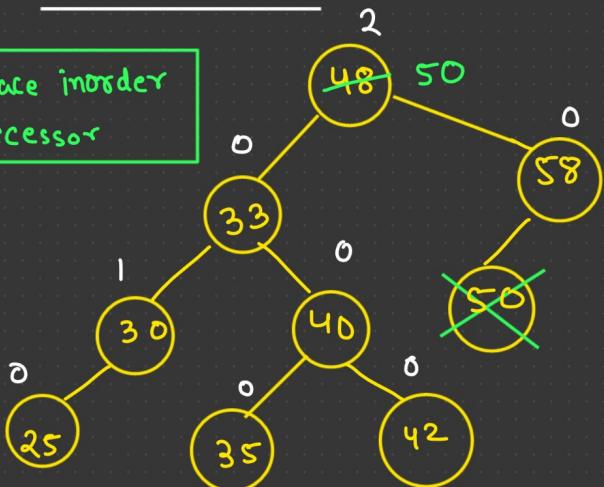
60, 48, 25, 30, 35, 33, 58

1) Delete 60 :-

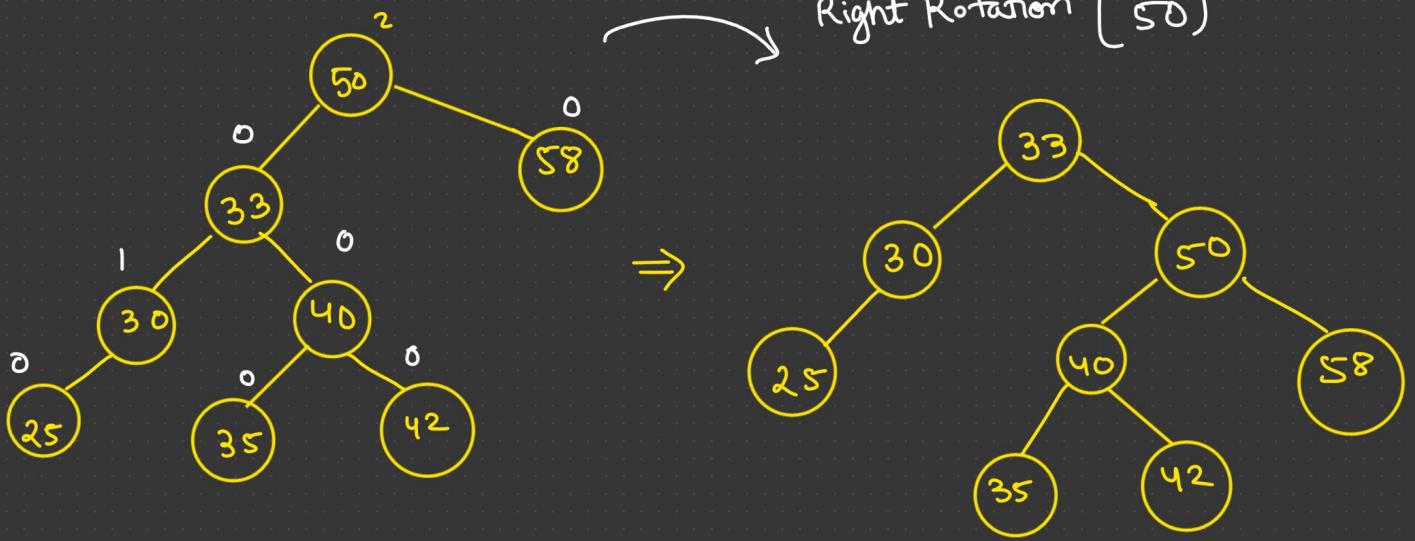


2) Delete 48 :-

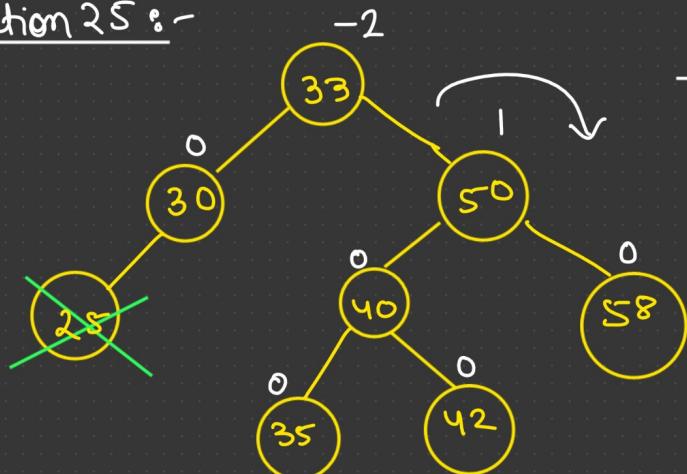
Replace inorder
successor



we need rotation to
make it AVL.

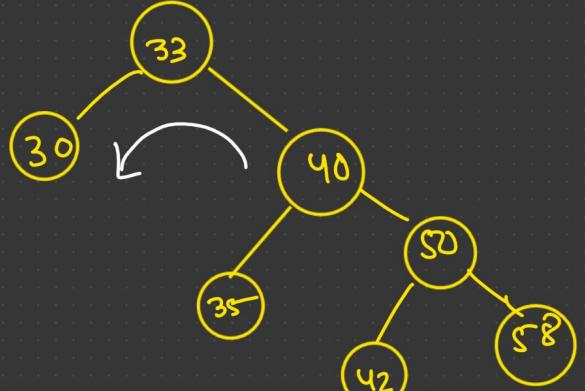


3) Deletion 25 :-

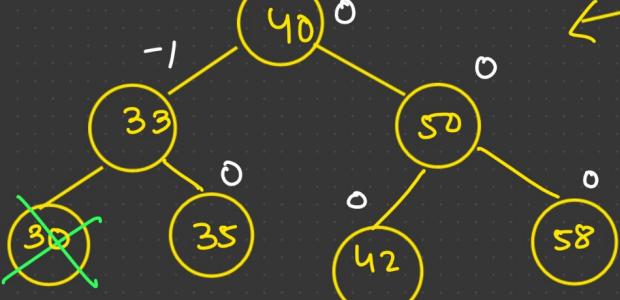


RL Rotation

① Right Rotation(50)

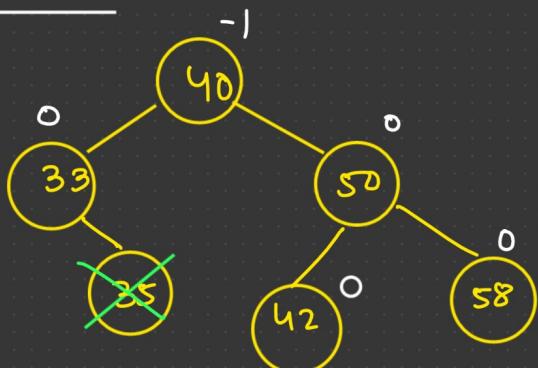


4) Deletion 30 :-

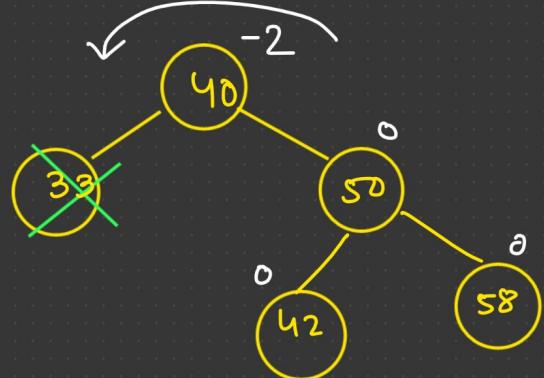


② Left Rotation(33)

5) Delete 35:-

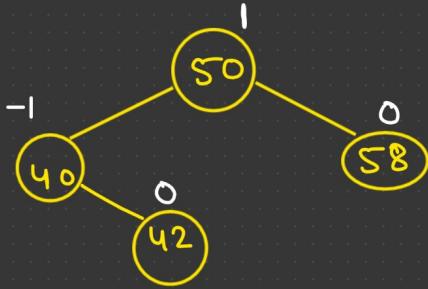


6) Delete 33 :-



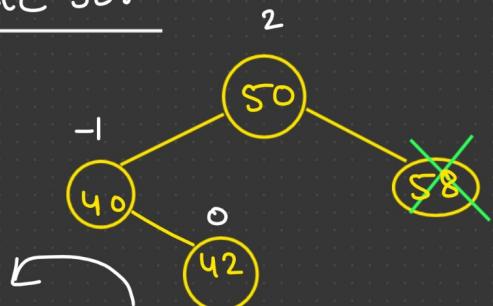
Now two possibility

RR
LR

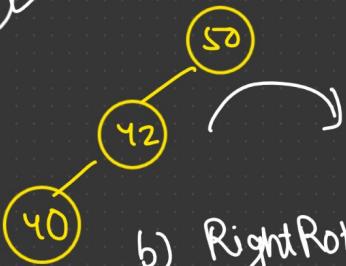


Left Rotation (40)

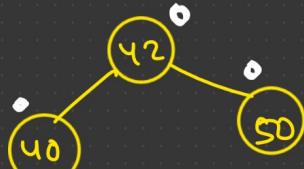
7) Delete 58:-



LL X
a) left Rotation (40)



b) Right Rotation (50)



1) if (BalanceFactor > 1)

 LL (Right Rotation)

 if (getBalanceFactor(Node->left) >= 0)

 RightRotation (Node)

 LR (left → right)

 if (getBalanceFactor (Node->left) < 0)

 leftRotation (Node->left)

 rightRotation (Node).

2) if (BalanceFactor < -1)

 RR (Left Rotation)

 if (getBalanceFactor (Node->right) <= 0)

 LeftRotation (Node);

 RL Rotation (Right → Left)

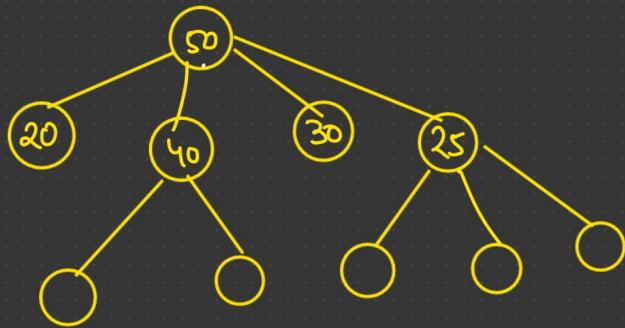
 if (getBalanceFactor (Node->right) < 0)

 RightRotation (Node->right);

 LeftRotation (Node);

General Tree

General tree is a tree in which a node can have either zero child or more than zero child.



- ★ No limitation on the degree of node.

```
struct Node  
{  
    int value;  
    struct Node * child[ ];  
};
```

M-way Tree / Multi-way Tree

It is generalised version of search tree. In m-way tree multiple elements can be found in a node.

In m-way tree, m- is known as order/degree of the node/tree.

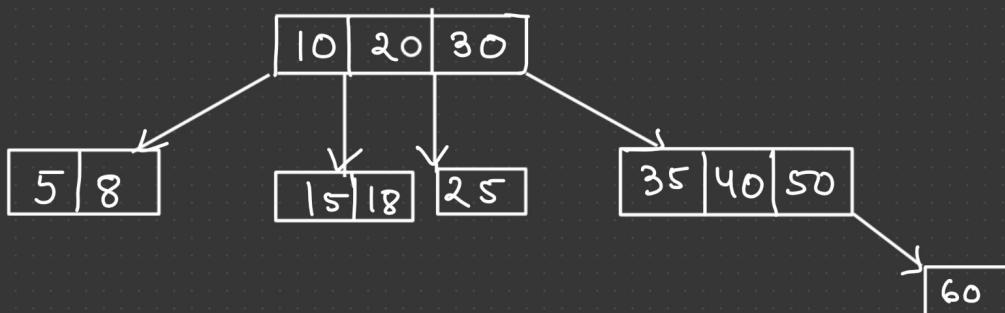
2-way Tree \Rightarrow Order = 2 \Rightarrow m = 2

3-way Tree \Rightarrow Order = 3 \Rightarrow m = 3

4-way Tree \Rightarrow Order = 4 \Rightarrow m = 4

\star
 m = maximum no. of child
 m-1 = maximum no. of elements in a node

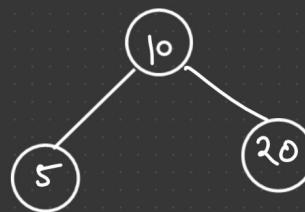
Ex:- 4-Way Tree :- $m=4$ max child = 4
max elements = 3



$O(\log n) = \text{height of Tree}$

2-way Tree / Binary Search Tree :-

$m=2$ \Rightarrow max child = 2
max element = 1

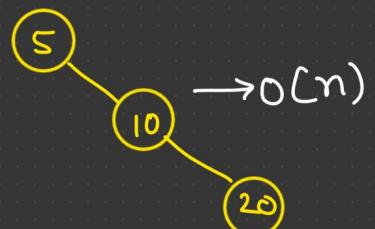


It is also :-

$\left\{ \begin{array}{l} \text{3-way tree} \\ \text{4-way tree} \\ \text{5-way tree} \end{array} \right.$
 ... So on.

Reason :-

- 1) No restriction on minimum no. of elements in a node.
- 2) No restriction on the height of the tree.



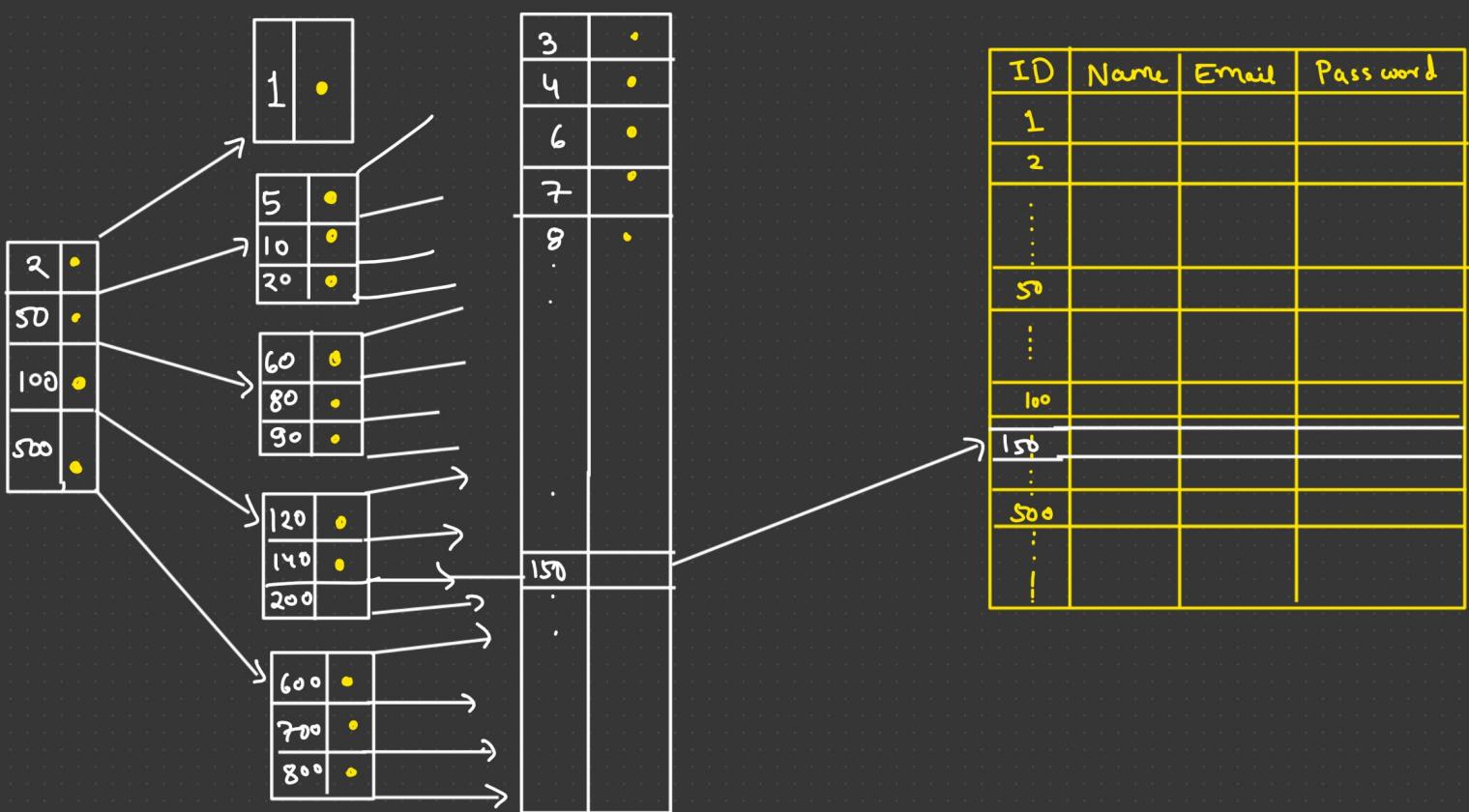
B-Tree

- * It is also a M-way Tree.
- * All the leaf nodes are at the same level.
- * Order of the tree must be determined.
- * All the elements of left sub-tree of a given element has less value & right sub-tree will have greater element.
- * If "m" is the order of B-Tree then it has "maximum of m child nodes" & "m-1 elements in a Node".
- * If "m" is the order of B-Tree then it has "minimum of ⌈m/2⌉ child nodes" & "minimum of ⌈m/2⌉-1 elements in a node".

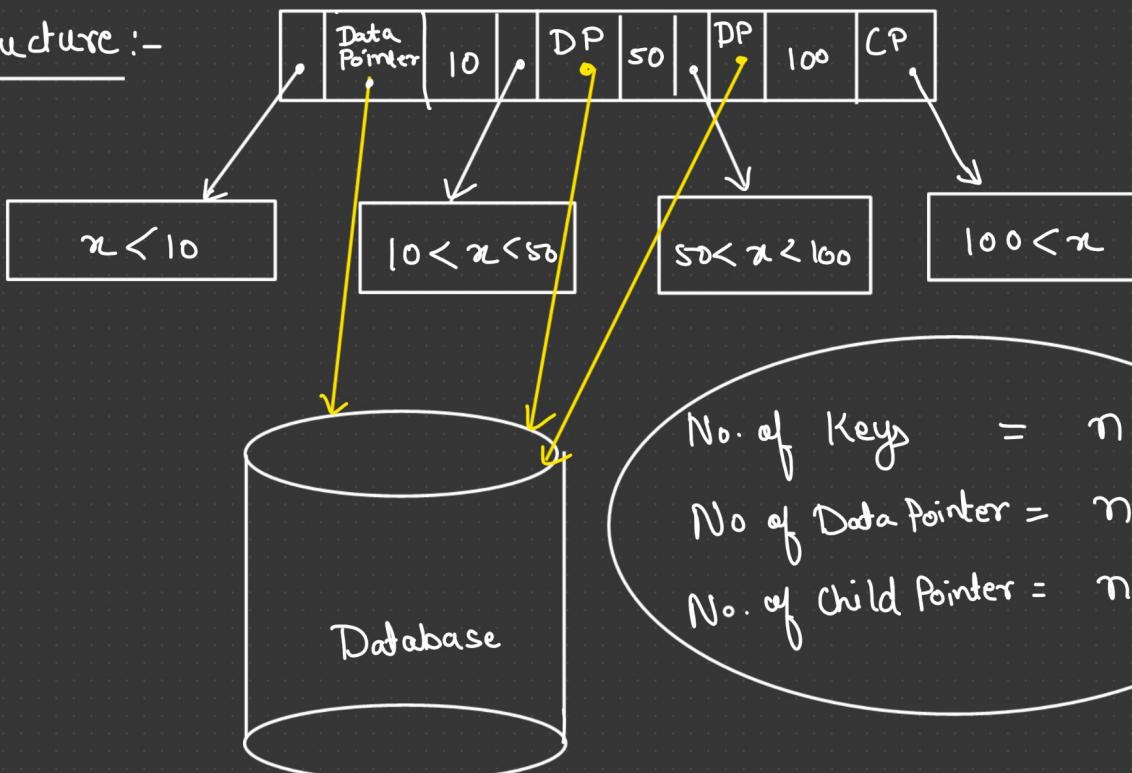
$$\begin{array}{l} \rightarrow \text{floor}(2) = \lfloor 2 \cdot 2 \rfloor = 2 \\ \rightarrow \text{ceil}(3) = \lceil 2 \cdot 2 \rceil = 3 \end{array}$$

- * Root Node & also Leaf Node then it can have 1 Key "or" 0 child.
- * Root Node & non-leaf Node then it can have 1 Key & minimum 2 child.

$\lfloor 3 \cdot 9 \rfloor = 3$	<p style="text-align: center;"><u>B-Tree of order 5</u></p> <p style="text-align: center;">$\rightarrow m = 5$</p> <p style="text-align: center;">Also it is 5-way Tree</p>	$\rightarrow \text{Max Child} = 5$
$\lceil 3 \cdot 9 \rceil = 4$		$\rightarrow \text{Min Child} = \lceil 5/2 \rceil = 3$
		$\rightarrow \text{Max Keys} = 4$
		$\rightarrow \text{Min Keys} = \lceil 5/2 \rceil - 1 = 2$



Node Structure :-



No. of Keys = n
 No. of Data Pointer = n
 No. of Child Pointer = $n+1$

Creation / Insertion in B-Tree :-

20, 40, 10, 30, 33, 50, 60, 5, 15, 25, 28, 31, 35, 45, 55, 65

Order:-

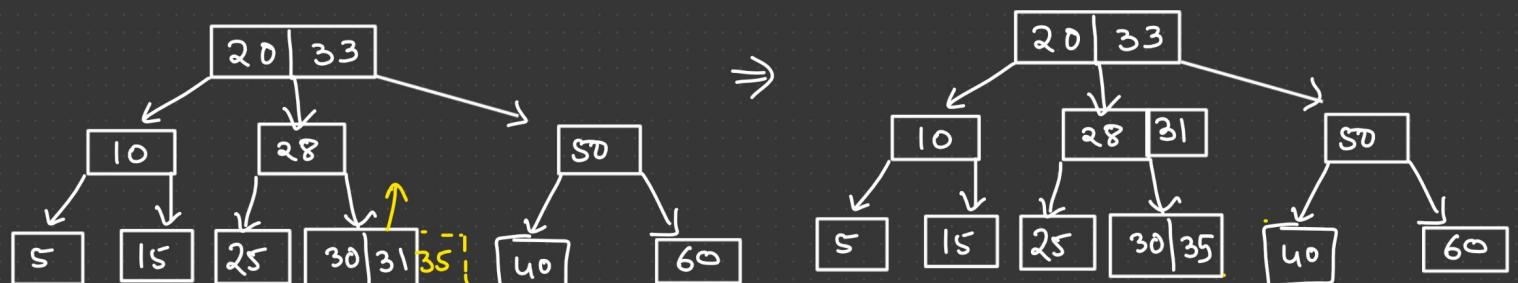
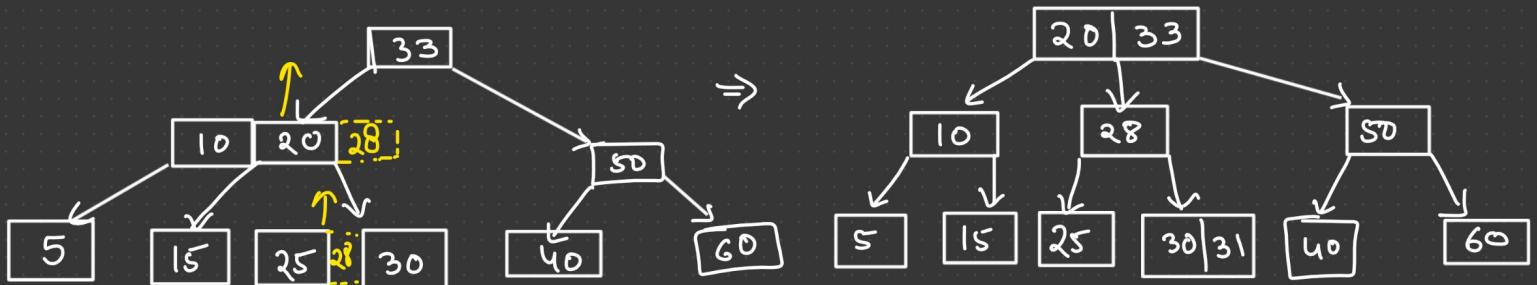
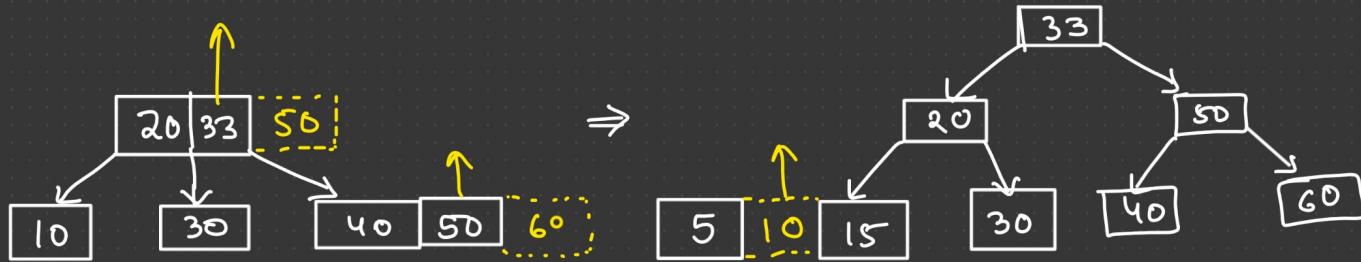
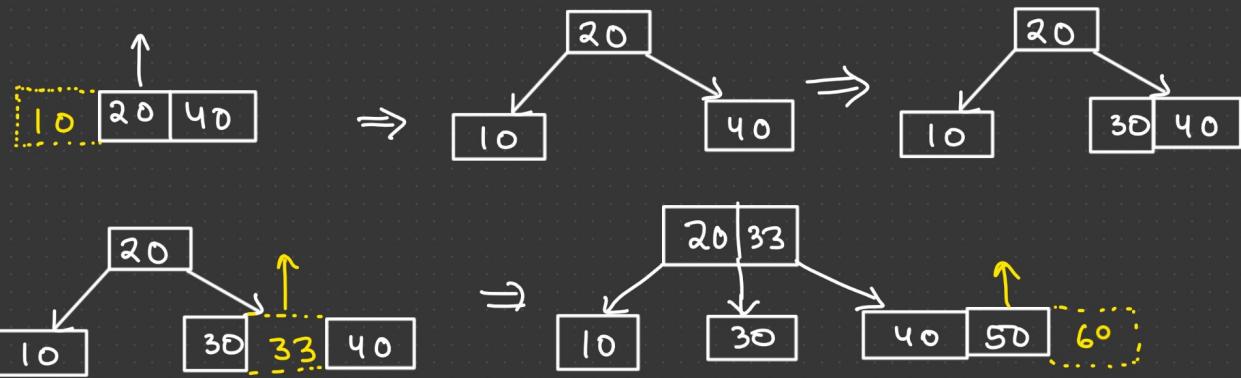
$$m = 3$$

Max Child = 3

Max Key = 2

Min Child = 2

Min Key = 1



To be continue....

Deletion in B-Tree

Rules:-

1) Leaf Node

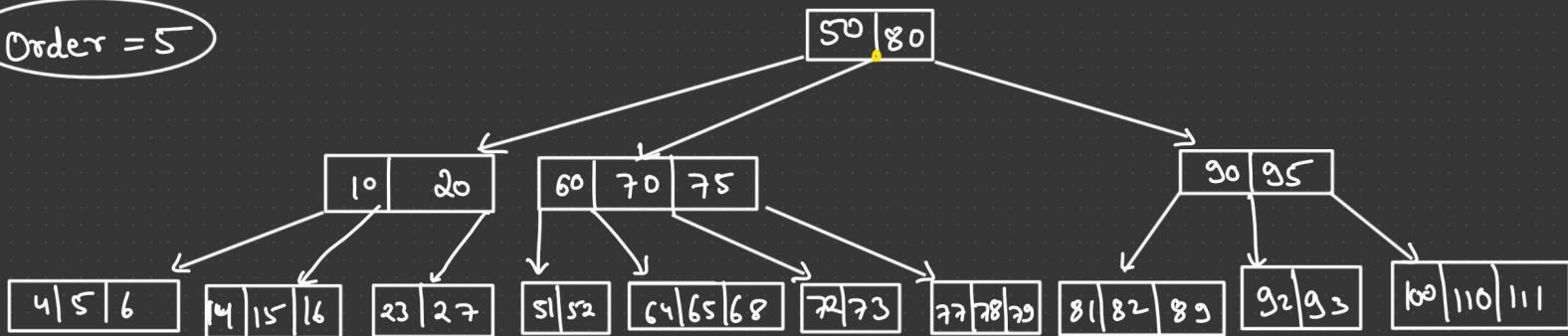
- a) If Node has more than min Key \Rightarrow Directly delete it.
- b) If Node has min Key \Rightarrow Borrow from Left/Right siblings.
If they have more than min Key. via parent node.
- c) If sibling has min key then merge left sibling + parent + right sibling
(parent goes down)

2) Internal nodes:-

- a) If Both left & right sibling exists then replace it with Inorder successor / Inorder predecessor if they have more than min Key.
- b) Merge left + parent + right.

Delete:- 64, 23, 72, 65, 20, 70, 95, 77, 80, 100, 6, 27, 60, 16, 50

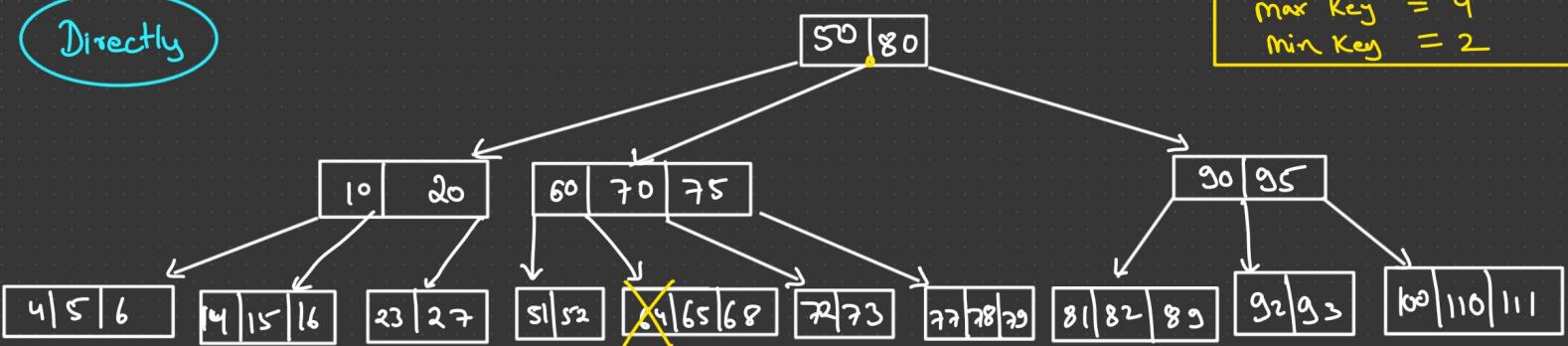
Order = 5



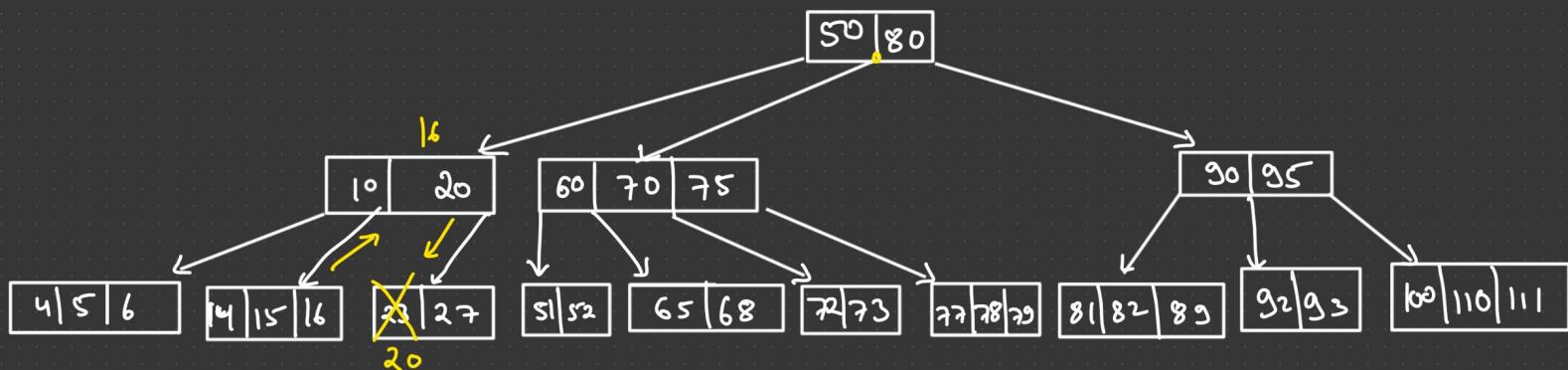
1) Delete 64:-

Directly

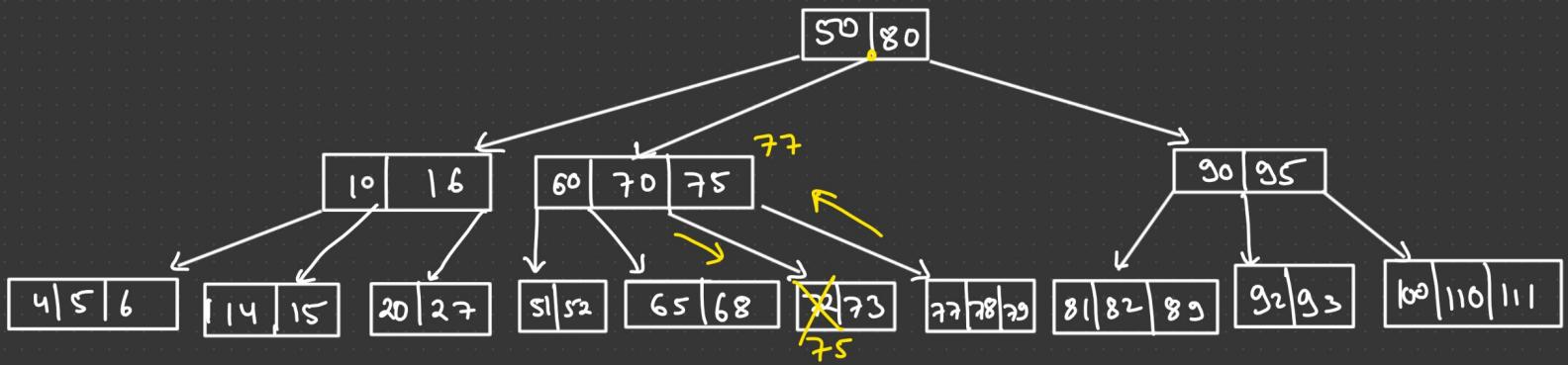
$m = 5$
max child = 5
min child = $\lceil \frac{5}{2} \rceil = 3$
max Key = 4
min Key = 2



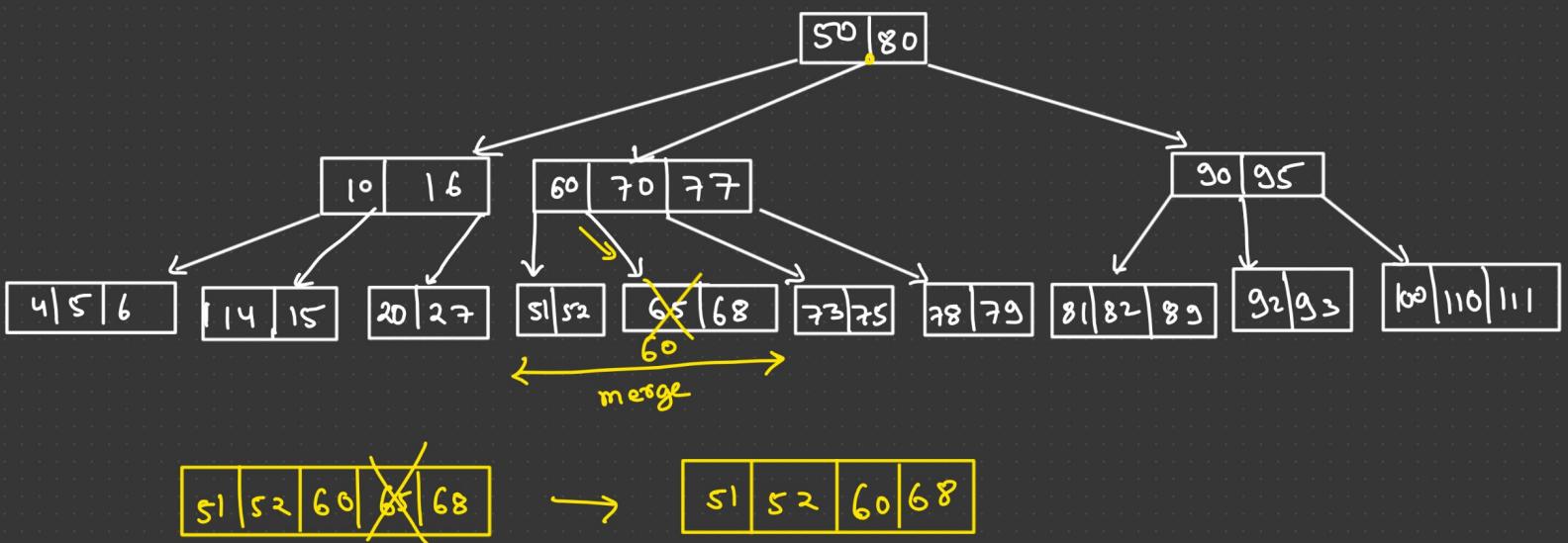
2) Delete 23:- Borrow Inorder predecessor via parent



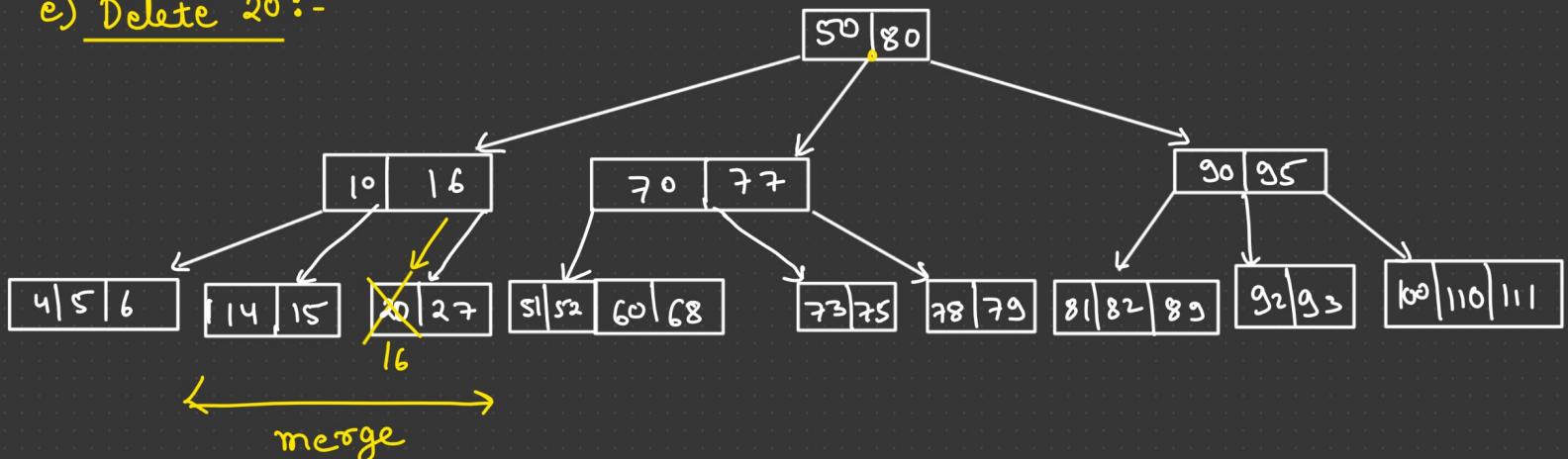
3) Delete 72:- Borrow inorder successor via parent.

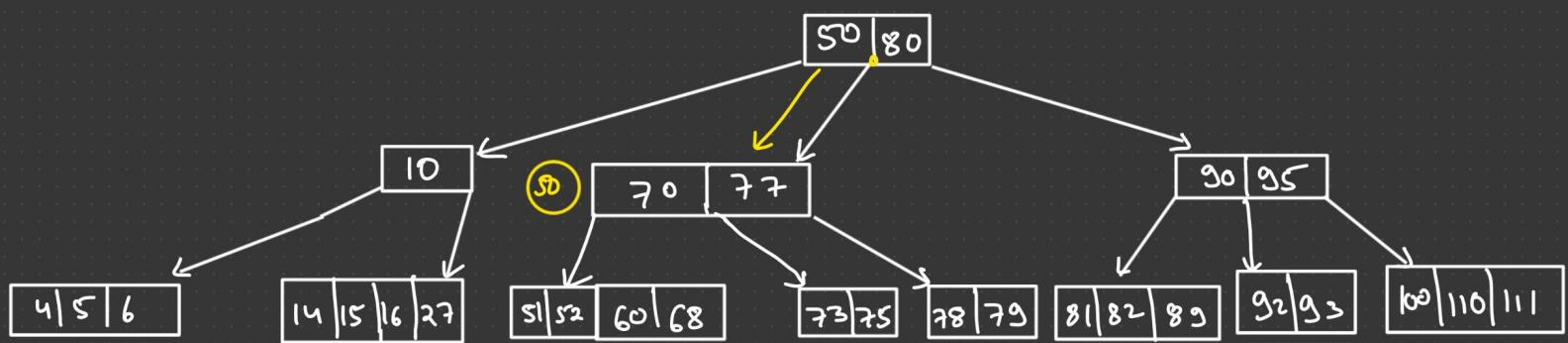


d) Delete 65:-



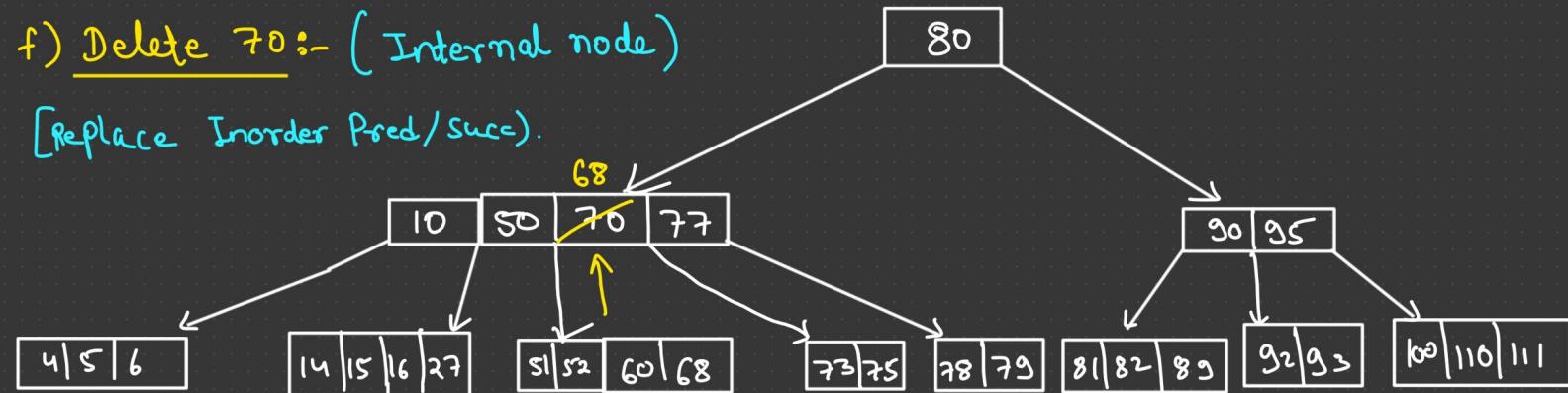
e) Delete 20:-



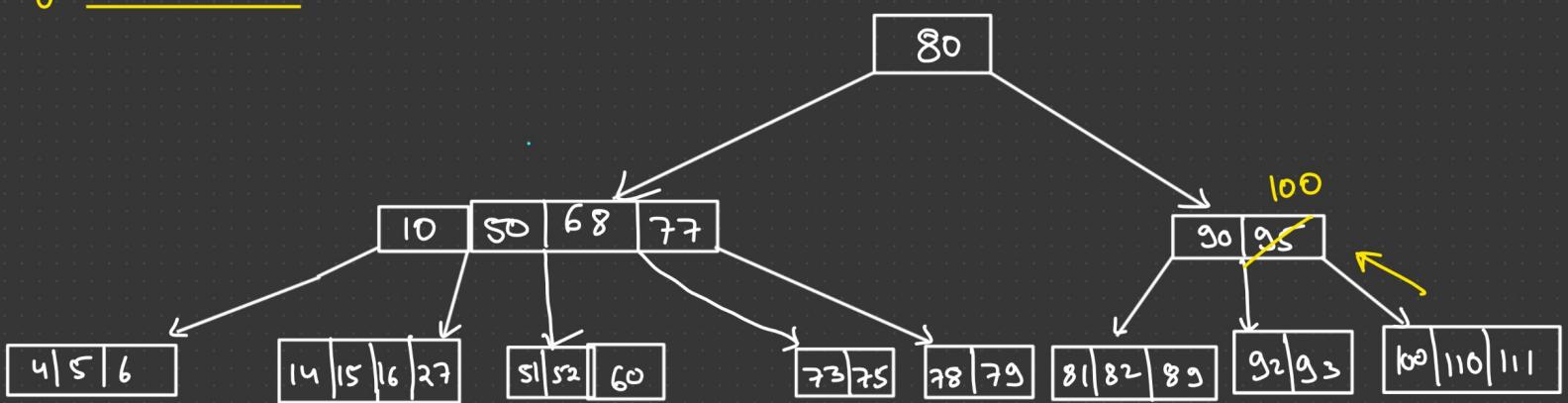


f) Delete 70 :- (Internal node)

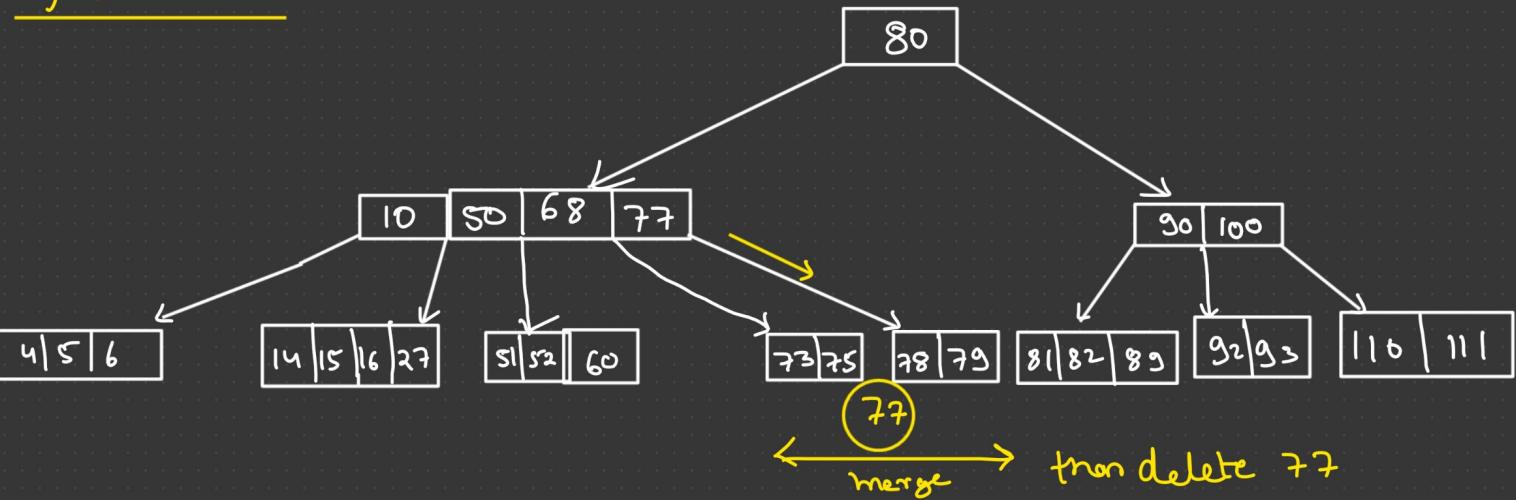
[Replace Inorder Prede/succ].



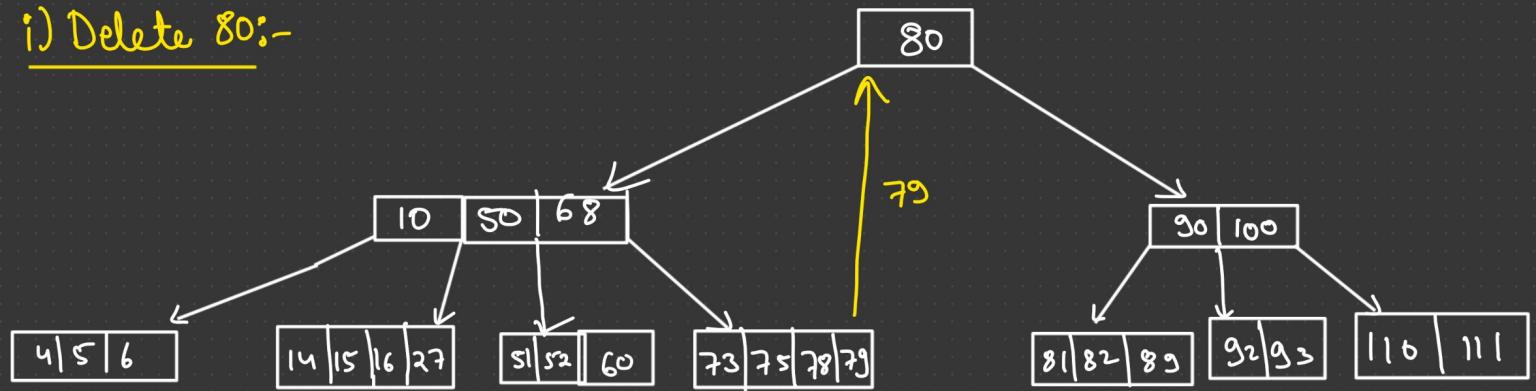
g) Delete 95 :-



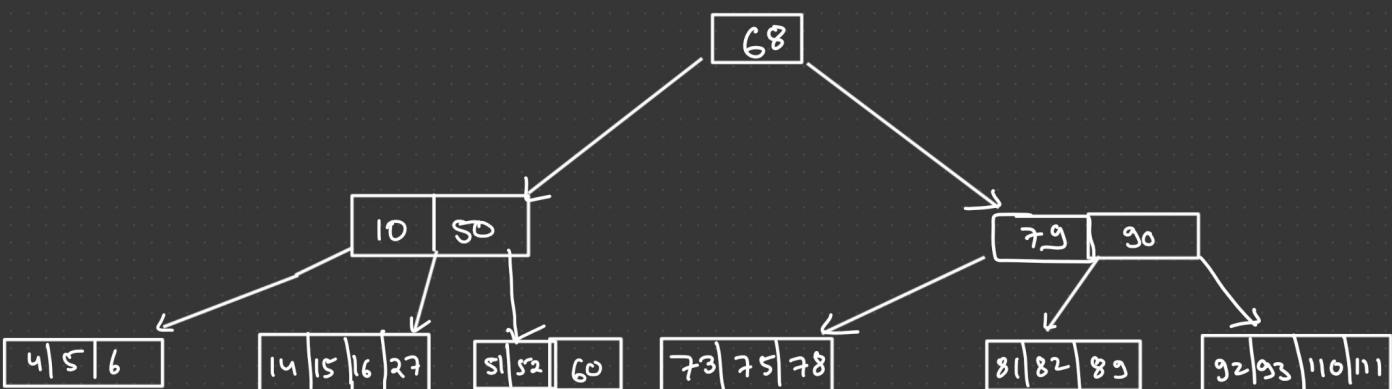
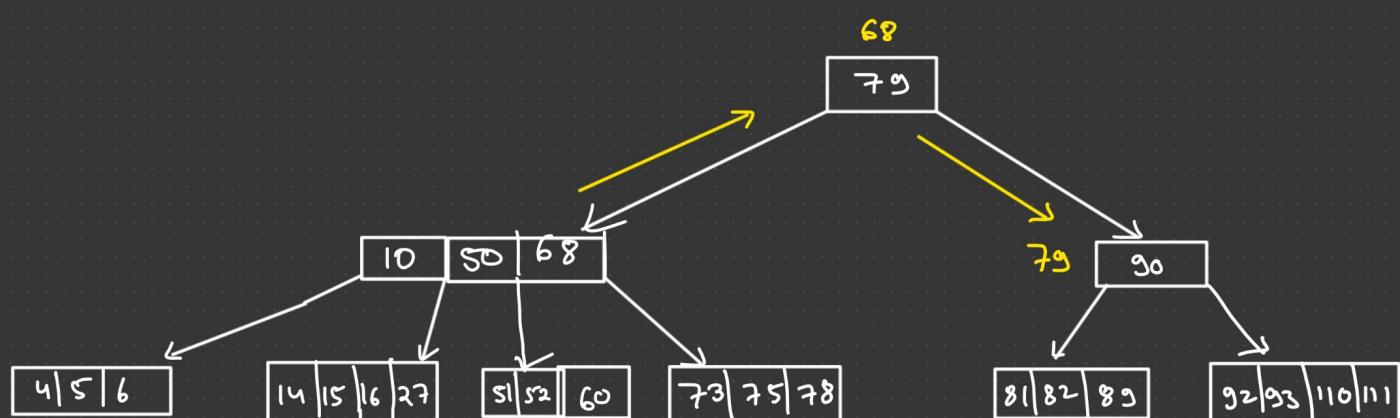
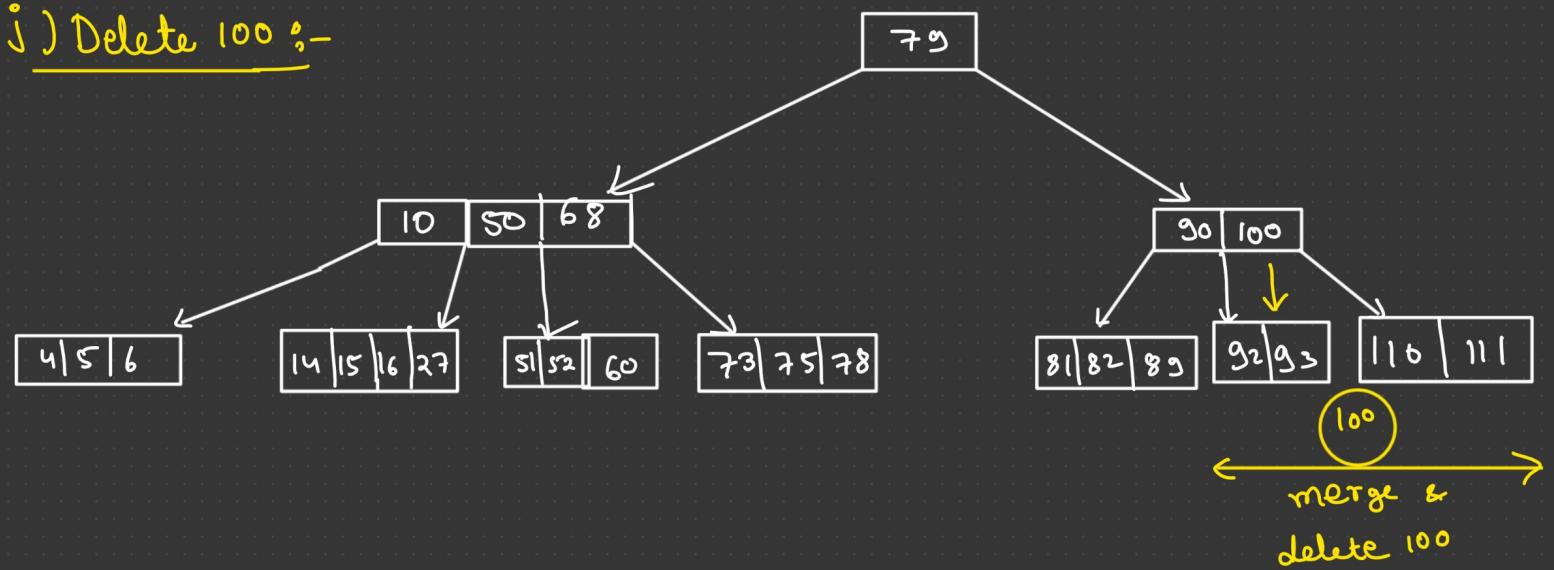
h) Delete 77 :-



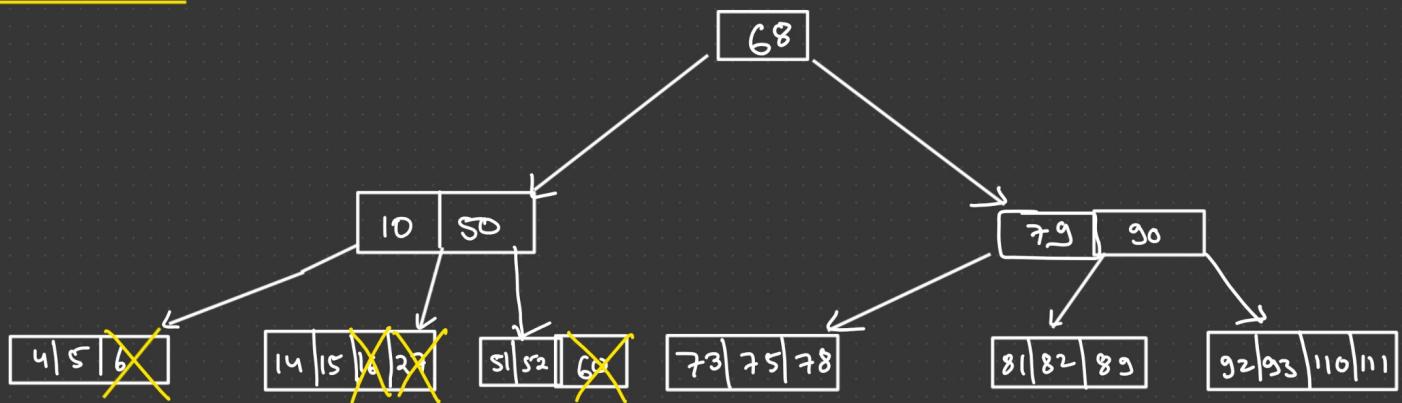
i) Delete 80:-



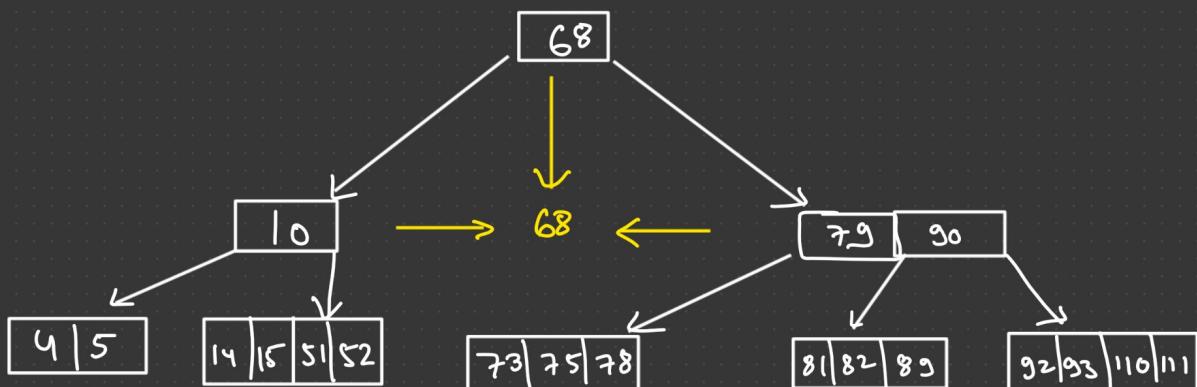
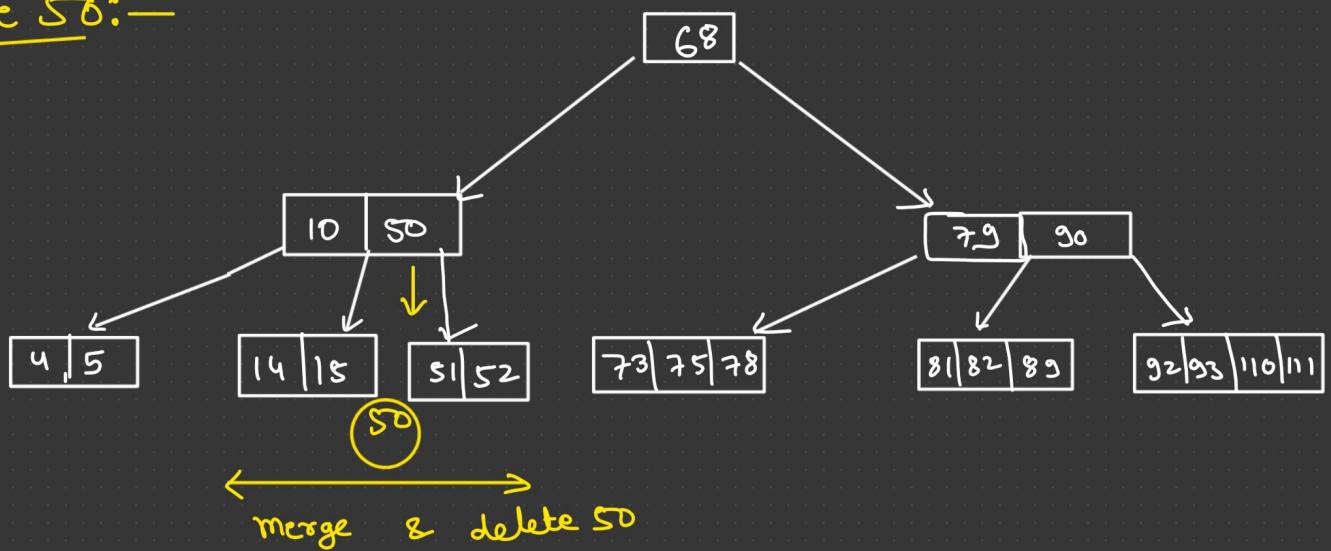
j) Delete 100 :-



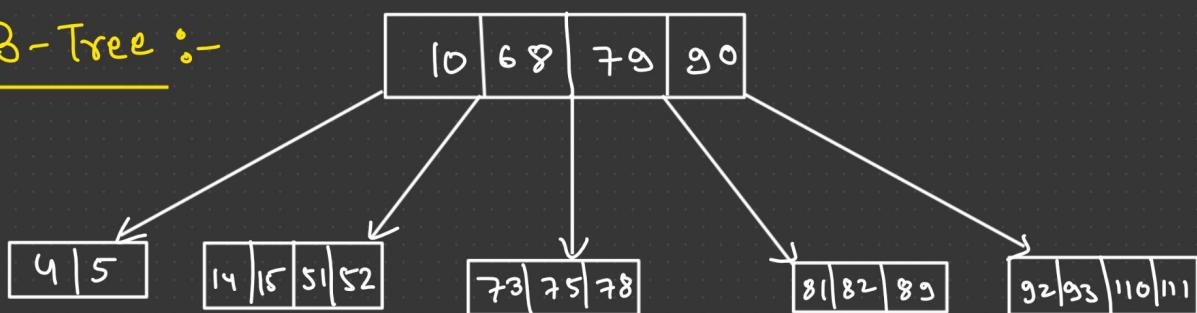
Delete :- [6, 27, 60, 16, 50]



Delete 50:-

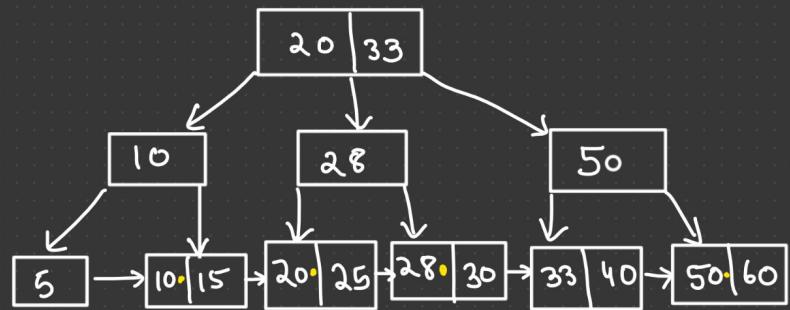
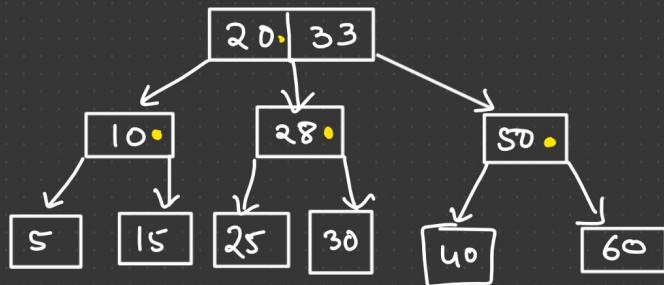


Final B-Tree :-



$m = 3$

B+ Tree



B-Tree

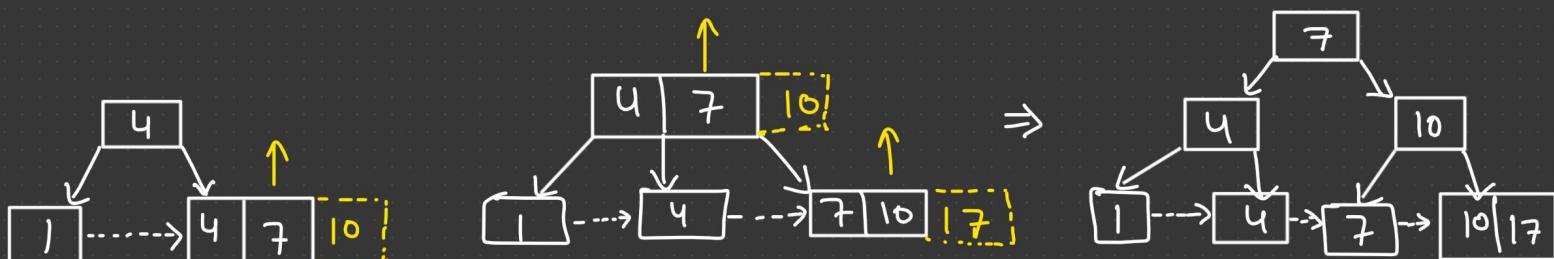
B+ Tree

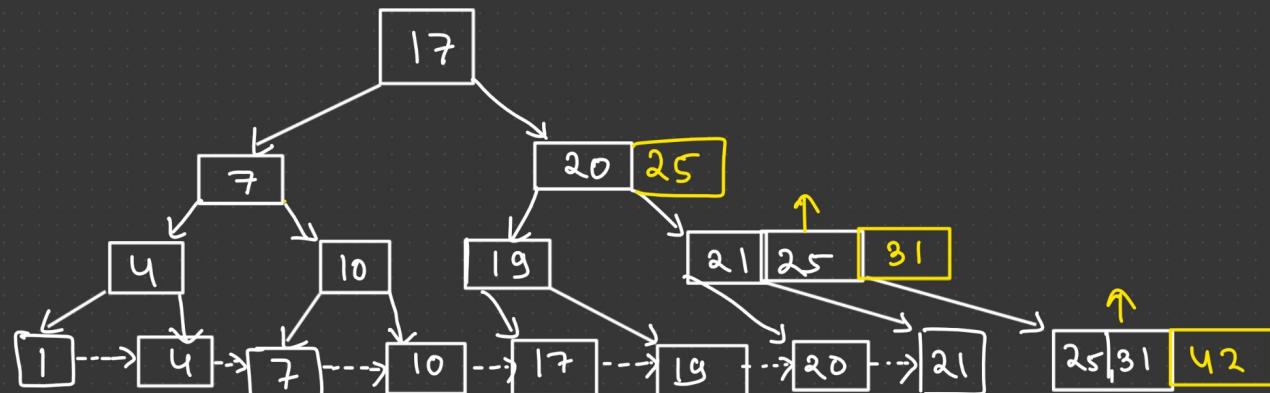
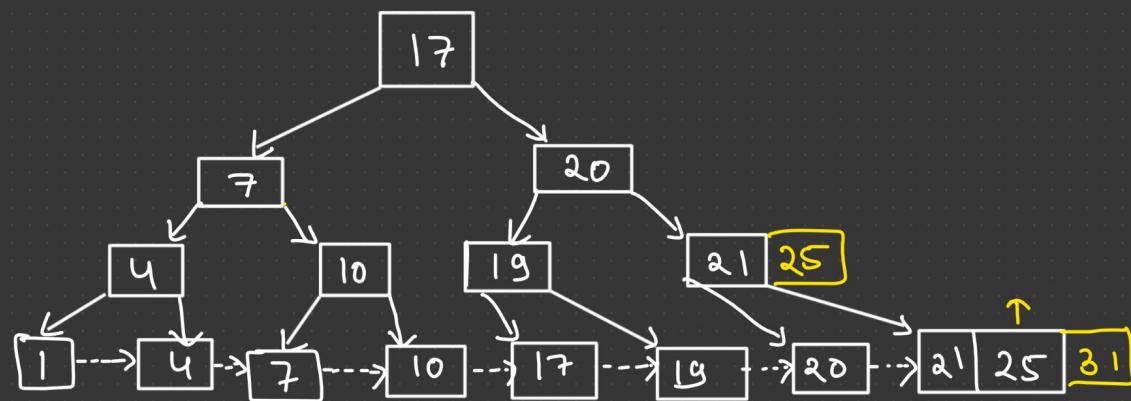
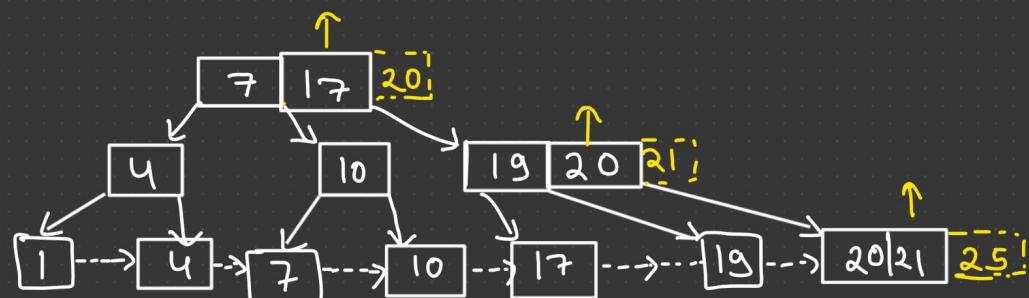
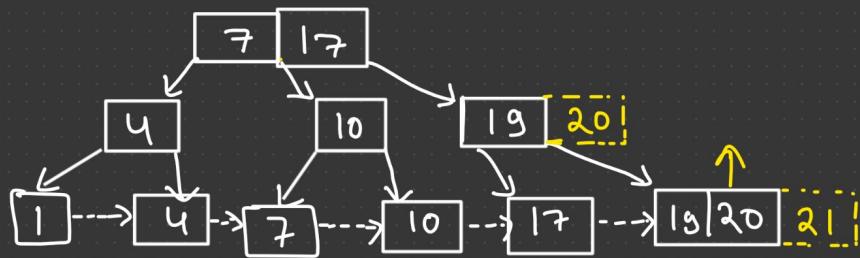
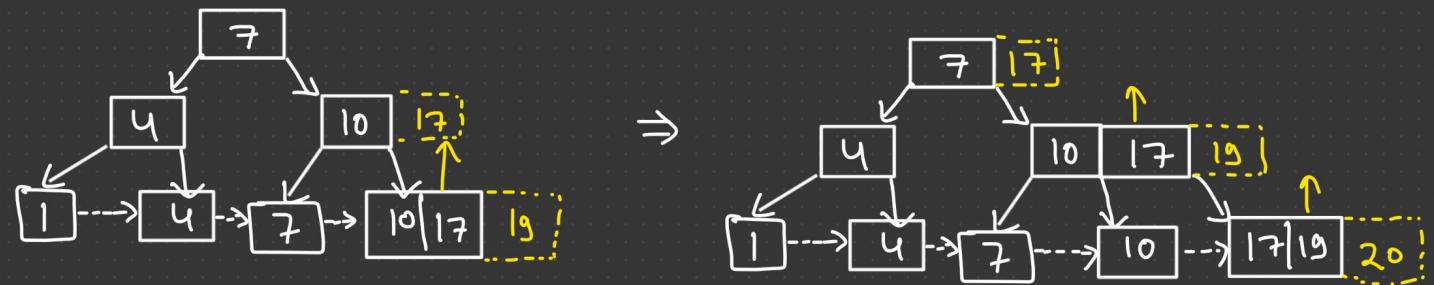
1, 4, 7, 10, 17, 19, 20, 21, 25, 31, 42

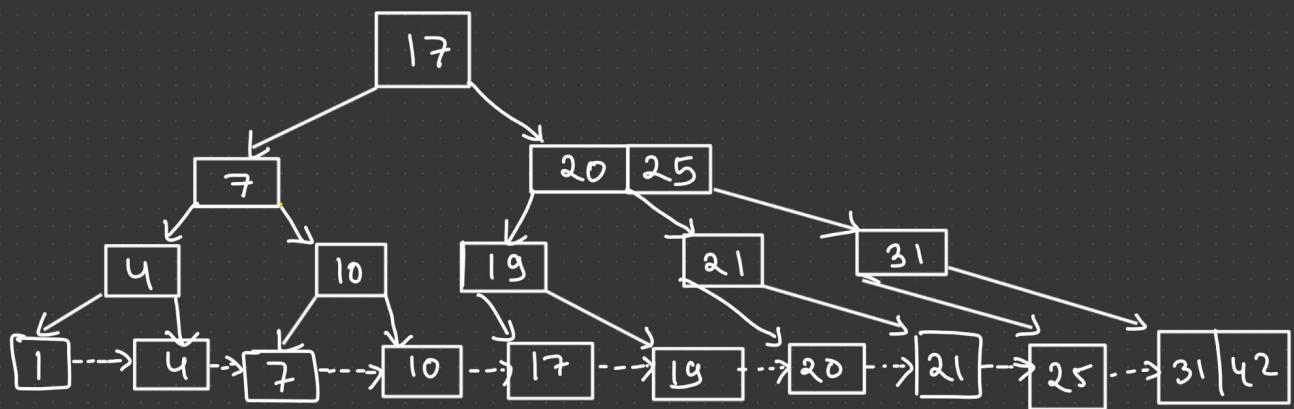


$m = 3 = \text{order}$
$\text{max child} = 3$
$\text{min child} = 2$
$\text{Max Key} = 2$
$\text{Min Key} = 1$

[Note - If leaf node breaks then, a copy of parent node is also present at leaf node. But if an internal node breaks its parent copy does not need to be copied.]







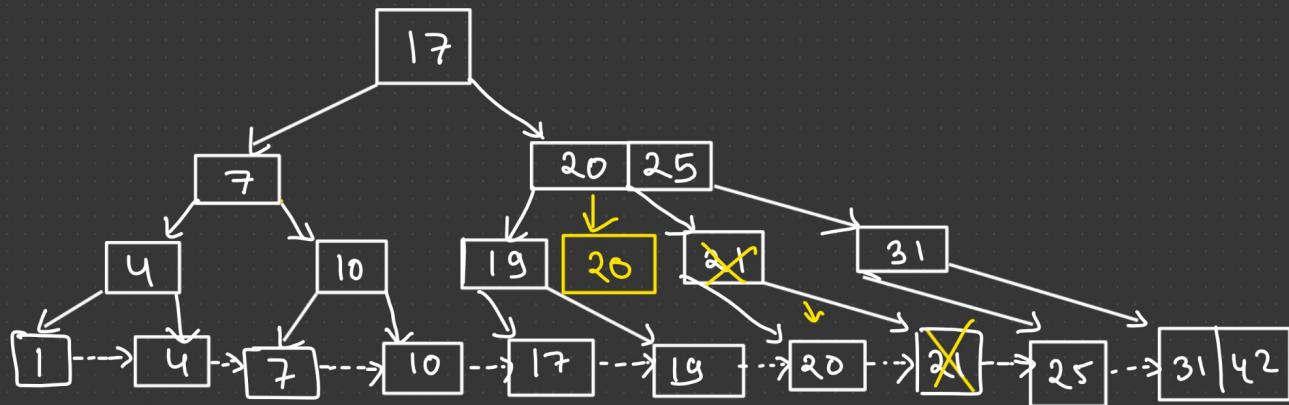
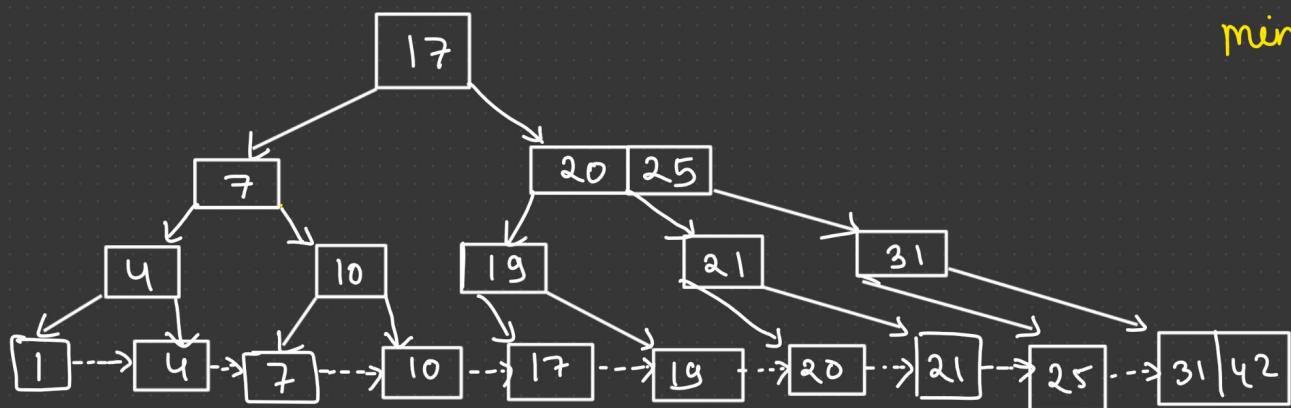
Deletion in B+ Tree :-

- 1) If node has more than min number of Keys then directly delete. And if that element is also present in its internal node then copy inorder successor/predecessor to that node.
- 2) If node has min number of elements then
 - a) It will borrow element from its siblings which have more than min elements via parent.
 - b) Else parent come down & merge with its child.

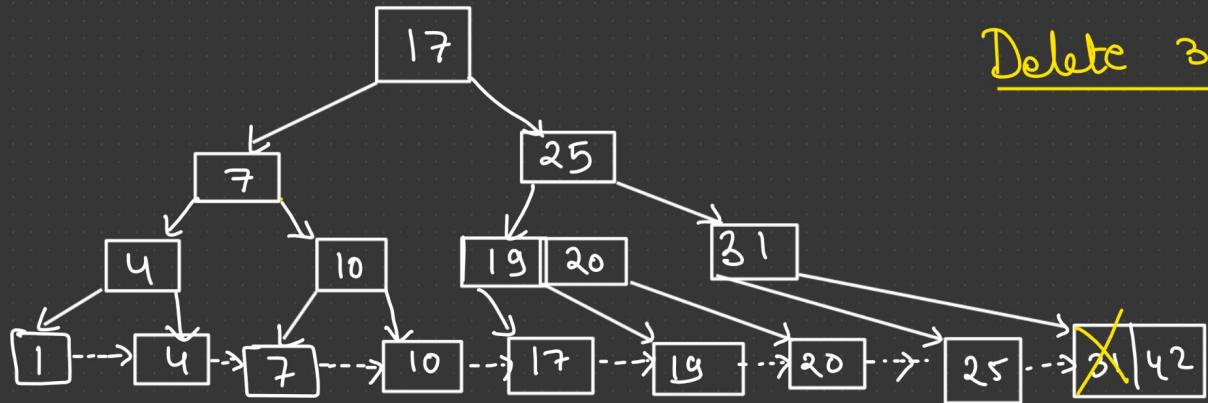
Delete :- 21, 31, 20, 10, 7, 25, 42, 4

$$m = 3$$

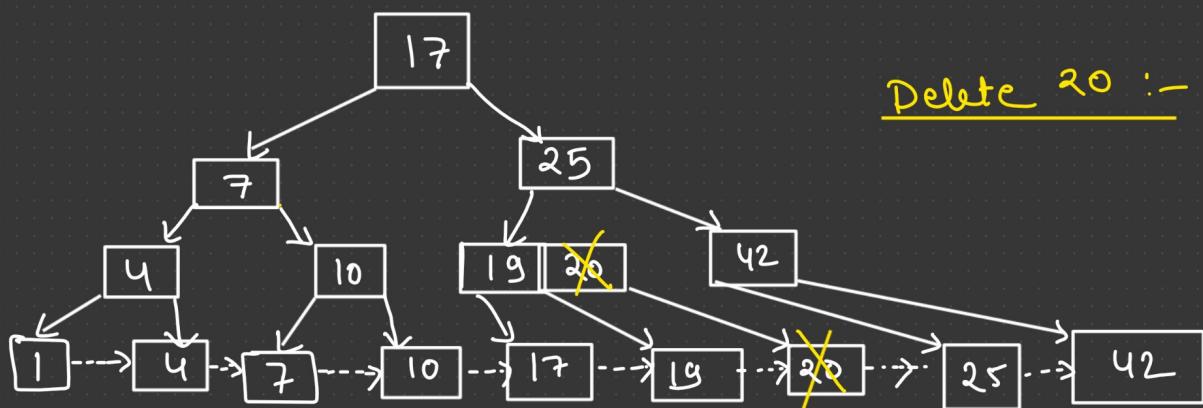
max Keys = 2
min Keys = 1



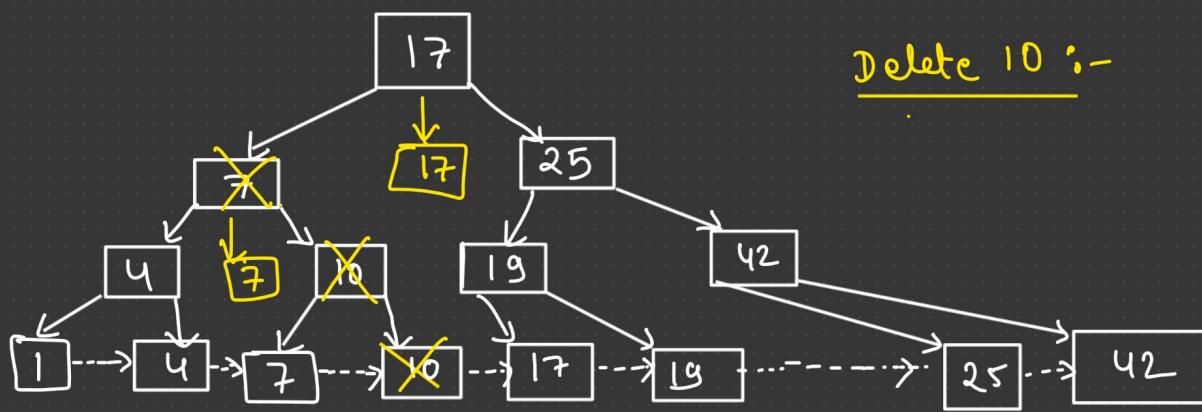
Delete 31



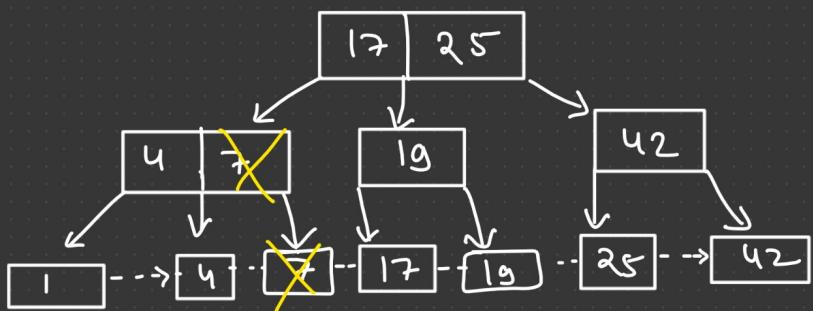
Delete 20 :-



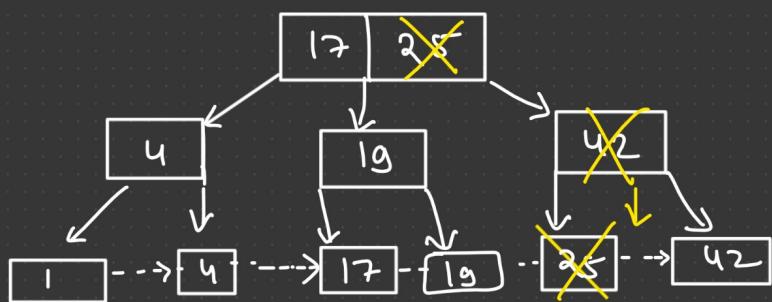
Delete 10 :-



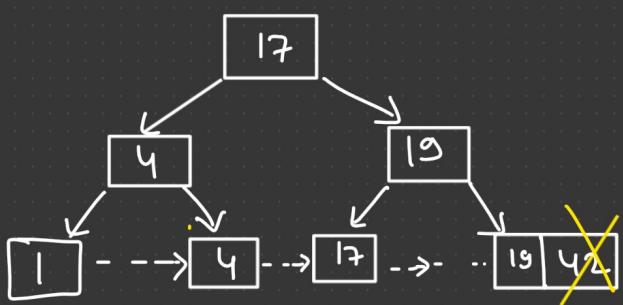
Delete 7 :-



Delete 25 :-



Delete 42 :-



Delete 4 :-



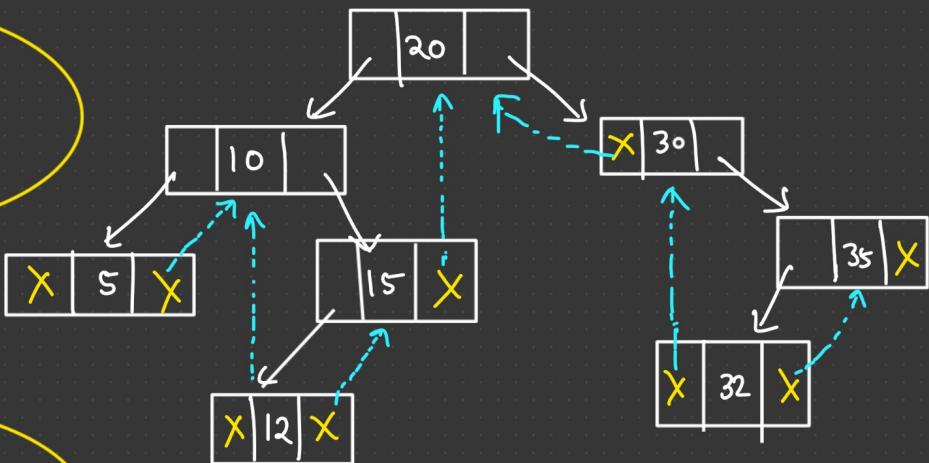
Threaded Binary Tree

Total Nodes = 8

Null pointers = 9

Total nodes = n

Null pointer = n+1



Right = Successor
Left = Predecessor

Inorder :- 5, 10, 12, 15, 20, 30, 32, 35

For Inorder Traversal :- we need stack.

Left ptr	L Tag	Data	R Tag	Right ptr

L Tag = 0 \Rightarrow ptr \rightarrow Tree
= 1 \Rightarrow ptr \rightarrow pred.

R Tag = 0 \Rightarrow ptr \rightarrow Tree
= 1 \Rightarrow ptr \rightarrow succ.

```
struct Node
{
    int data;
    Node *left, *right;
    int LTag, RTag;
};
```

