Todays Content:

- Binary Search Tree basics
- Insert / update /
- IsBST()
- Recover BST
- delete

Binary Search Tree (BST):

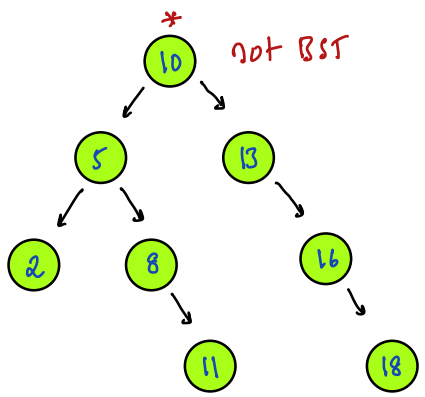
A binary Tree is said to BST if

∀ all nodes { All lST < node.data < All RST }

For all nodes ⟺ node.data

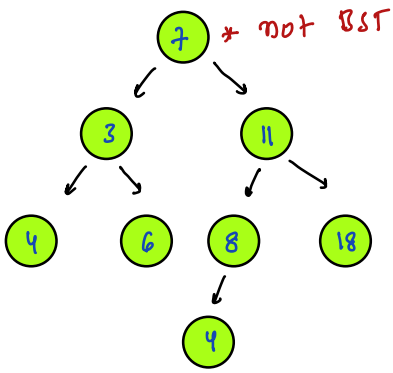
< Entire lST < Entire RST

Ex1:



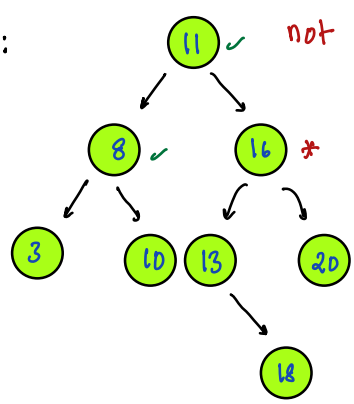
not BST

Ex2:



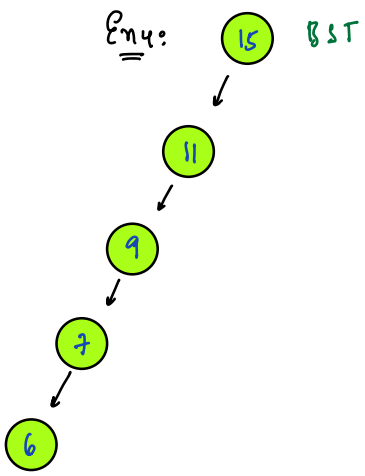
not BST

Ex3:



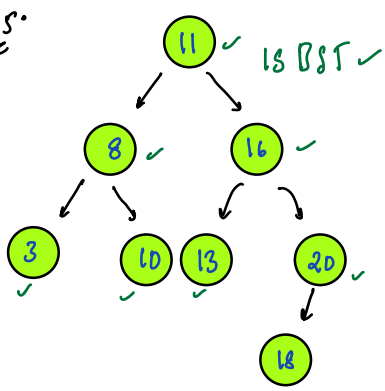
not BST

Ex4:



BST

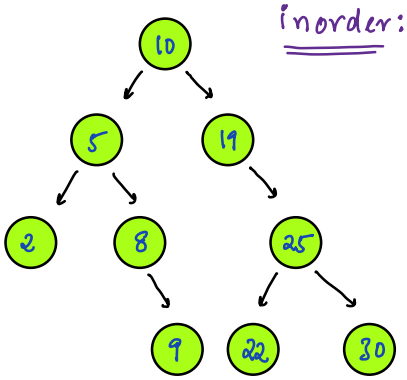
Ex5:



is BST

Note: If something is null we can assume it will satisfy condition

BST property:

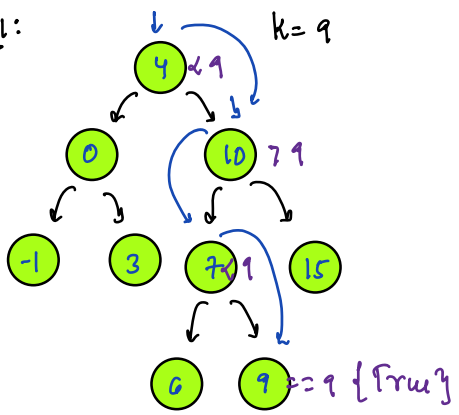


inorder: 2 5 8 9 10 19 22 25 30

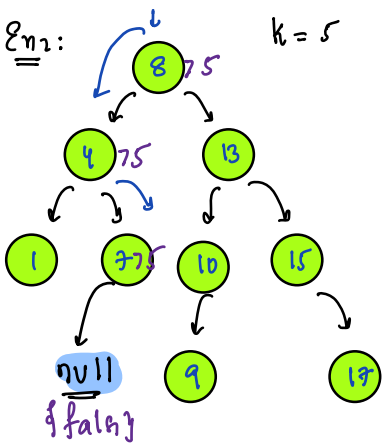
Sorted in ascending order : Inorder

Search K in BST

Ex1: k = 9



Ex2: k = 5

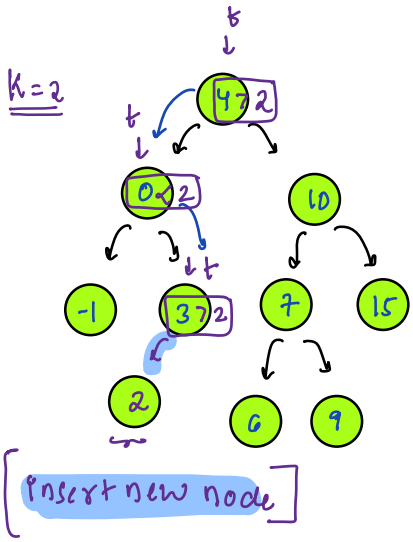


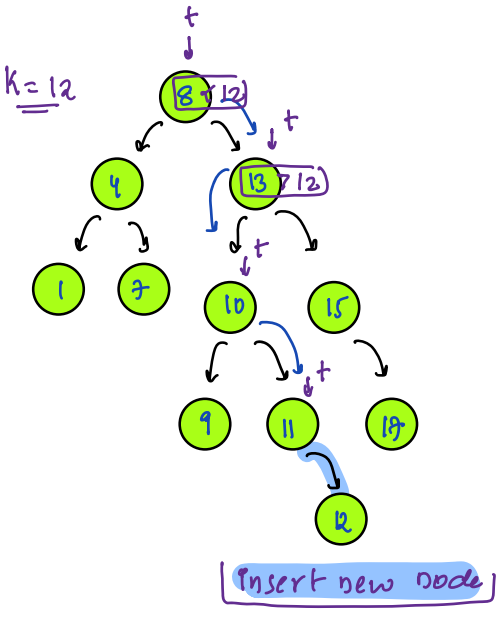
{false}

```
bool Search(Node root, int k){   TC: O(H)   Height → Worst Case: N
                                            → Best Case: log N ?
    while(root != null){
        if(root.data == k){ return True }
        if(root.data > k){ // goto left & search
            root = root.left
        }
        else{
            root = root.right
        }
    }
    return false
}
```

[Given N nodes min height, we can get is log N]

Insertion






```
Node insert(Node root, int k){   TC: O(H)  SC: O(1)
    if(root==null){ root = new Node(k) return root }

    Node tmp = root;
    while(root!=NULL){
        if(root.data > k){ // goto left
            if(root.left==null){
                root.left = new Node(k)
                break;
            }
            root = root.left
        }
        else{ // goto right
            if(root.right==null){
                root.right = new Node(k)
                break
            }
            root = root.right
        }
    }
    return tmp;
}
```

Eg: root = null, k = 5

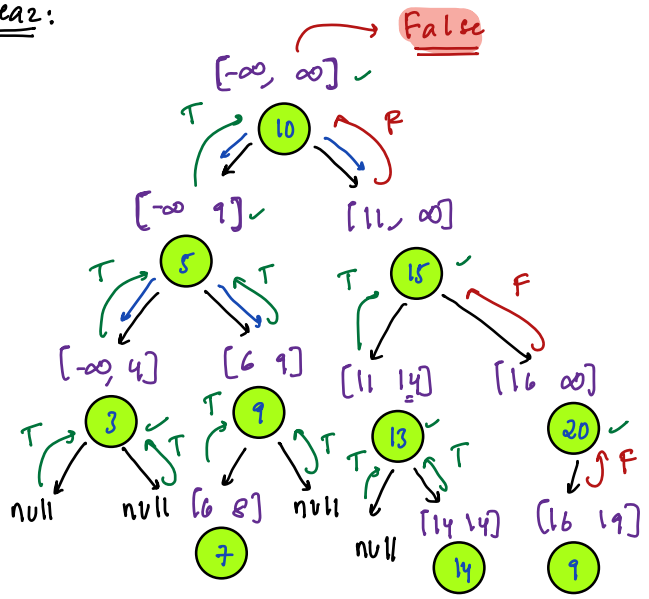
root = 5

30) Check if a given Tree is BST or not?

Idea1: Get Inorder to arr[], check if it's Inc or not?

TC: O(N) SC: O(N) {TODO: comp every ele with prev & update prev every time}

Idea2:




```
bool isBST(Node root, int s, int e){ TC: O(N) SC: O(H) recursive stack size
    if(root == null){return true}
    if( s <= root.data && root.data <= e){
      // check if lst & RST are in range
        bool l = isBST(root.left, s, root.data-1)
        bool r = isBST(root.right, root.data+1, e)
    }   return l && r // if both are true we anyway return true
    return false
}
```

8:20 ⟶ 8:30 am

48) Recover sorted arr[]

↳ Given an arr[], which is formed by swapping 2 distinct index positions in a unsorted arr[], get original sorted arr[]

Ex1: sort[] = { 4 10 12 14 18 19 25 28 }

Input arr[] = { 4 10 19 14 18 12 25 28 }

arr[2] < arr[3] * 1st ele = arr[2]

arr[4] < arr[5] * 2nd ele = arr[5]

Iterate arr[] & compare 2 adj ele

1) 1st time comparison fails: consider 1st ele & consider 2nd ele

2) 2nd time compare fails: consider 2nd ele

arr[] = { 2 6 23 10 14 19 8 40 51 }

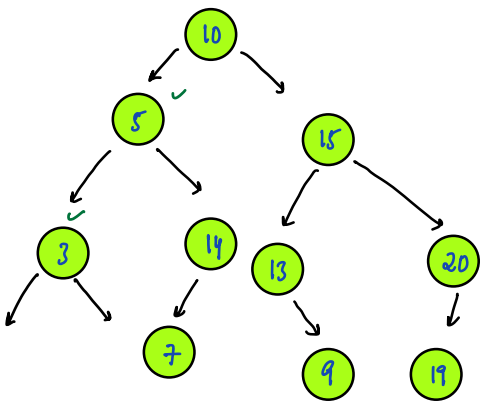
arr[] = { 3 6 10 15 12 17 20 33 }

1st ele 2nd ele

arr[] = { 3 < 6 < 9 < 31 < 14 < 18 < 24 < 12 35 40 }

1st ele 2nd ele ..... 2nd ele

508) Given a BST, which is formed by swapping 2 distinct nodes, recover original BST.



Idea: Do the inorder on tree & find mis matches

inorder:

3 5 7 14 10 13 9 15 19 20

1st 2nd ... 2nd

```
Node prev = null
Node first = null
Node sec = null
void inorder(Node root){ TC: O(N) SC: O(H) recursive stack
    if(root == null){return}
    inorder(root.left)
    if(prev != null && prev.data > root.data && first == null){
        first = prev, sec = root
    }
    else if(prev != null && prev.data > root.data){
        sec = root
    }
    prev = root // update prev
    inorder(root.right)
}
// swap first.data & sec.data
```

68) floor(root, k) : Find greatest ele < k

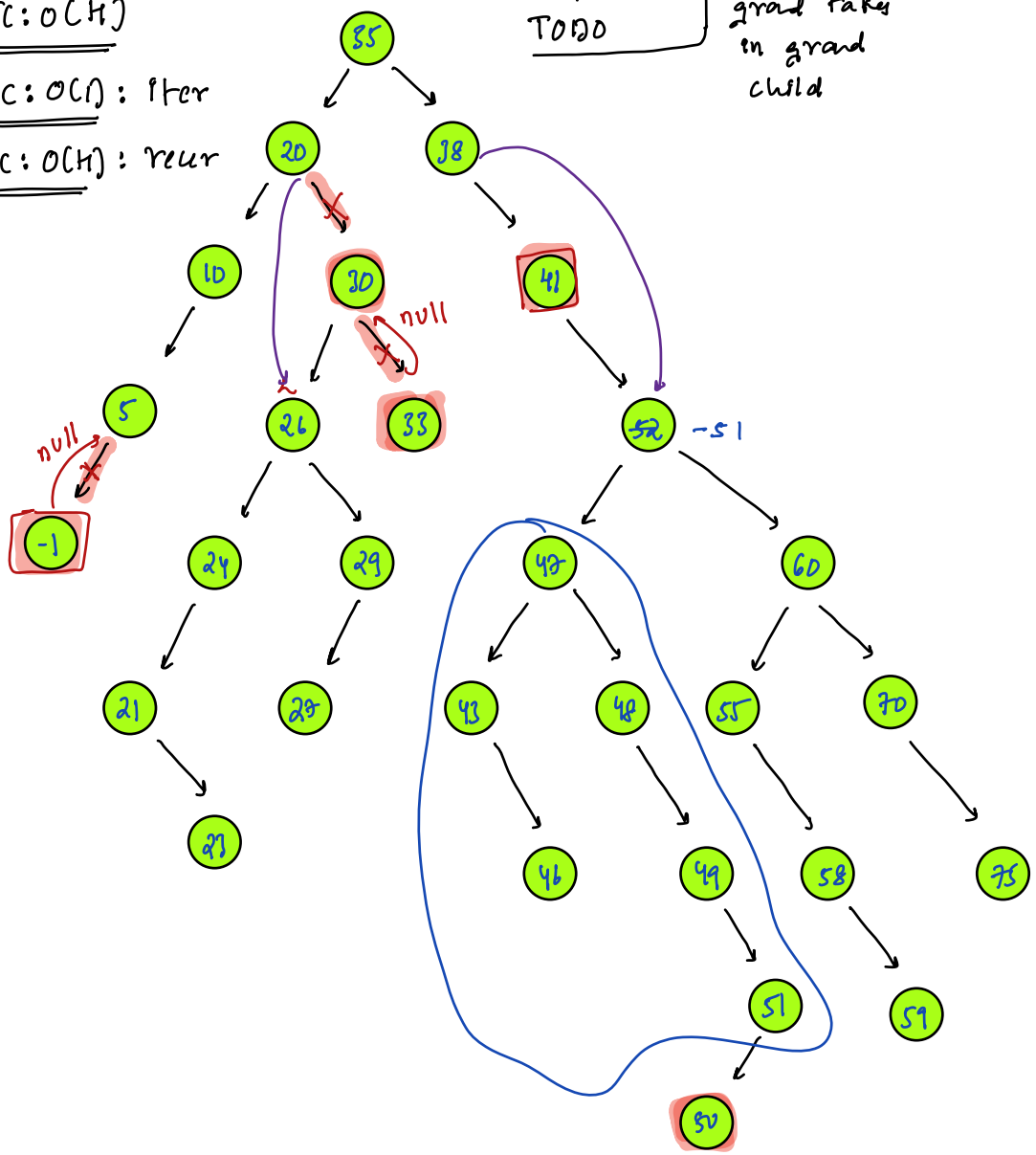
78) ceil(root, k) : Find smallest ele > k

88) Delete BST → -1 33 50 (leaf node, TODO) | 30 41 (single child, grand takes in grand child)

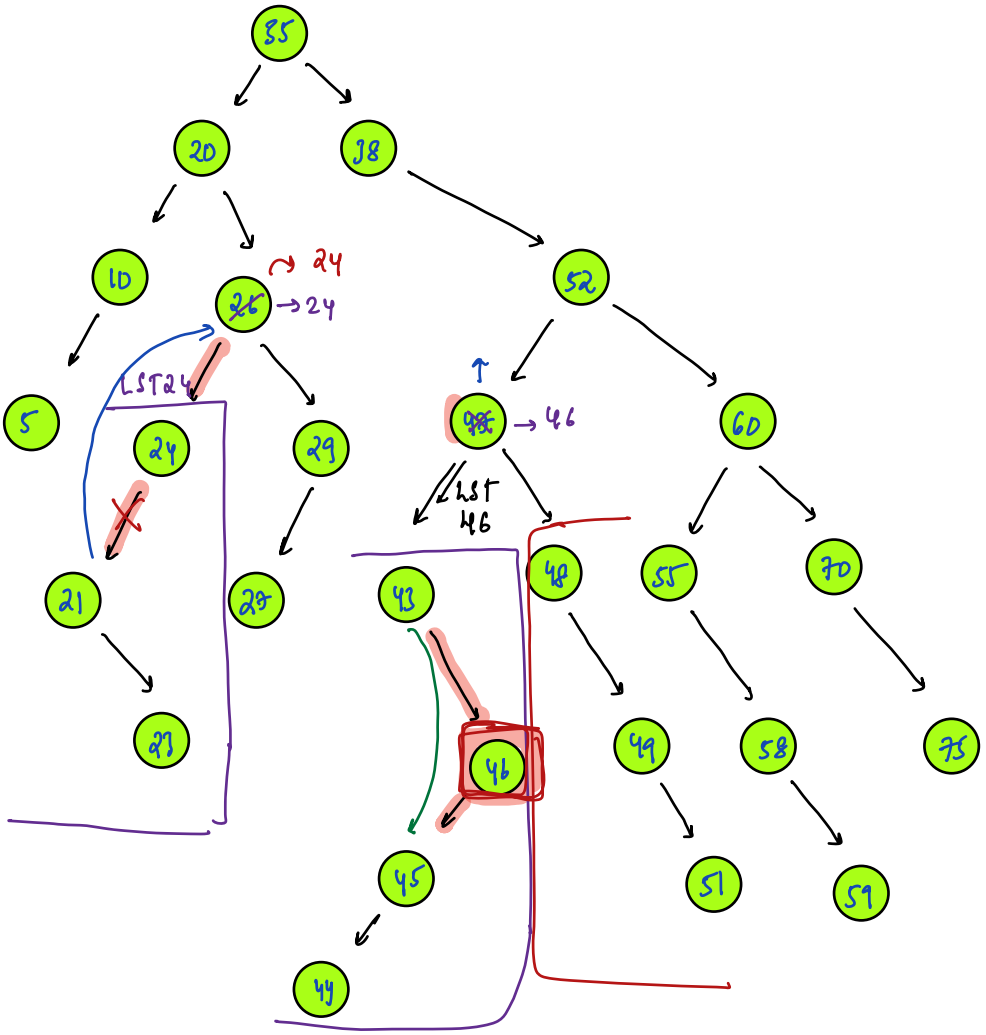
TC: O(H)

SC: O(1) : Iter

SC: O(H) : recur



Case-[1] : 47 26



Obs1: max node in a tree can have can only have 1 child at max.

Case 1: Deleting leaf node

Case 2: Deleting node 1 child:

→ hand over 1 child responsi to it's parent node

Case3: Deleting node 2 child

→ replace node with max of LST

→ delete max of LST

↳// max will at max have 1 child