Agenda

1) BST intro and its properties
2) Search in BST (TC comparison with BT)
3) Insert in BST
4) Is BST
5) Sorted array to balanced BST
6) Recover BST (if possible)

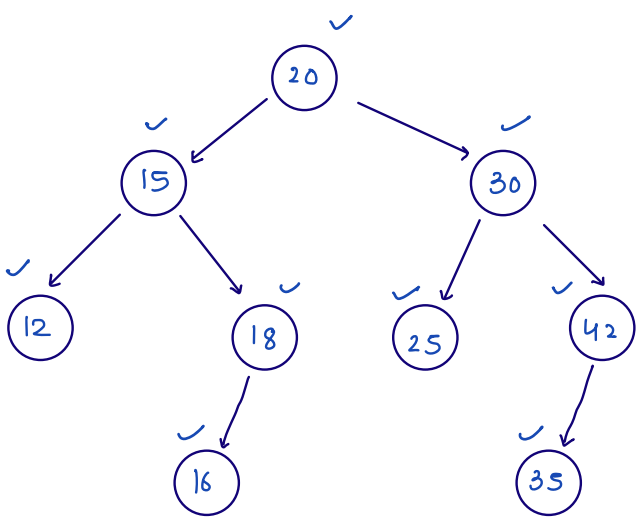
Binary Search Tree (Introduction)

A binary tree is a BST if

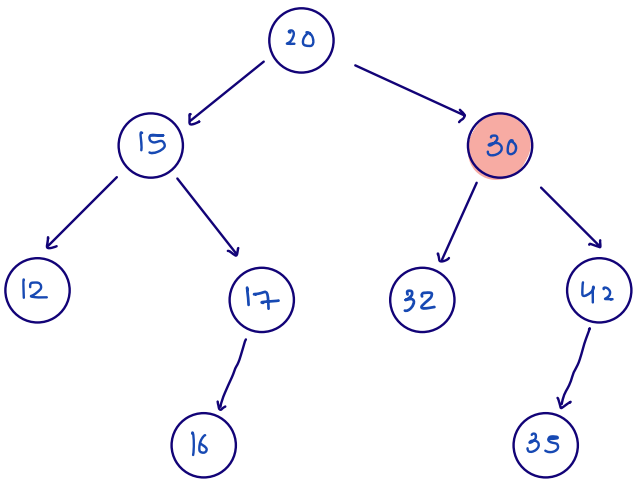
all nodes < node.val < all nodes

coming in left subtree — coming in right subtree

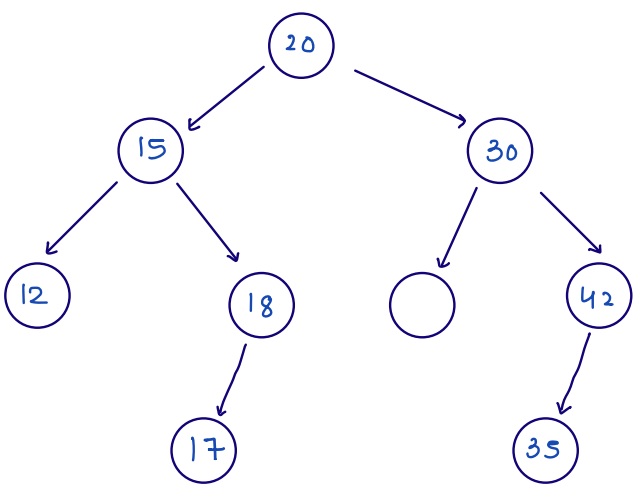
} this condition should be true for all nodes



→ Binary search tree

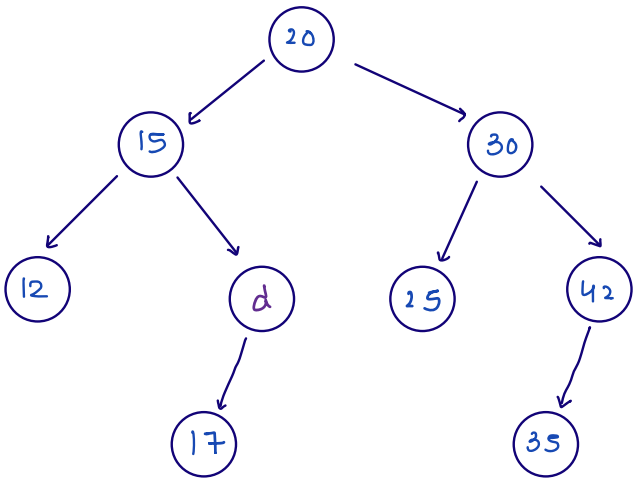


not a BST



0 - 29 ✗

21 - 29 ✓

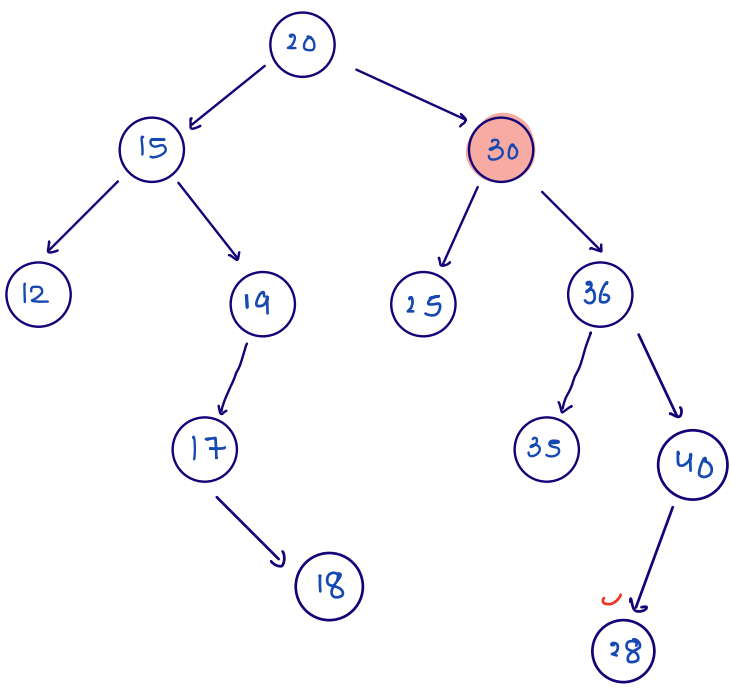


due to 20 and 15

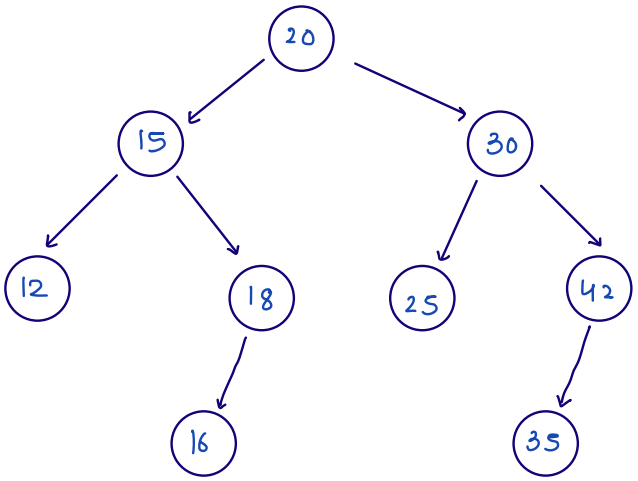
$15 < d < 20$

On left of d there is 17 so d can be only

18 & 19



Not a BST



Inorder

L N R

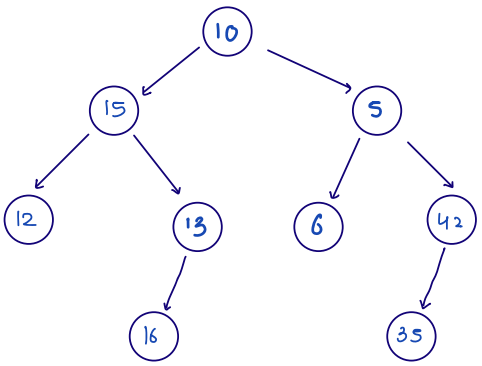
L < N < R

Inorder: 12 15 16 18 20 25 30 35 42

In a BST inorder is always sorted

Binary tree vs Binary search tree

Binary tree



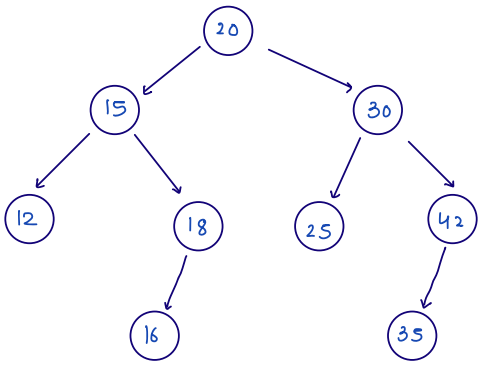
Search for 16

TC : O(n)

SC : O(h)

↳ height of Binary tree

Binary search tree

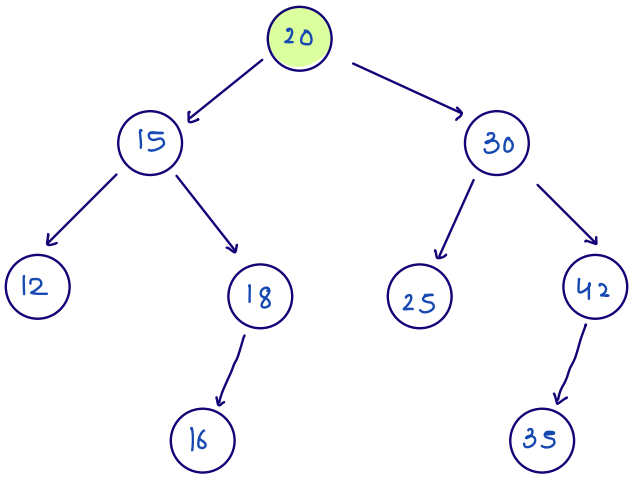


Search for 16

TC : O(h) } travelling at max length of longest branch

SC : O(h)

Q.1 Given root node of a BST, search if K exists or not.



K = 28 (false)

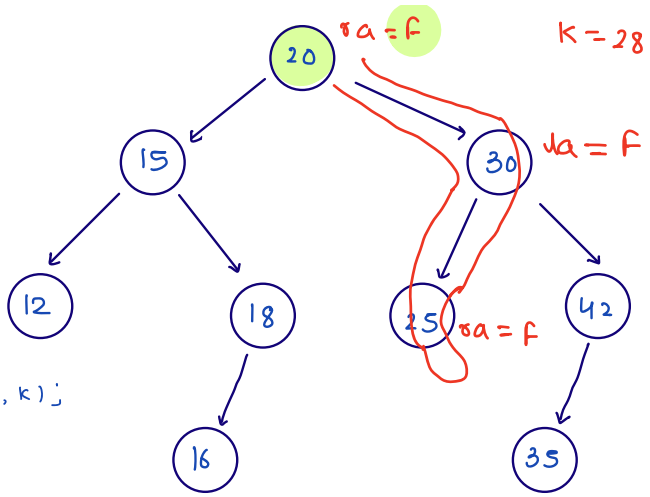
K = 18 (true)

```
boolean searchInBST(Node node, int K) {
    if (node == null) {
        return false;
    }
    if (node.val == K) {
        return true;
    }
    else if (node.val < K) {
        boolean ra = searchInBST(node.right, K);
        return ra;
    }
    else {
        boolean la = searchInBST(node.left, K);
        return la;
    }
}
```

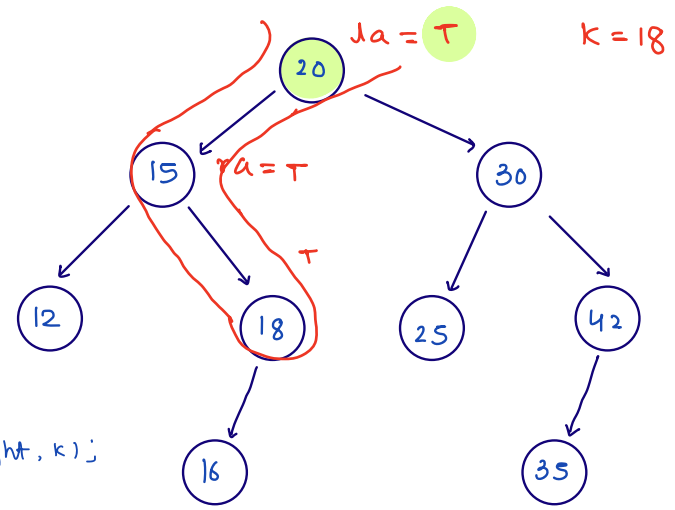
TC: O(h)

SC: O(h)

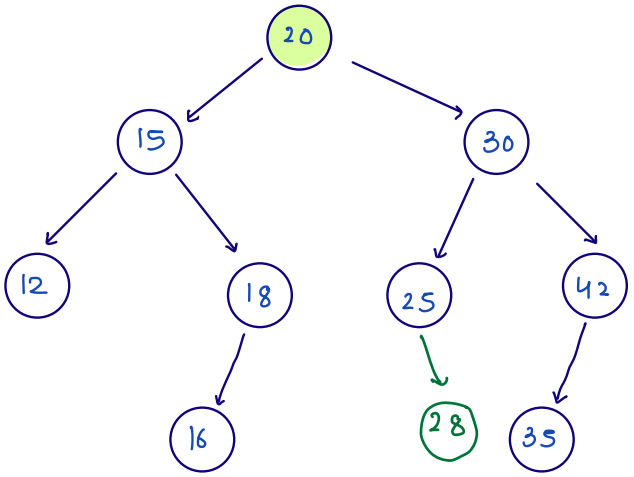
```
boolean searchInBST(Node node, int k) {
    if (node == null) {
        return false;
    }
    if (node.val == k) {
        return true;
    }
    else if (node.val < k) {
        boolean ra = searchInBST(node.right, k);
        return ra;
    }
    else {
        boolean la = searchInBST(node.left, k);
        return la;
    }
}
```



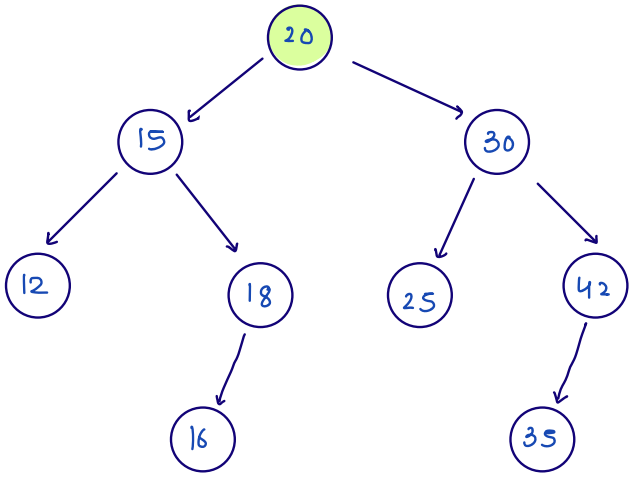

```
boolean searchInBST(Node node, int k) {
    if (node == null) {
        return false;
    }
    if (node.val == k) {
        return true;
    }
    else if (node.val < k) {
        boolean ra = searchInBST(node.right, k);
        return ra;
    }
    else {
        boolean la = searchInBST(node.left, k);
        return la;
    }
}
```



Q.2 Given root of a BST, insert node with data k in this BST.
(Insertion should be done without shuffling the existing node)



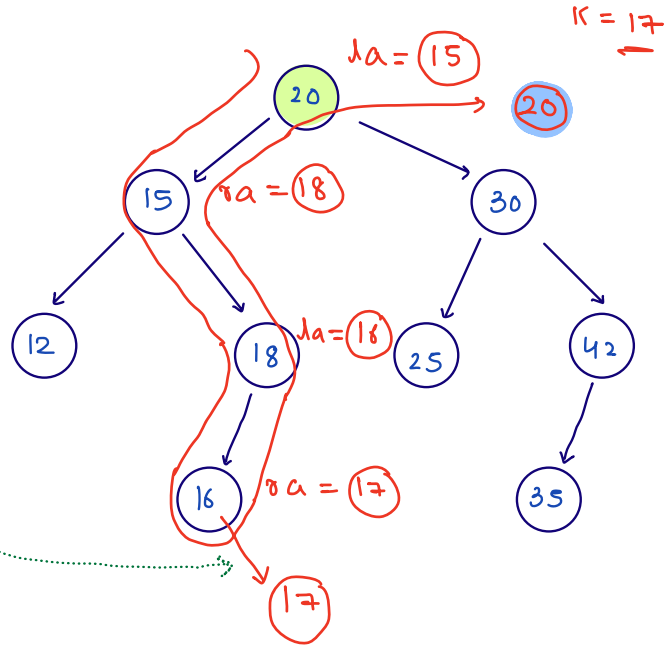
k = 28



k = 17

```
Node insert (Node node, int k) {
    if (node == null) {
        Node nn = new Node(k);
        return nn;
    }
    if(node.val == k) {
        return node;  // no need of work
    }
    else if(node.val < k) {
        Node ra = insert(node.right, k);
        node.right = ra;
        return node;
    }
    else {
        Node la = insert(node.left, k);
        node.left = la;
        return node;
    }
}
```

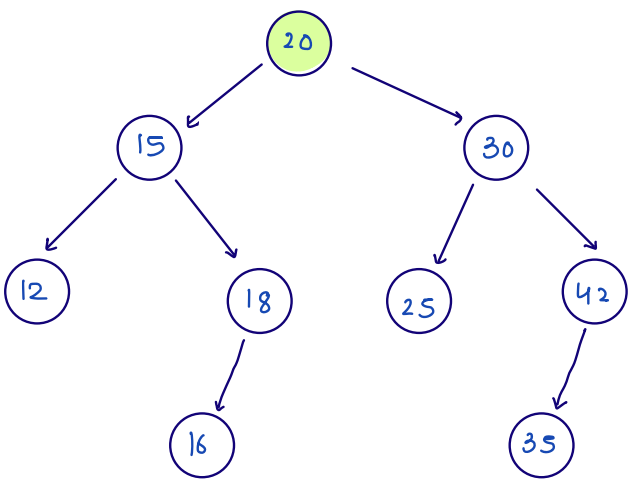
```
Node insert (Node node, int k) {
    if (node == null) {
        Node nn = new Node(k);
        return nn;
    }
    if (node.val == k) {
        return node;   // no need of work
    }
    else if (node.val < k) {
        Node ra = insert (node.right, k);
        node.right = ra;
        return node;
    }
    else {
        Node la = insert (node.left, k);
        node.left = la;
        return node;
    }
}
```



todo: above ques with return type void

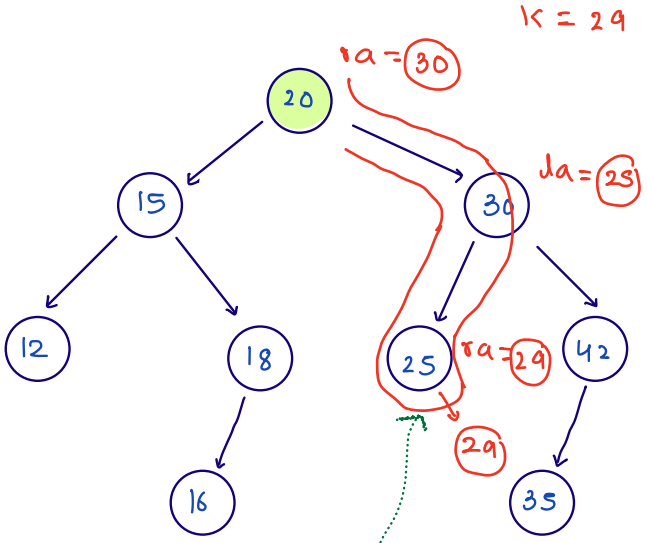
Hint: proactive ↓ that child existing or not

extra: Check

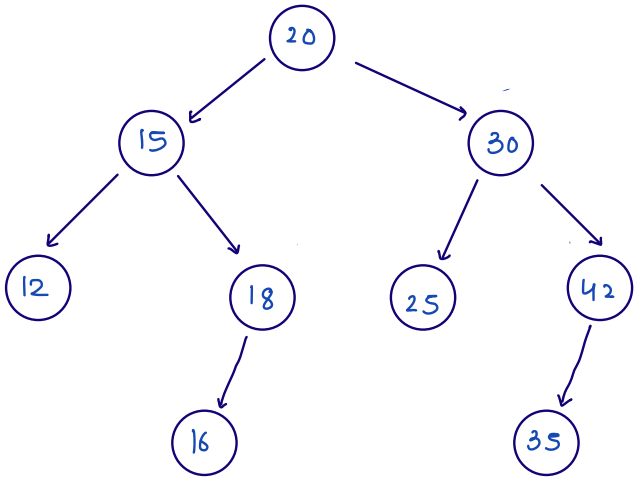


Dry run

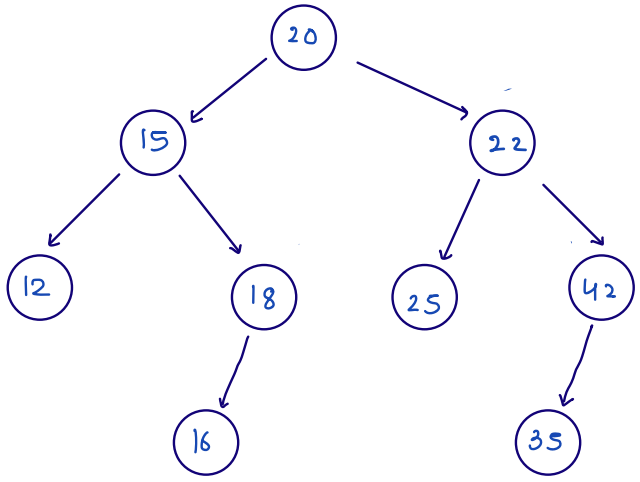
```
Node insert (Node node, int k) {
    if (node == null) {
        Node nn = new Node(k);
        return nn;
    }
    if (node.val == k) {
        return node;   // no need of work
    }
    else if (node.val < k) {
        Node ra = insert (node.right, k);
        node.right = ra;
        return node;
    }
    else {
        Node la = insert (node.left, k);
        node.left = la;
        return node;
    }
}
```



Q.3 Given root of a binary tree, check if it is BST or not.



→ true

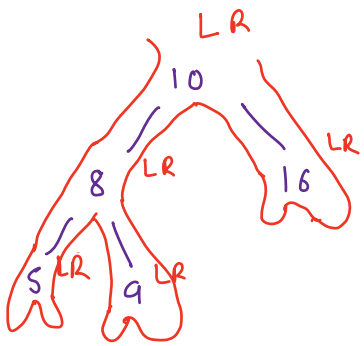


false

Idea 1 : Fill the inorder of binary tree using inorder traversal, now check if this list is sorted or not.

```
AL<Integer> list = new AL<>();

void travel(Node node) {
    if (node == null) {
        return;
    }
    travel(node.left);
    list.add(node.val);
    travel(node.right);
}
```



list : 5 8 9 10 16

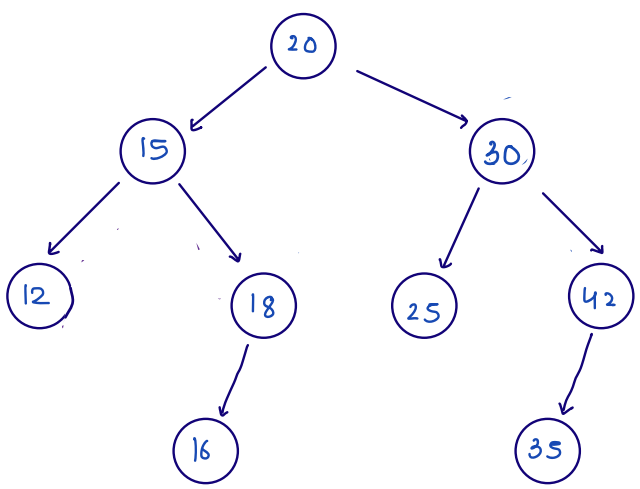
TC : O(n)

SC : O(h) + O(n) => O(n)

↓ ↓

travel() List

can you do it by taking by recursion space ?



prev =

In : L W R

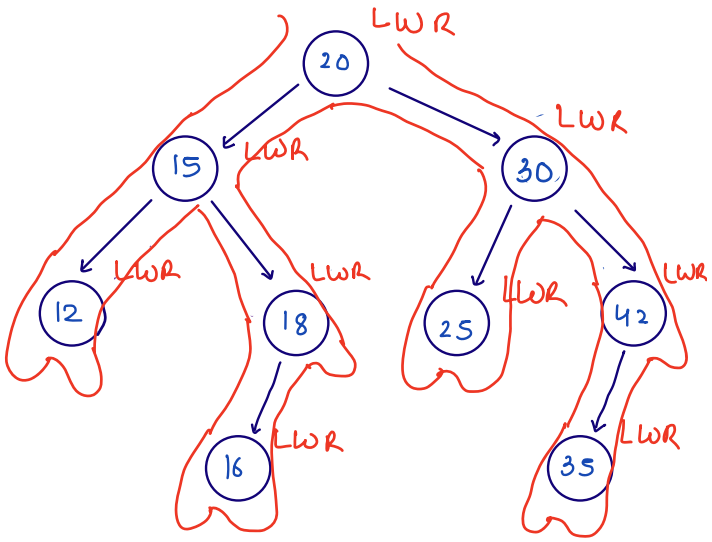
prev < node.val

↳ good

else ans = false

```
boolean ans = true;
int prev = -∞;

void travel (Node node) {
    if (node == null) {
        return;
    }
    travel (node.left);
    if (prev > node.val) {
        ans = false;
        return;
    }
    prev = node.val;
    travel (node.right);
}
```



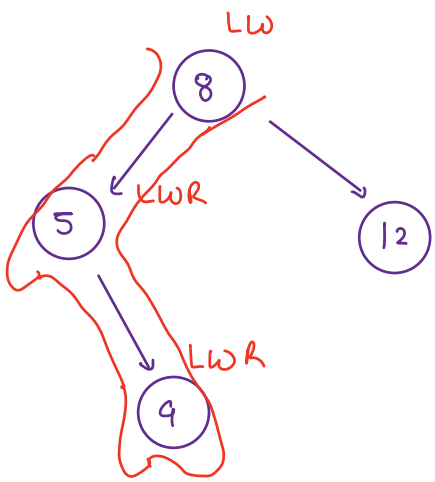
Inorder: 12 15 16 18 20 25 30 35 42

prev = ~~-∞ 12 15 16 18 20 25 30 35~~ 42

ans = true

```
boolean ans = true;
int prev = -∞;

void travel (Node node) {
    if (node == null) {
        return;
    }
    travel (node.left);
    if (prev > node.val) {
        ans = false;
        return;
    }
    prev = node.val;
    travel (node.right);
}
```



Inorder: 5 9 8 12

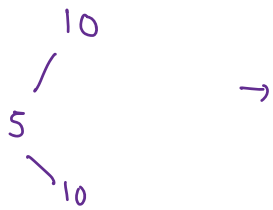
prev: ~~-∞ 5~~ 9

ans = ~~true~~ false

note:

```
boolean solve (Node node) {
    ans = true;
    prev = -∞;
    travel (node);
    return ans;
}
```

left should be strictly less than node.val and right should be strictly greater:

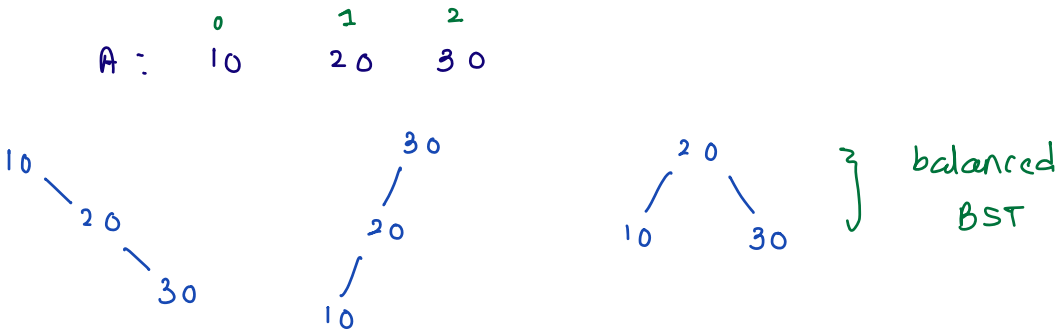


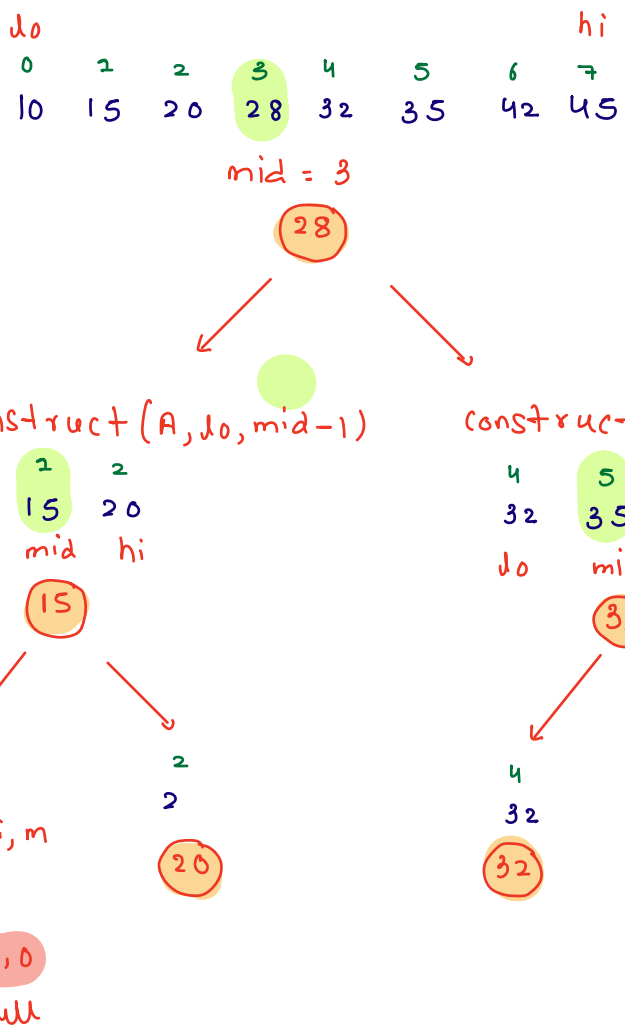
→ Inorder : 5 10 10

Condition

```
if (prev >= node.val) {
    ans = false;
    return;
}
```

Q.4 Given a sorted array, construct **balanced BST** using this array and return its root node.

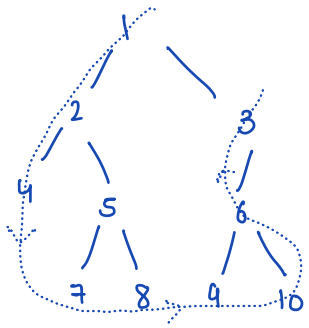




```
Node solve(int[] A) {
    return construct(A, 0, A.length-1);
}

Node construct(int[] A, int lo, int hi) {
    if(lo > hi) {
        return null;
    }
    int mid = (lo+hi)/2;
    Node nn = new Node(A[mid]);
    nn.left = construct(A, lo, mid-1);
    nn.right = construct(A, mid+1, hi);
    return nn;
}
```

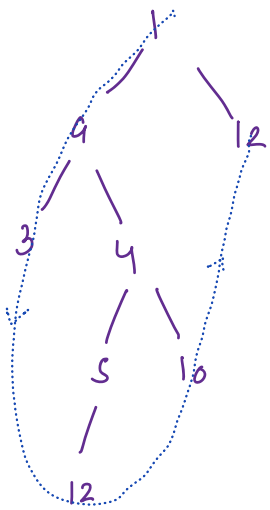
Doubts



left boundary : 1 2

leaf nodes : 4 7 8 9 10

right boundary: 3 6 rev → 6 3



lb : 1 9

leaf : 3 12 10 12

rb :

left boundary

```
void travel(Node node) {
    if(node == null) { return; }
    if(node is non leaf) { lb.add(node.val); }
    if(node.left != null) {
        travel(node.left);
    }
    else {
        travel(node.right);
    }
}
```