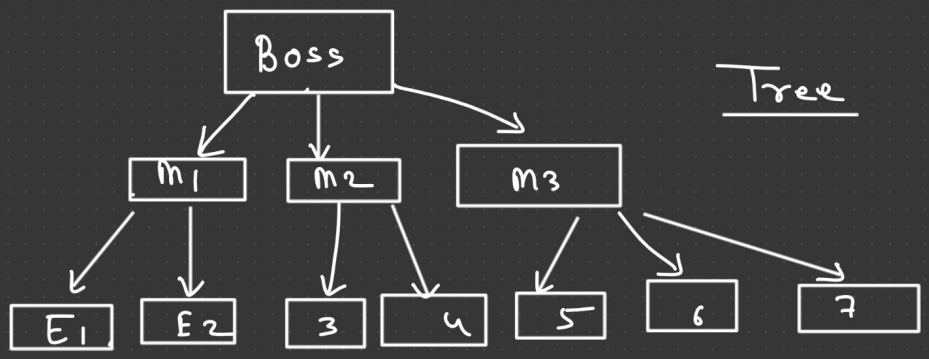
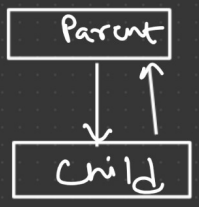
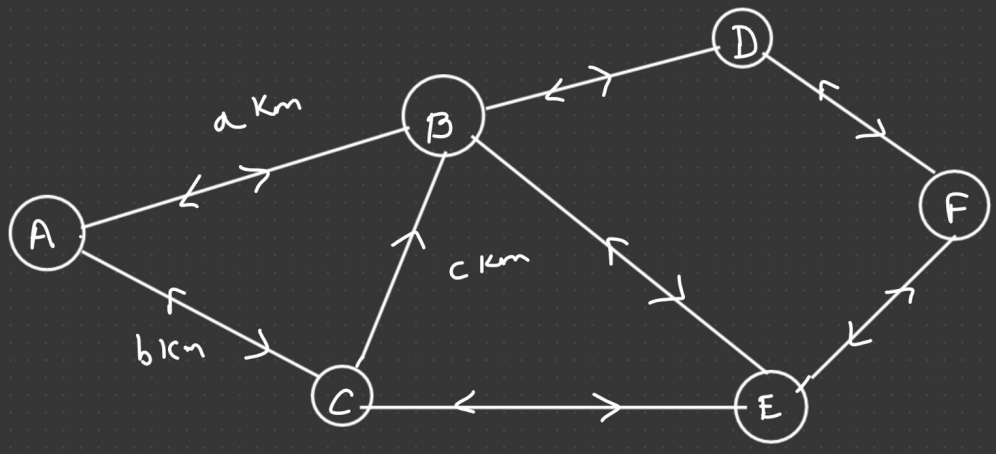
Tree

DS → 1) Linear → (Array, DA, Stack, Queue)
DS → 2) Non-Linear → (Tree, Graph)

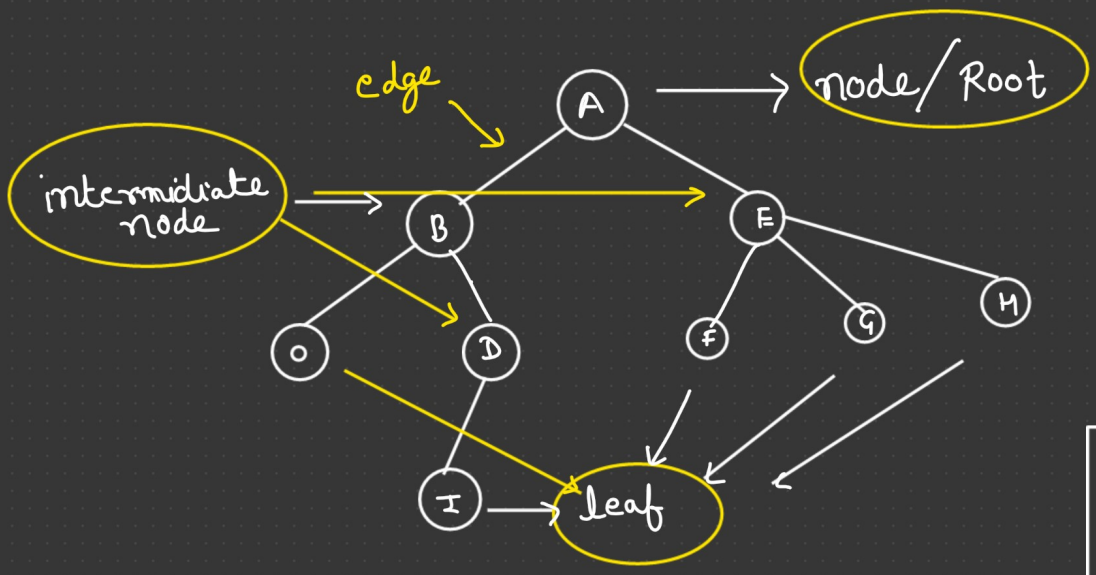


Tree





→ Graph



Root

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

Leaf Node ⇒ which do not have any child node.

1) Root → first node.

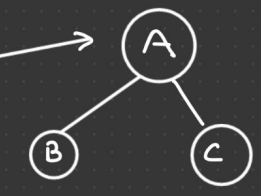
2) Leaf

3) Intermidiate Note

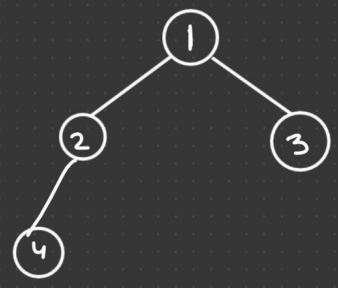
4) Edges 

5) Parent Node →

6) Child →



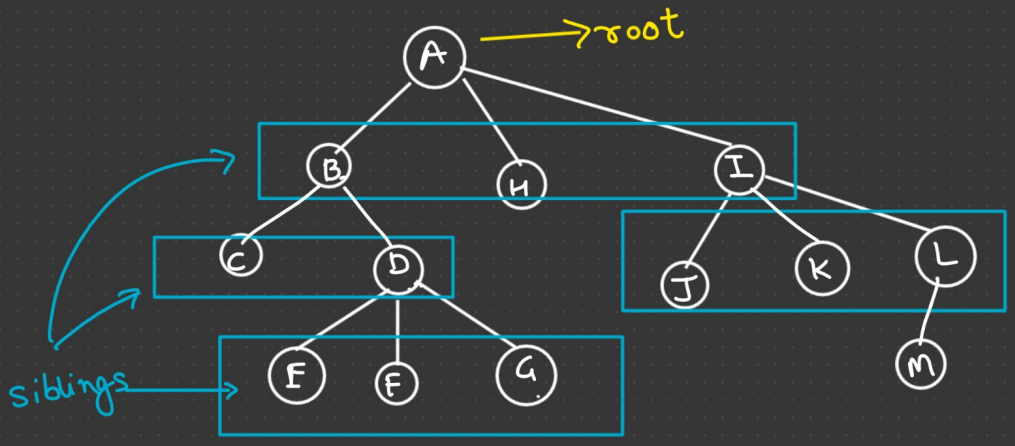
# of Nodes = 3
# of Edges = 2



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

no. of edges + 1 = No. of Nodes

7) Siblings

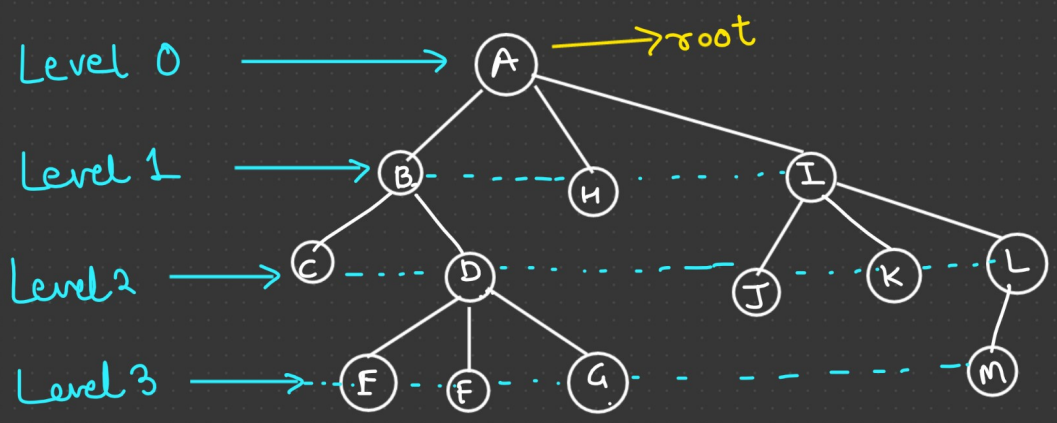


Root = A

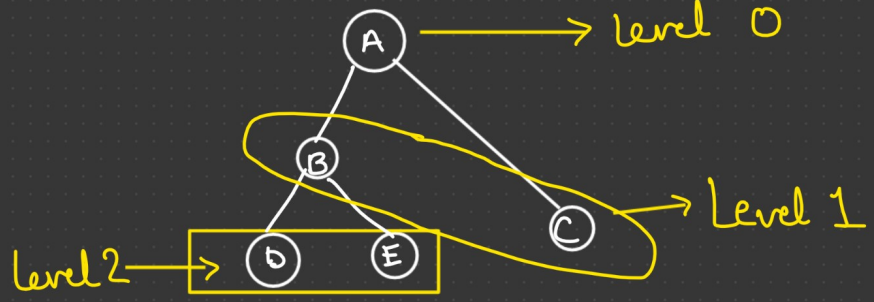
Leaf = C, E, F, G, H, J, K, M

Intermidiate = B, D, I, L

Level :-



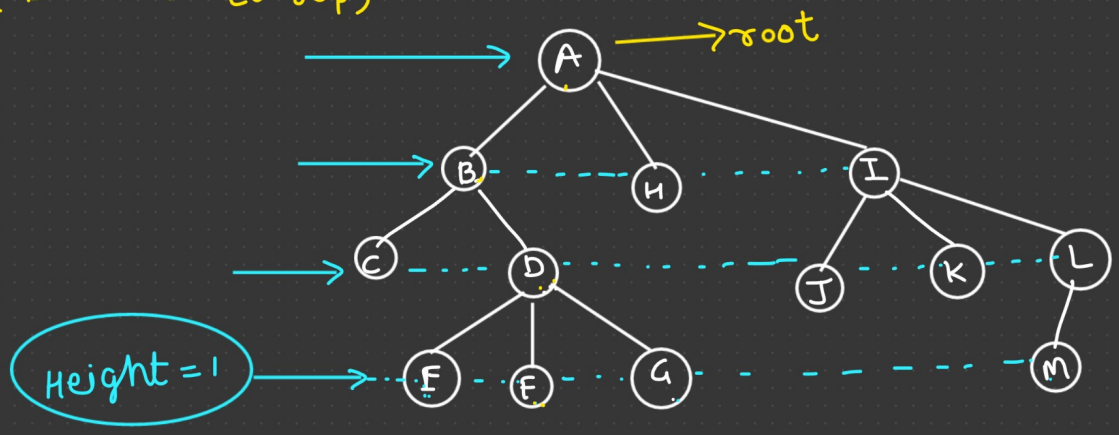
Total 4 levels → [0 - 3]



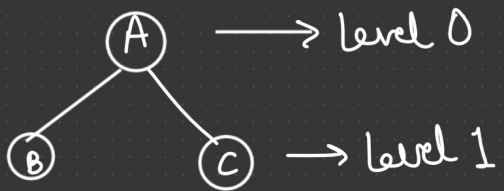
Height :- (Bottom to top)

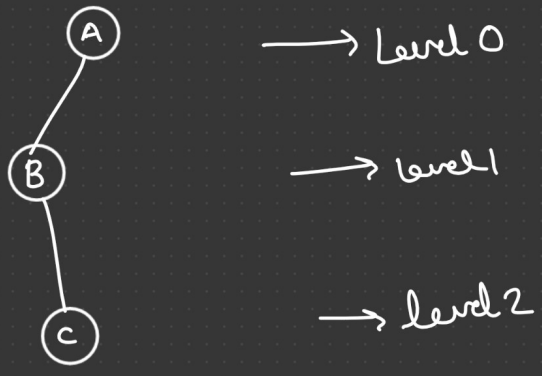
edge / node

Height = 3 / 4



Depth :- (Top to Bottom)





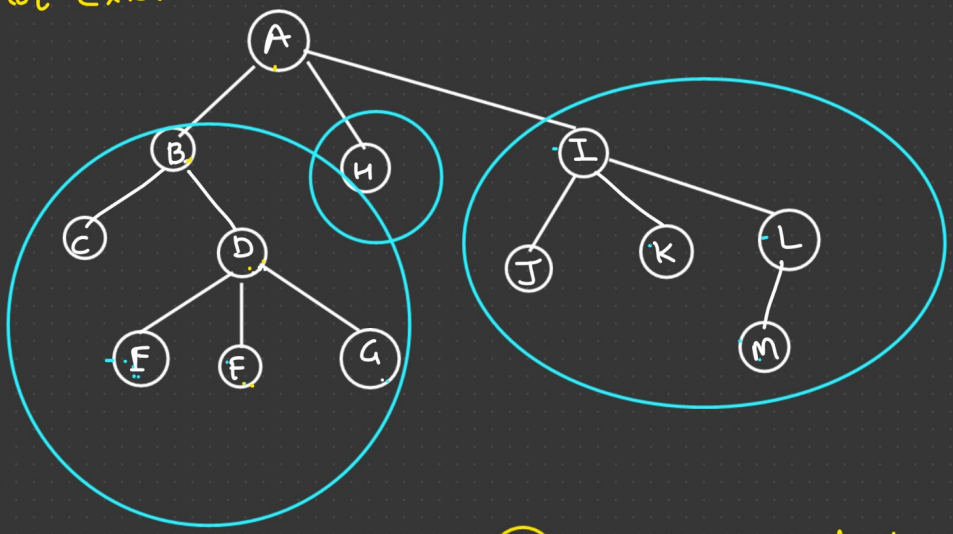
Path :-

B - K → Does not exist

A - G → A-B-D-G

A - M → A-I-L-M

B - E → B-D-E



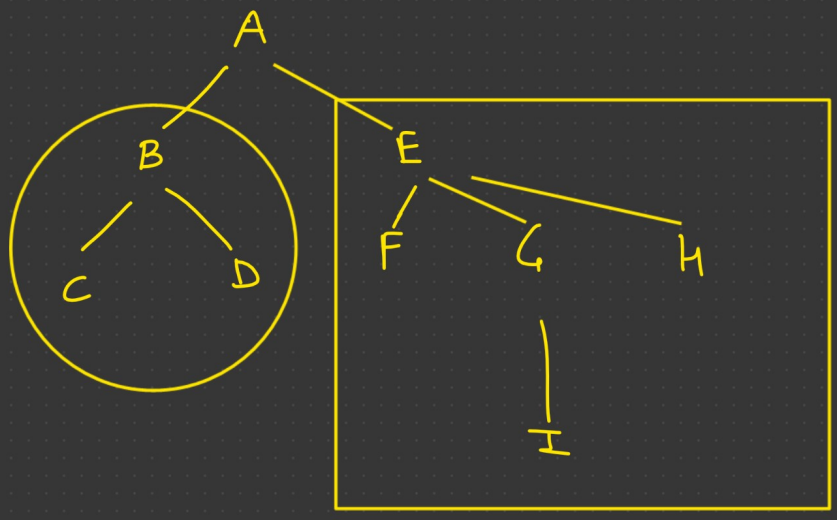
Subtree :-

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

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

$B \subseteq A$

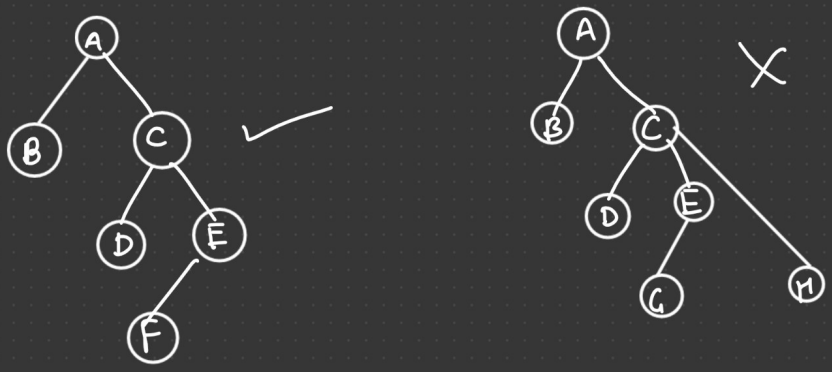
○ → single node is also a tree.



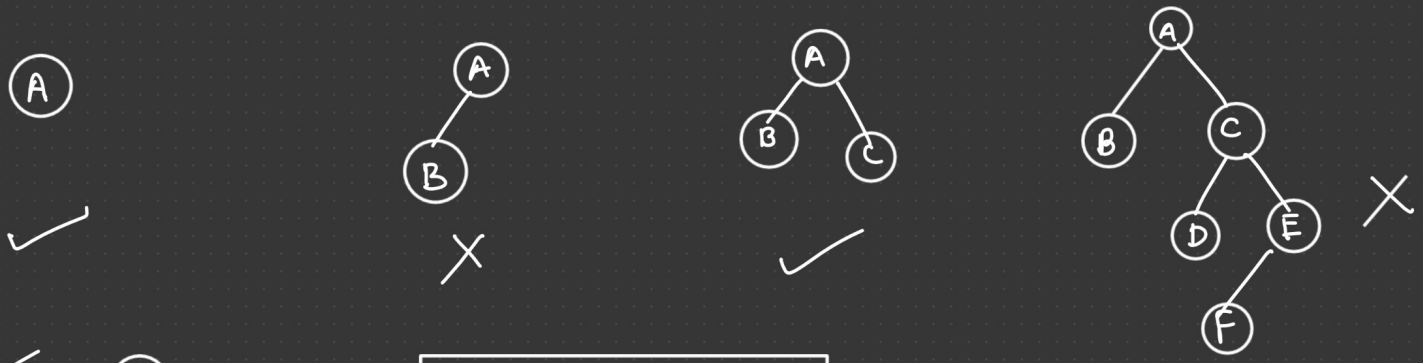
Types of Trees :-

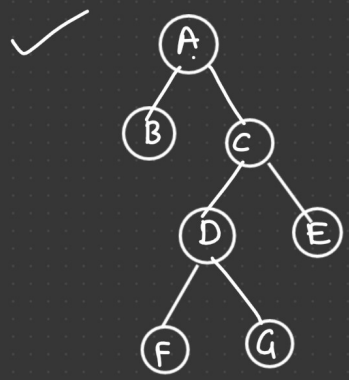
1) **Binary Tree** :- A tree is called BT if every node in it has maximum of two child nodes.

(Because it 0 child)



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





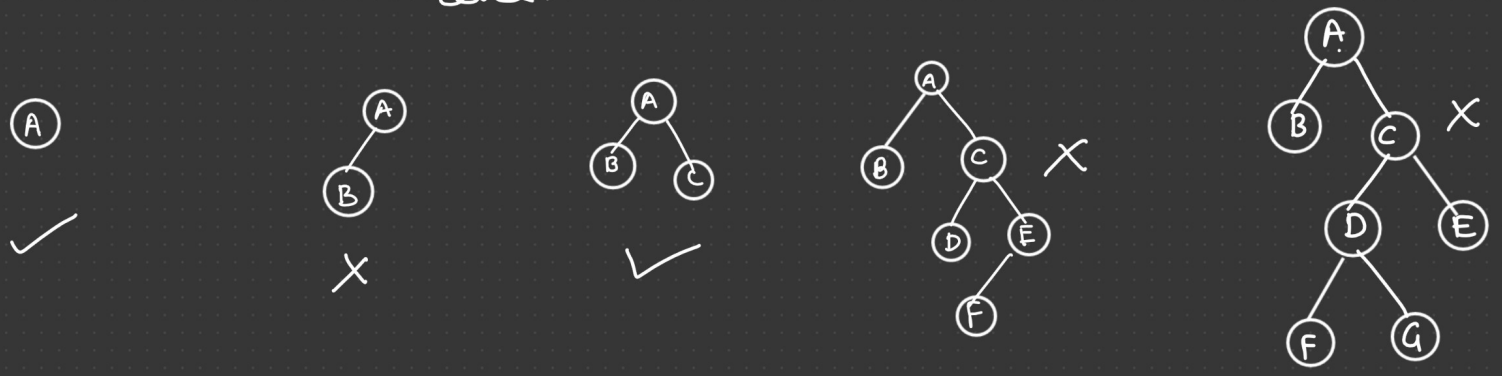
Leaf = 4

non leaf = 3

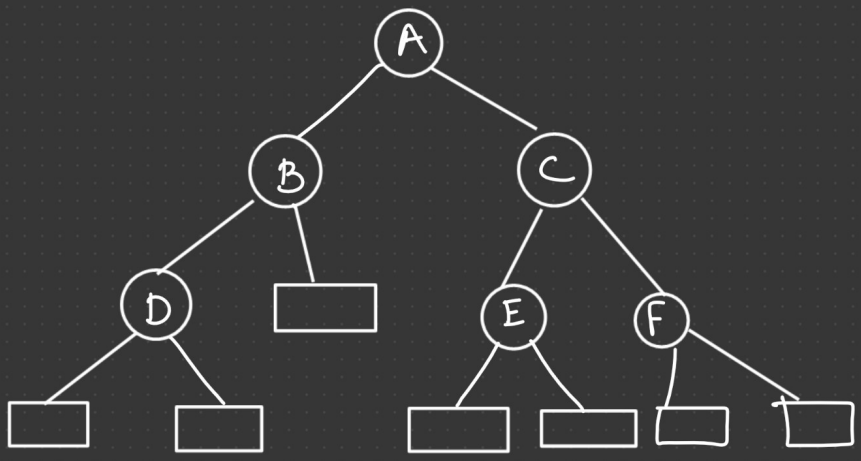
* if it has "n" non leaf node then, it must have "n+1" leaf nodes.

Complete BT :-

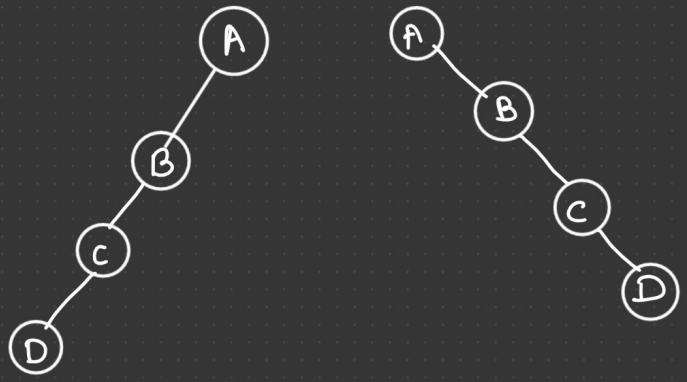
Every internal node must has exactly 2 child node. And all the leaf nodes are at the same level.



Extended BT:-



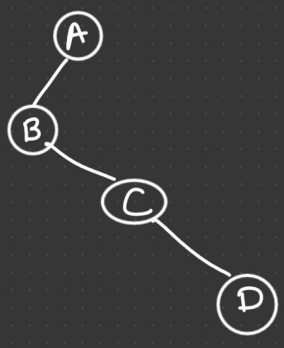
Skewed BT :-



Left - skewed

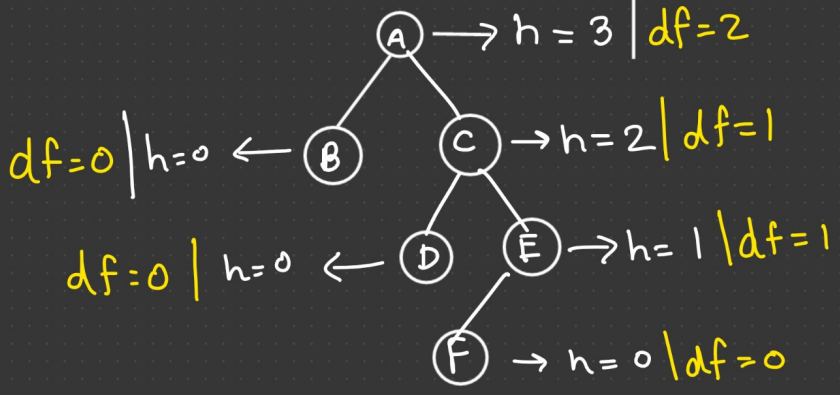
Right - skewed

Degenerate Tree :-

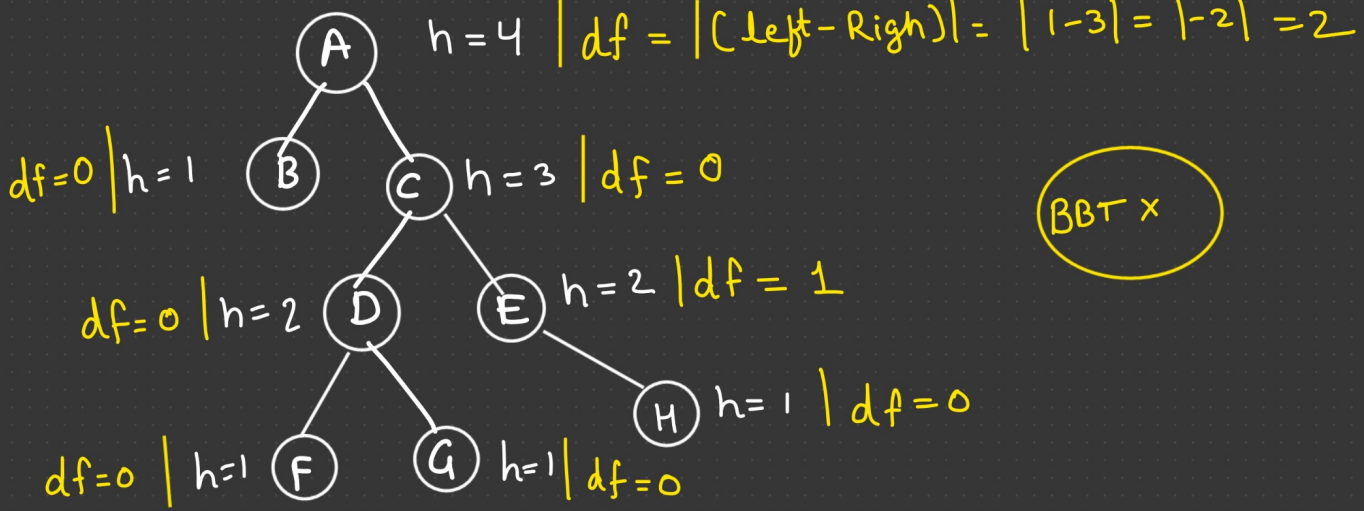


Balanced BT :-

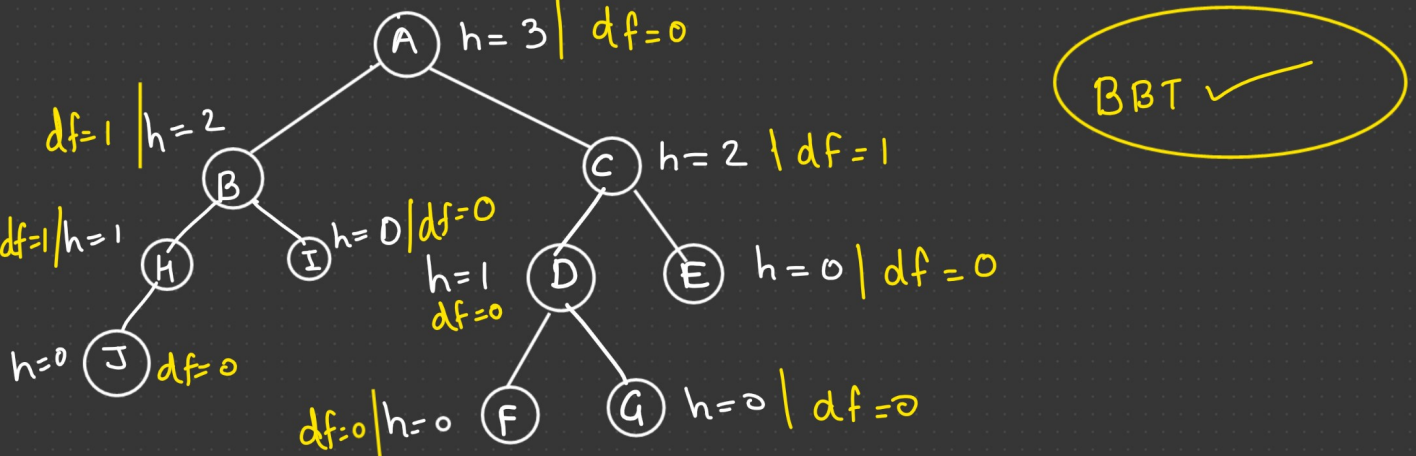
if (df = 0 'or' 1) ⟶ BBT



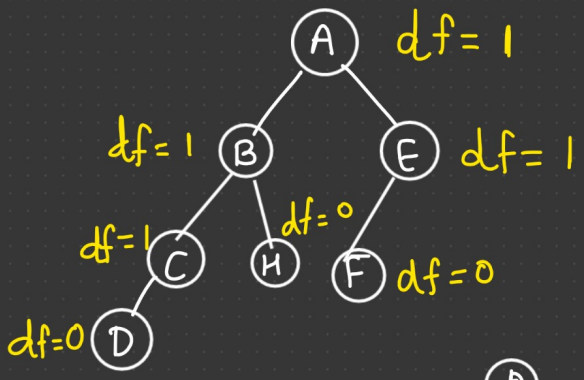
difference (df) = |(height of left subtree) − (height of right subtree)|



BBT ✗



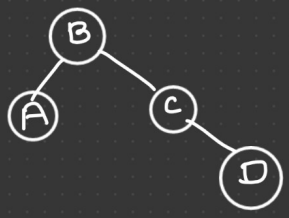
BBT ✓

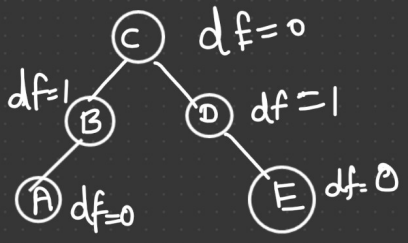

A
df = 1
df = 1
B
E
df = 1
df = 1
C
H
df = 0
F
df = 0
df = 0
D

→ BBT ✓


A
B
C
D


A
B
C
D


B
A
C
D


C
df = 0
df = 1
B
D
df = 1
A
df = 0
E
df = 0

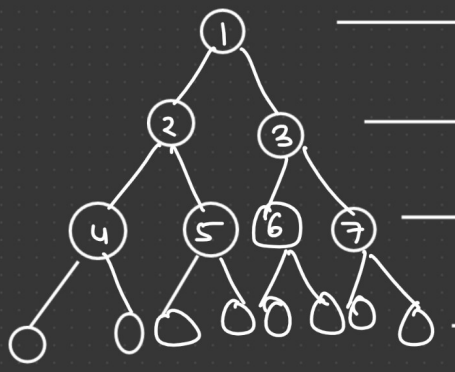
Properties of BT :-

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

minimum

degenerated BT

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



$\longrightarrow 1 \longrightarrow$ level = 0 $= 2^0$

$\longrightarrow 2 \longrightarrow$ level = 1 $= 2^1$

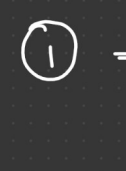
$\longrightarrow 4 \longrightarrow$ level = 2 $= 2^2$

$\longrightarrow 8 \longrightarrow$ level = 3 $= 2^3$

$\vdots$

level = l $= 2^l$

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



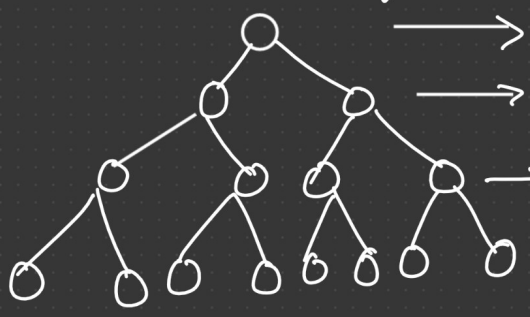
⇒ Level = 0
No. of nodes = 1
height = 1

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.



$\longrightarrow l = 0 \Rightarrow 1 \Rightarrow 2^0$

$\longrightarrow l = 1 \Rightarrow 2 \Rightarrow 2^1$

$\longrightarrow l = 2 \Rightarrow 4 \Rightarrow 2^2$

$\longrightarrow l = 3 \Rightarrow 8 \Rightarrow 2^3$

$\vdots$

$\Rightarrow 2^l$

l = 2
h = 3
n = 7

Max node = $2^h - 1 = 2^{l+1} - 1$

$= 8 - 1 = 7$

Total no. of nodes $= 1+2+4+8 \ldots +2^{l}$

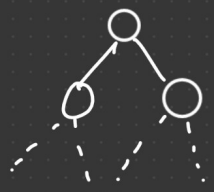
Sum of G.P $= \frac{a(r^n-1)}{(r-1)} = \frac{1.(2^{l+1}-1)}{(2-1)} = \boxed{2^{h}-1 = 2^{l+1}-1}$

4) $\boxed{e = n-1}$

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

$n = 1$
null = 2

$n = 2$
null = 3



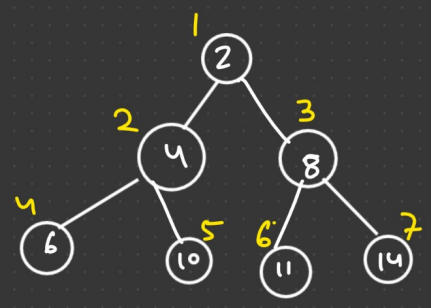
$n = 3$
null = 4

Binary Tree Representation :-

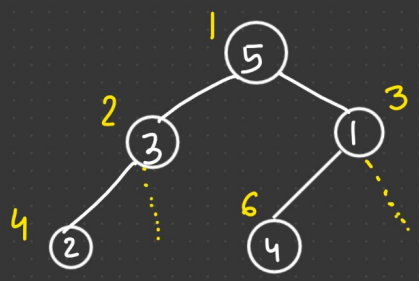
- 1) Array Representation
- 2) Linked list Representation

1) Array Representation :-

1 (Parent) = LC = 2 , RC = 3
2 (parent) = LC = 4 , RC = 5

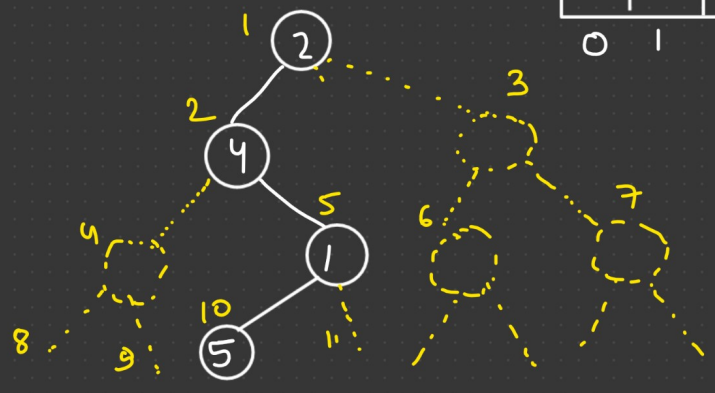


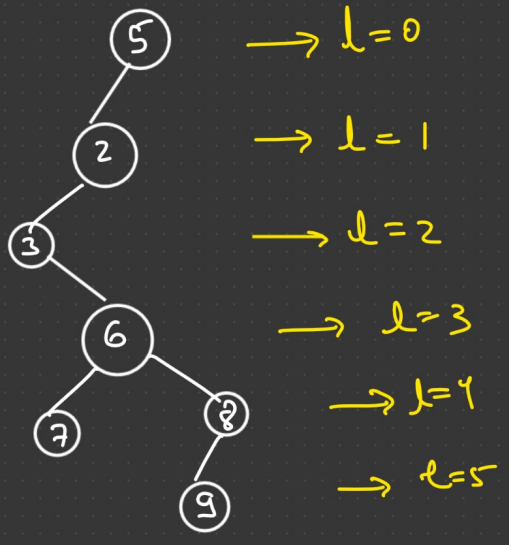
	2	4	8	6	10	11	14	
0	1	2	3	4	5	6	7	8



	5	3	1	2	–	4	–	
0	1	2	3	4	5	6	7	8

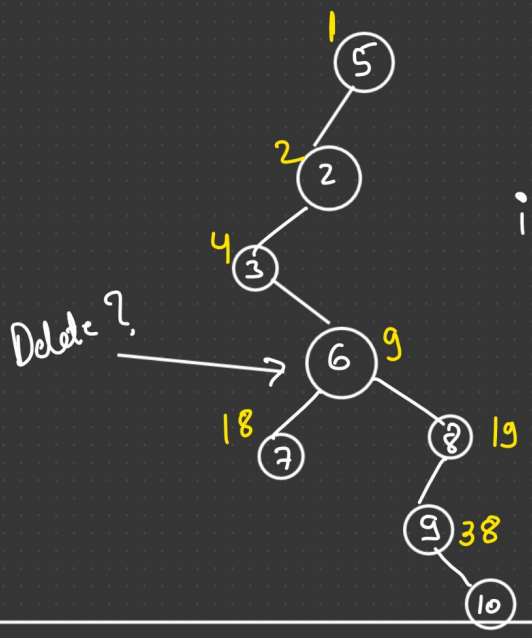
	2	4	–	–	1	–	–	–	–	5
0	1	2	3	4	5	6	7	8	9	10





$= 2^6 - 1 = 63$

$= 2^5 - 1 = 31$



7 elements ⇒ 38 nodes

if (K is parent node index)
then
2K = left child index
and
2K+1 = Right child index

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

$\lfloor 3.5 \rfloor = 3$, $\lfloor 3.1 \rfloor = 3$, $\lfloor 4.9 \rfloor = 4$, $\lfloor 5 \rfloor = 5$

```
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;
}
```

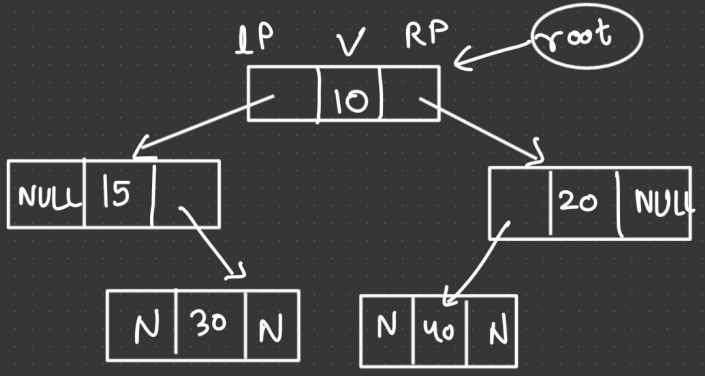
Drawbacks of Array implementation :-

1) we can't change the size of Tree dynamically.

2) Wastage of memory.

---

2) Linked List Reprentation :-



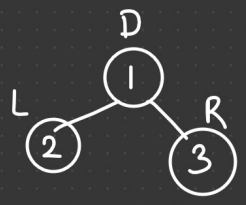

```
class node
{
 Public:
    int data;
    node *left;
    node *right;
    node(int v)
    {
      data = v;
      left = NULL;
      right = NULL;
    }
};
```

```
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



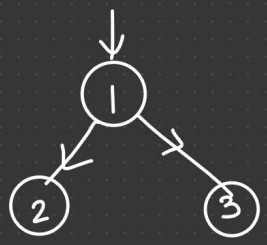
D, L, R
D, R, L → 1 (Root) → Preorder

L, D, R
R, D, L → 2 (Root) → Inorder

L, R, D
R, L, D → 3 (Root) → Postorder

→ Level order

1, 2, 3
1, 3, 2
2, 1, 3
2, 3, 1
3, 1, 2
3, 2, 1

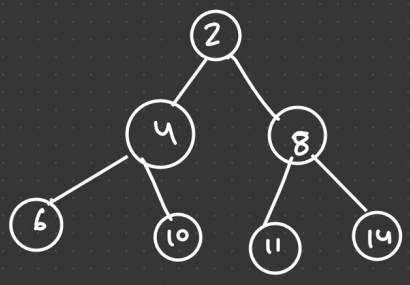


6

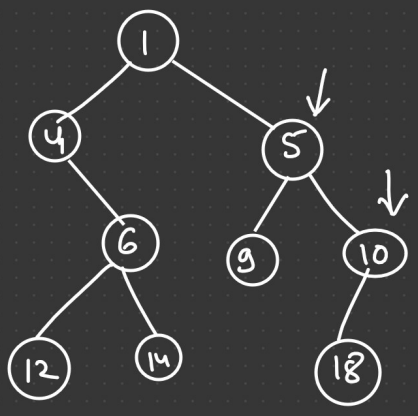
Preorder Traversal :-

(Root → left subtree → Right subtree) (It is same as DFS)

(we can use stack)

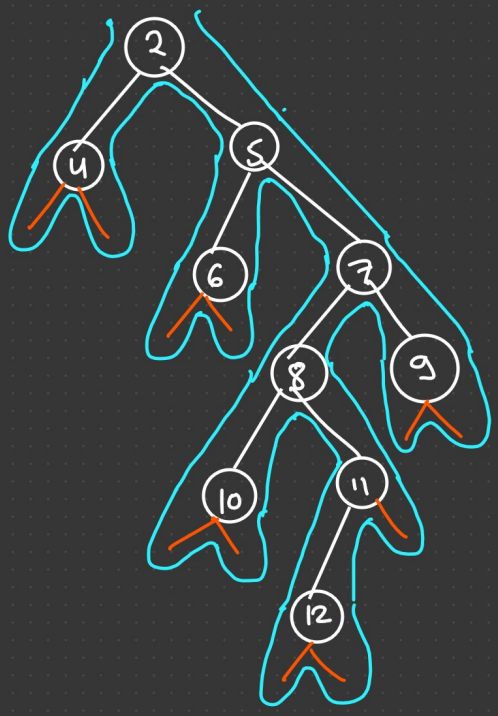


2, 4, 6, 10, 8, 11, 14



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

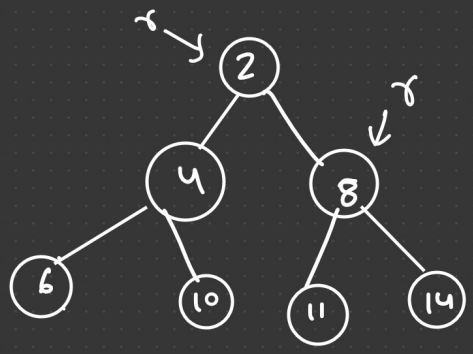
2nd (Inorder) [L – D – R]

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

3rd (Postorder) [L – R – D]

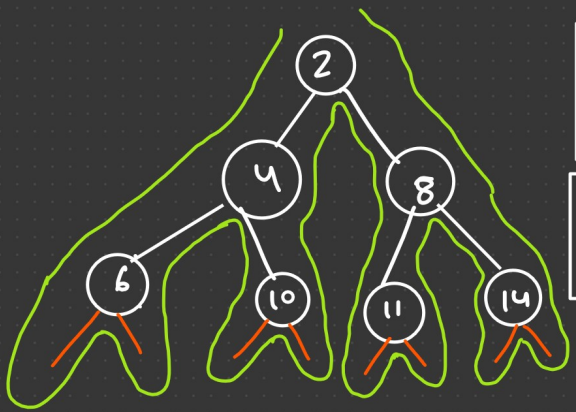
4, 6, 10, 12, 11, 8, 9, 7, 5, 2

Inorder Traversal :- (Left → Root → Right)



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

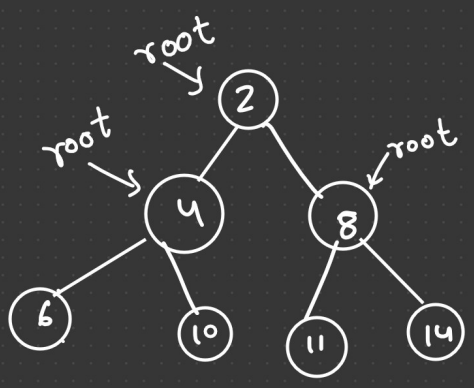
Post order Traversal :— (Left Subtree → Right Subtree → Data)



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



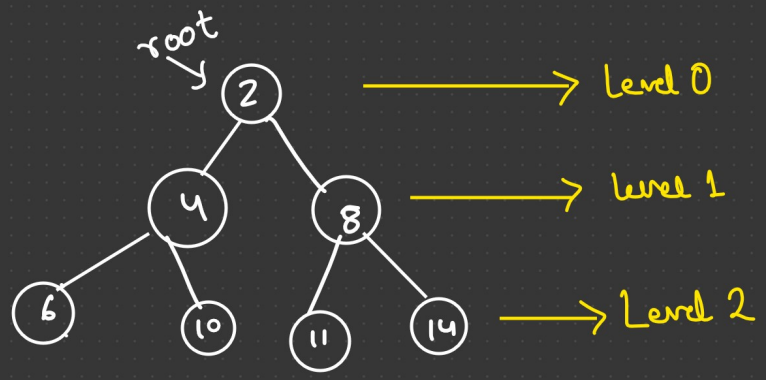
Preorder Traversal :-

[D → L → R]

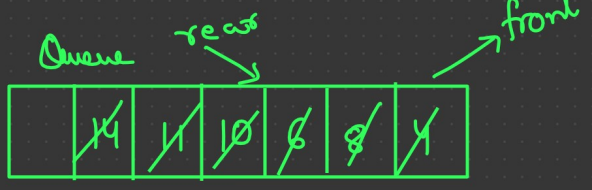
print Visit Visit

```
Void Preorder( Node * root)
{
    if ( root != NULL)
    {
        cout << root → data << " ";
        preorder (root → left);
        preorder (root → right);
    }
}
```

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

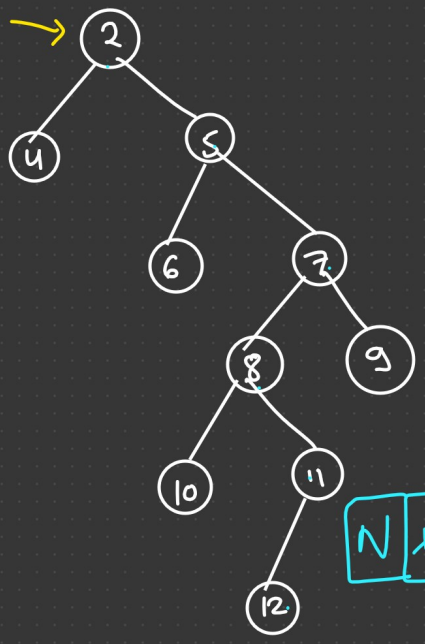


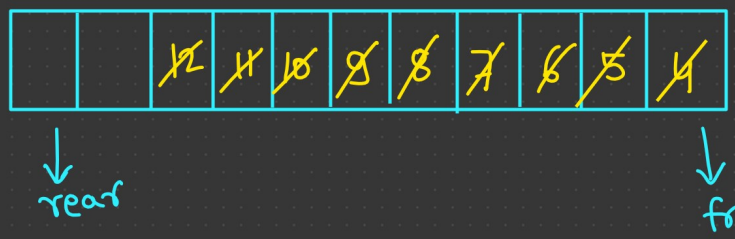
2, 4, 8, 6, 10, 11, 14



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

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

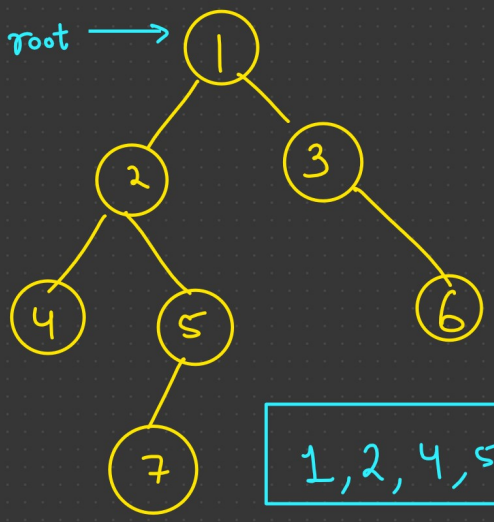




2, 4, 5, 6, 7, 8, 9, 10, 11, 12

count = 1 2 3 4 5 6

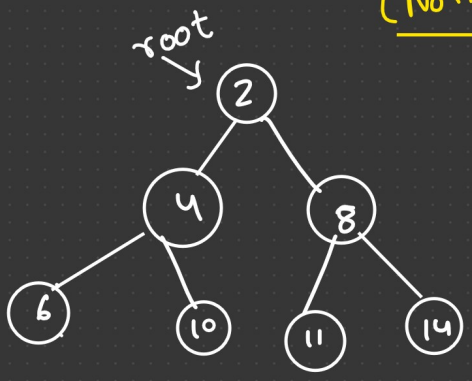
```
Void preorder ( Node *root)
{
    if( root != NULL)
    {
        cout<< root->data;
        preorder (root->left);
        preorder (root->right);
    }
}
int main()
{
  preorder (root);
}
```

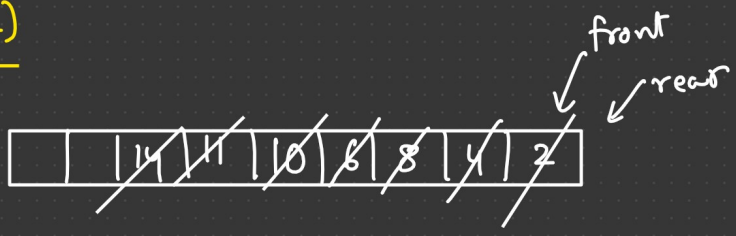


1, 2, 4, 5, 7, 3, 6

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

(Non- Recursive)

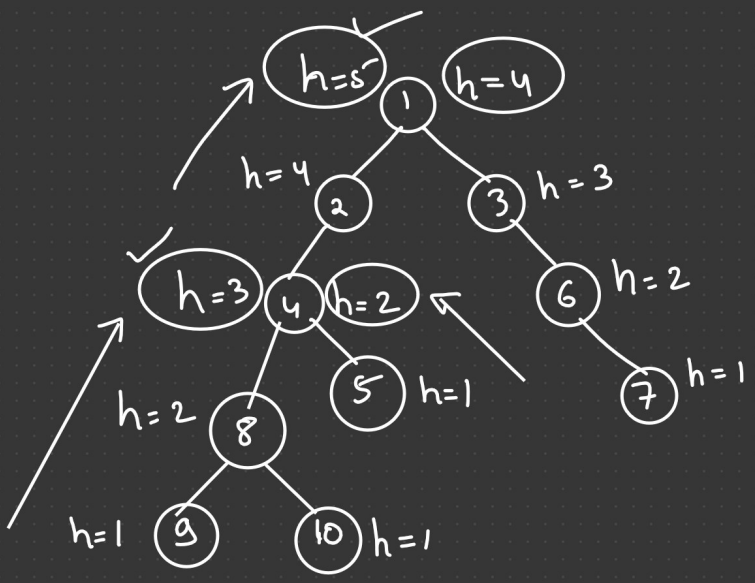




2, 4, 8, 6, 10, 11, 14,

```
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 ⇒ Height(1) = 5

Height(4) = 3

Height(1) = [lTree = Height(2) = 4

rTree = Height(3) = 3

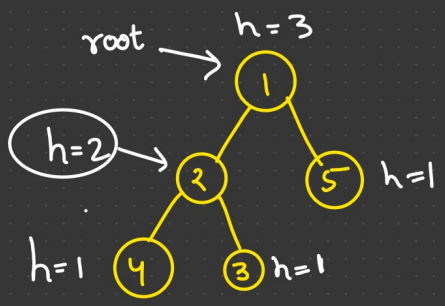
if(lTree > rTree)

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

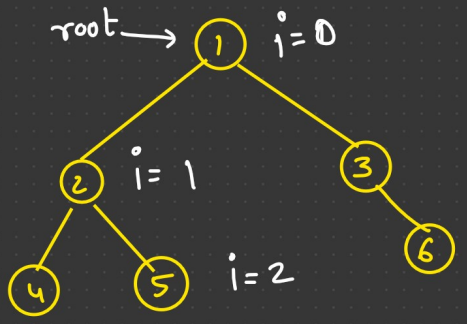
l = H(2) ⇒ 2

r = H(5) = 1

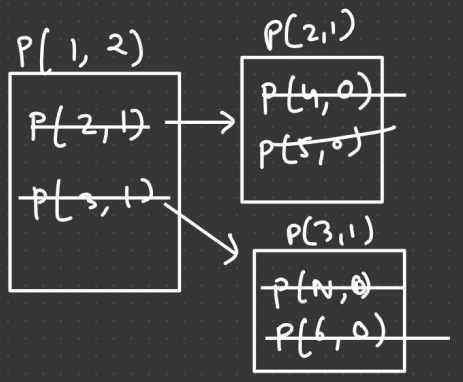
return 3

Level order Traversal (Recursive) :-

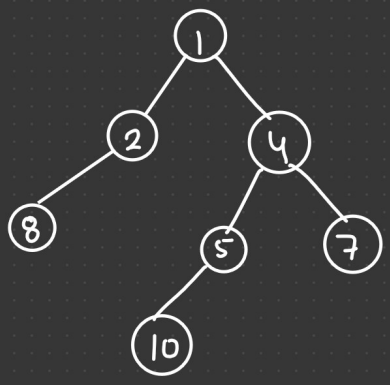
```
void printLevelOrder(Node *root)
{
    int h = Height(root);        = 3

    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);
}
```



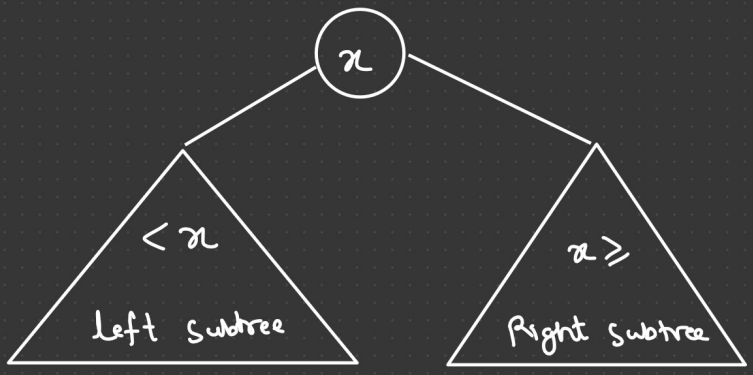
1, 2, 3, 4, 5, 6



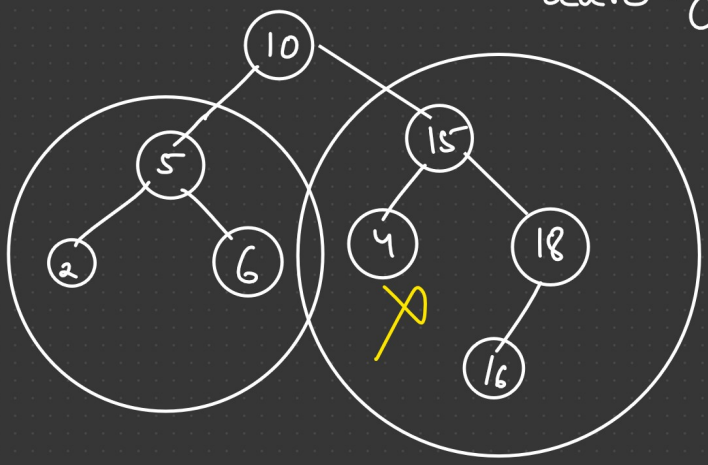
Binary Search Tree



Time Complexity :- $O(n)$

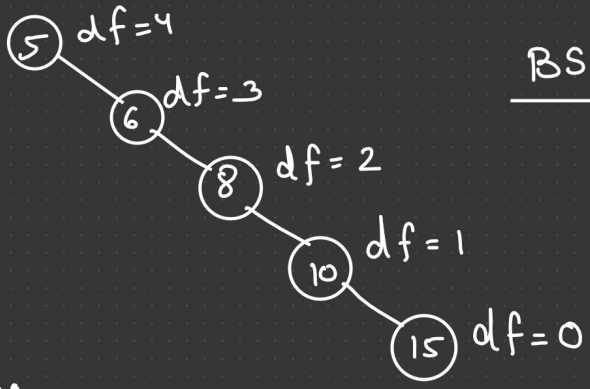


Searching TC = $O(\log n) < O(n)$



X BST

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



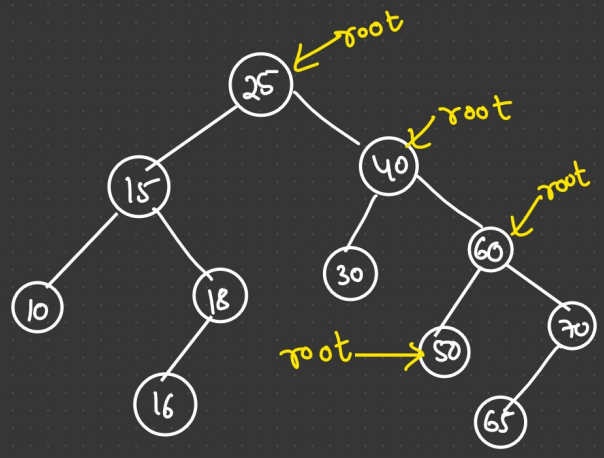
BST ✓

Problems:-

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

Searching :—

Key = 50

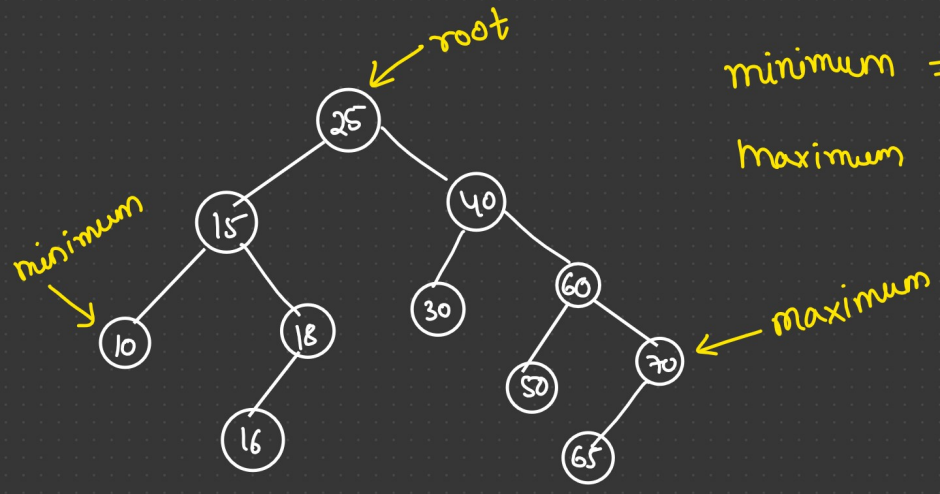



```
Node *Search(Node *root, int key)
{
  if(root == Null )
      return NULL;
  if(root->data == key)
      return root;
  else if (root->data > key)
  return search(root->left, Key);
  else
  return search(root->right, Key);
}
```

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

```
Node *root;
```

Minimum & Maximum value in a BST :-



minimum ⇒ Left most element of Tree

maximum ⇒ Right most element of Tree

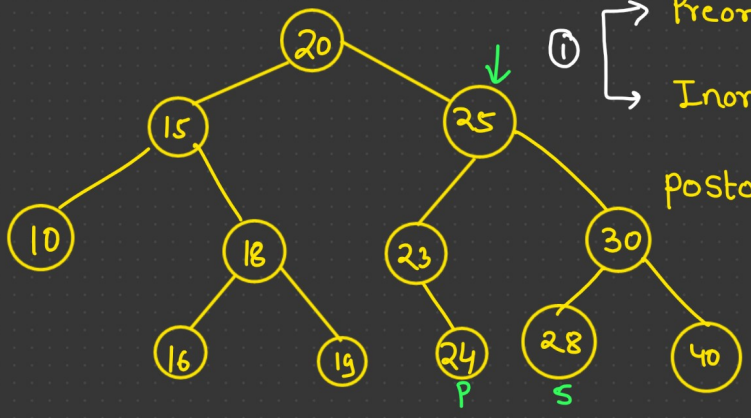
```
Node * MinElement ( Node *root)
{
    if (root == NULL)
        return NULL;
    else if ( root->left == = NULL)
        retuen root ;
    else
        minElement (root->left);
}
```

```
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)
        retuen 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.

[Node → Right → left most] (1 step)

[Node → Left → Right most] (1 step)

```
Node * InorderSuccessor (Node * node)
{

    return MinElement (node → right);

}
```

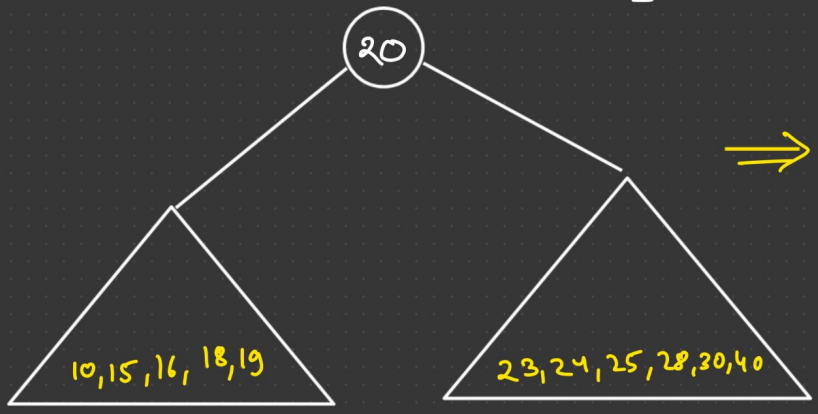
```
Node *InorderPredecessor (Node * node)
{
    return MaxElement (node → left);
}
```

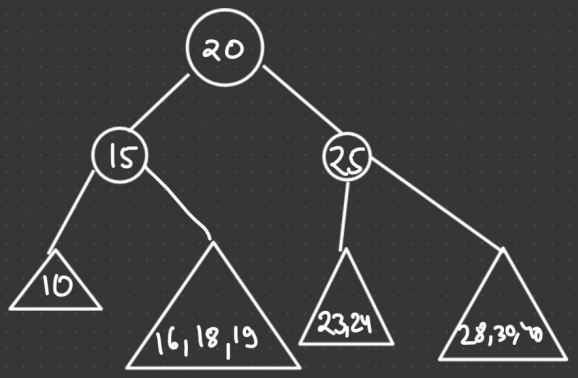
Create BST using Inorder & preorder Traversal :-

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

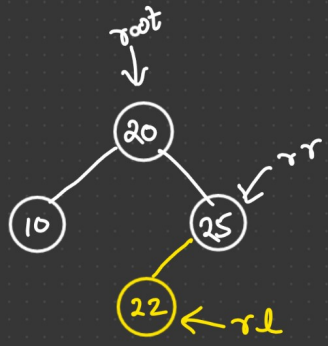
Postorder & Inorder ✓

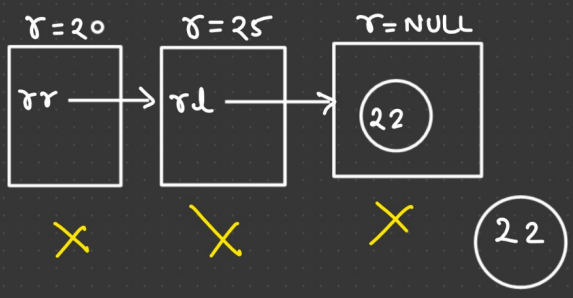




Preorder & Postorder = ? (Not possible)

Creation / Insertion in BST :-



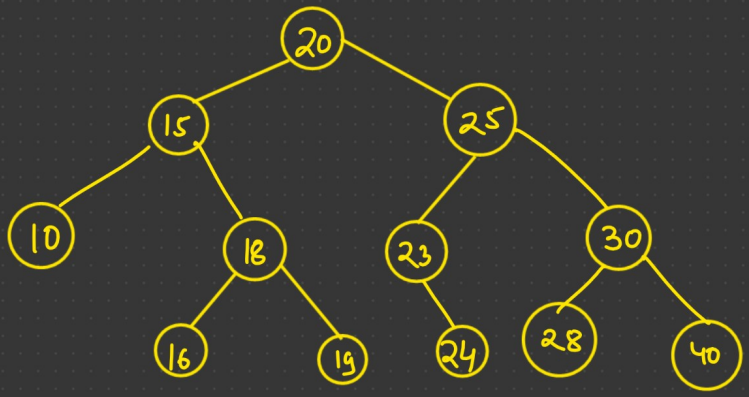



```
Node * Insert(Node *root, int value)
{
    if (root == NULL)
        return new Node(value);
    if (root -> data < value)
    root->right = Insert(root->right, value);
    if (root -> data >= value)
    root->left = Insert(root->left, value);
}
```

1) Time complexity of Inserting a node = O(logn)

2) Time complexity to create BST = O(n logn)

Deletion in BST :-

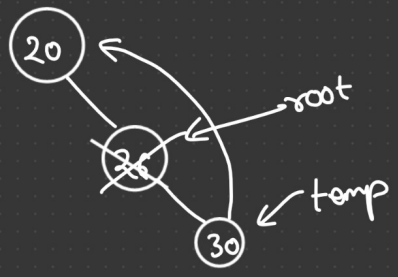


1) Leaf Node (zero child)

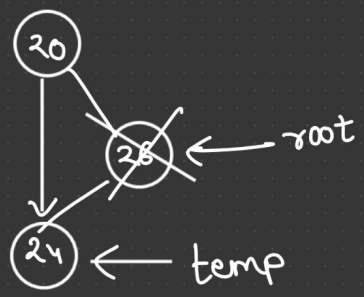
2) Non-Leaf Node (one child)

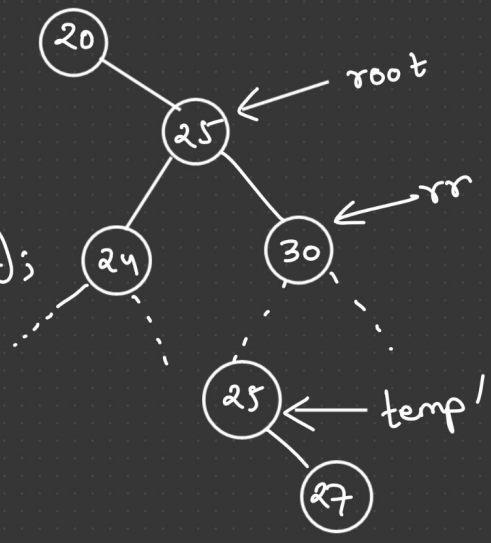
3) —•— (two child)

```
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;
    }
```

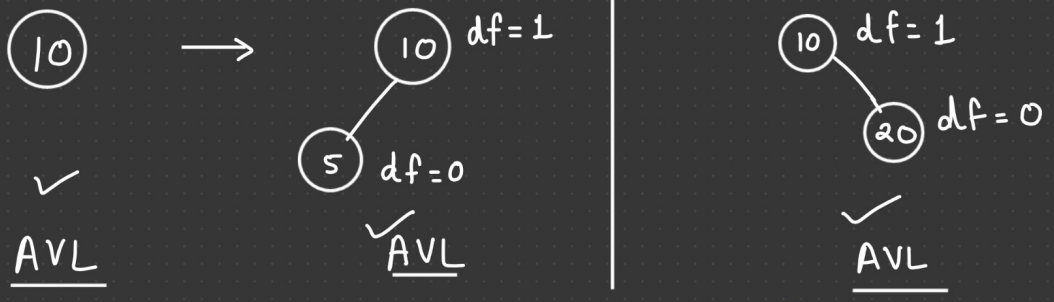


```
else if (root->right == NULL)
{
    Node *temp = root->left;
    delete root;
    return temp;
}
else {          // two child
    Node* temp = minElement(root->right);
    root->data = temp->data;
    root->right = delete(root->right, temp->data);
}
return root;
}
```

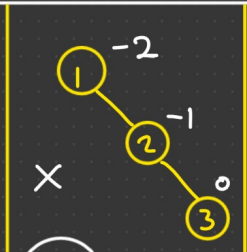




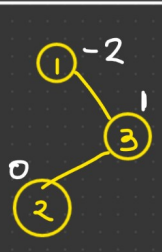
AVL Tree :- It is same as BST + Balanced Tree



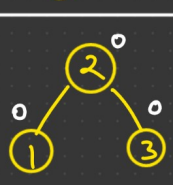
	1	2	3	4	5	6
1)	1 2 3					
2)	1 3 2					
3)	2 1 3					
4)	2 3 1					
5)	3 1 2					
6)	3 2 1					



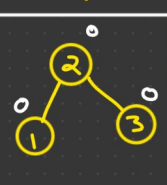
× RR (2) if(C.N→R<key)



× RL (4) if(C.N→R>Key)



✓ AVL



✓ AVL

× LR (3) if(C.N→L<K)

× LL (1) if(C.N→L>Key)

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

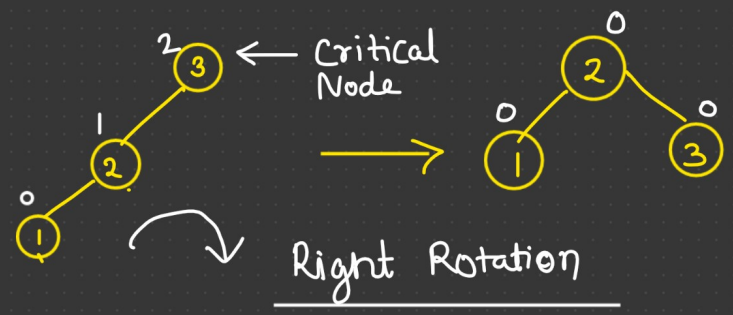
AVL ✓ ⇒ BF = 0, +1, −1

- AVL Tree is self Balancing Tree in nature.
- AVL = Adelson Velskii & Landis

---

1) LL Tree :−

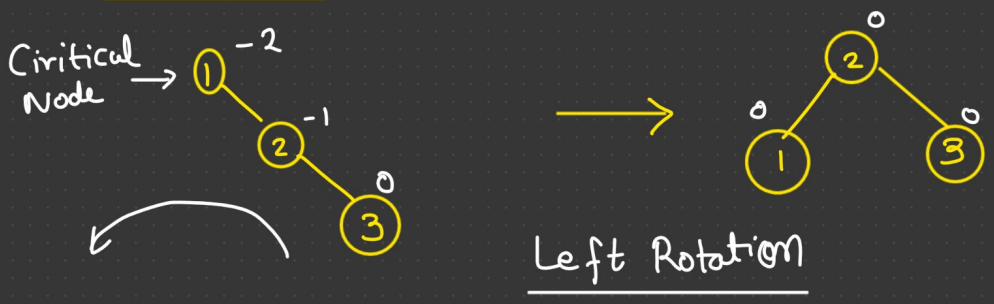
Reason :− Left-Left insertion



---

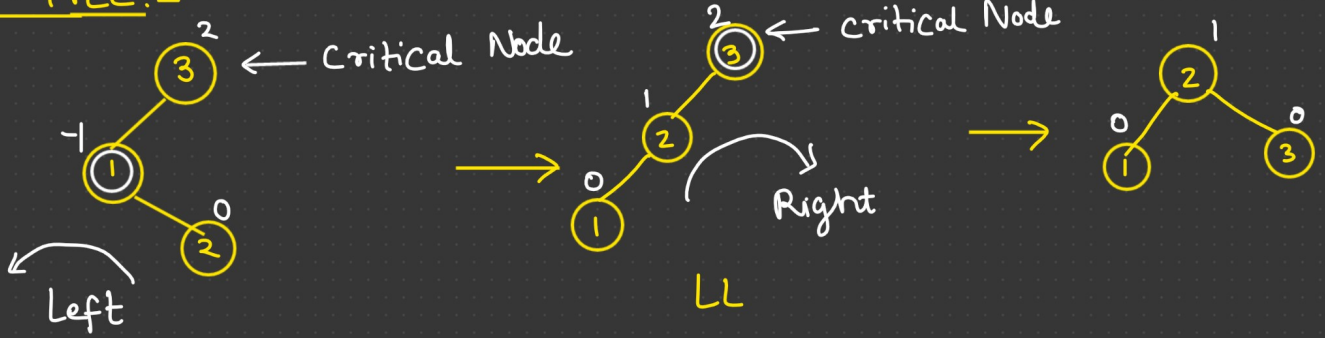
2) RR Tree :−

Reason :− Right-Right insertion



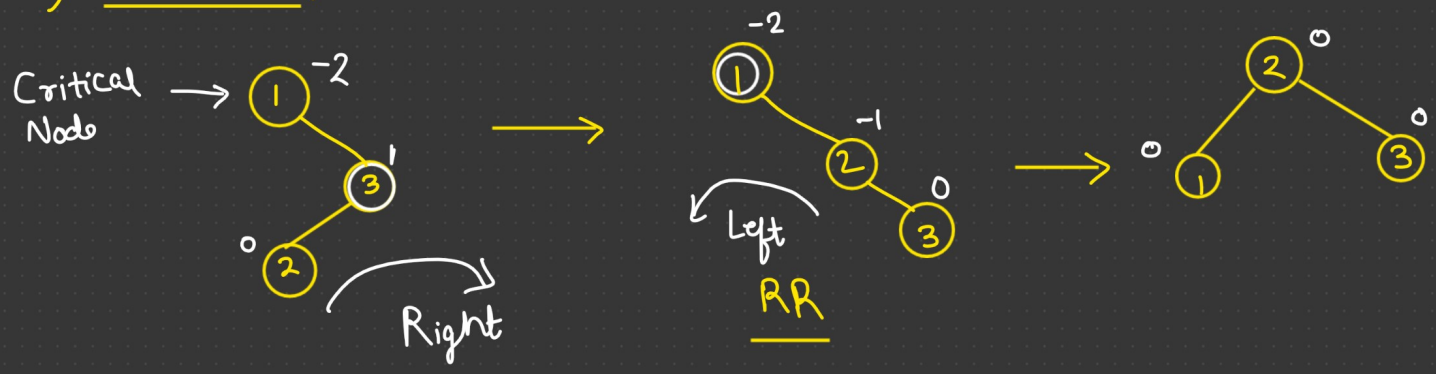
---

3) LR Tree :−



Reason :- Left - Right insertion

4) RL Tree :-

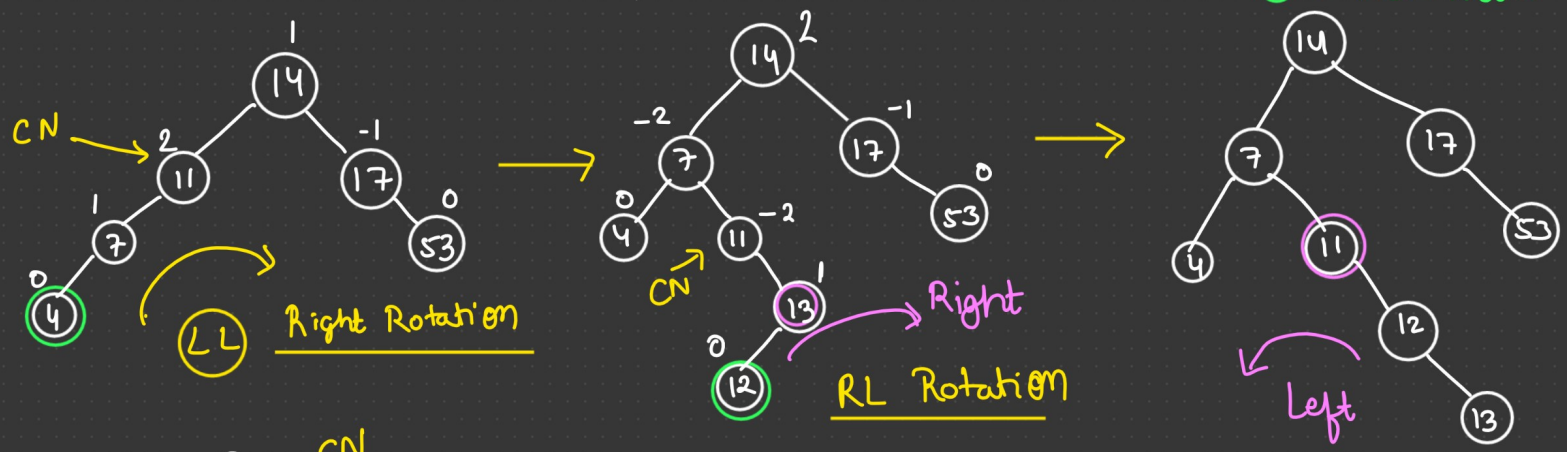


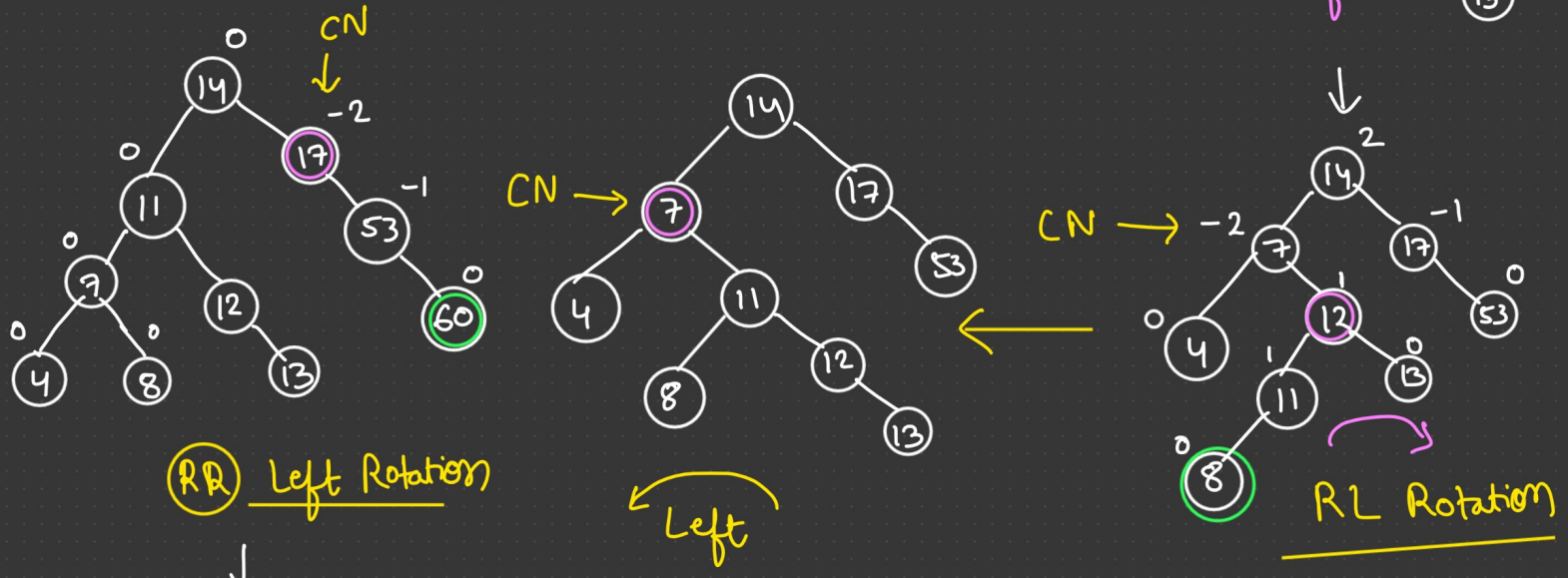
Reason :- Right - Left insertion.

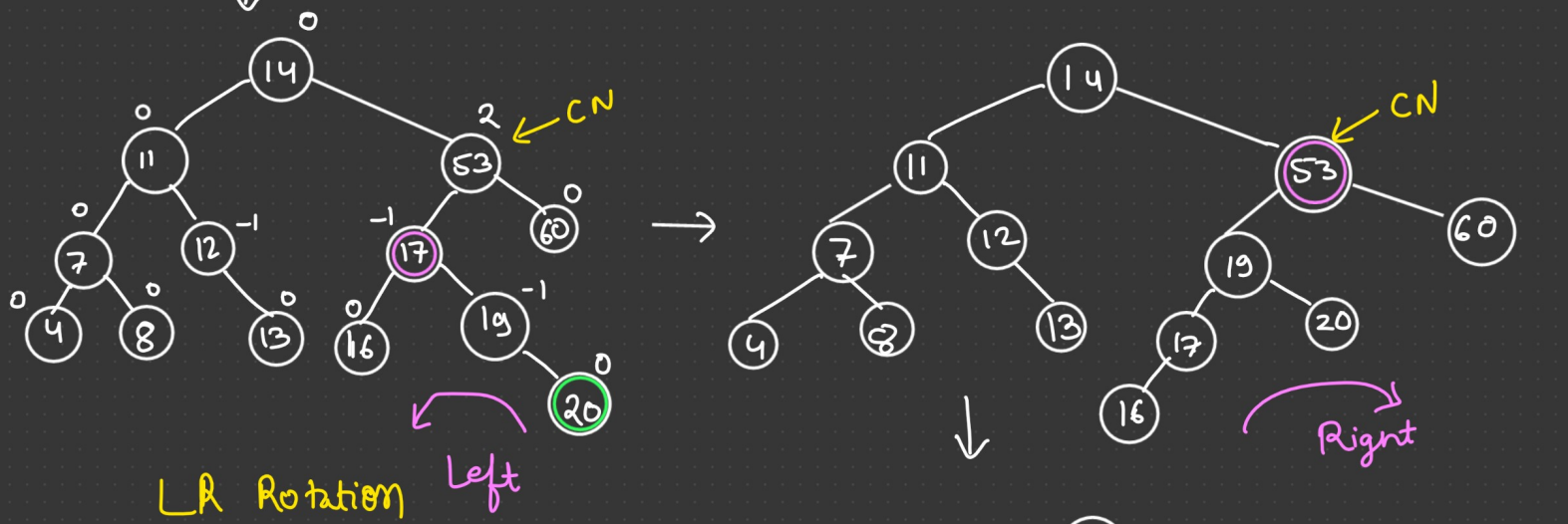
Ex:- 14, 17, 11, 7, 53, 4, 13, 12, 8, 60, 19, 16, 20

○ → Rotation About which Node

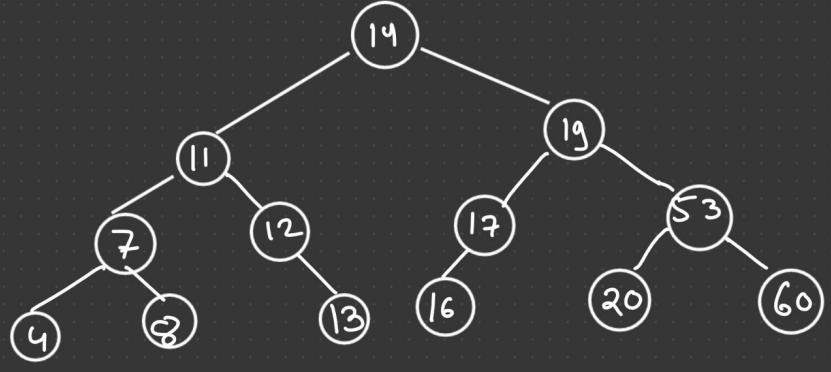
○ → New Node



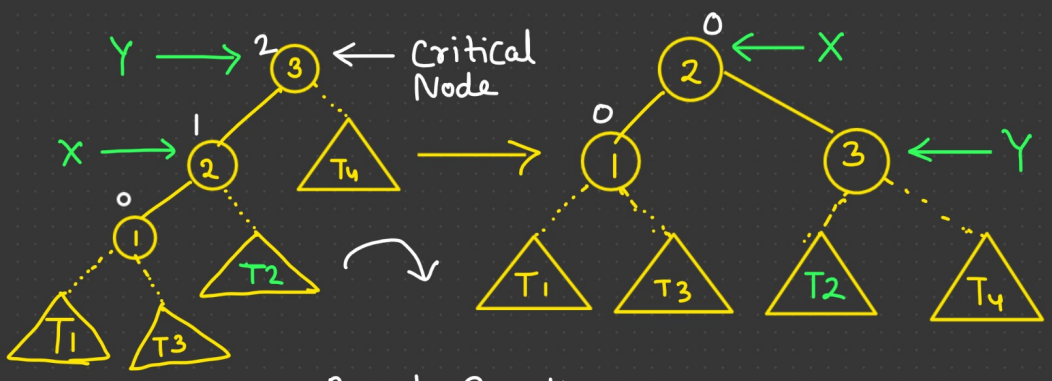




Final AVL Tree →



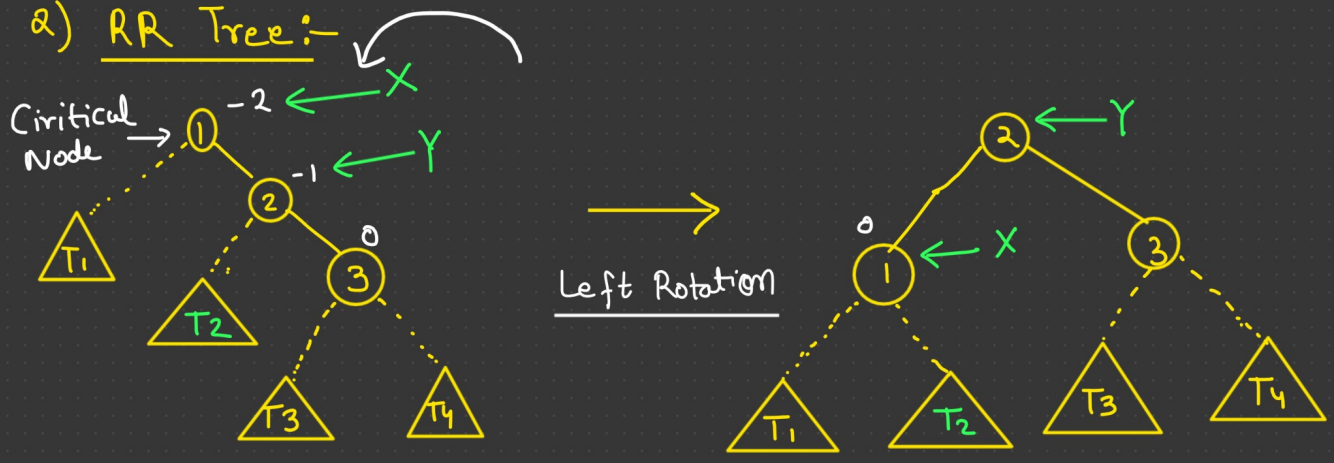
1) LL Tree :-




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

    Node(int v)
    {
        key = v;
        left = NULL;
        right = NULL;
        height = 1;
    }
};
```

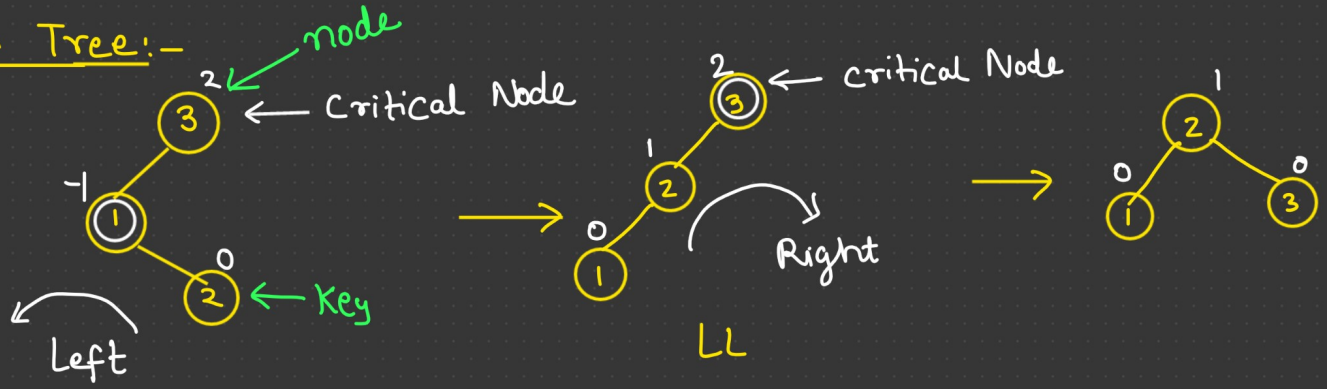
```
Node* RightRotation(Node* Y)
{
Node * X = Y->left;
Node * T2 = X->right;
X->right = Y;
Y->left = T2;
Y->height = max( H(Y->left), H(Y->right)) +1
X->height = max(H(X->left), H(X->right)) +1;
  return X;
}
```

2) RR Tree :-




```
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;
}
```

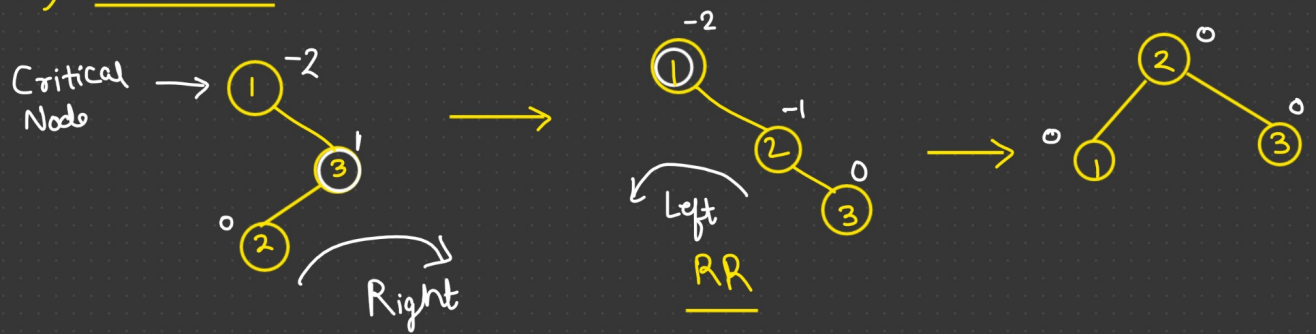
3) LR Tree:-



1) Check BF
2) Node = C.N
3) C.N → left < > Key
   - LL ✓
   - LR ✓

```
int Balancefactor = getBalancefactor(Node);

if( Balancefactor > 1)
{
    if(Node->left->value > key)
    return RightRotation( Node );          } LL

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

4) RL Tree:-




```
if( Balancefactor < -1)
{
    if( Node->right->value < key)
        return LeftRotation (Node);
    else {
        node->right = RightRotation (Node->right);
        return LeftRotation (Node);
    }
}
```

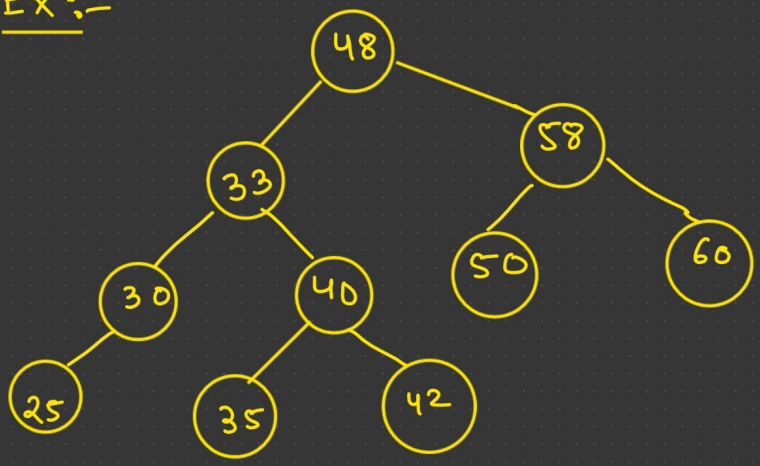
Deletion in AVL Tree:-

You must know 2 things:-

1) Deletion BST.

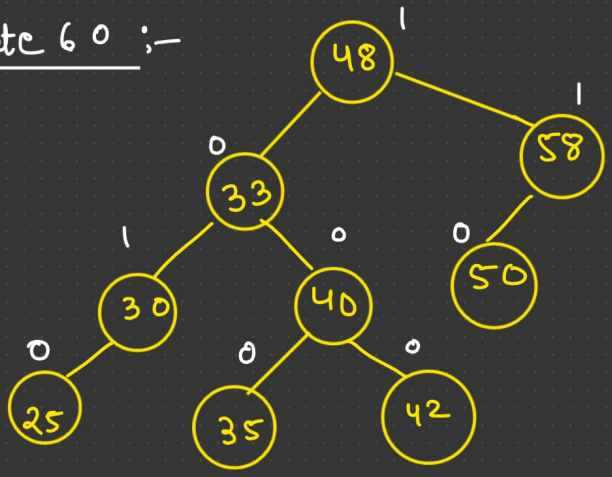
2) All types of Rotation.

Ex:-



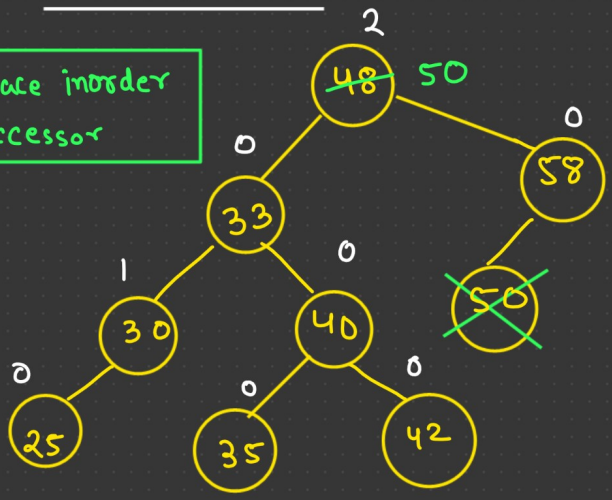
60, 48, 25, 30, 35, 33, 58

1) Delete 60 :-



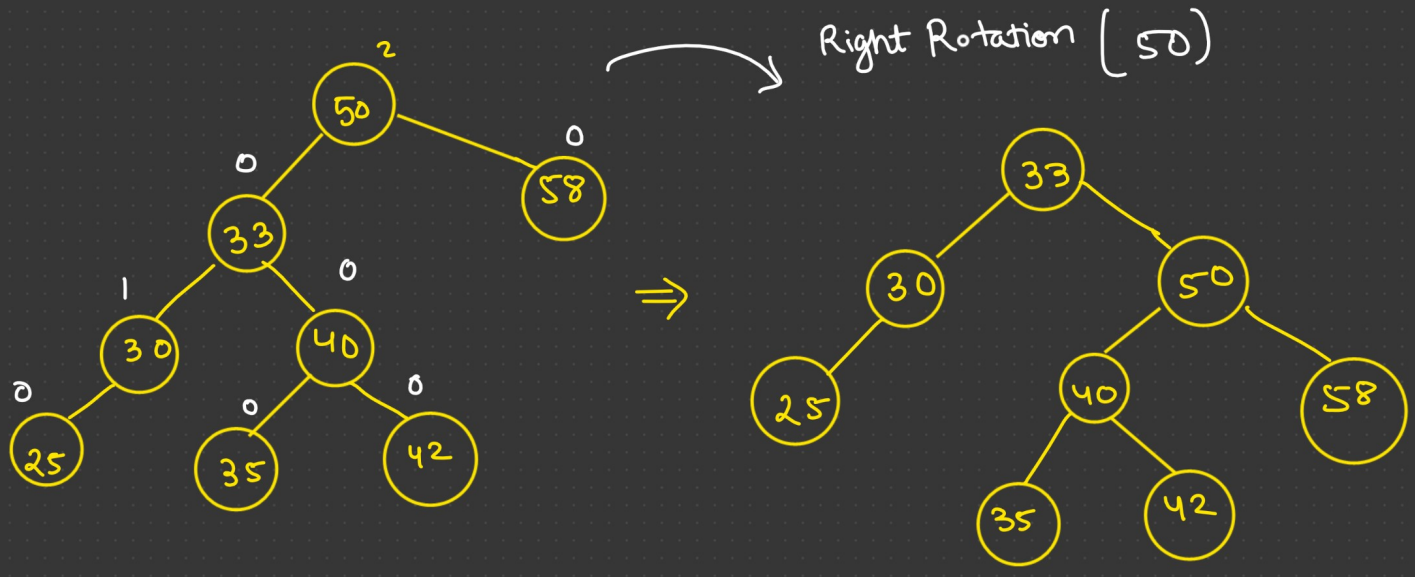
2) Delete 48:-

Replace inorder successor

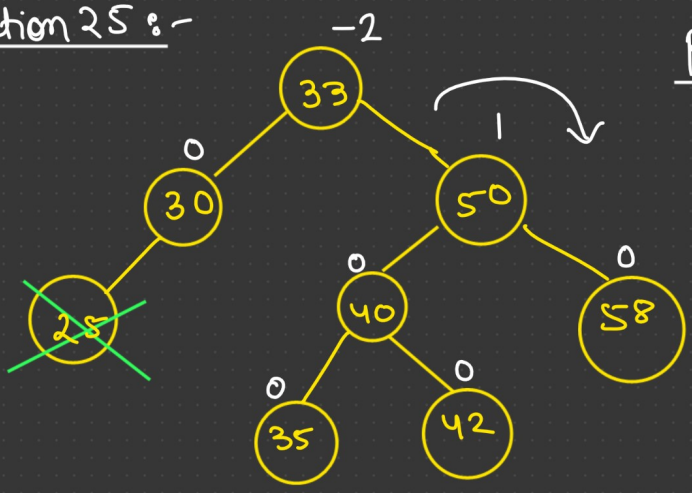


⇓

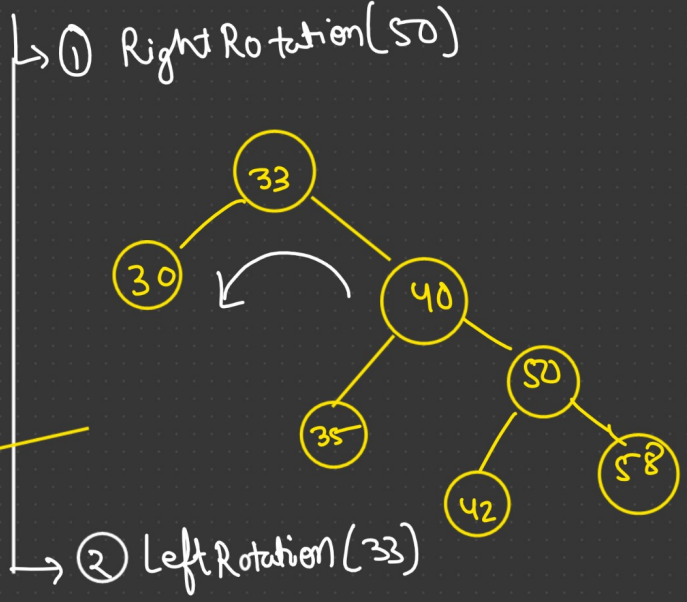
we need rotation to make it AVL.



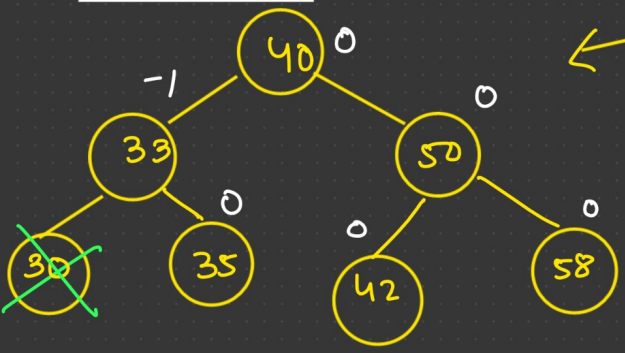
3) Deletion 25 :-



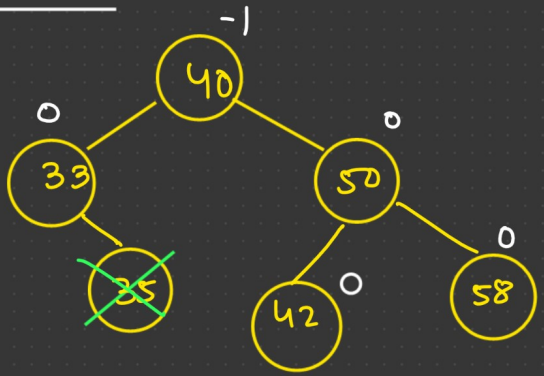
RL Rotation



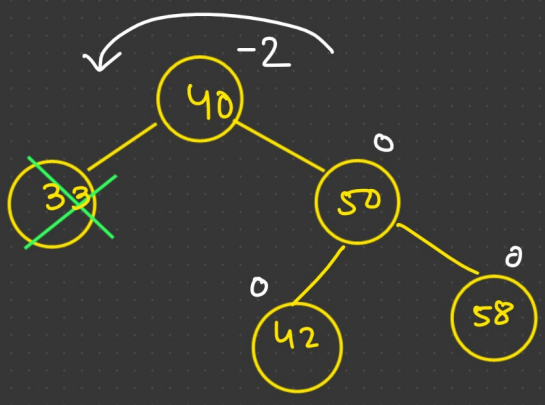
4) Deletion 30 :-



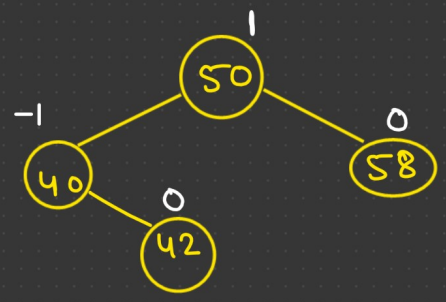
5) Delete 35 :-



6) Delete 33 :-

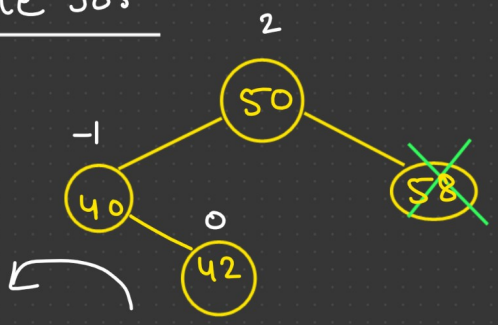


Now two possibility → RR / LR



Left Rotation (40)

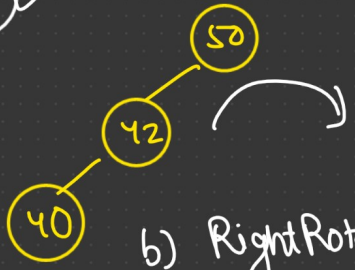
7) Delete 58:-



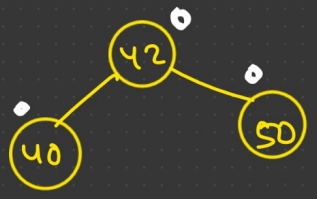
→ LL ✗

→ LR ✓

a) left Rotation (40)



b) RightRotation (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)

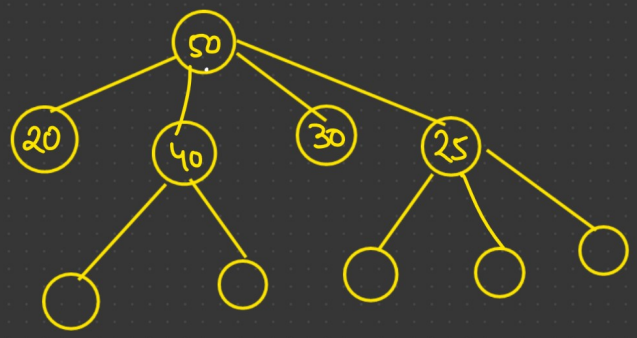
if (getBalanceFactor (Node→right) < 0)

RightRotation (Node→right);

LeftRotation (Node);

General Tree

General tree is a tree in which a node can have either zero child or more then 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 ⇒ Order = 2 ⇒ m = 2

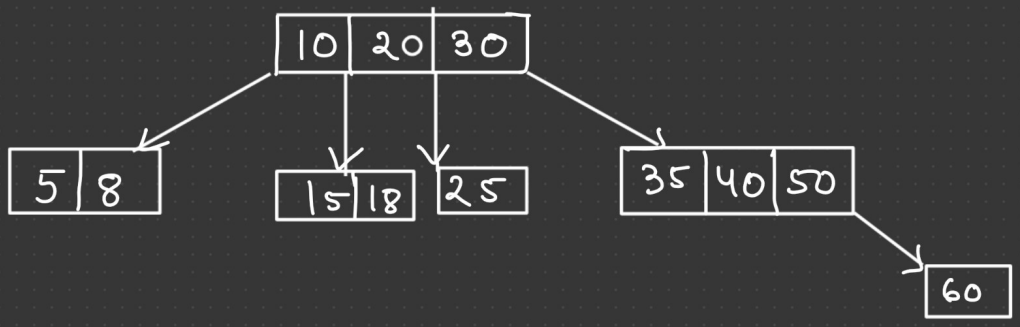
3-way Tree ⇒ Order = 3 ⇒ m = 3

4-way Tree ⇒ order = 4 ⇒ m = 4

m = maximum no. of child

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

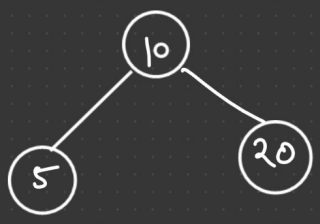
Ex:- 4-Way Tree:- (m = 4) max child = 4, max elements = 3



O(log n) = height of Tree

2-Way Tree / Binary Search Tree :-

(m = 2) ⇒ max child = 2, max element = 1



It is also :-

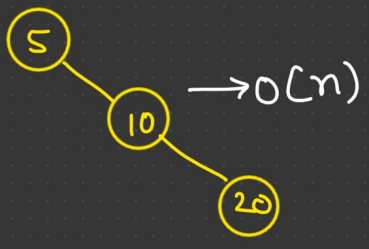
- 3-way tree
- 4-way tree
- 5-way tree

.... 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 its has "minimum of $\lceil m/2 \rceil$ child nodes" & "minimum of $\lceil m/2 \rceil - 1$ elements in a node".

$2.2 \rightarrow$ floor $(2) = \lfloor 2.2 \rfloor = 2$

$2.2 \rightarrow$ Ceil $(3) = \lceil 2.2 \rceil = 3$

- 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.

$\lfloor 3.9 \rfloor = 3$

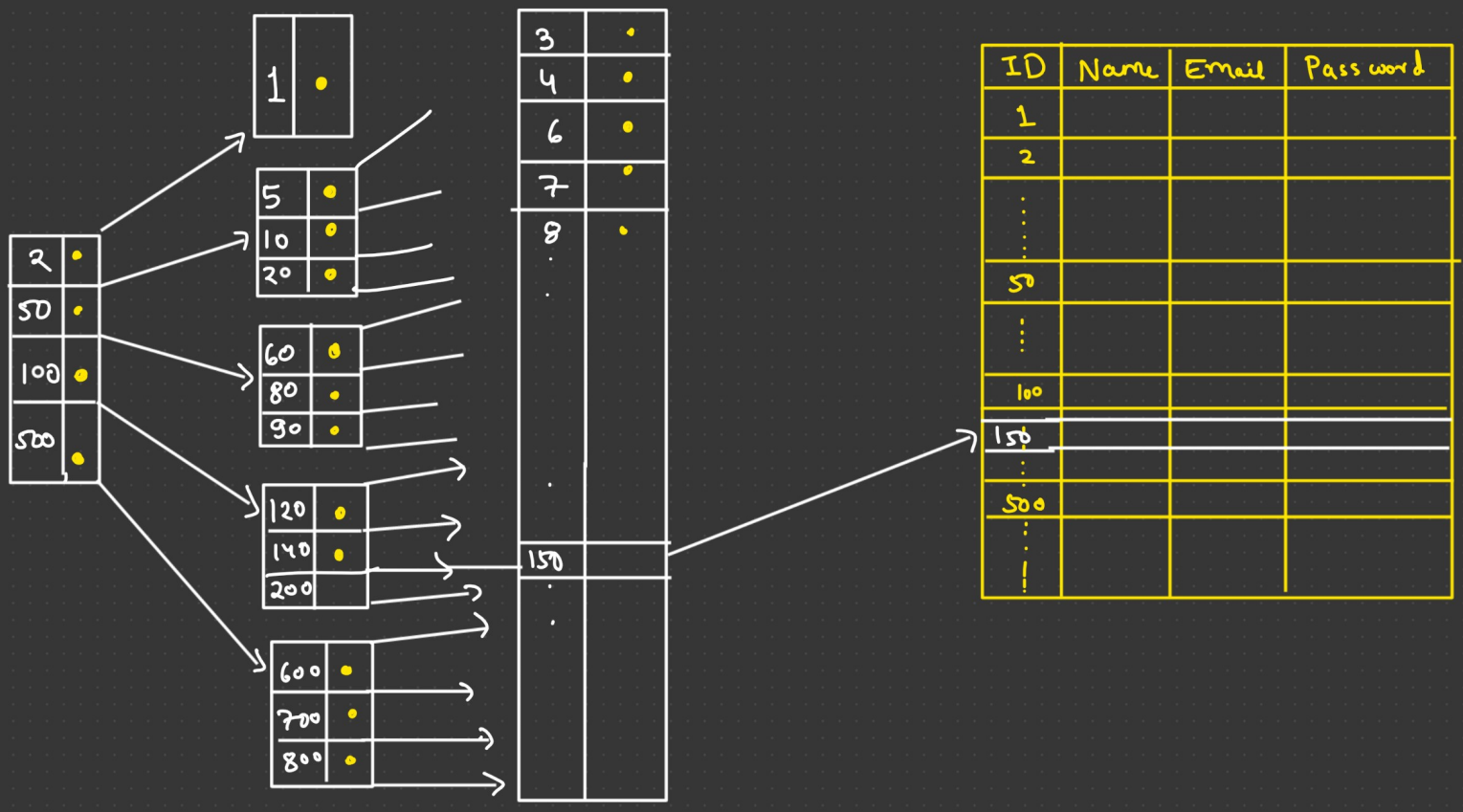
$\lceil 3.9 \rceil = 4$

B-Tree of order 5

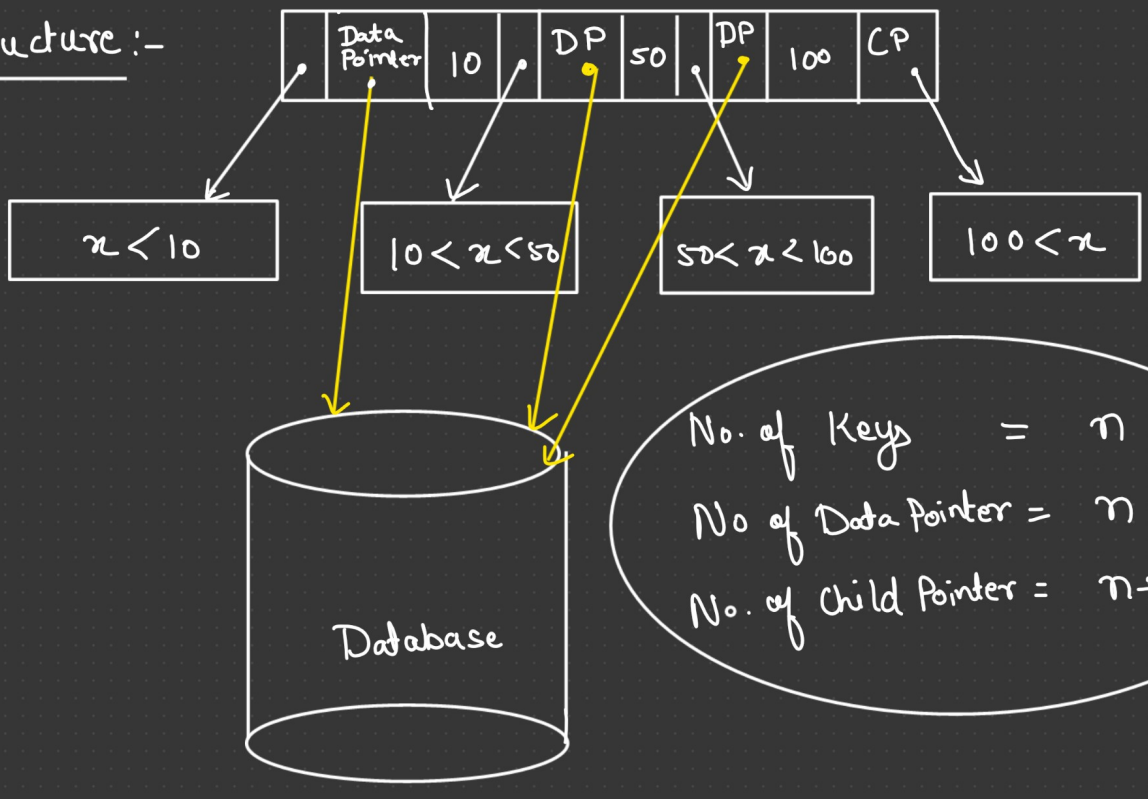
$m = 5$

- Max child = 5
- Min child = $\lceil 5/2 \rceil = 3$
- Max keys = 4
- Min keys = $\lceil 5/2 \rceil - 1 = 2$

Also it is 5-way Tree



Node structure :-



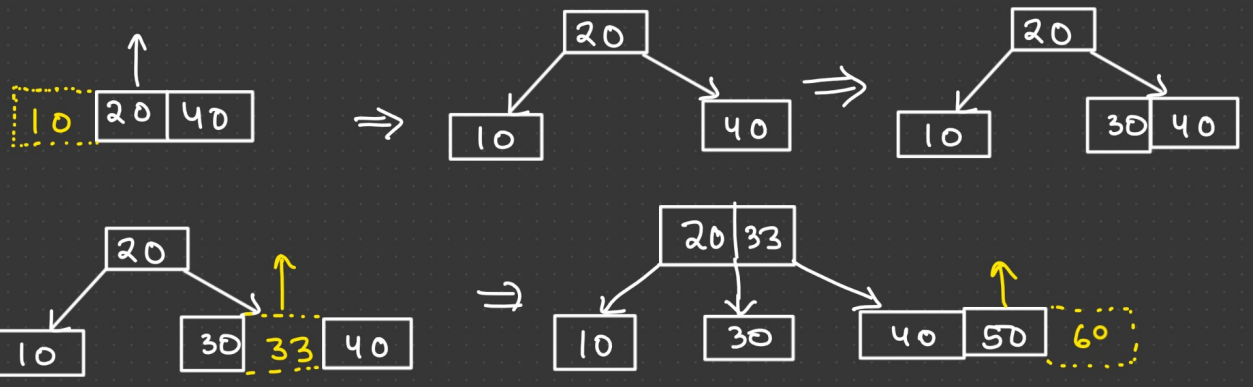
Creation / Insertion in B-Tree :-

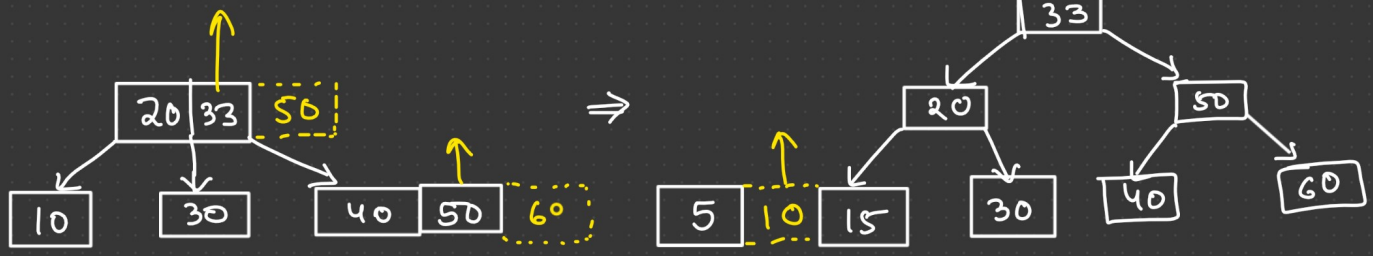
20, 40, 10, 30, 33, 50, 60, 5, 15, 25, 28, 31, 35, 45, 55, 65

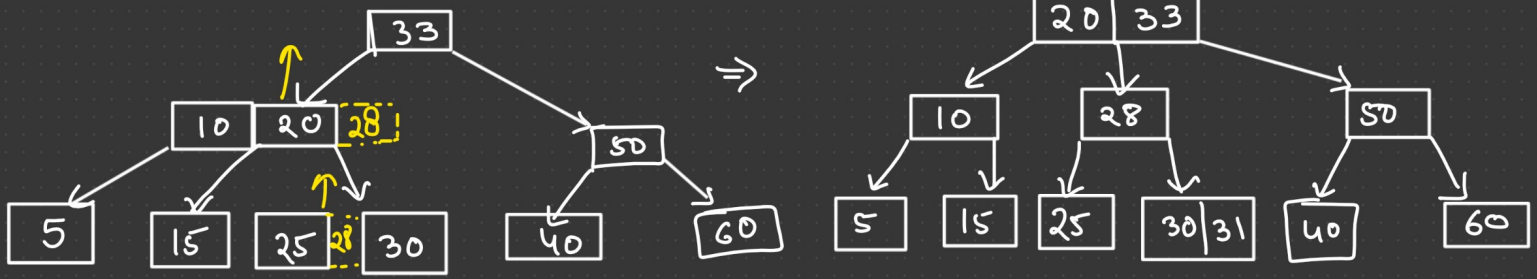
Order :-

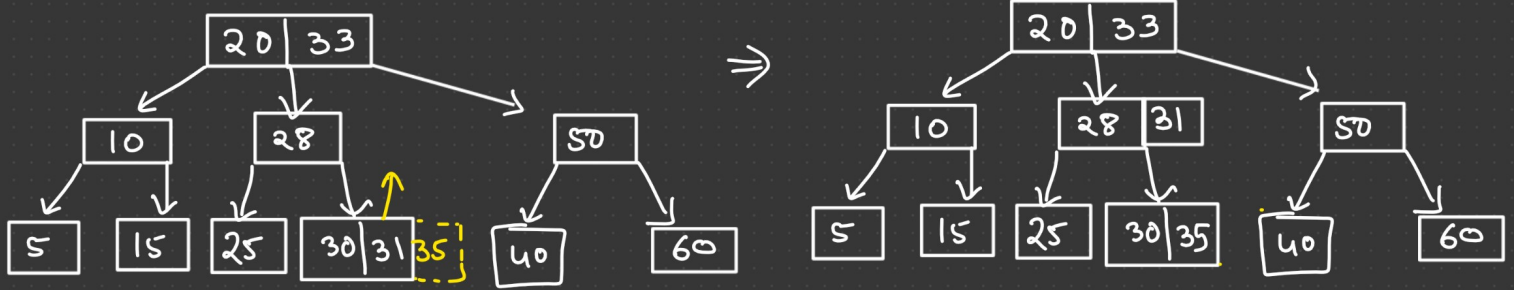
m = 3

- Max Child = 3
- Max Key = 2
- Min Child = 2
- Min Key = 1









To be continue....

Deletion in B-Tree

Rules:-

1) Leaf Node

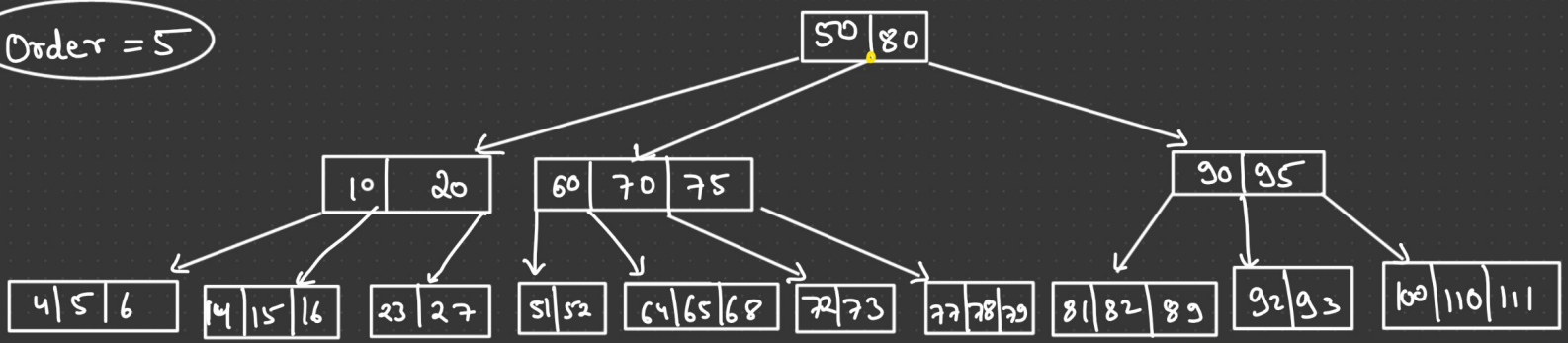
- a) If Node has more than min Key ⇒ Directly delete it.
- b) If Node has min Key ⇒ Borrow from left/Right siblings. If they have more than min key. via parent node.
- c) If sibling has min key then "merge" left sibling + parent + right sibling (parent go down)

2) Internal nodes:-

- a) If Both left & right sibling exists the replace it with Inorder Successor / Inorder predessor 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

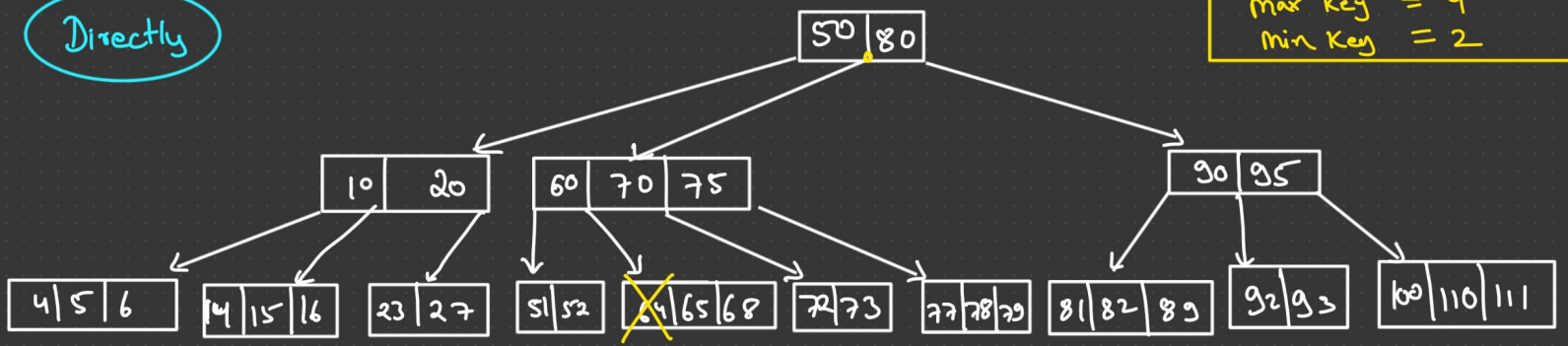
Order = 5



m = 5
max child = 5
min child = $\lceil 5/2 \rceil = 3$
max key = 4
min key = 2

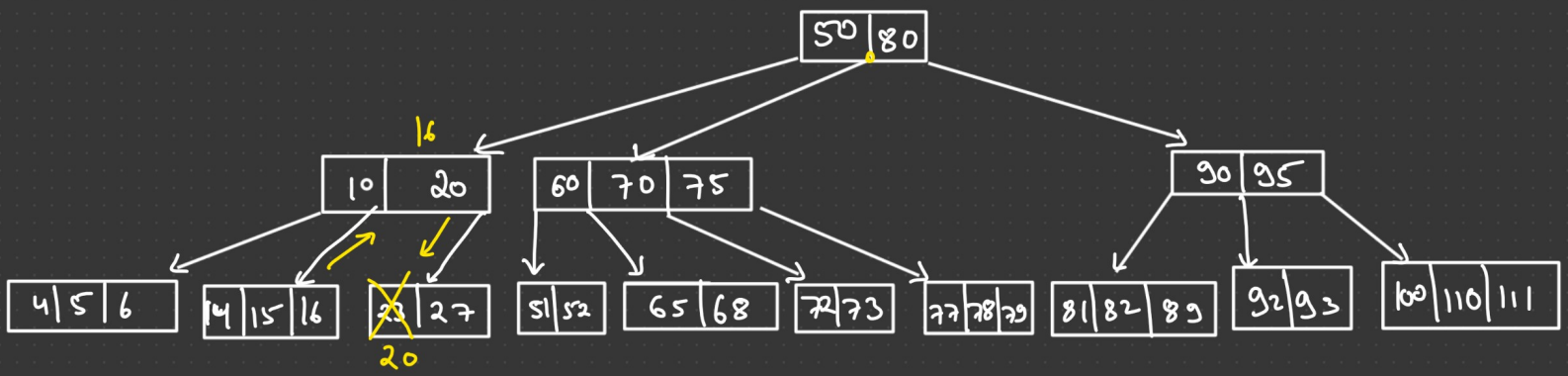
1) Delete 64 :-

Directly

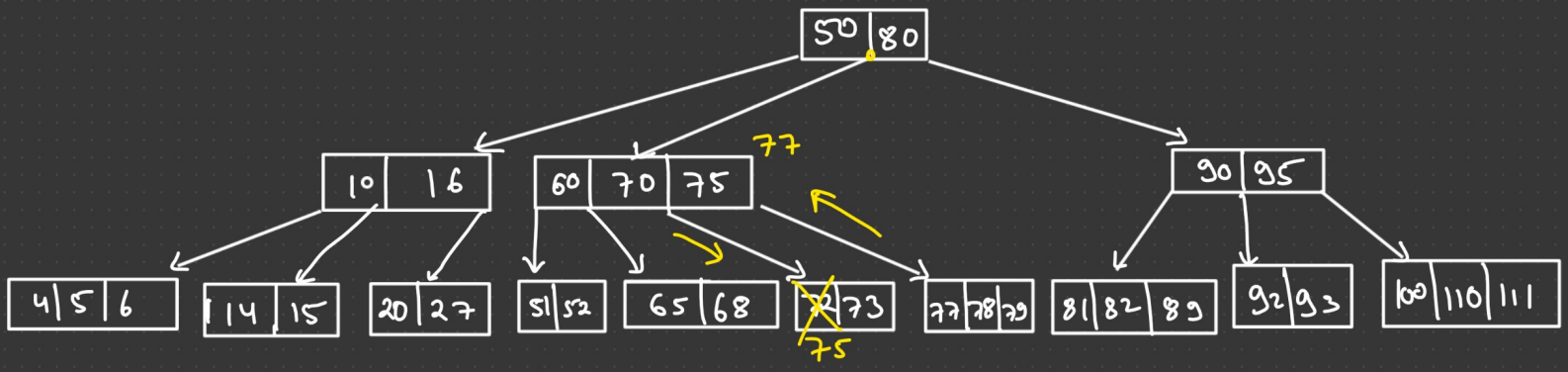


2) Delete 23 :-

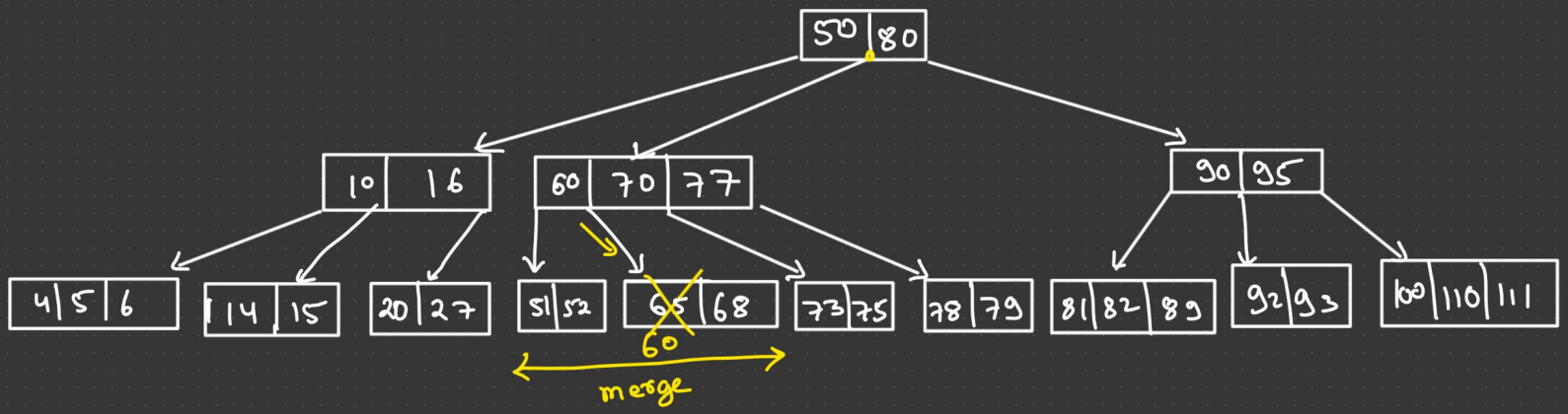
Borrow Inorder predecessor via parent



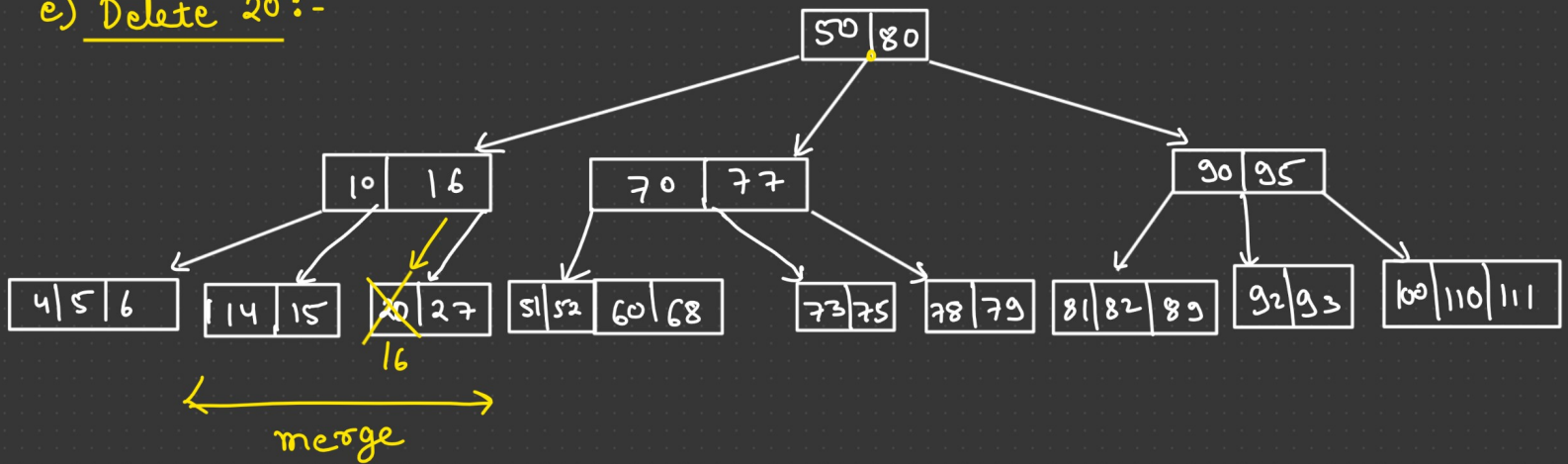
3) Delete 72:- Borrow inorder successor via parent.

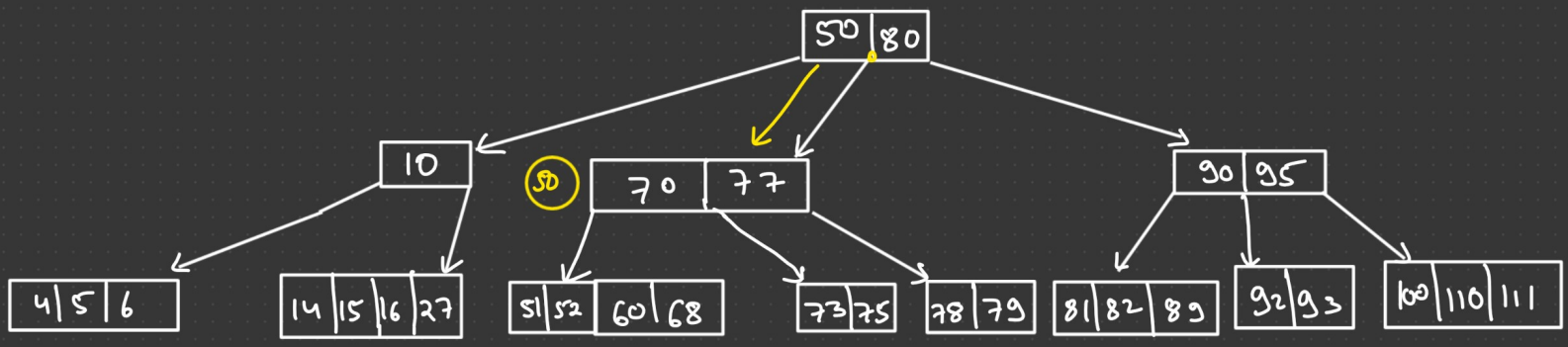


d) Delete 65:-



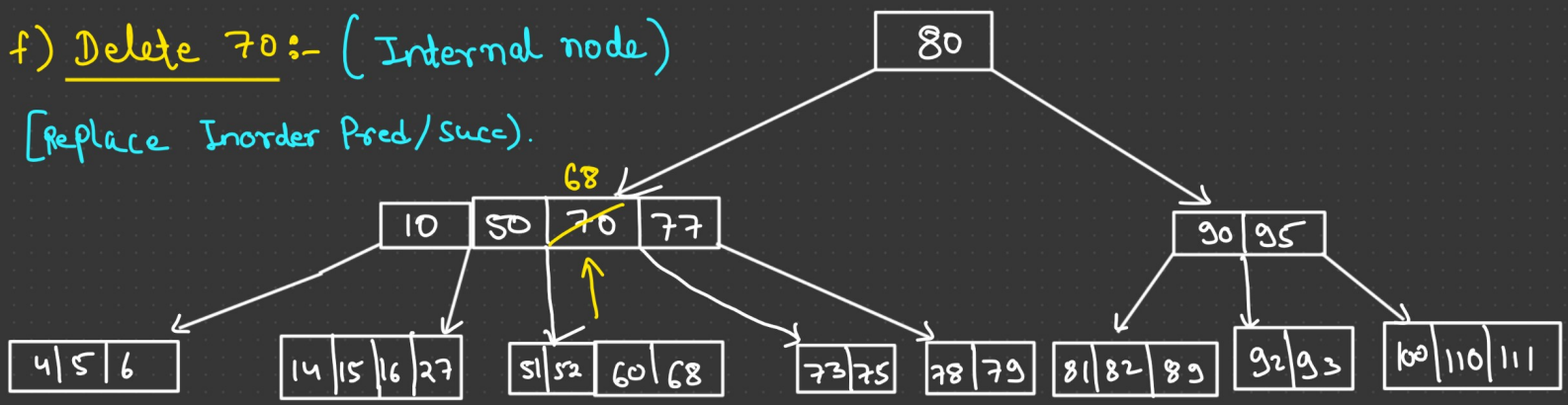
e) Delete 20:-



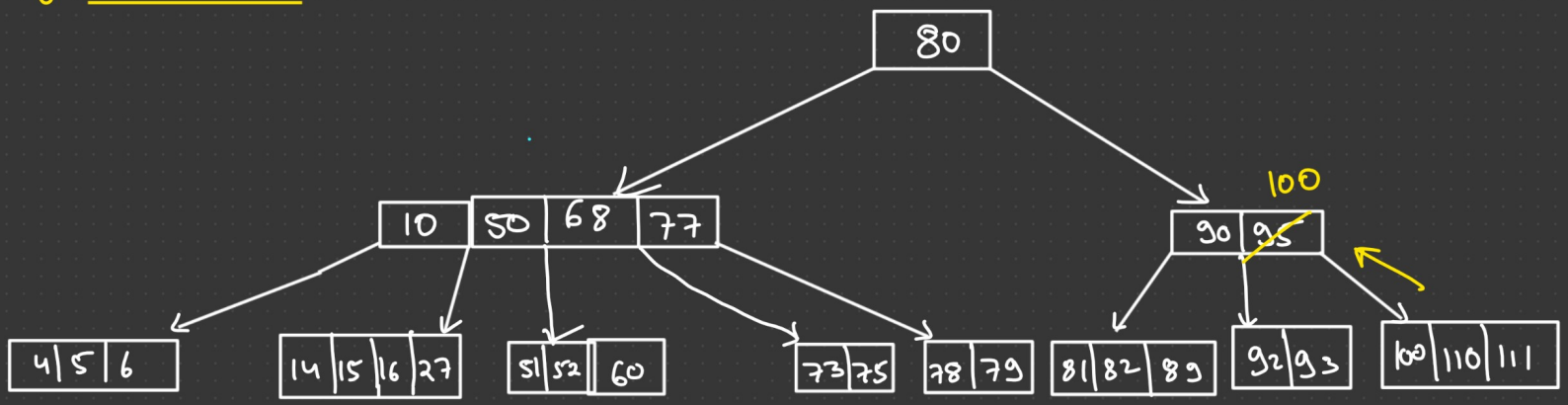


f) Delete 70 :- (Internal node)

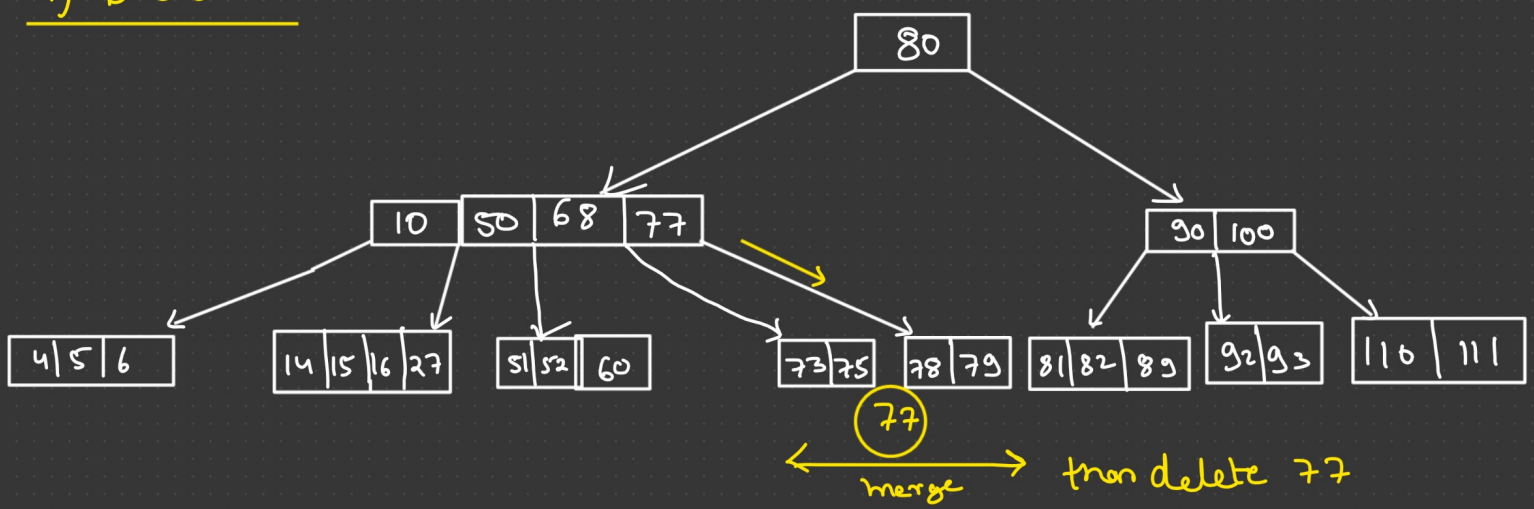
[Replace Inorder Pred/Succ).



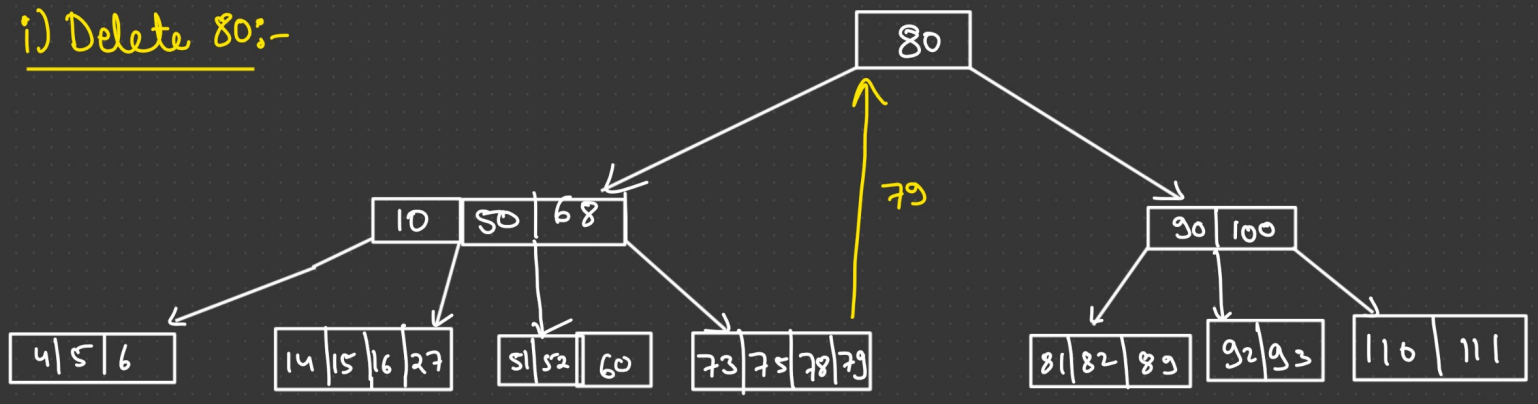
g) Delete 95 :-



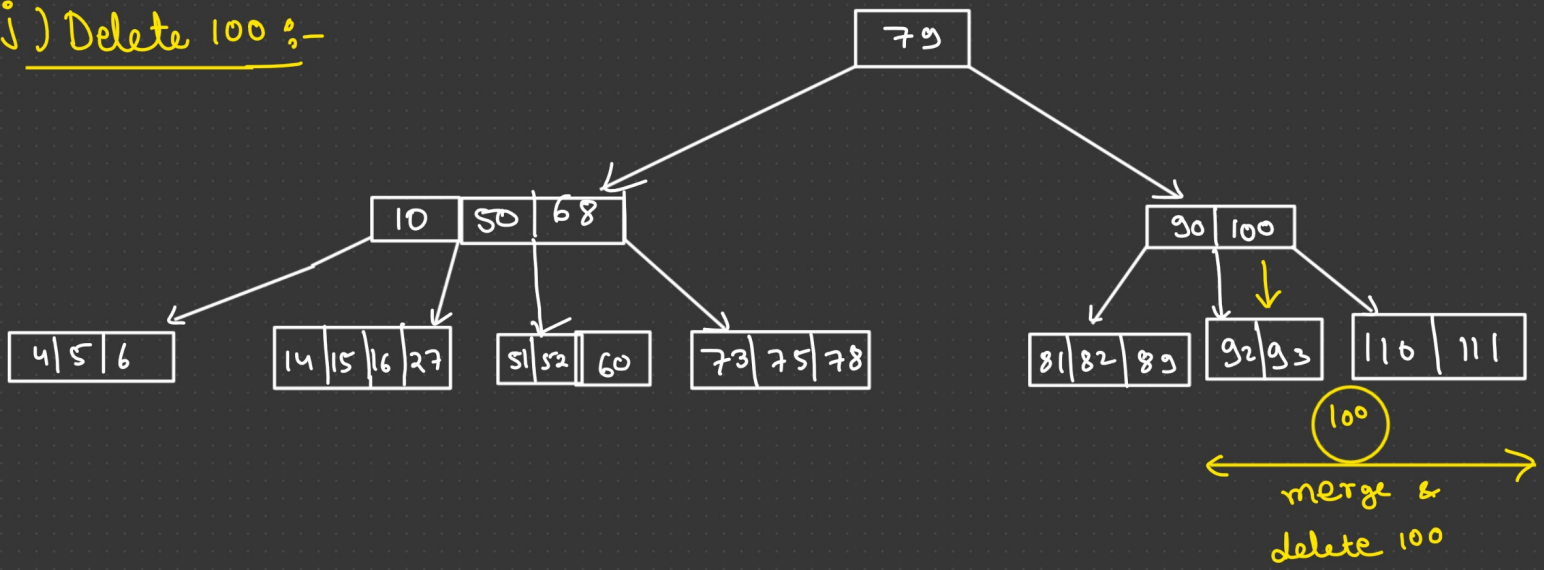
h) Delete 77 :-

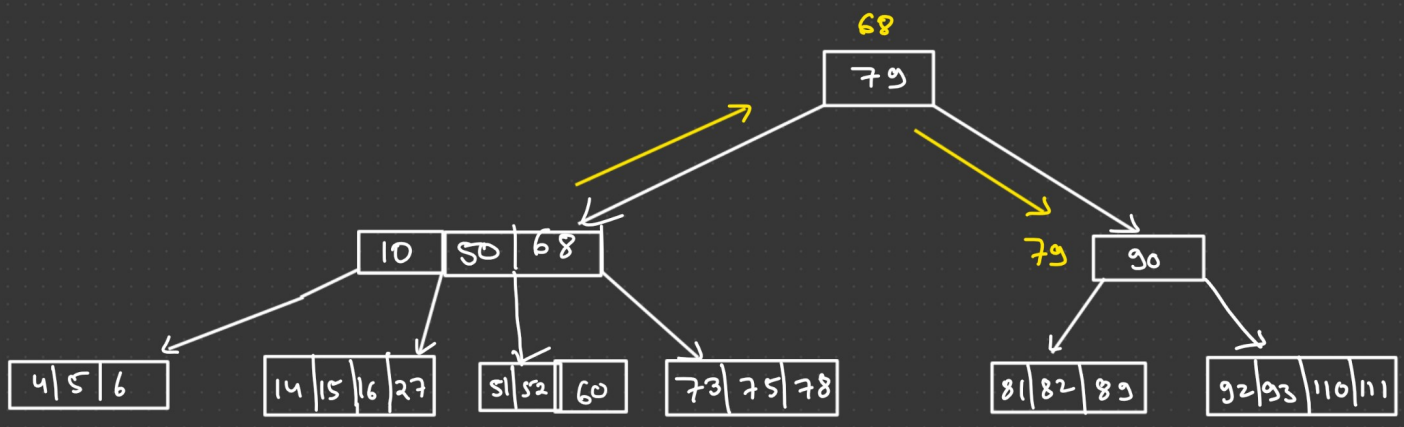


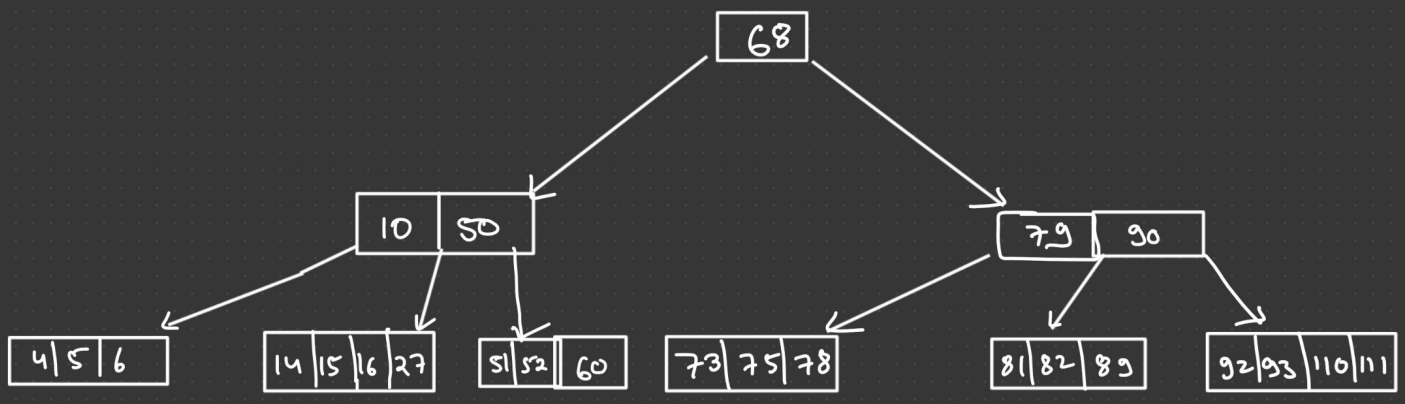
i) Delete 80:-



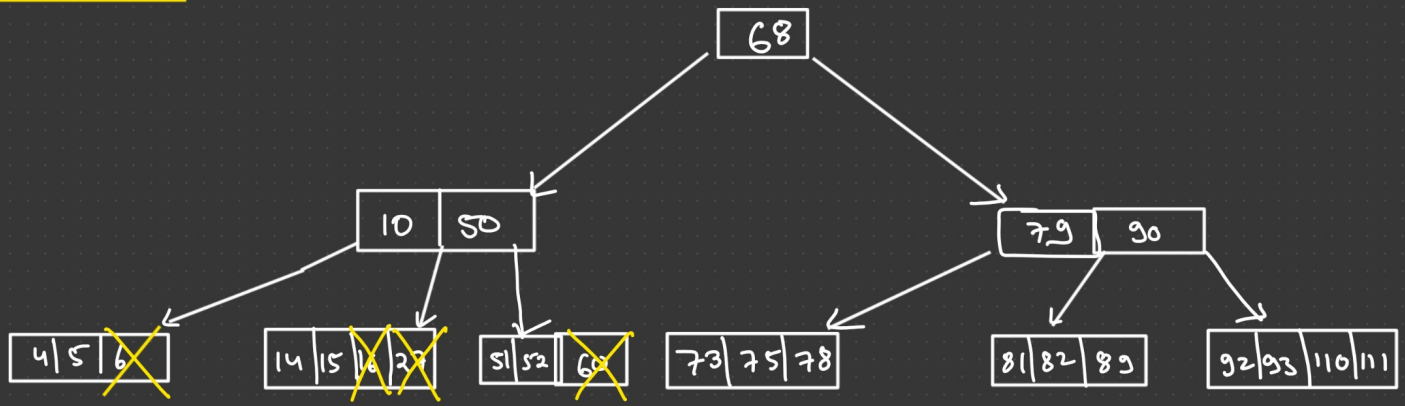
j) Delete 100 :-



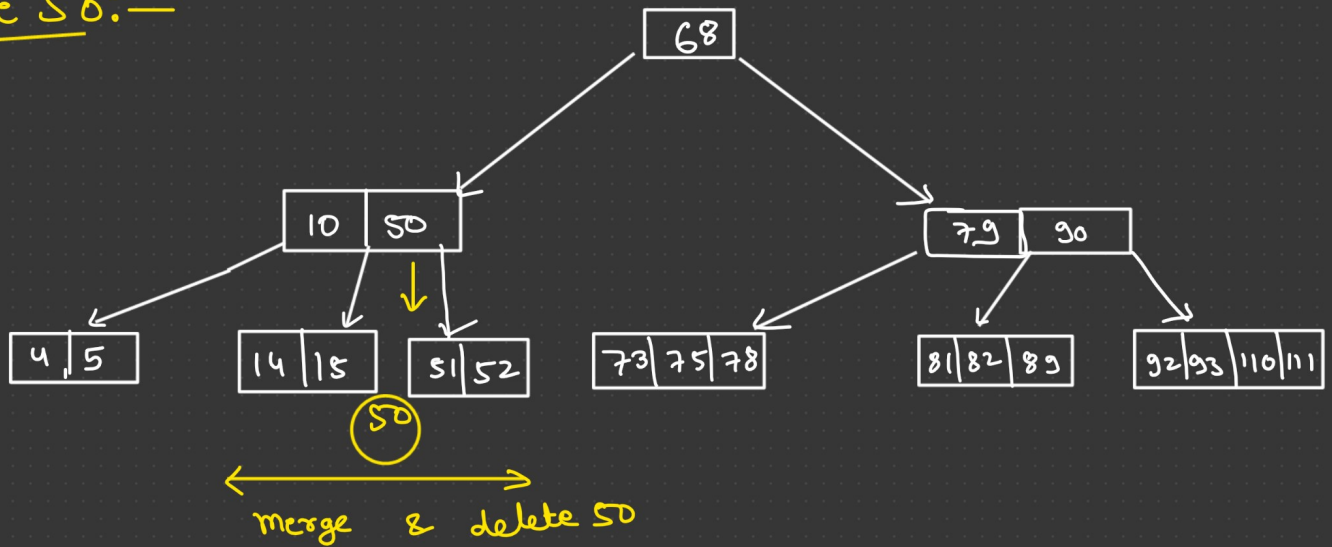


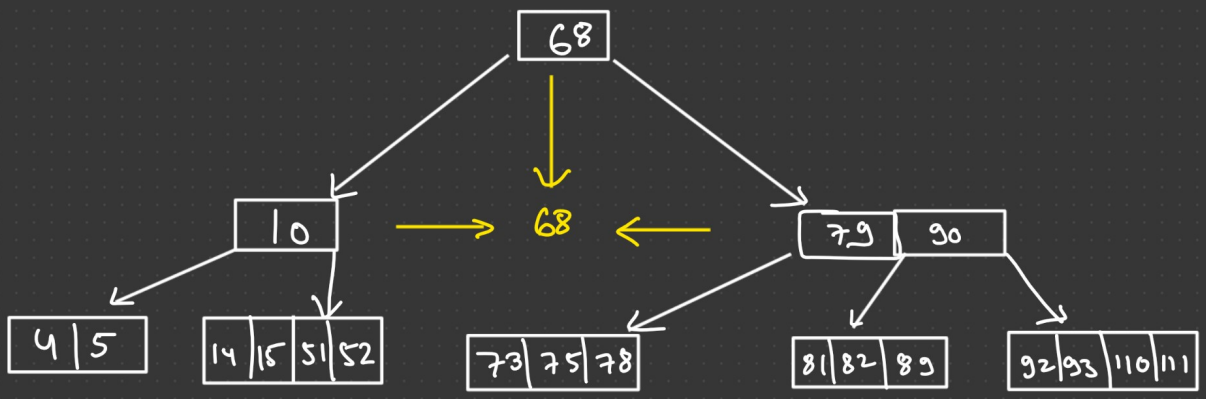


Delete :- [6, 27, 60, 16, 50]

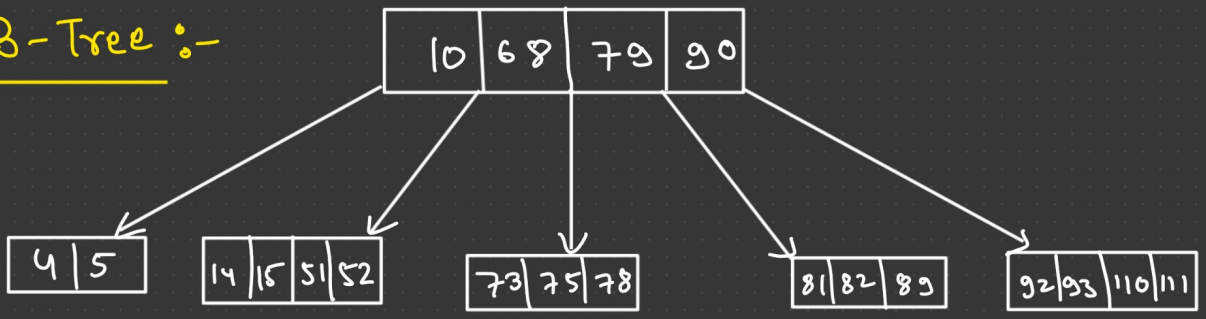


Delete 50:-



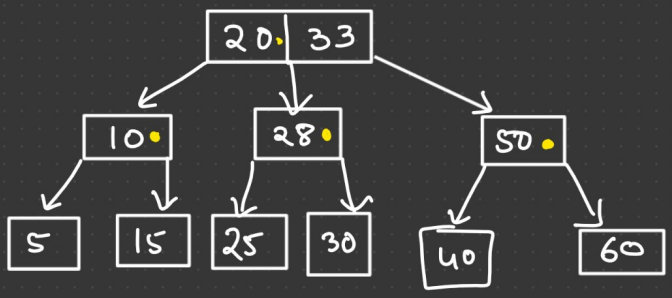


Final B-Tree :-

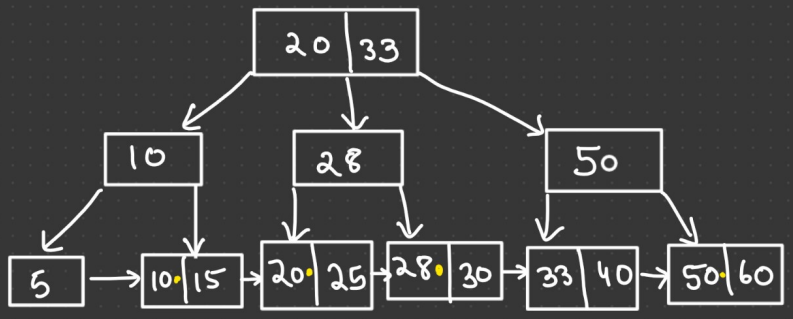


m = 3

B+ Tree



B-Tree



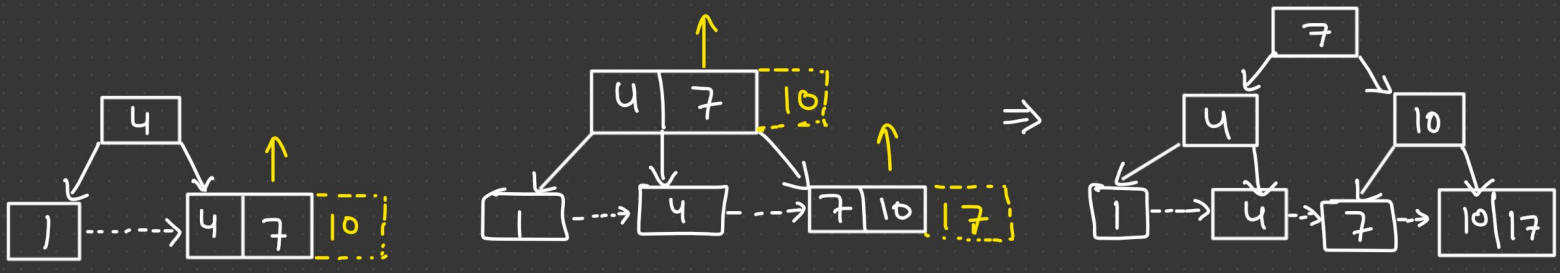
B+ Tree

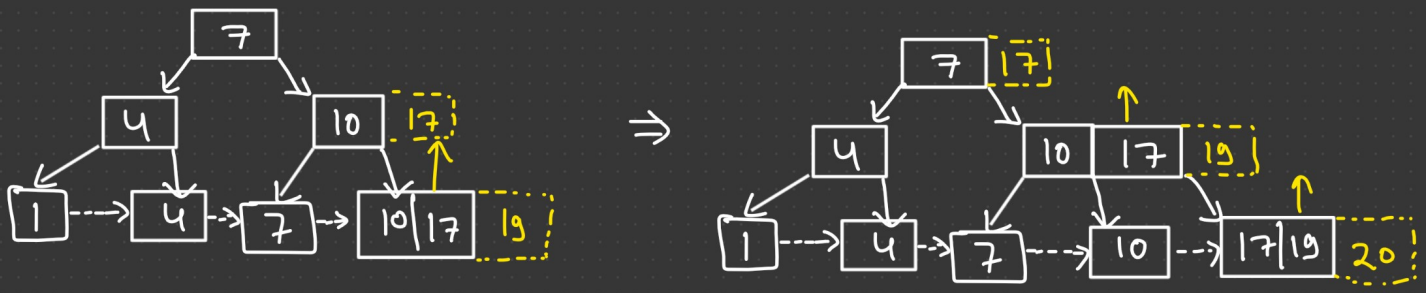
Creation / Insertion in B+ Tree

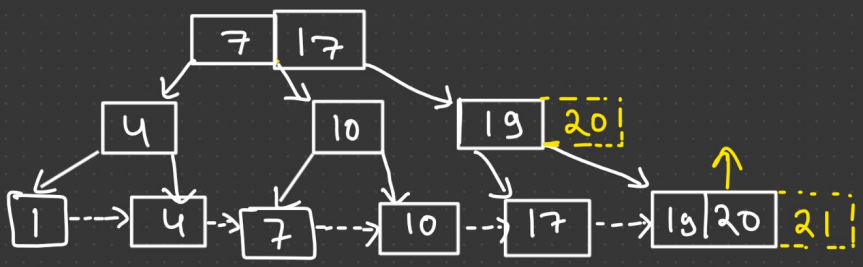
1, 4, 7, 10, 17, 19, 20, 21, 25, 31, 42

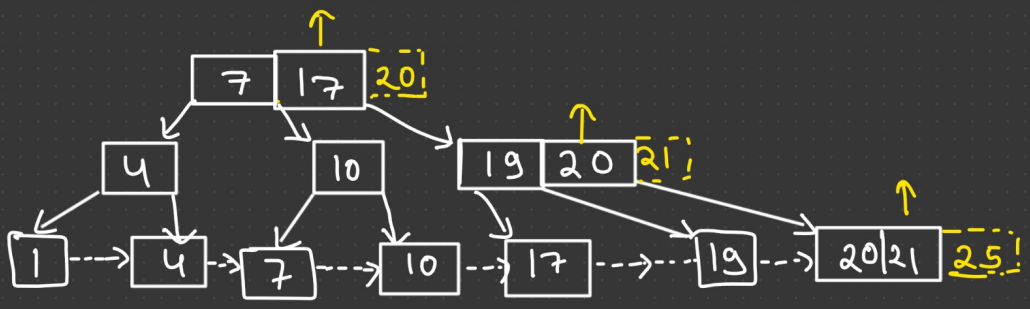
m = 3 = order
max child = 3 min child = 2
max key = 2 min key = 1

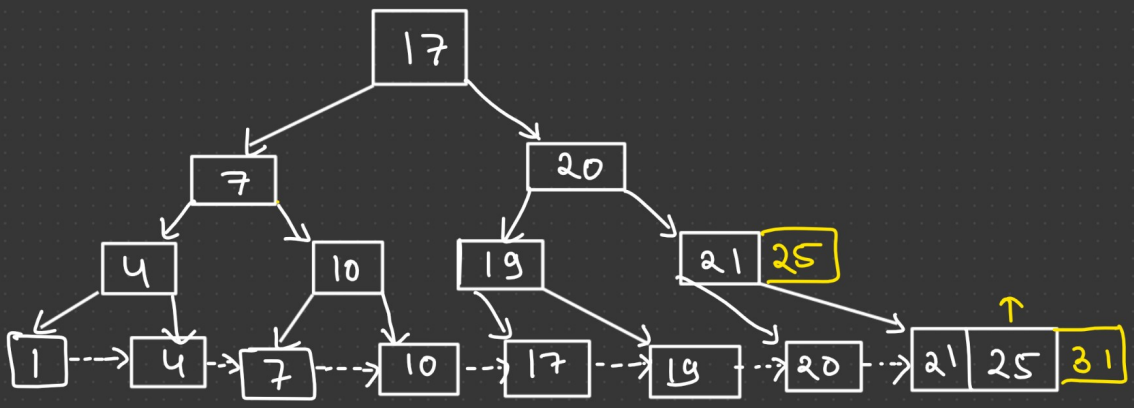
[Note – If leaf node breaks then, a copy of parent node is also present at leaf node. But if an internal node breaks its parent copy does not need to be copied.]

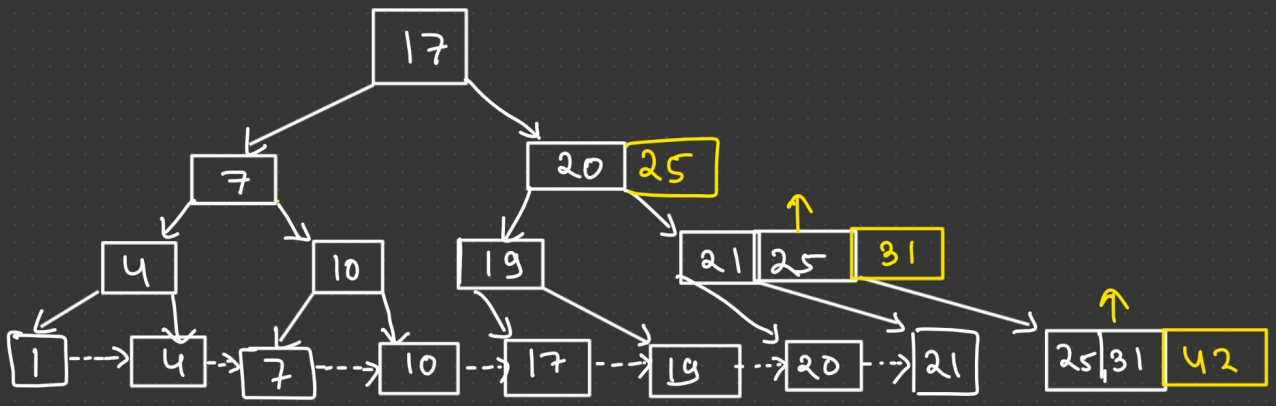


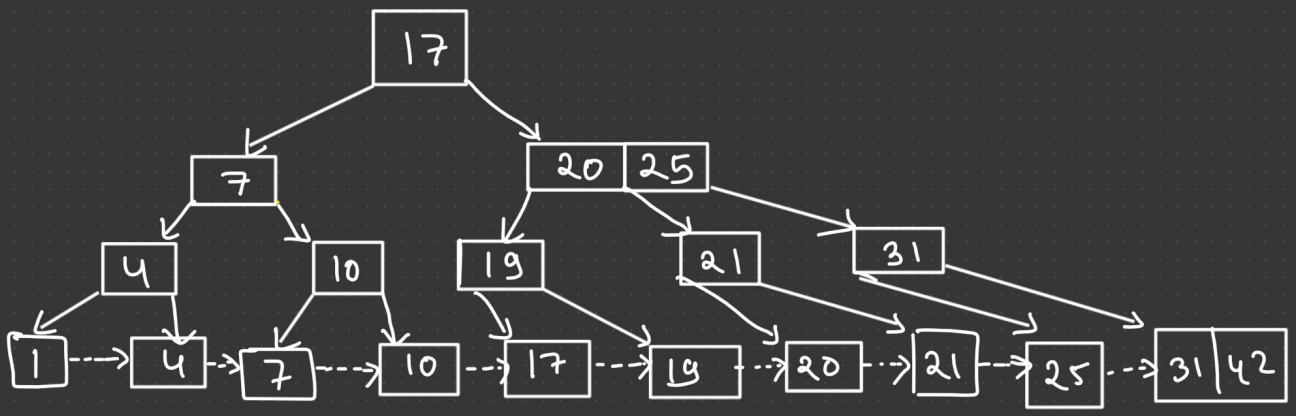

7
4
10
17
1
4
7
10
17
19
⇒
7
17
4
10
17
19
1
4
7
10
17
19
20


7
17
4
10
19
20
1
4
7
10
17
19
20
21


7
17
20
4
10
19
20
21
1
4
7
10
17
19
20
21
25


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


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



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.

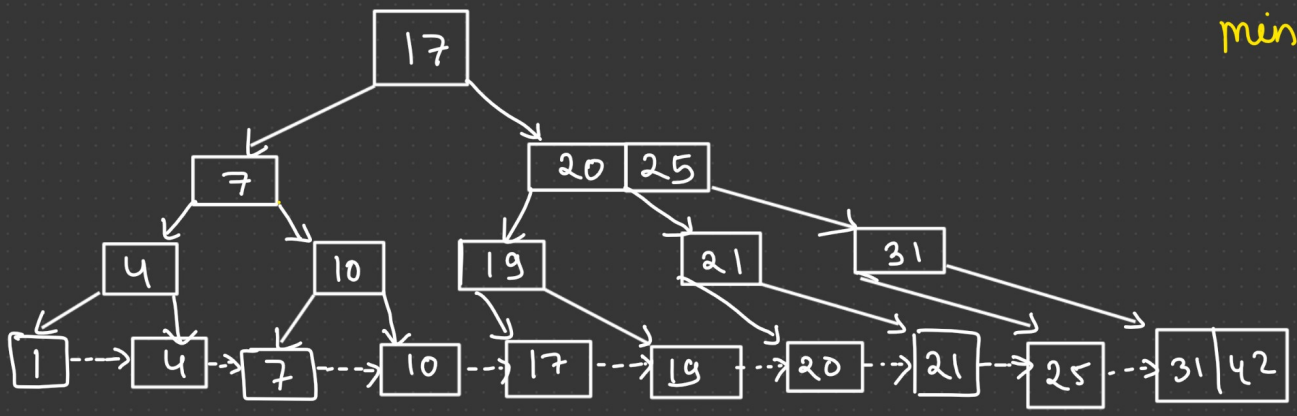
2) If node has min number of elements then
   - a) It will borrow element from its siblings which have more than min elements via parent.
   - b) Else parent come down & merge with its child.

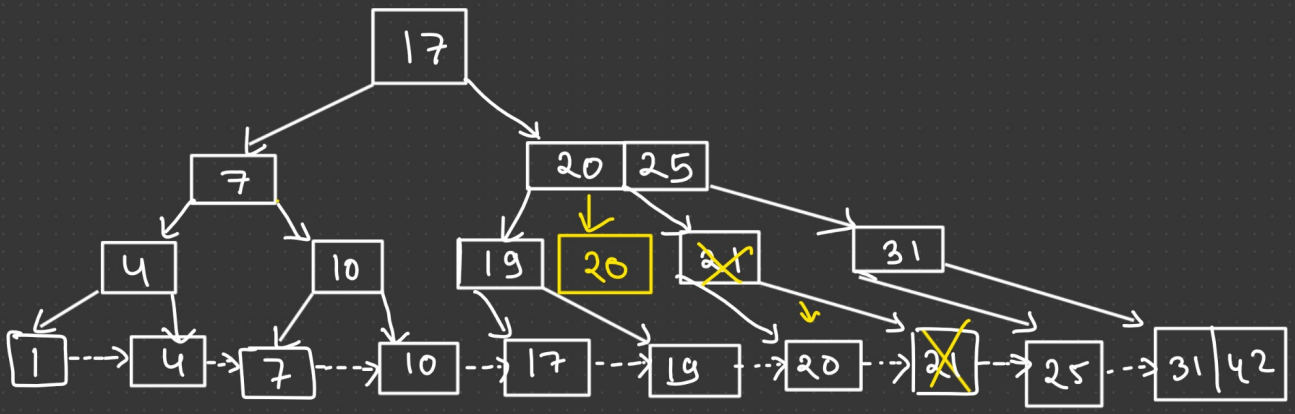
Delete :- 21, 31, 20, 10, 7, 25, 42, 4

m = 3

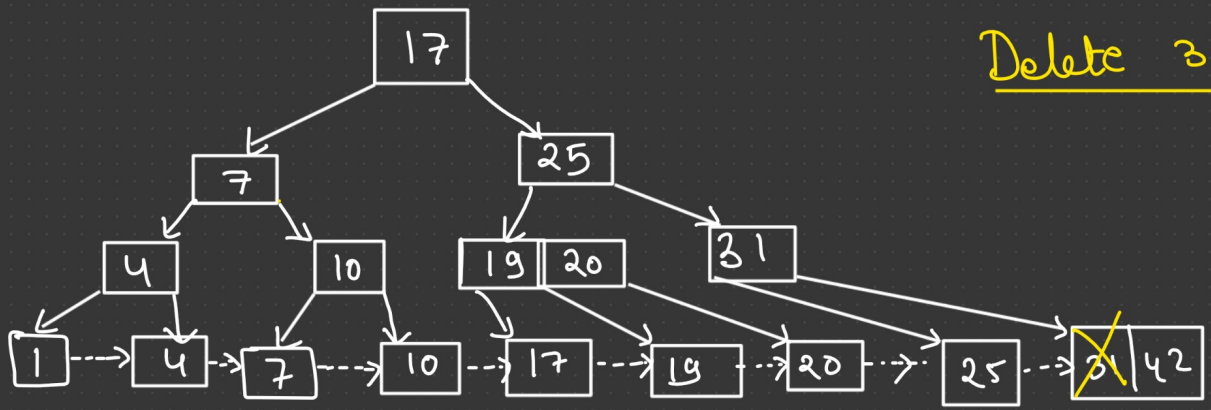
max Keys = 2

min Keys = 1

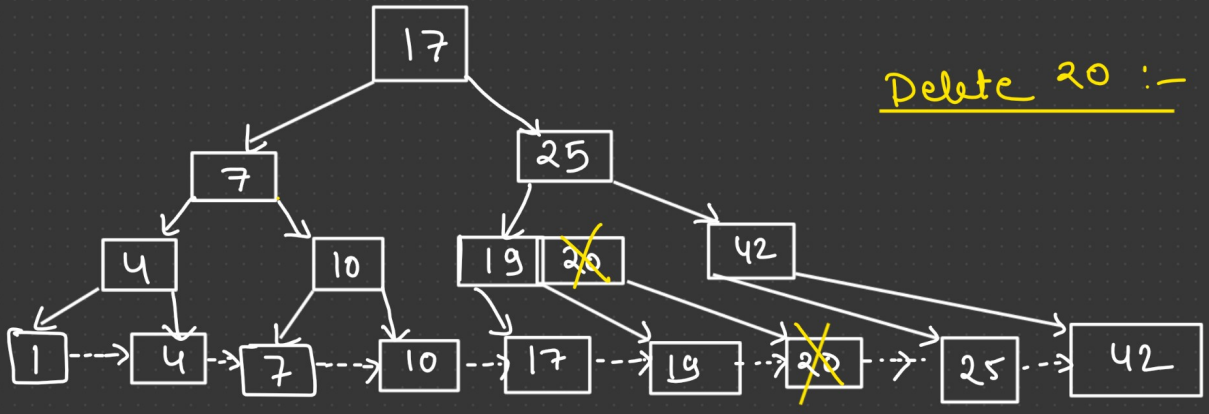


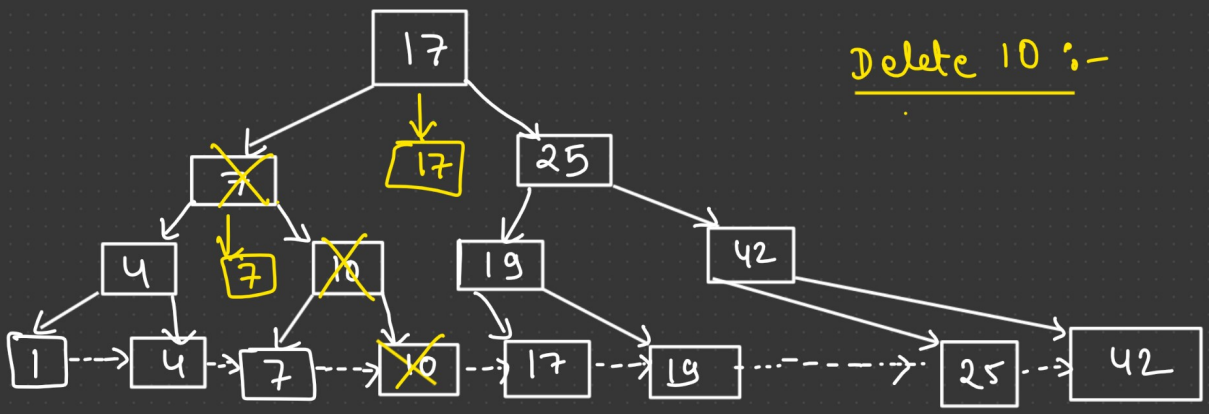


Delete 31

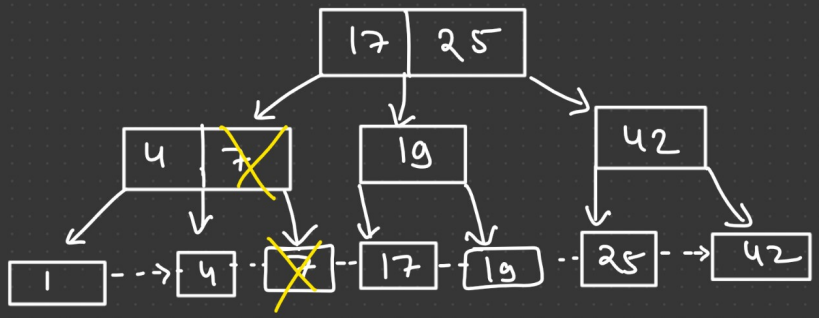


Delete 20 :-

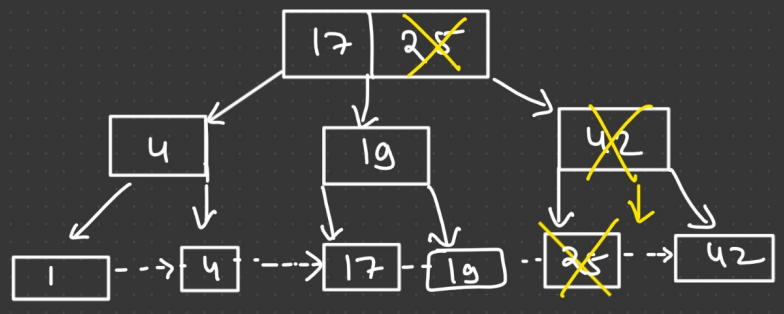




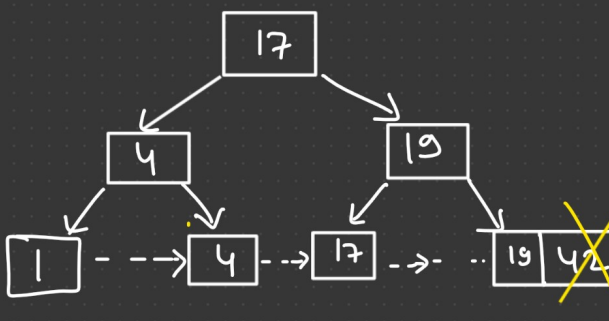
Delete 10 :-



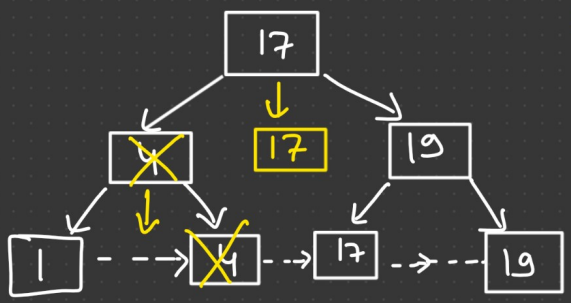
Delete 7 :-



Delete 25 :-

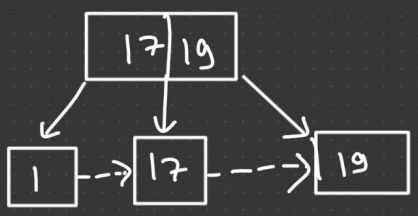


Delete 42 :-



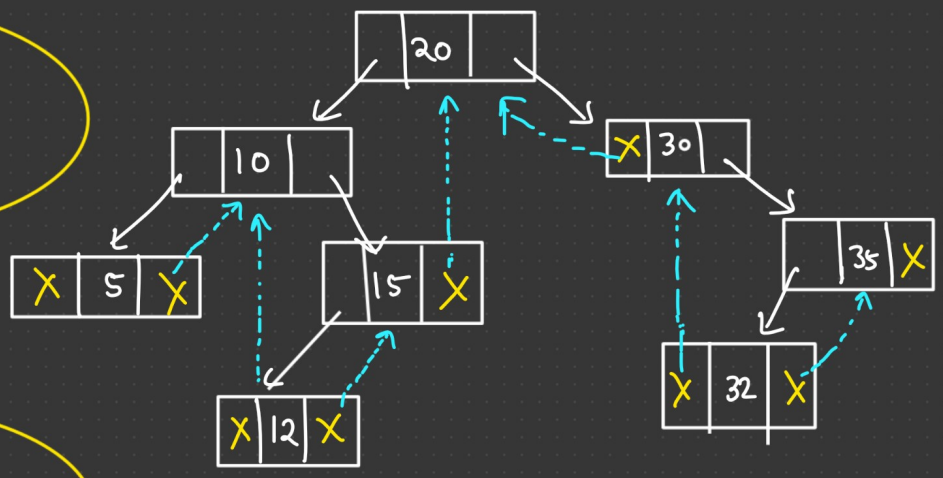
Delete 4 :-

⇒



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

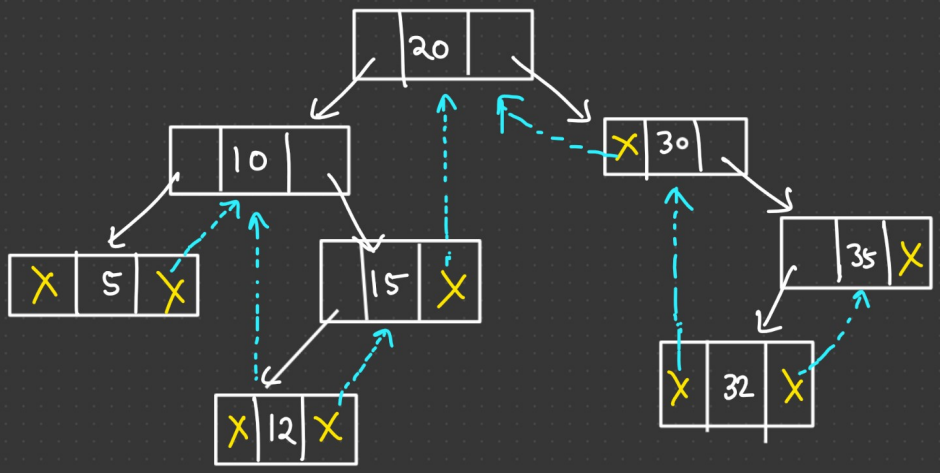
For Inorder Traversal :- we need stack.

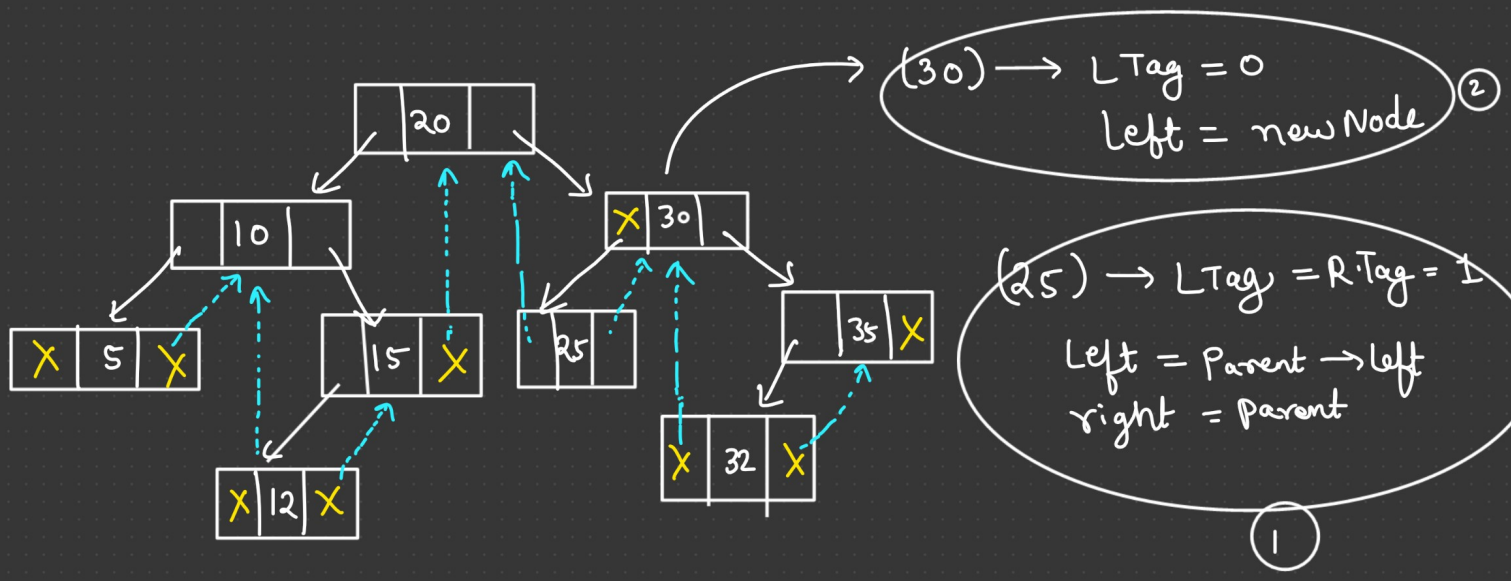
Left Ptr	L Tag	Data	R Tag	Right Ptr

L Tag = 0 ⇒ Ptr → Tree
= 1 ⇒ Ptr → pred.

R Tag = 0 ⇒ ptr → Tree
= 1 ⇒ ptr → succ.

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


20
10
30
X 5 X
15 X
35 X
X 12 X
X 32 X


20
10
X 30
X 5 X
15 X
25
35 X
X 12 X
X 32 X
(30) → LTag = 0
Left = newNode
2
(25) → LTag = RTag = 1
Left = Parent → left
right = Parent
1