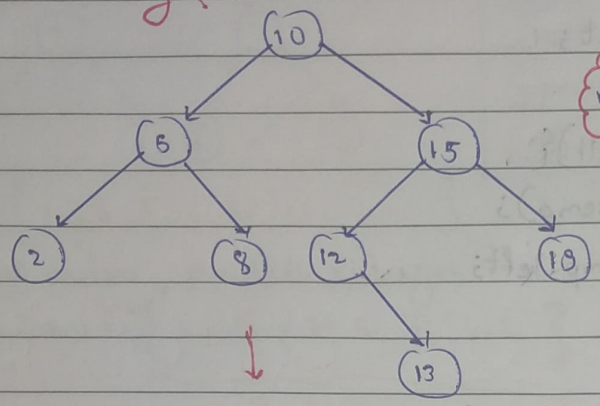
Binary Search Trees.

→ What & Why ?

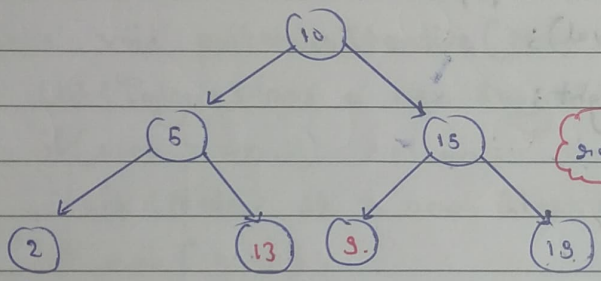


min(RST) > node.val > max(LST)

→ for Every 'node' in the tree.

This is A VALID BST.

Common Mistake



root.left.val < root.val < root.right.val

→ Wrong Definition.

This is NOT a BST.

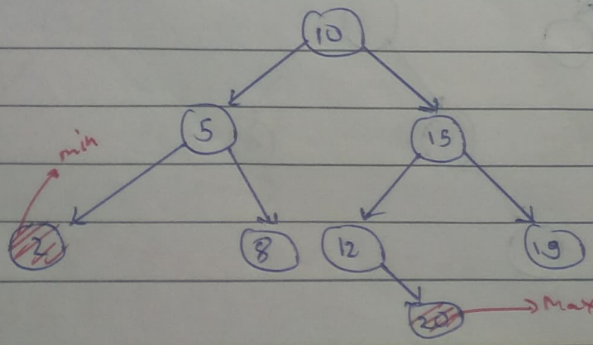
Can a BST Contain Duplicate Elements?

min(RST) > root.val > max(LST).

→ ~~In~~ General, a BST Contains Unique Elements.

→ But if it given that BST Can Contain duplicate, then properly see the definition of BST in question.

Find the min & max ele in a given BST → Without Traversing the entire tree.



→ Pass Root. Left.
Not root.

⇒ Code :- ps

```
int min(Node root){
    if(root == null)
        return Integer.MAX_VALUE;
    int a = root.val;
    int minLst = min(root.left);
    int minRst = min(root.right);
    return Math.min(a, Math.min(minLst, minRst));
}
```

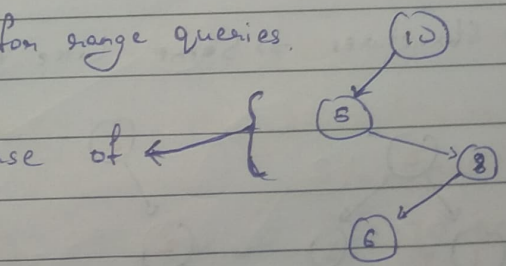
→ Pass root.right
Not root.

⇒ Code :- ps

```
int max(Node root){
    if(root == null)
        return Integer.MIN_VALUE;
    int a = root.val;
    int maxLst = max(root.left);
    int maxRst = max(root.right);
    return Math.max(a, Math.max(maxLst, maxRst));
}
```

#Advantages
- → Efficient Searching
- → Efficient insertion & deletion.
- → Usage in implementation of Other data structure like. Sets, maps, priority Queue, etc.

#Disadvantages
- → Lack of support for range queries.
- → Not that efficient in case of Unbalanced Trees.



#Application
- → Phonebook
- → Dictionary
- → Stock Market Analysis.

Binary Search Trees
O(log n),

Concept Builder.

M T W T F S S

Page No.:

Date:

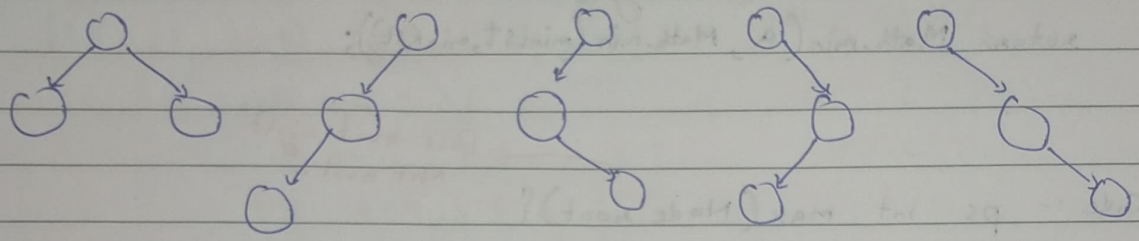
YOUVA

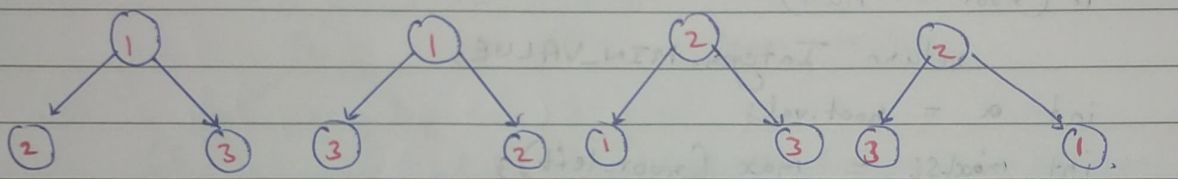
=> You have 3 Nodes with different values. How many Unique Binary Trees can be formed?

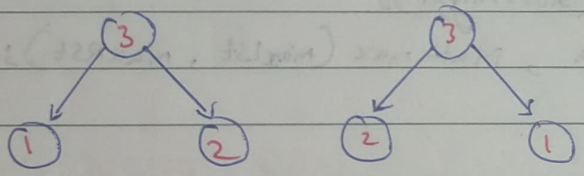
-> 1, 2, 3

Structures (5) × 3! = 30 => Unique BT can formed.

=> Structures :- (5).



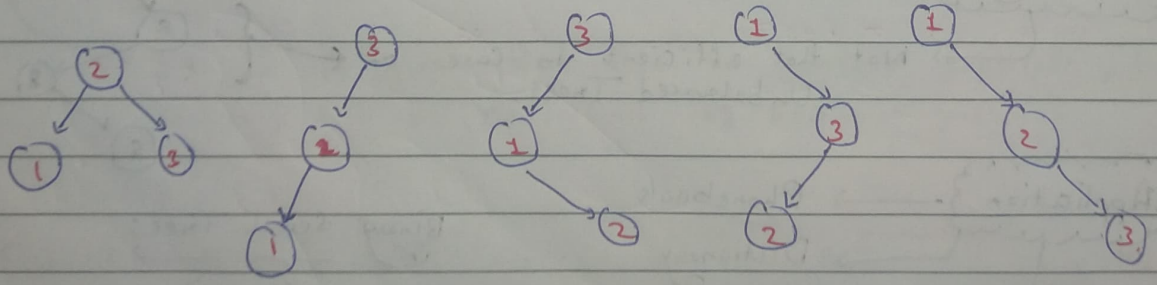




For same # Structure we have 6 diff. Binary Trees. => 3!

=> You have 3 Nodes with different values. How many unique Binary Search Trees can be formed?

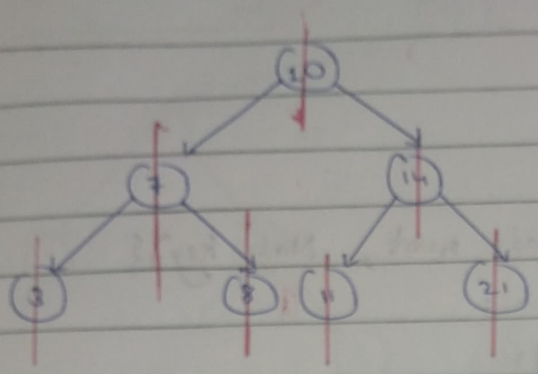
=> Structures => Same like BT.



No. of unique BST = No. of Structures.

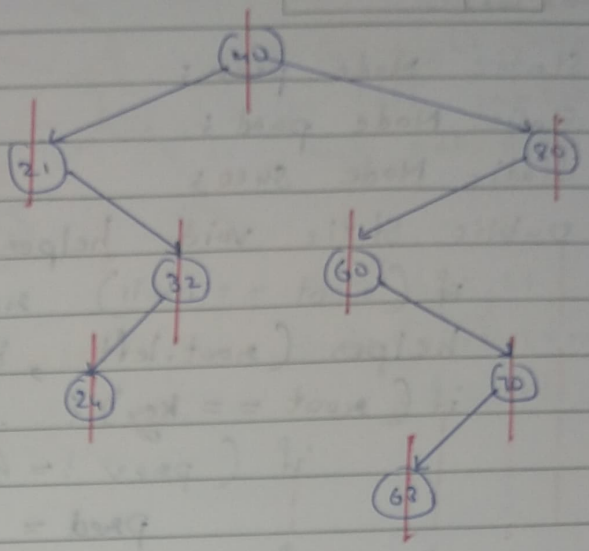
→ Left Root Right

Inorder Traversal in BST (Operation)



In :- 3, 7, 8, 10, 11, 14, 21

Sorted Hai.



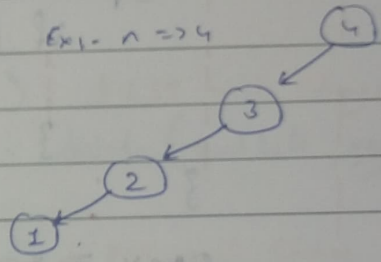
In :- 21, 24, 32, 40, 60, 63, 70, 80

MCQ - 1 :- Consider a binary search tree with n nodes, what is the maximum possible height of the tree?

↳ or level

Ans → n //

Ex:- n => 4



for minimum possible height its Balanced Binary Search Tree.

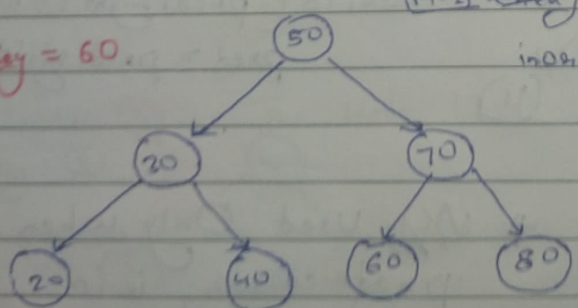
So. → log n //

MCQ → 2 :- Consider a BST with n nodes, what is the minimum no. of comparisons required to search for a value in the worst-case scenario?

Ans → O(n) //.

Inorder Predecessor And Successor for a given key in BTS

M-1 Using Array List



Key = 60.

inorder = {20, 30, 40, 50, 60, 70, 80}

Predecessor

Successor

M-2 Without using Array List.

p → null 20, 30, 40, 50, 60, 70, 80

r

if (r == key) Pred = prev.

if (prev == key) Succ = root.

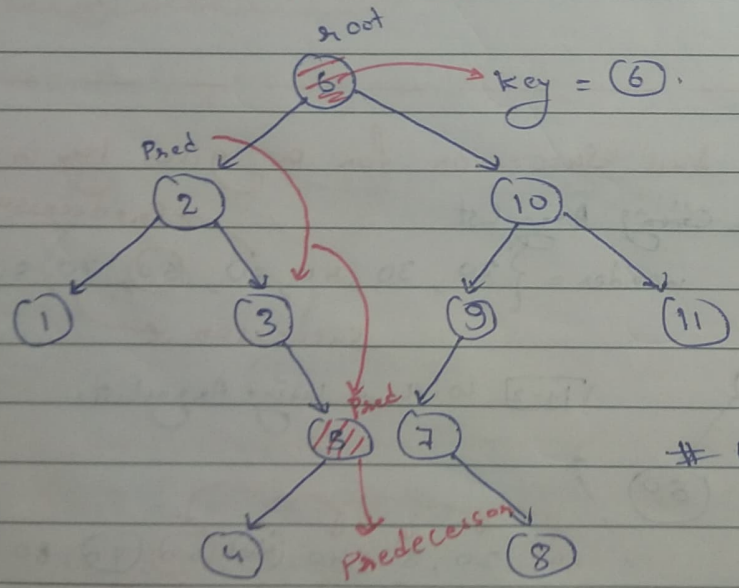
Inorder Predecessor is 50

Inorder Succesor is 70.

=> Code:-

```
Static Node prev;
Static Node pred;
Static Node succ;
public Static void helper(Node root, Node key){
    if (root == null) return;
    helper (root.left , key);
    if ( root == key) {
        if ( prev != null)
            pred = prev;
        }
    }
    if ( prev == key){
        if (root != null){
            succ = root;
        }
    }
    prev = root;
    helper (root.right , key);
}
```

In Order Predecessor :

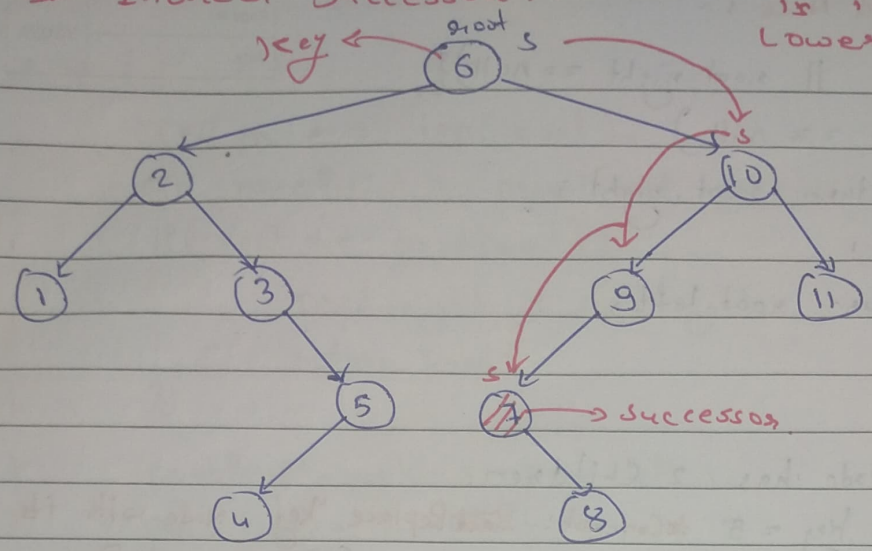


Go Left once :
Go right jab tak ~~null~~ null Nahi Aata.

pred = ~~root~~ root.left.
while (pred.right != null)
    pred = pred.right;

~~Only~~ Used Only when predecessor is in Lower ~~level~~ level of key.

Inorder Successor :-



use only when successor is in Lower Level of key.

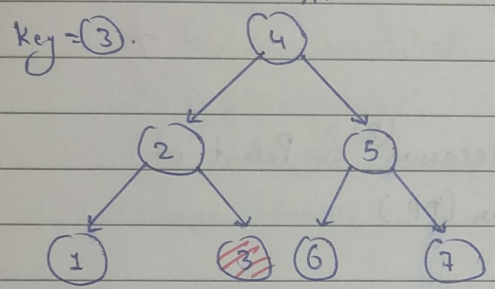
Go right Once
Go left jab tak null nahi Aata.

```
s = r.right;
while(s.left != null)
    s = s.left;
```

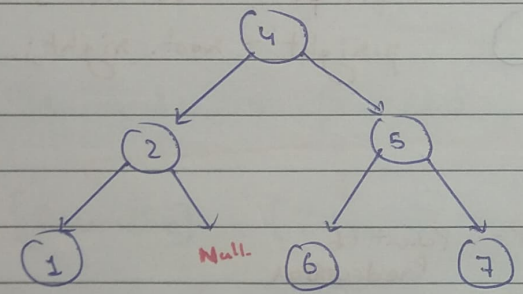
Delete Node in BST.

delete(root, key) → Integer.

⇒ Case (1) :- The Node has 0 Child. / Leaf Node.

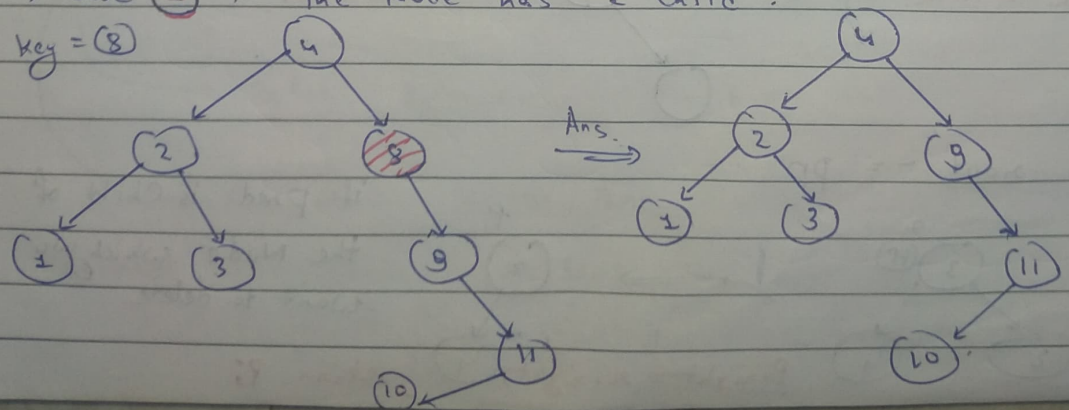


Ans




```
if(root.val > key)   // LST will Change
    root.left = delete(root.left, key);
if(root.val < key)   // RST will Change
    root.right = delete(root.right, key);
if(root.val == key)
    return null;
// Case (1) : 0 Child Node :-
if(root.left == null && root.right == null)
    return null;
```

⇒ Case (2) :- The Node has 1 Child.

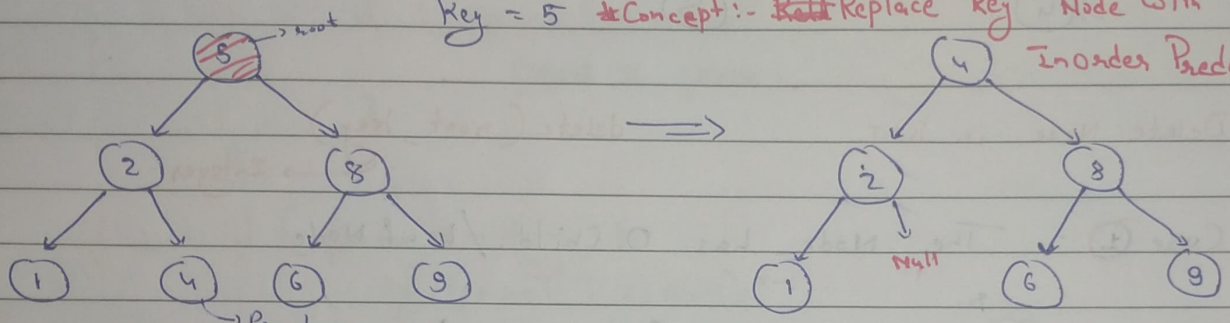


// Case (2) : 1 Child Node :-

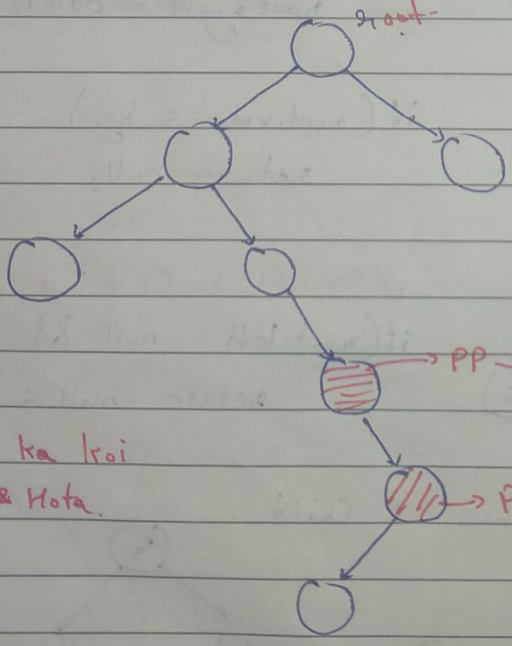
```
if(root.left == null || root.right == null){
    if(root.left == null)
        return root.right;
    else
        return root.left;
}
```

Case (3) :- The Node has 2 Children.

Key = 5 *Concept :- ~~Root~~ Replace 'key' Node with its Inorder Predecessor



Need to find Predecessor (P) & Parent of Predecessor (PP)

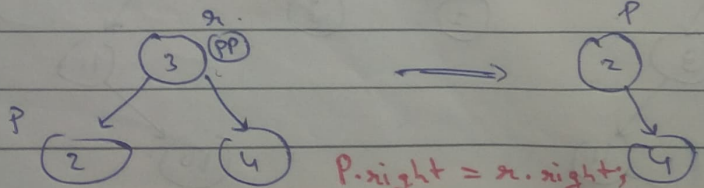



```
pp.right = p.left;
p.left = root.left;
p.right = root.right;
```

* Predecessor ka koi Right Nahi ☒ Hota.

1 → If root == pp :-

key = 3.



P.right = r.right;

return P;

if pred is Child of the Node which you want to delete

// Case (3) :- 2 Child Node :-

```
else {
    TN pred = iop(root);
    TN predParent = parent(root, pred);
    if(root == predParent){
        pred.right = root.right;
        return pred;
    }
    predParent.right = pred.left;
    pred.left = root.left;
    pred.right = root.right;
    return pred;
}
```

```
public TN iop(TN root){
    TN temp = root.left;
    while(temp.right != null)
        temp = temp.right;
    return temp;
}
```

```
public TN parent(TN root, TN pred){
    if(root.left == pred || root.right == pred)
        return root;
    TN temp = root.left;
    while(temp.right != pred)
        temp = root.right;
    return temp;
}
```

★ # Morris Traversal — Google

Traversals → preorder, postorder, Inorder [recursive]
- T.C => O(n)
- S.C => O(h)

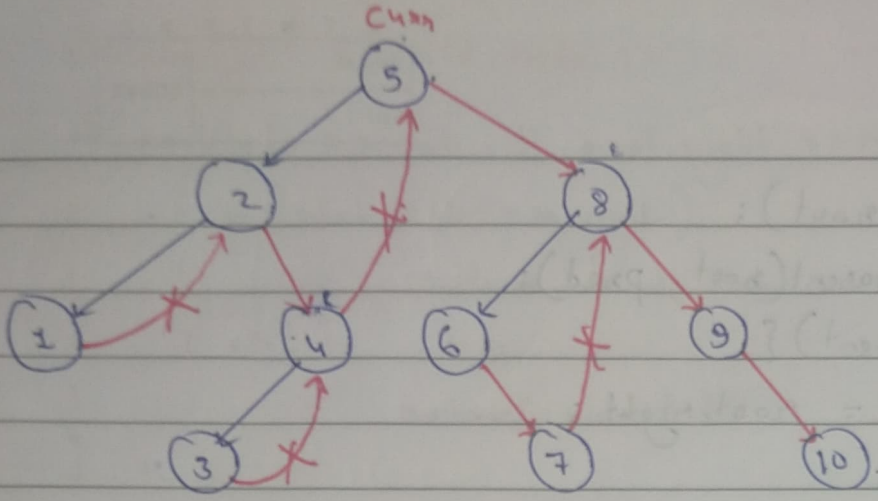
h = log n
h = n.

[Recursive] [Iterative] [Morris]

Morris → Inorder traversal in O(1) space

- Before Interview → Ek baar solve krna hai
- Need to know Inorder Pred.

Concept :- Linking & Unlinking.
- Linking → Change tree structure.
- Unlinking → Bring back Original Structure.



ans = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}

```
while(curr != null){
    if(curr.left != null){
        Node pred = curr.left;
        while(pred.right != null &&
              pred.right != c)
            pred = pred.right;
        if(pred.right == null){
            pred.right = curr.
            curr = curr.left.
        }                          Linking
        else{
            visit(curr);
            curr = curr.right
            pred.right = null
        }                          Unlinking
    }
    else{
        visit(curr);
        curr = curr.right;
    }
}
```