Todays Content:

- Binary Search Tree basics
- Insert / search /
- IS BST()
- Recover BST
- delete()

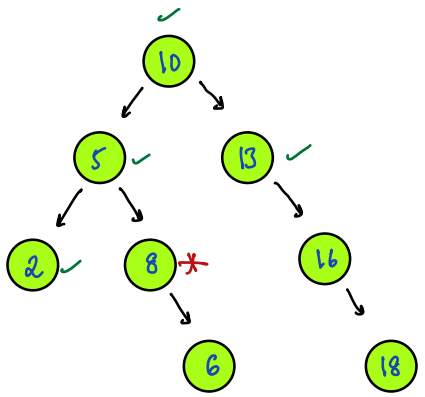
Binary Search Tree (BST):

A binary Tree is said to BST if

For all nodes { All nodes in LST < node.data < All nodes in RST }

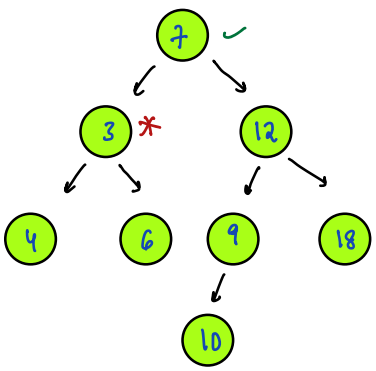
not BST

Ex1:

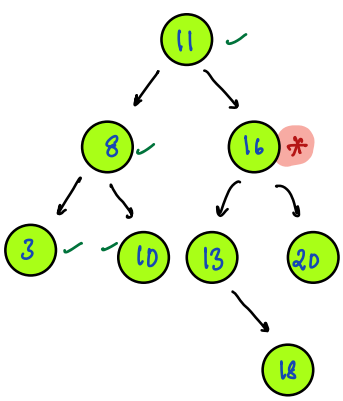


not BST

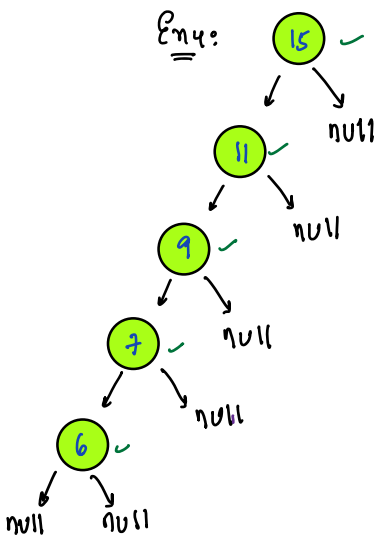
Ex2:



Ex3: not BST

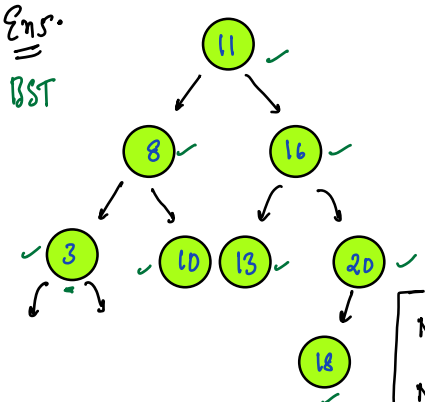


Ex4: BST



Ex5:

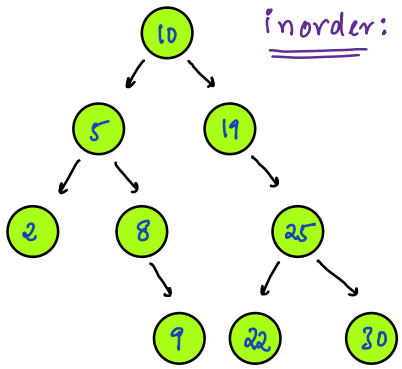
BST



Note1: If we have a null assume it holds property

Note2: In BST our value are distinct

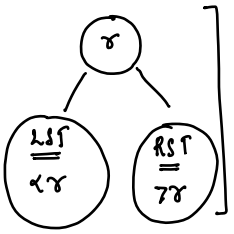
BST property:



inorder: { 2 5 8 9 10 19 22 25 30 }

: inorder is sorted in asc order

:

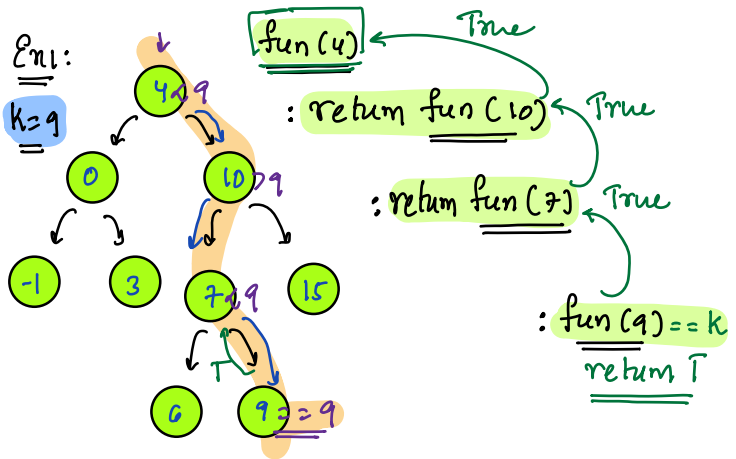


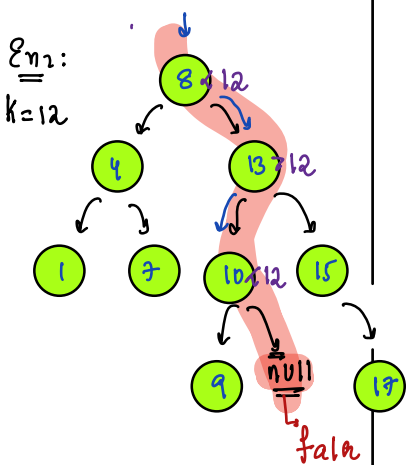
inorder: L D R

↳ : lr, r, > r

obs: In BST inorder is sorted in ascending order

#Search k in BST



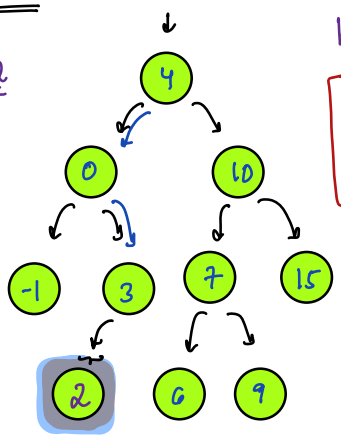


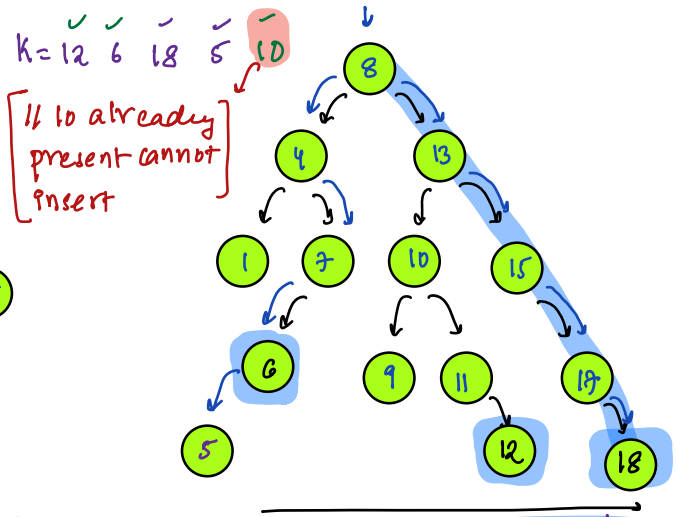
Ass: Check k in given BST & return True or false

```
bool Search(Node root, int k){ TC: O(H)  SC: O(H)
                                  ↳ recursive call stack
    if(root == null) {return false}

    if(root.data == k) {return true}

    if(root.data > k){ // goto left & search
        return Search(root.left, k) // if present in LST
    }
    else{                                  Trace:
        return Search(root.right, k) // if present in RST
    }
}
```

TODO: iterative as well

Insertion



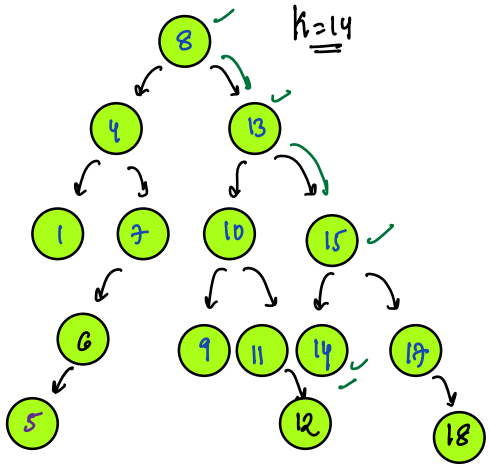


Note: When ever we insert an ele, we insert at empty slot
↳ we insert a leaf node

Ass: Given k, insert node in given BST & return root node of BST

```
Node insert(Node root, int k){                    TC: O(H)  SC: O(H)
1   if(root==null){
        Node nn = new Node(k)
        return nn;
    }
    if(root.data==k){ return root;}
2   if(root.data>k){
        root.left= insert(root.left, k) // return root of BST
    }
3   else
        root.right= insert(root.right, k) // return root of BST
    return root
}
```

Trace Inserting:



insert(root == 8)

→ 8.right = insert(13, 14) = 13

return 8

13

insert(root = 13)

13.right = insert(15, 14) = 15

return 13

15

insert(root == 15)

15.left = insert(null, 14) = nn

return 15

nn

insert(root = null)

nn = new Node(14)

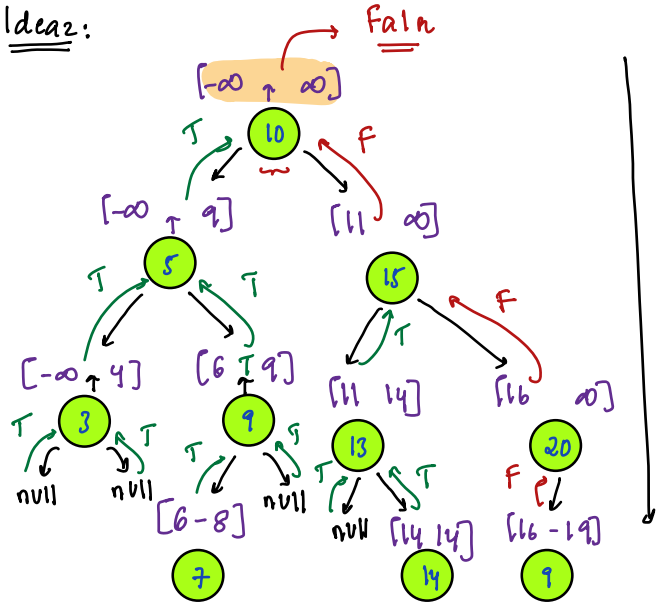
return nn

3Q) Check if a given Tree is BST or not?

Idea1: Check if Inorder traversal is in increasing order or not?

TC: O(N) SC: O(N+H) → recursive stack size

Idea2:



Idea: From top-down we are checking if node is in given range, if it's, check for LST & RST if it's not return false

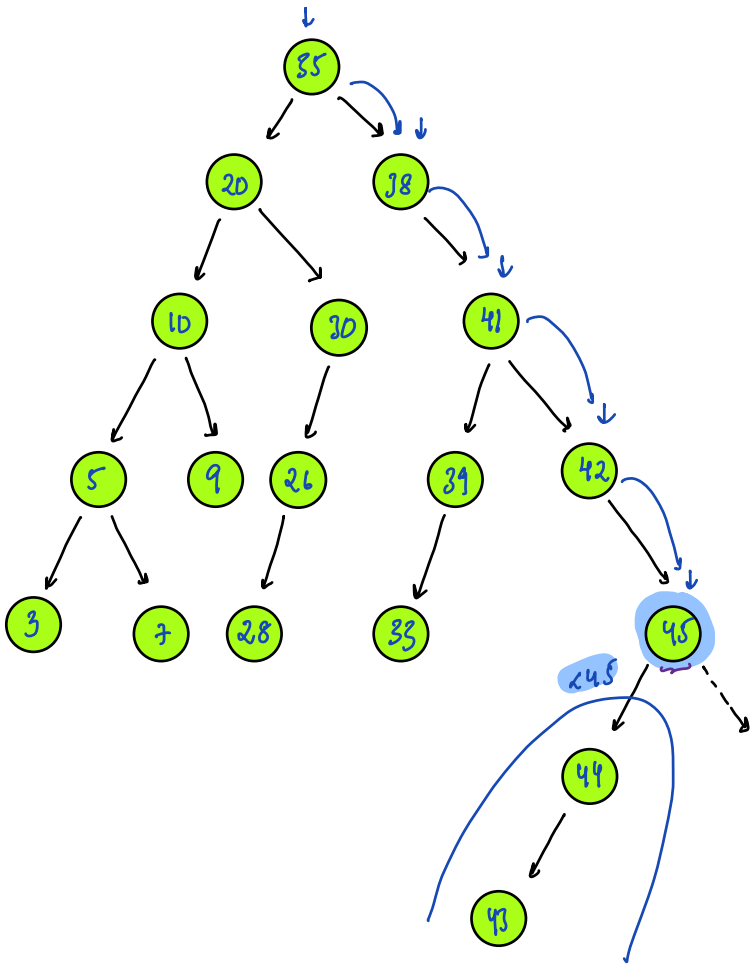
Ans: Given root node Check if all nodes are in given range [s e]

Call IsBST(root, INT_MIN, INT_MAX)

```
bool IsBST(Node root, int s, int e){   TC: O(N) SC: O(H)
    if(root==null) {return true}
    if( s <= root.data && root.data <= e){
        //Check if LST & RST are in given range
        bool l= IsBST(root.left, s, root.data-1) // return if LST is BST
        bool r= IsBST(root.right, root.data+1, e) // return if RST is BST
        return l&&r
    }
    else
    | return false
    }
}
```

Max in BST:

Given root node of a BST, return max of BST.



Idea1: last element in inorder, TC: O(N)

Idea2: keep traveling until temp.right == null : TC: O(H)

↳ Note1 Max node cannot have a right child SC: O(1) ⇒ Iterative code

TODO: Min of BST: traverse to left TC: O(H)

Note2: Min node cannot have a left child SC: O(1) ⇒ Iterative code

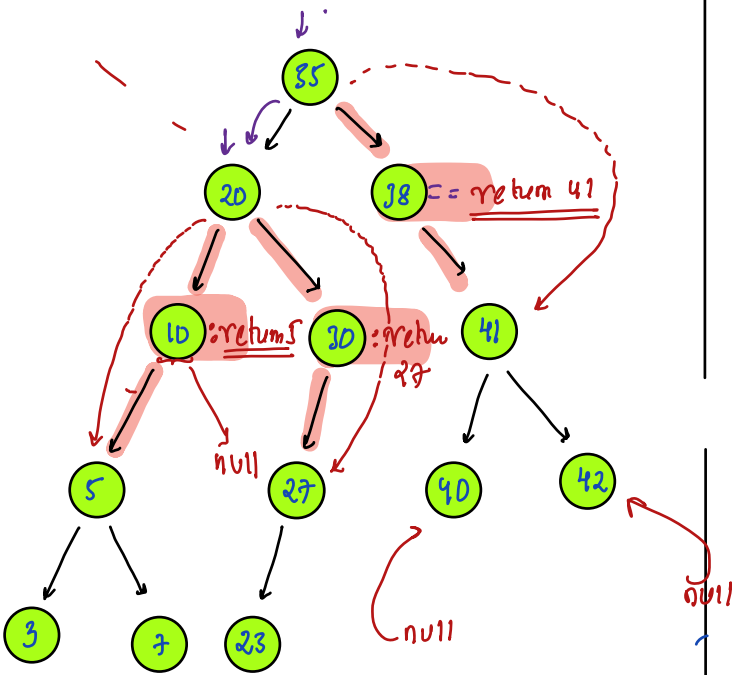
Delete node :

→ leaf node = {39, 19, 45}

→ 1child {10, 38, 30}

: return 1child of node we are deleting

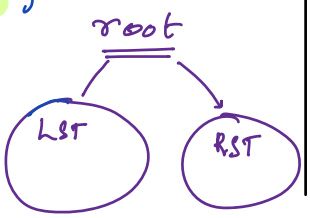
→ 2child



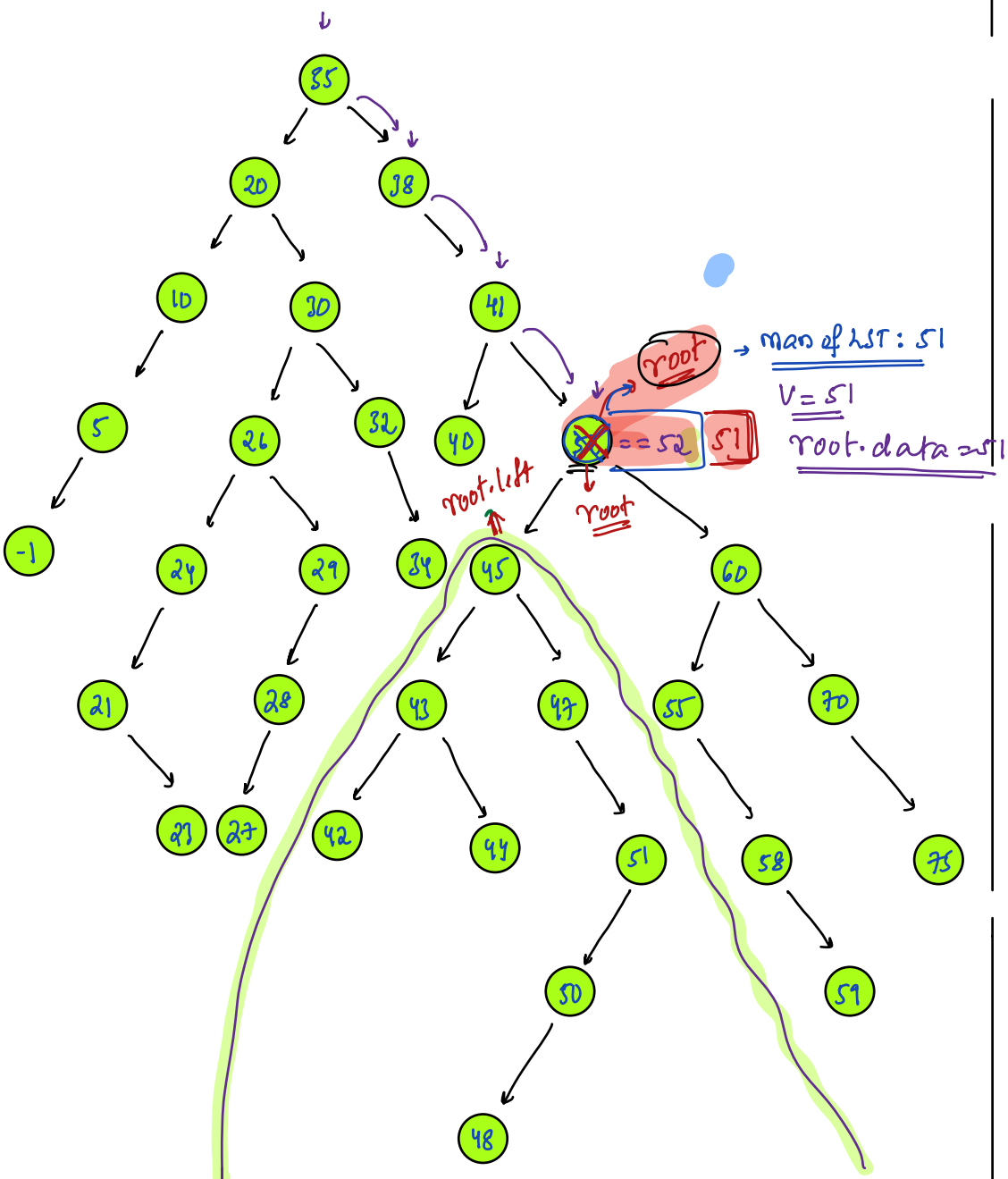
Ass: Given root of BST, delete node with value k & return node of BST

↳ // Assume k is always present in given Tree

```
Node delete(Node root, int k){ TC: O(H) SC: O(H)
    if(root.data == k){
        //Case-I leaf
        if(root.left==null && root.right==null){return null}
        Case-II: Single Child, return
        if(root.left==null){return root.right}
        if(root.right==null){return root.left}
        Case-III: 2 child.
        int v = max(root.left)
        root.data = v;
        root.left = delete(root.left, v) //root of LST
        return root;
    }
    if(root.data > k){
        root.left = delete(root.left, k) //return root of LST
    }
    else{
        root.right = delete(root.right, k) // return root of RST
    }
return root;
}
```

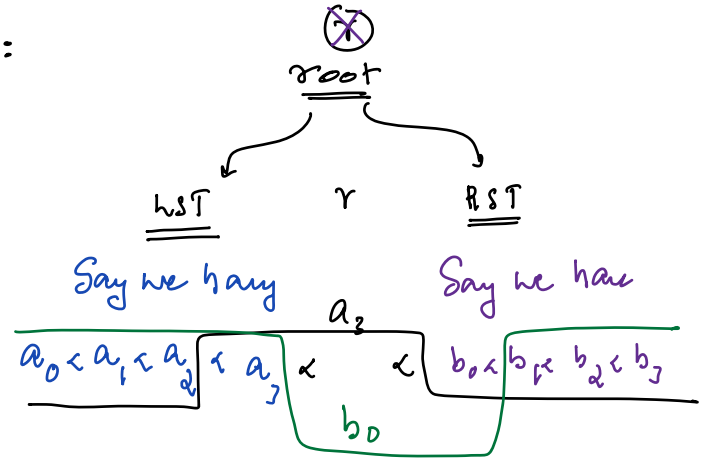


808) Delete BST 52



→

BST :



Obs: With value in tree, I can replace r, such that it should still hold property of BST

In above eg: r, can be plac a_2 or b_0

Obs: a_j = max of LST

: b_0 = min of RST

Final obs: node replaced max of LST or min of RST, it can still retain BST