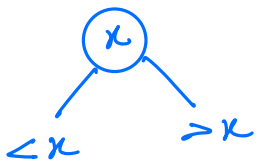
Binary Search Trees (B.S.T)

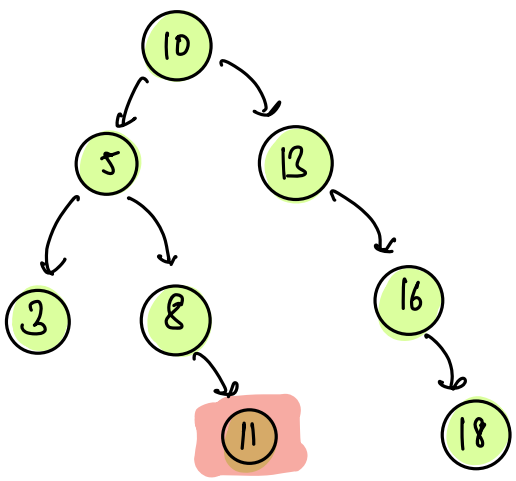
⇓

Binary Tree

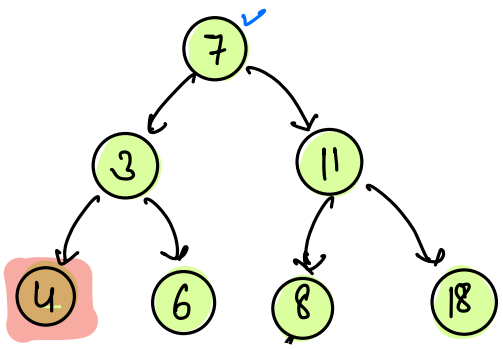
∀nodes

[all values in the l.s.t < node.data < all values in the r.s.t]

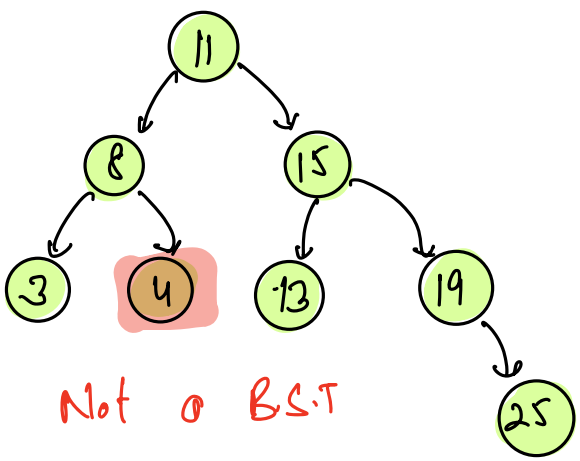




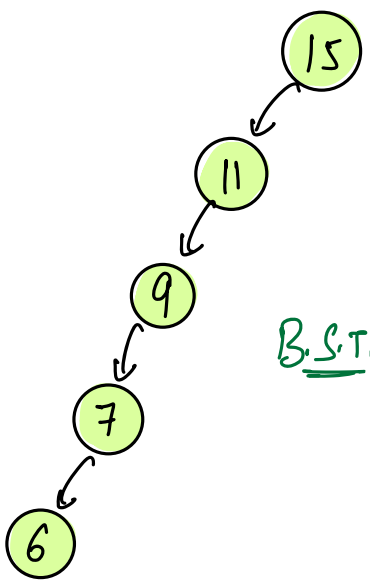
Not a B.S.T



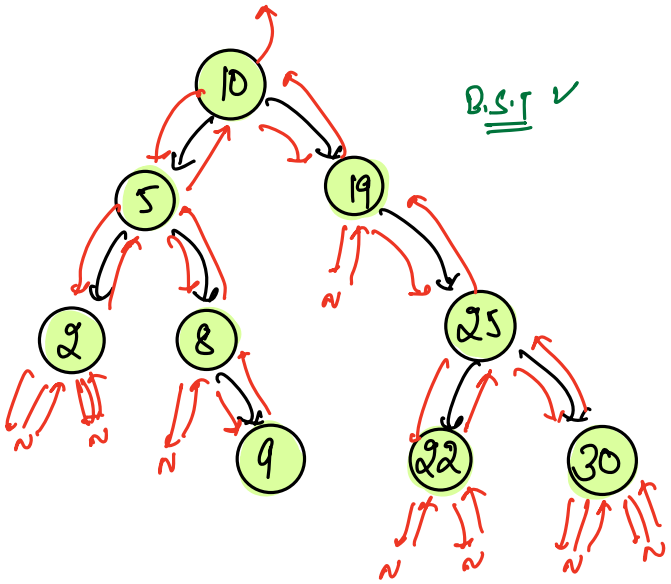
Not a B.S.T



Not a BS.T



B.S.T. ✓



B.S.T ✓

inorder → 2, 5, 8, 9, 10, 19, 22, 25, 30

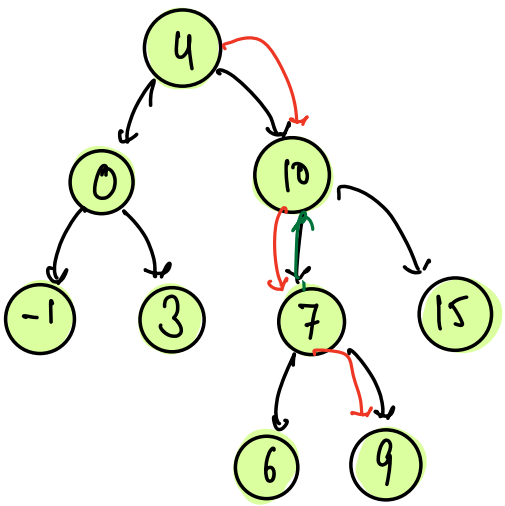
L.S.T root R.S.T

∴ [in-order of B.S.T ⇒ sorted array]

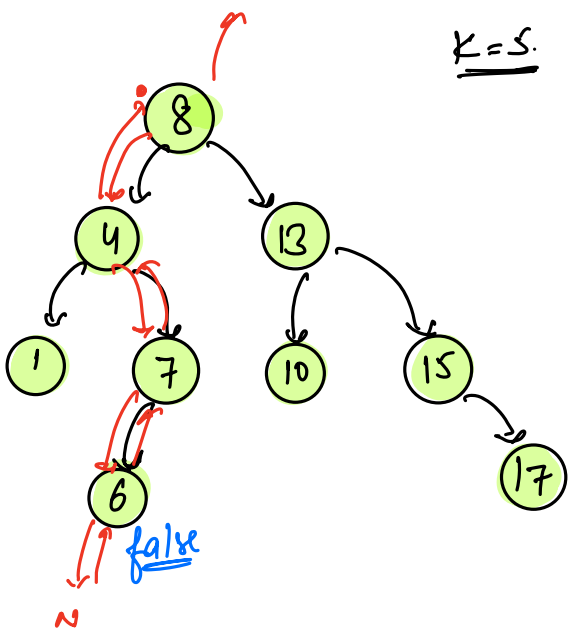
Q) Search an element K in Binary Search Tree.

B.F. idea. - Pre/ In/ Post [Just like Binary tree] T.C → O(N)

K=9.



K=5.



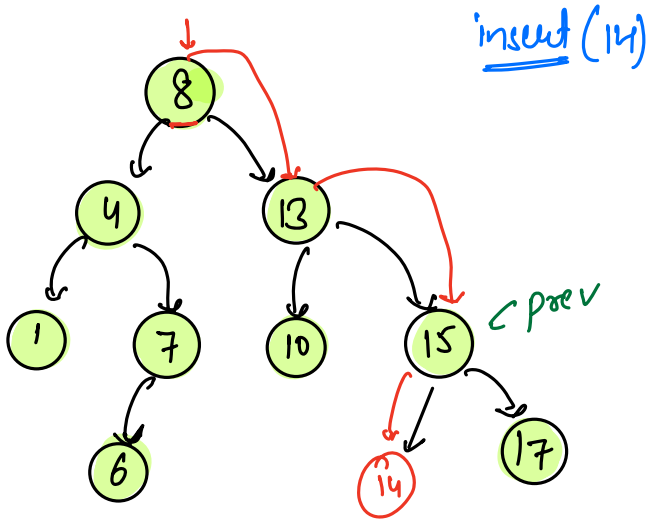
```
boolean search ( Node root, int k){
    Node temp = root;
    while (temp != null){
        if (temp.val == k){
            return true;
        }
        else if ( temp.val < k){
            temp = temp.right;
        }
        else{
            temp = temp.left;
        }
    }
    return false;
}
```

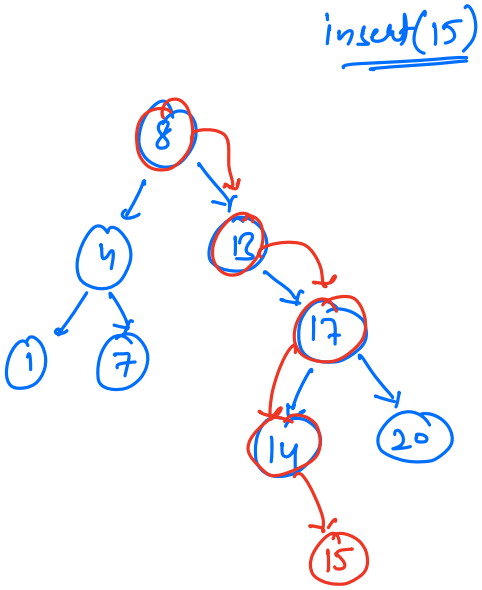
balanced.
$\log_2 N$

skewed
N

T.C → O(HT)
S.C → O(1)

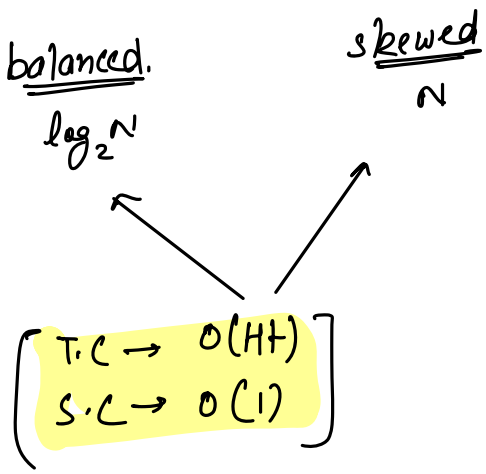
Q: Insert an element x in B.S.T.



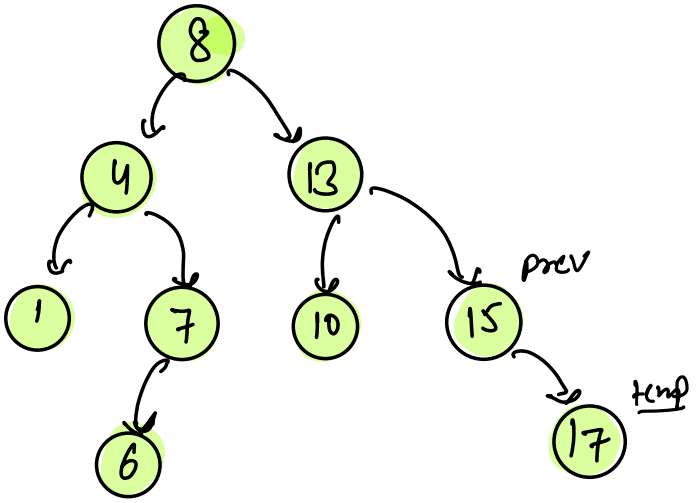


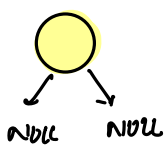
#code.-

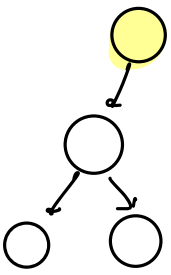
```
Node temp = root , prev = NULL;

while( temp != NULL){
        prev = temp;
    if ( temp.val == x) {
        return;
    }
    else if ( temp.val < x){
        temp = temp.right;
    }
    else{
        temp = temp.left;
    }
}

if(prev == NULL){  root = new Node(x) , return }
if ( x < prev.val) { prev.left  = new Node(x) }
   else            { prev.right = new Node(x) }
```

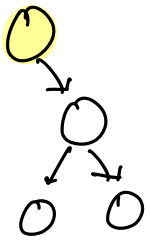


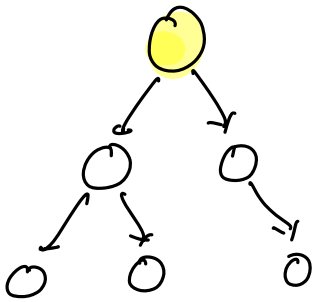
Deletion of a Node from BST.











→ search(x) by keeping track of prev node.

① `if (temp.left == NULL && temp.right == NULL){`

```
    // temp is a leaf node
    ____________
    ____________
}
```

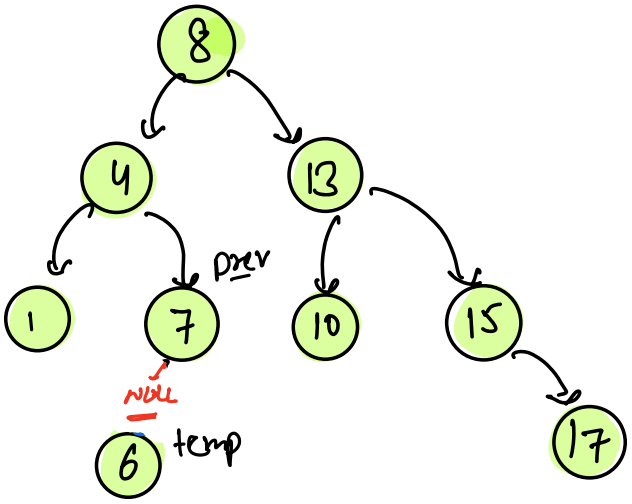
`else if ( temp.left == NULL || temp.right == NULL){`

②
```
    // temp with single child
    ____________
    ____________
}
```

`else{`

③
```
    // temp with both the children
    ____________
    ____________
}
```

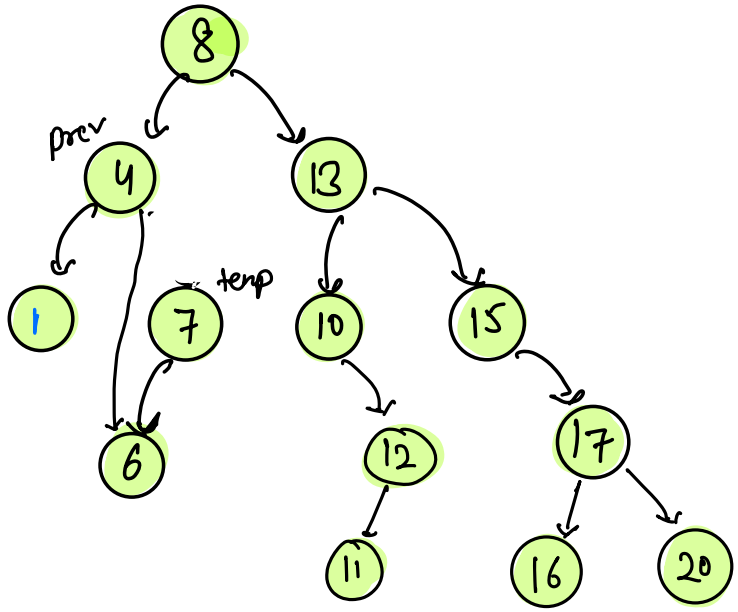
Case-1 When temp is leaf node



remove(6)

```
if( prev.left == temp){
    prev.left = NULL;
}
else{
    prev.right = NULL;
}
```

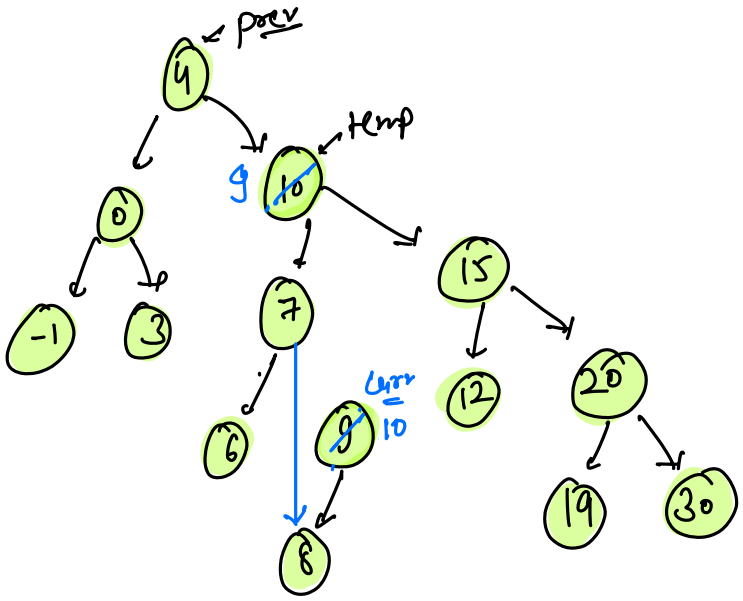
Case-2.: temp with single child



remove(7)

```
if( temp.left != NULL ){
        if( prev.left == temp){
            prev.left = temp.left;
        }
        else{
            prev.right = temp.left;
        }
}
else if( temp.right != NULL ){
        if( prev.left == temp){
            prev.left = temp.right;
        }
        else{
            prev.right = temp.right;
        }
}
```

Case-1. temp with both the children



remove(10)

[All nodes in lst < node.data < nodes in r.s.t]

||

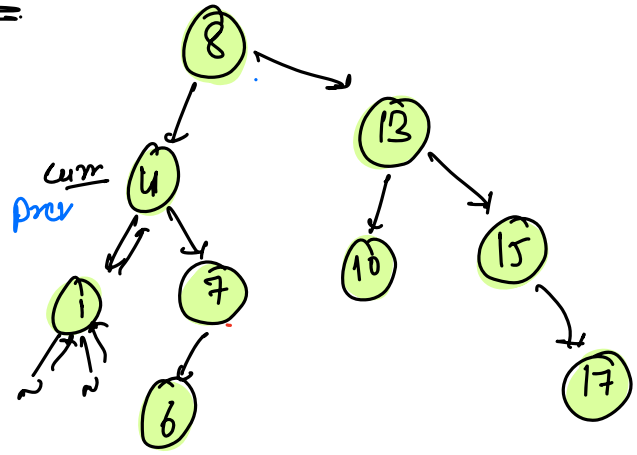
[Max in l.s.t < node.data < Min in r.s.t]

→ Max of l.s.t.

```
Node curr = temp.left;
if (curr.right != NULL){
    curr = curr.right;
}
swap(temp.value with curr.value);
//Make a recursive call to remove 10 from temp.left
remove(temp.left, K);
```

[T.C → O(Ht)
S.C → O(1)]

Validate BST.



[1, 4, 6, 7, 8, 10, 13, 15, 17]

idea-1 Do inorder traversal → Array is sorted → Yes.

[T.C → O(N)
S.C → O(N)]

↓

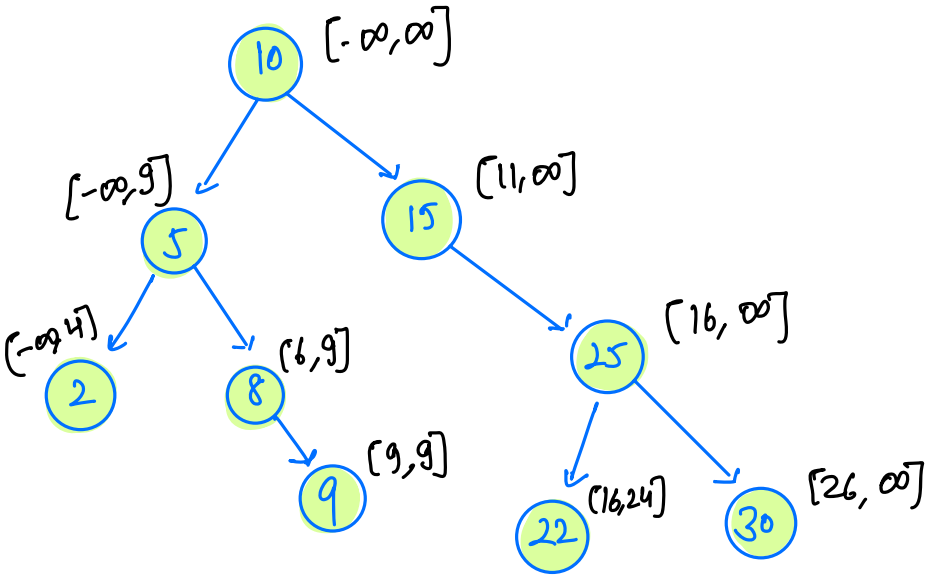
[while doing traversal
prev, curr]

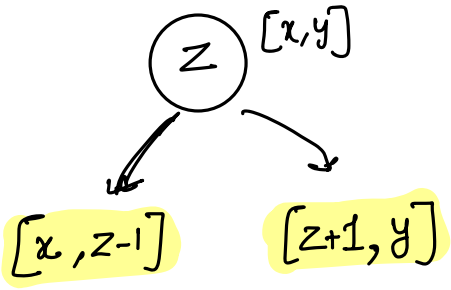
Node prev = NULL, boolean isBST = true;

```
void traversal( Node curr){
    if(curr == NULL) { return}

    traversal(curr.left);

    if(prev != NULL && prev.val ≥ curr.val){
        isBst = false;
    }

    prev = curr;

    traversal (curr.right);
}
```

[T.C → O(N)
S.C → O(ht.)]

② Using pre-order →



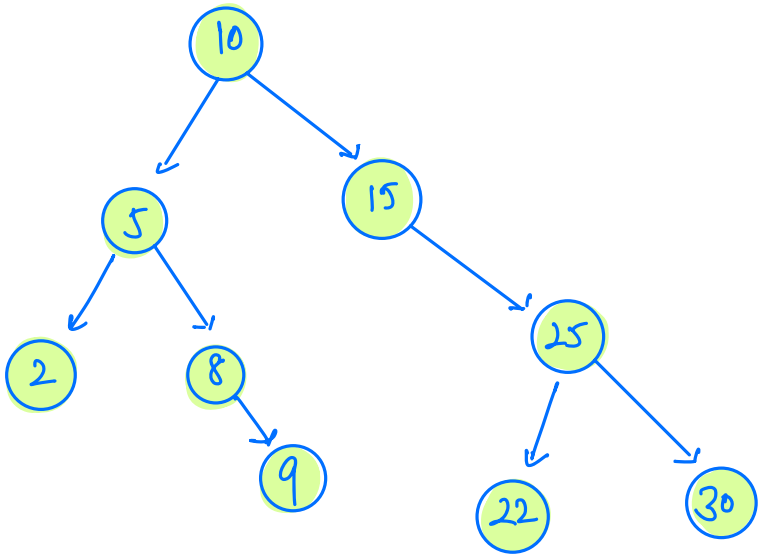


#code →

LONG.MIN.VALUE LONG.MAX.VALUE.

```
boolean isBst( root, l, r){
    if(root == NULL) { return true }
    if( root.val ≥ l && root.val ≤ r){
        boolean f1 = isBst(root.left, l, root.val - 1);
        boolean f2 = isBst(root.right, root.val +1, r);
        return (f1 && f2);
    }
    return false;
}
```

T.C → O(N)
S.C → O(Ht.)



Context → [L.L, Stack, Queue, Trees]