Todays Contant:

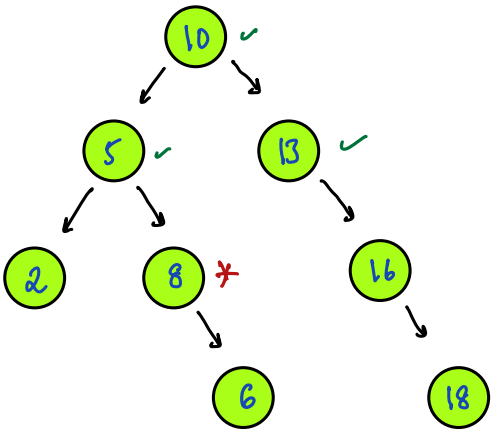
- Binary Search Tree basics
- Insert / search /
- Is BST()
- Recover BST
- Sorted arr[] to BBST

Binary Search Tree (BST):

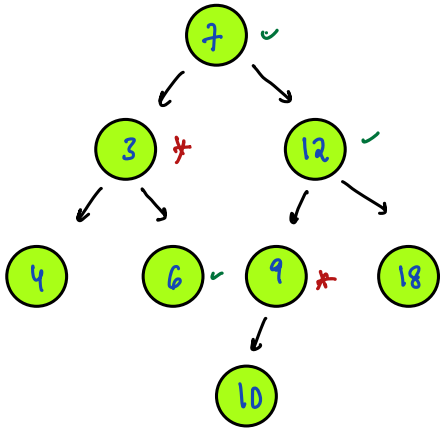
A binary Tree is said to BST if

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

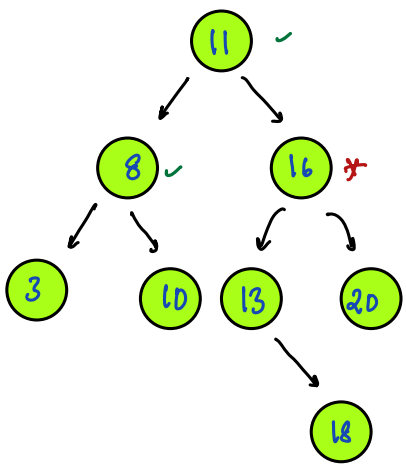
Ex1: NOT BST



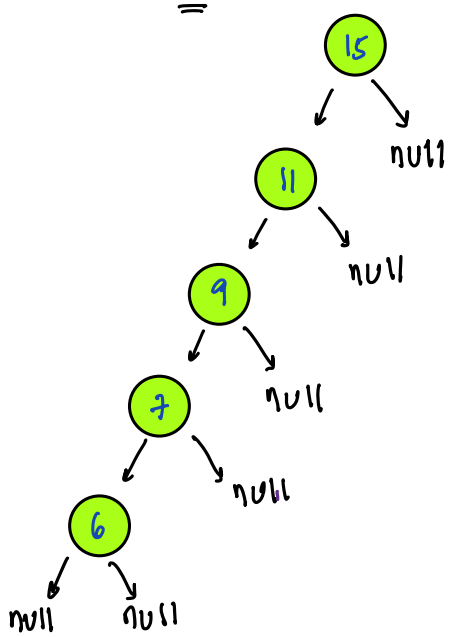
Ex2: NOT BST



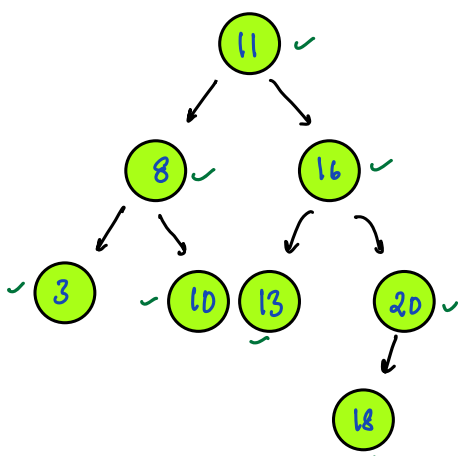
Ex3: NOT BST



Ex4: BST



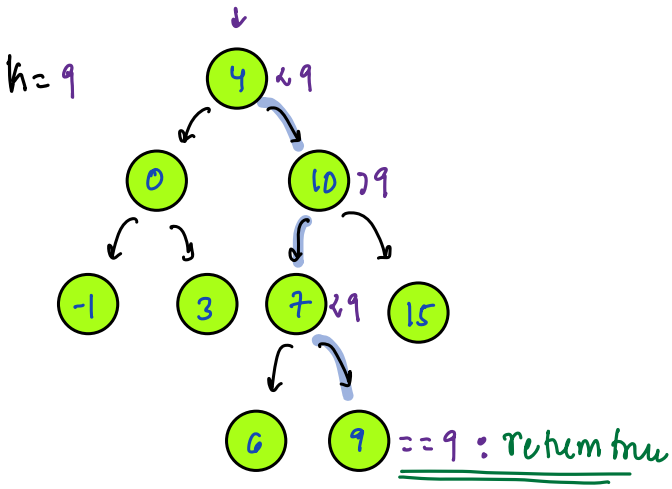
Ex5: BST

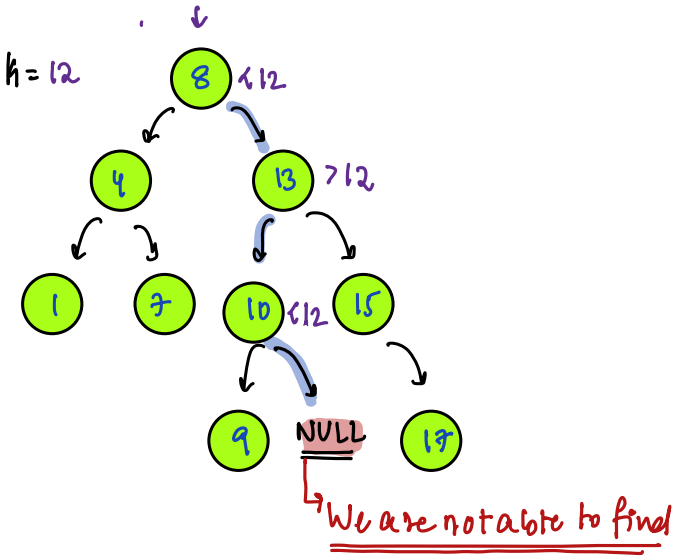


Note1: If we have a null assume it holds property

Note2: In BST, values are distinct.

#Search k in BST



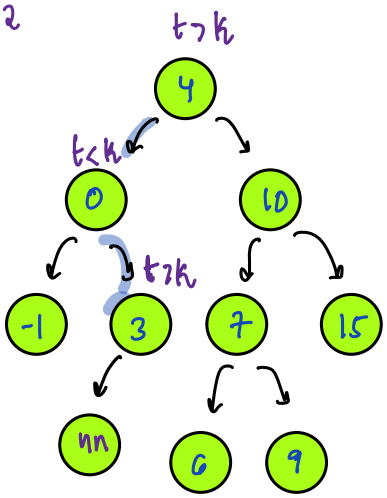



```
bool Search(Node root, int k){ TC: O(H) SC: O(1)
    Node temp = root; // temp is obj reference
    while(temp != NULL){      ↳ temp.data temp.next
        if(temp.data == k){return true}
        if(temp.data > k) { temp = temp.left}
        else{temp = temp.right}
    }
    return false;
}
```

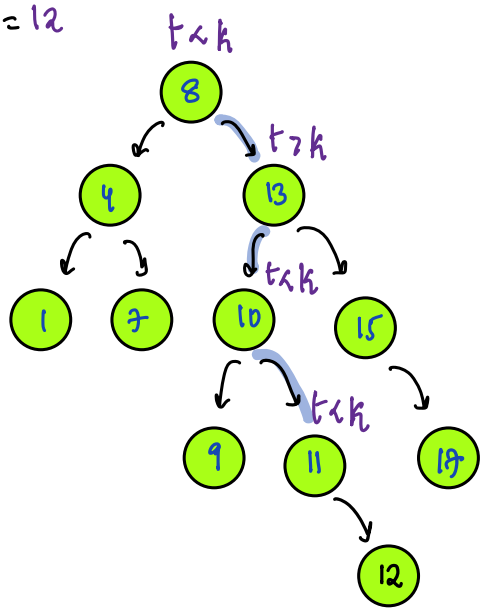
Note: When ever we insert an ele, we insert as leaf node

Insertion in BST

k=2



k=12

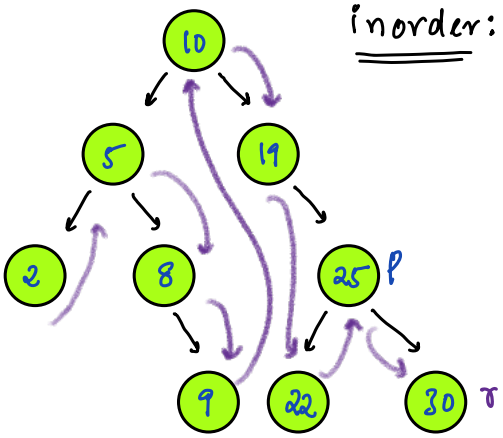


Q: Insert & return root node

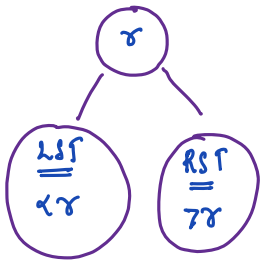
```
Node insert(Node root, int k){ TC: O(H) SC: O(1)
    Node nn = new Node(k);
    Node temp = root;
    if(root==null){return nn}
    while(temp!=NULL){
        if(temp.data>k){ //goto left
            if(temp.left==NULL){ temp.left=nn; break}
            else{ temp=temp.left}
        }
        else{ //goto right
            if(temp.right==NULL){ temp.right=nn; break}
            else{ temp=temp.right}
        }
    }
    return root;
}
```

{ Insertion happens }

BST property:



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



inorder: L D R

↳ : <r, r, >r

Obs: In BST: Inorder in increasing

Q: Given BT check if its BST or Not?

Ideal: Find inorder & check sorted or not?

Way1: Store entire inorder in arr[] & check increasing?

TC: O(N+N) SC: O(N) → // Storing in arr[]?

Way2: { 2 5 8 9 10 19 22 25 30 }

Compare each element to previous element to check if data is sorted or not?

↳ Store prev node & use it to compar

```
Node prev = NULL
boolean flag = True;
void isBST(Node root){ TC: O(N) SC: O(H)
    if(root==null){return}
    isBST(root.left)
    if(prev!=NULL && prev.data > root.data){
        flag = False;
    }
    prev = root; // We update prev; Root is updated by recursion.
    isBST(root.right) // prev is following root.
}
```

```
boolean solve(Node root){
    prev = NULL;
    flag = True;
    isBST(root);
    return flag;
}
```

Reinitialize global variable before function call

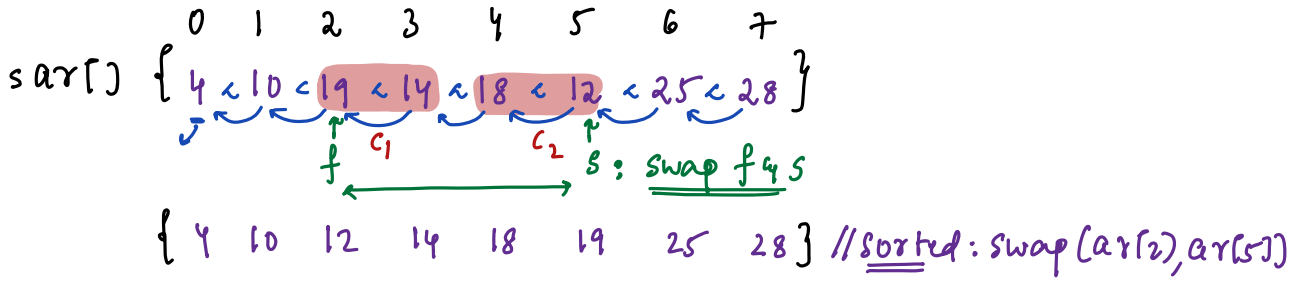
Note: Prev it won't always be on left

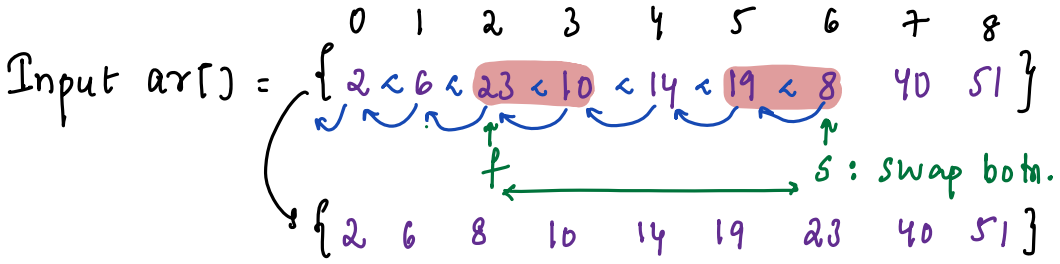
Recover Sorted arr[]

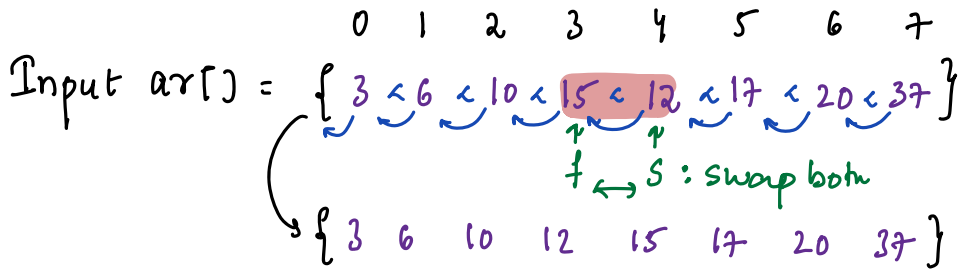
Given arr[N], which is formed by swapping 2 distinct index positions in a sorted (inc) arr[] only 1 time.

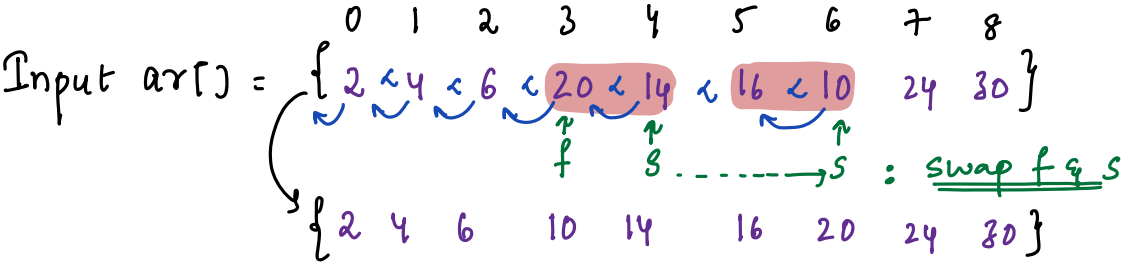
Get original sorted arr[], Expected TC: O(N)

Input









Idea: Iterate on arr[]: compare ele with prev

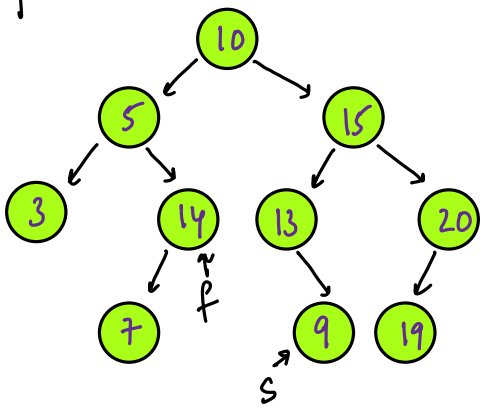
If 1st comp fails: Update f & s else if 2nd comp fails: Update s.

Once entire traversal done: Swap f & s TC: O(N) SC: O(1)

Recover BST:

Given a BT, which is formed by swapping 2 distinct nodes data recover original BST, by swapping 2 nodes data back

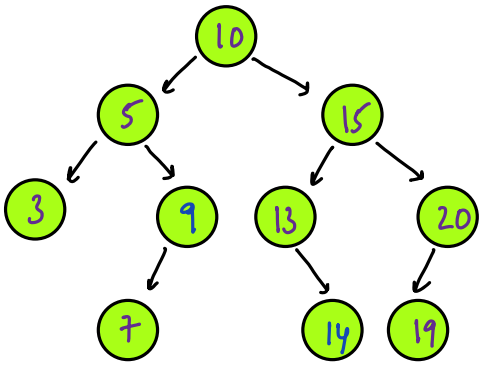
Input:



Node f = 14

Node s = 9

Swap f & s



Idea: Apply Inorder traversal & compare ele with prev:

if compn fails 1st time: update f & s

if compn fails 2nd time: update s

```
Node p = null, f = null, s = null
void inorder(Node root){ TC: O(N) SC: O(H)
    if(root==null){return}
    inorder(root.left)
    if( prev!=NULL && prev.data > root.data){
        if( f==null){//condition first fail  p < r
            f=prev; s=root;
        }else{ //condition fail 2nd  p < r
            s=root;
        }
    }
    p=root;
    inorder(root.right)
}
```

```
Node solve(Node root){
    f=null; s=null; prev=null
    inorder(root);
    swap f & s => data
    int temp = f.data
    f.data = s.data
    s.data = temp
}
```

4. BBT : Balanced Binary Tree

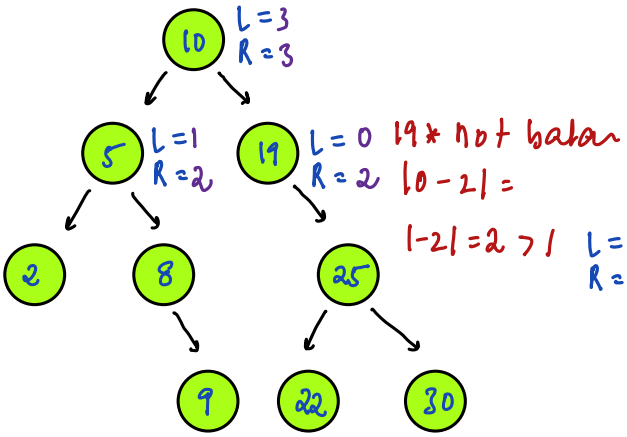
A BT is said to be balanced,

If at every node, abs|Height(left) - Height(Right)| <= 1

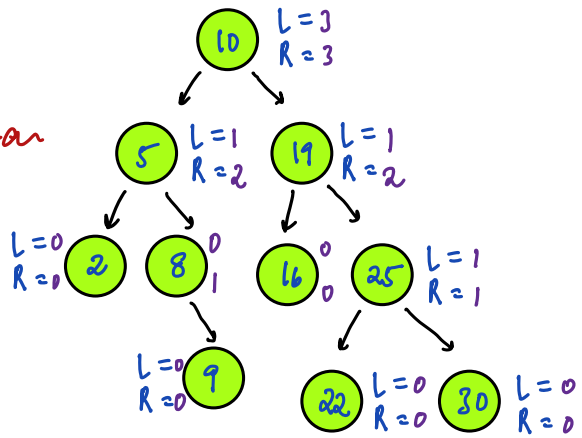
Note: Height calculate using nodes: = Height with Edges+1

Note2: abs(-2) = 2 abs(5) = 5

Ex1: Not Balanced



Ex2: Balanced.



Note: In a Balanced Binary Tree = $h \approx \log_2(N)$

Where N is no: of nodes in Tree.