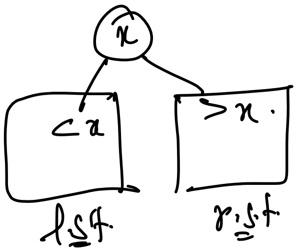
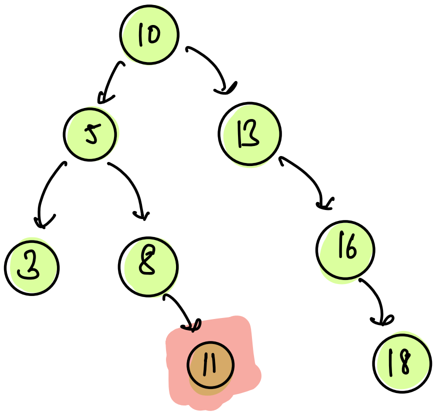
Binary Search Trees (B.S.T)

⇓

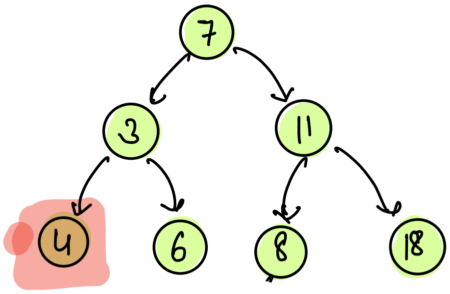
Binary tree.

∀ node. [all nodes in l.s.t < node.data < all nodes in r.s.t]

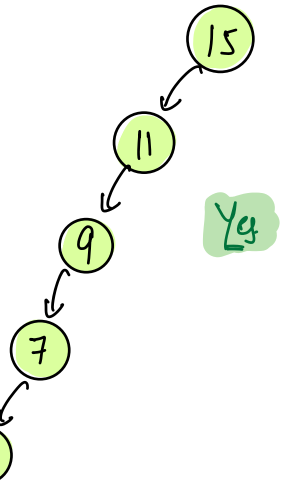




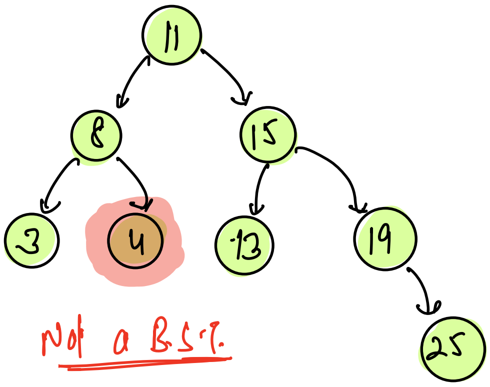
Not a B.S.T



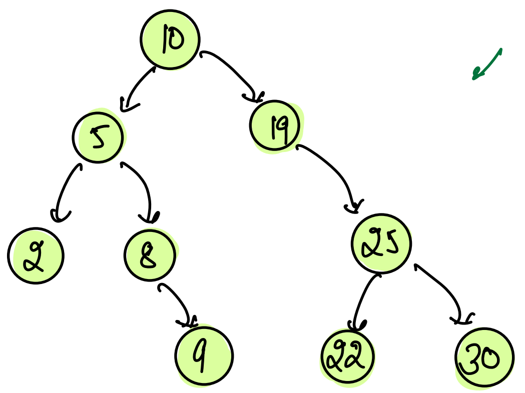
Not a B.S.T.



Yes



Not a B.S.T.



✓ This is a B.S.T.

In.order.

2,5,8,9,10,19,22,25,30.

A in.order traversal of B.S.T → sorted.

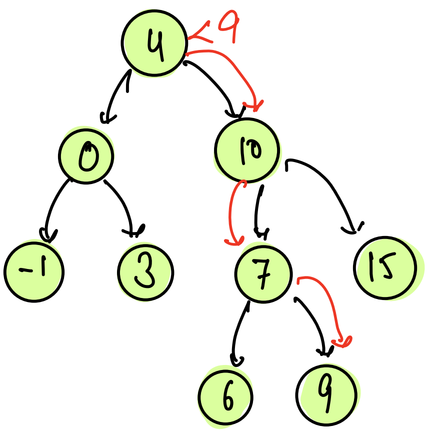
L.S.T < Root < R.S.T.

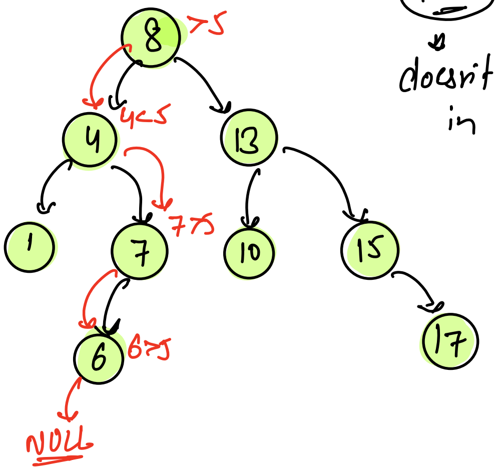
Q: Search an element K in Binary Search Tree.

Pre/ Post/ In

T.C → O(N)   S.C → O(H).

K=9.





K=5

doesn't exist in tree.

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

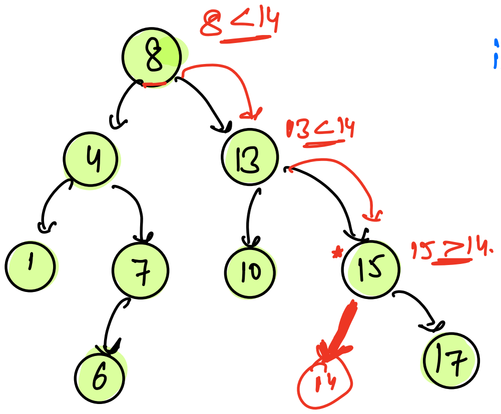
$H = \log_2 N$ balanced

$H = N$ skewed

T.C → $O(H)$

S.C → $O(1)$

Q1 Insert an element x in B.S.T.



insert(14) :

```
Node temp = root, Node parent = NULL;
while( temp != NULL) {
        parent = temp;
      if (temp.data == k){
            return;
      } else if (temp.data < k){
            temp= temp.right
      }else{
            temp= temp.left.
      }
}

if( parent == NULL){ return new Node(x) };

  if ( K < parent.data){
          parent.left = new Node(K);
  }else{
          parent.right = new Node(K);
  }
```

T.C → O(H)
S.C → O(1)

Q3) Check if the given B.T is B.S.T.

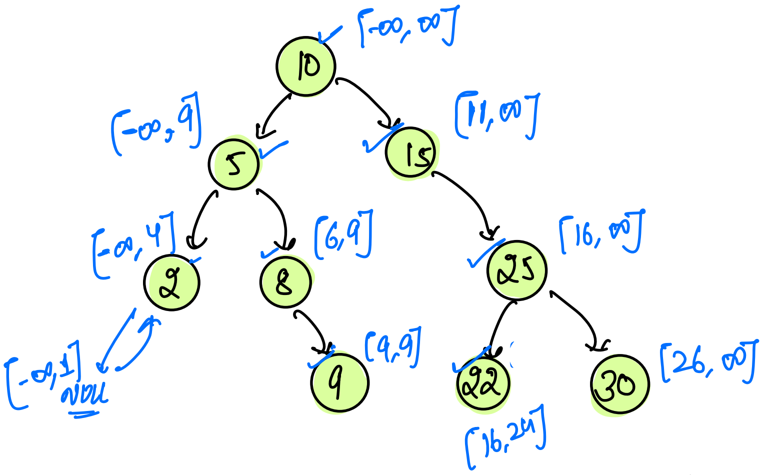
A.1. → Do inorder traversal → sorted → Yes.

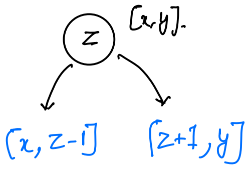
T.C → O(N) , S.C → O(H)

⇓

[while doing the traversal
curr, prev.]

A.2. Using Pre-order





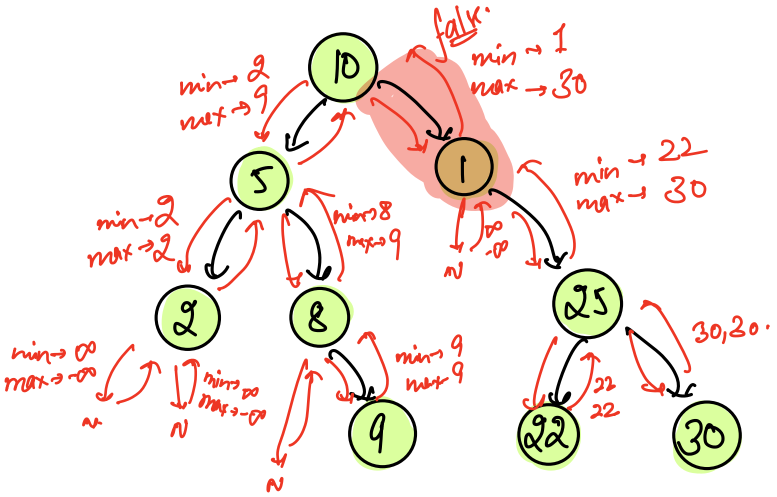
INT-MIN INT-MAX

```
boolean isBST( Node root, int l , int r){
            ↑              ↑
         INT-MIN        INT-MAX
    if(root == NULL){ return true }
    if ( root.data > l && root.data < r){
        boolean x = isBST(root.left , l , root.data-1);
        boolean y = isBST(root.right, root.data+1 , r );
        return x && y ;
    }
    return false ;
}
```

A.3. Using post-order.

L.S.T < node.data < R.S.T.

{ max of L.S.T. < node.data < min of R.S.T. }



min → infinity ⇒ no minimum element.

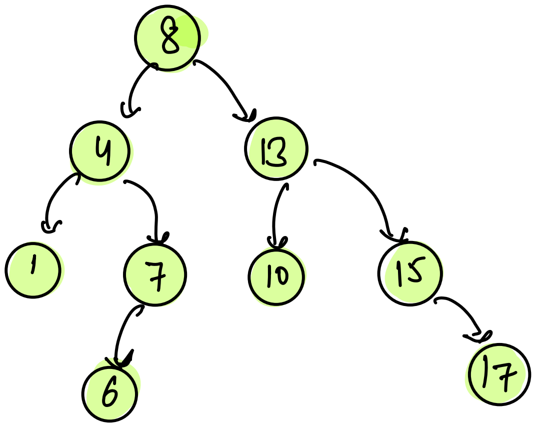
max → - infinity ⇒ no maximum element.

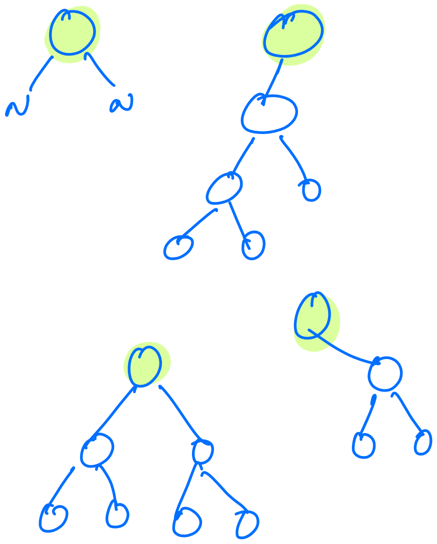
```
Triplet isBst ( Node root ) {
  if(root == NULL){ return new Triplet( true, +∞, -∞);

    Triplet left = isBst (root.left);
    Triplet right = isBst ( root.right);
     Triplet mt = new Triplet( );
    if( left.bst && right.bst && left.max < node.data &&
                                         node.data < right.min){
    {          mt .bst = true;
    }
     mt.min = Min( left.min, right.min, node.data);
     mt. max = Max( left.max, right.max , node.data);
     return mt;
}
```

```
Triplet {
   boolean bst;
    int min;
   int max
}
```

{ T.C → O(N)
S.C → O(H) }

Q4) Deletion of a Node from B.S.T.



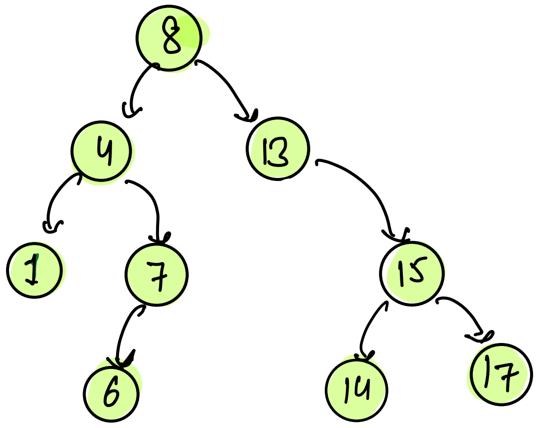


Case.1. When Node is leaf node.

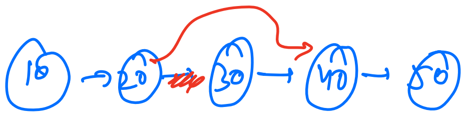
remove(6)

```
parent, curr
  //search curr node & keep a track of
     parent node.

if (parent.left == curr) {
        parent.left = null
}
else{
        parent.right = null
}
```

Case-2. When node Is having a single child



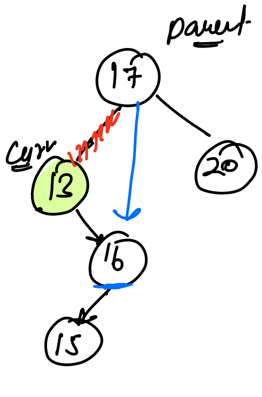
remove(13)



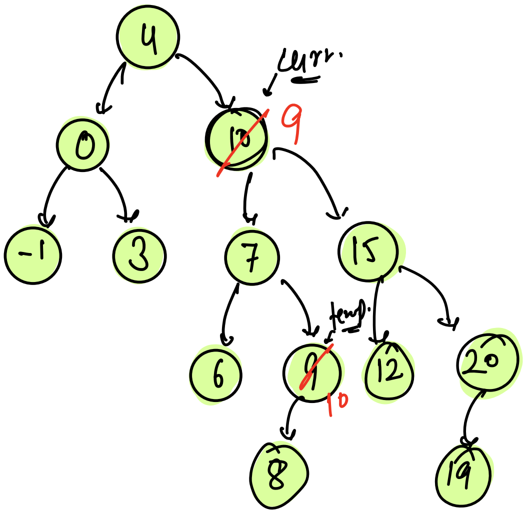
parent, curr

//search curr node & keep a track of parent node.

```
if ( curr.left == NULL && curr.right != NULL) {
    if ( curr == parent.left) {
        parent.left = curr.right;
    }
    else {
        parent.right = curr.right;
    }
}
else if ( curr.left != NULL && curr.right == NULL) {
    if (curr == parent.left) {
        parent.left = curr.left
    }{
        parent.right = curr.left
    }
}
```



Case-3. Node with both the children.



remove(10);

$\left[\underline{\text{max}}\ \underline{\text{l.s.t}} < \text{node.data} < \underline{\text{min}}\ \underline{\text{r.s.t}} \right]$

① search the node to be removed.

② find max of l.s.t → curr.left.

```
temp = curr.left
while( temp.right != NULL){
            temp = temp.right
}
```

③ swap(curr.data, temp.data)

④ make a recursive to remove 10.

remove(curr.left , 10);

try to code. (#todo)

```
remove( root, x ) {
search the element in B.S.T.
  if (node is leaf node) {
①
  }
  else if (node is having a single child) {
② {
  }
  else {
③
  }
}
```

structure.

(dry-run / tracing) is.