* TC of Search operation in Binary Tree : O(N)
  {wc}

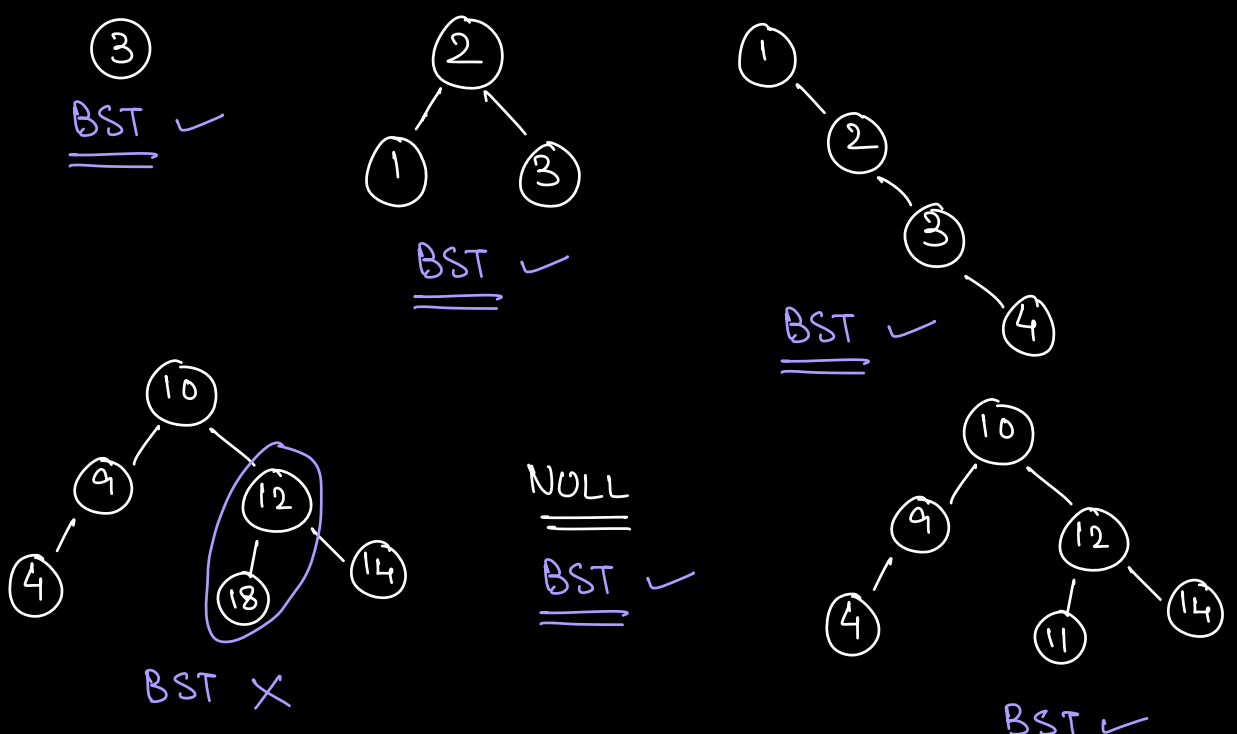
* TC of Search operation in Array : O(N)

* TC of Search operation in Sorted Array : O(log N)
  (Binary Search)

* If we arrange the elements in some specific order in Binary Tree the search TC can be optimised.

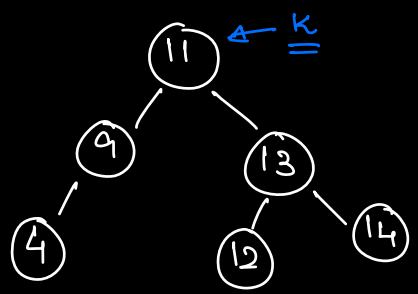
* Rule

  Value ≤ Root ⇒ LST
  Value > Root ⇒ RST
  } Binary Search Tree (BST)

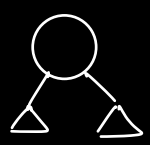


# Given a BST, Insert a value maintaining the BST Property.

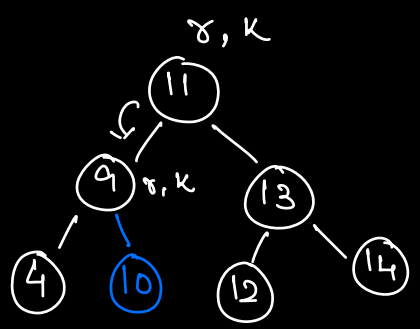
(NO duplicates)




K = 10



```
TreeNode insert (root, K) {
    if (root == Null) {
        return new TreeNode(K);
    }
// Assumption: insert(root, K) fun^ inserts the
   new value K at its correct pos^ & returns
   the root node.
    if (root.val > K)
        root.left = insert(root.left, K);
    else
        root.right = insert(root.right, K);
    return root;
}
```




K = 10

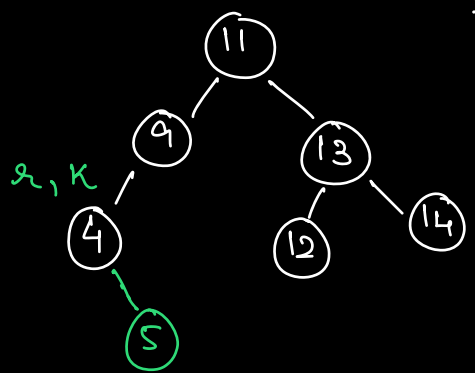
r

9 . right = insert ( NULL , K )

Base Case

10

K = 5

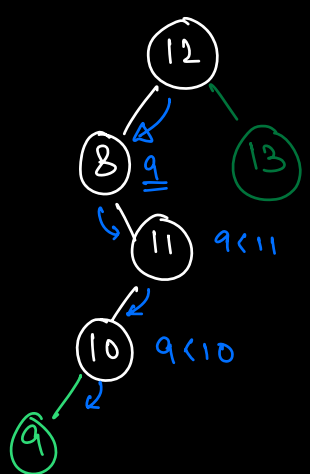



11 . left = insert ( 9 , 5 )

9 . left = insert ( 4 , 5 )

4 . right = insert ( null , 5 )

5




K = 9

10 . left = insert ( NULL , 9 )

9

K = 13

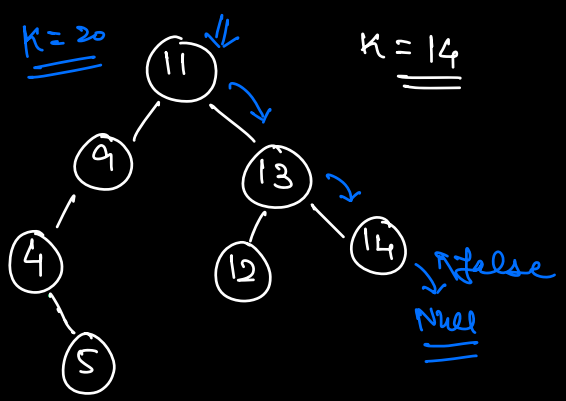
12 . right = insert ( null , 13 )

= 13

TC : O(N)
SC : O(N)

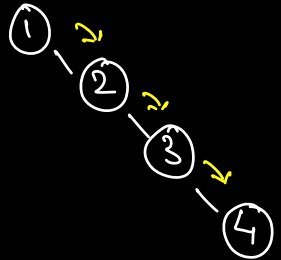
} W.C (Skewed Tree)

Q: Given a BST, search a value K in it.




```
bool search(root, K) {
    if(root == NULL)
        return false;
    if(root.data == K)
        return true;
    if(root.data > K)
        return search(root.left, K);
    return search(root.right, K);
}
```

TC: O(N) {Worst Case}



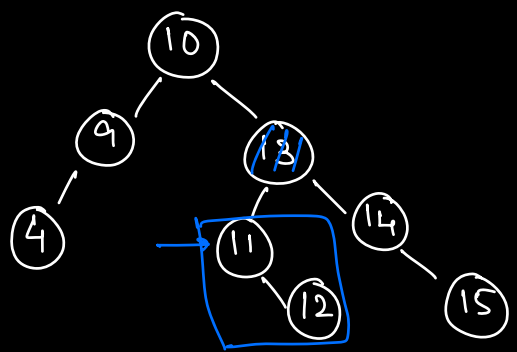

⇒ O(N)

SC: O(N) {Worst Case}

Q: Given a B.S.T, Delete a value (K) from it.

Amazon
MS.

[NO duplicates]




Case I :-

Node to be deleted is a leaf node.

⇒ Make it NULL.

Case II :-

Node to be deleted has (1) child :

⇒ Return the NON NULL child

Case III :

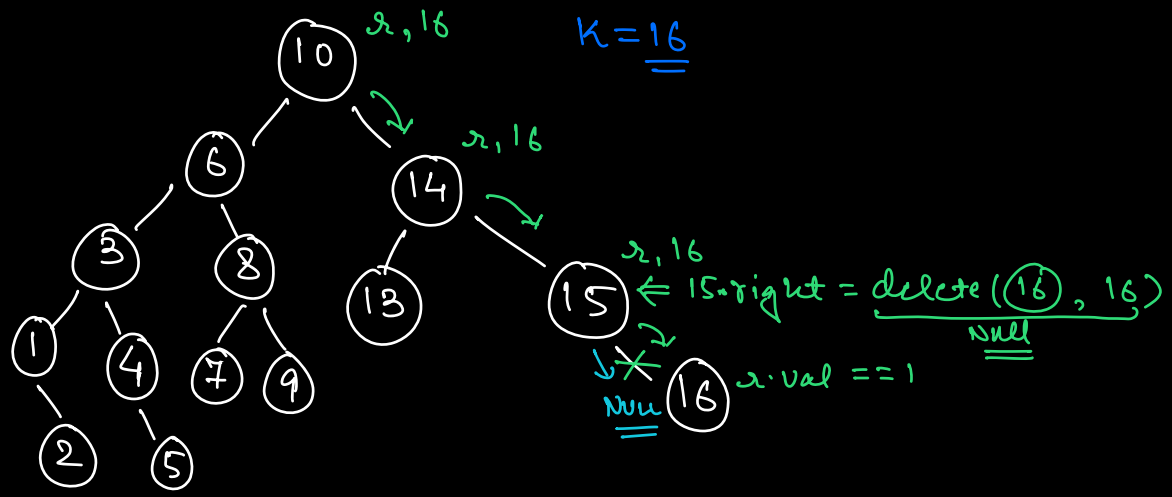
Node to be deleted has both children.

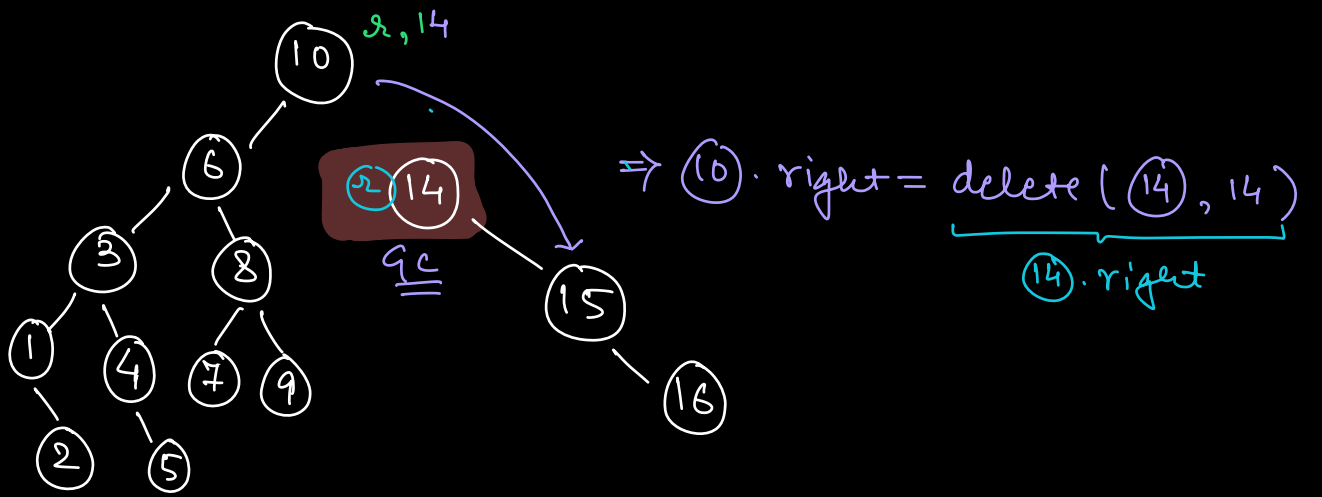
→ Replace the node with max of LST and delete max of LST OR min of RST and delete min of RST.

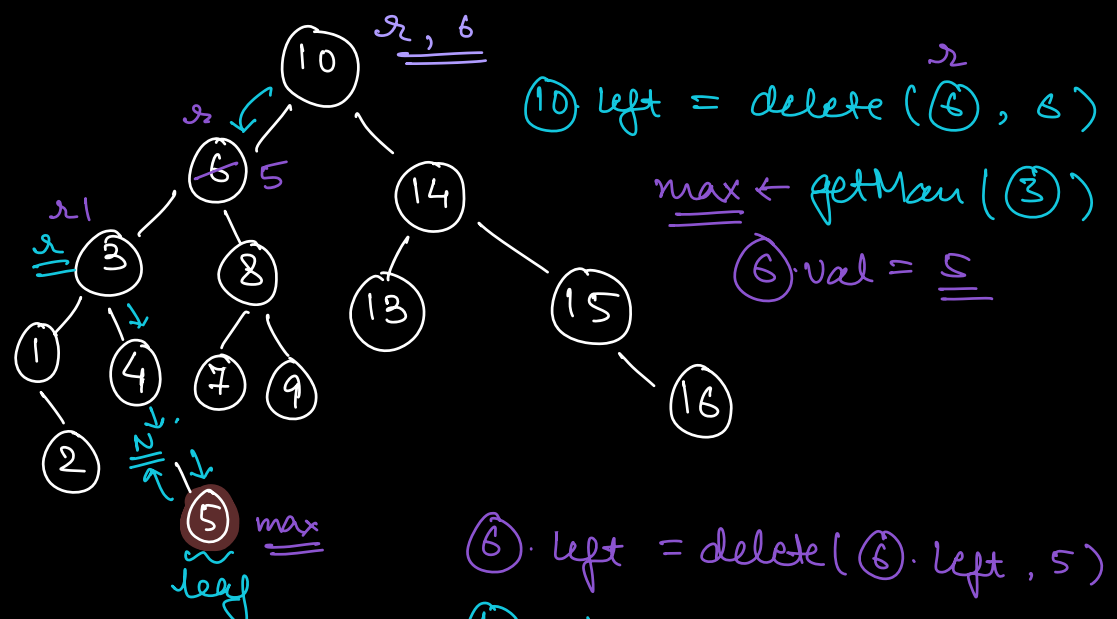
Max of LST
or
Min of RST

} Can have at max (1) child.

```
TreeNode deleteNode(root, k) {
    if (root == NULL)
        return NULL;
    if (root.val > k)
        root.left = deleteNode(root.left, k);
    else if (root.val < k)
        root.right = deleteNode(root.right, k);
    else { // root.val == k => 3 cases.
        // Case I: -> O(1)
        if( isLeaf(root) )
            return NULL;
        // Case II: Root has (1) child
        if (root.left == NULL)
            return root.right;
        if (root.right == NULL)
            return root.left;
        // Case III: Root has both children.
        max = getMax(root.left);
        root.val = max.val;
        root.left = deleteNode(root.left, max.val)
    }
    return root;
}
```


r, 16
K = 16
r, 16
r, 16
15.right = delete(16, 16)
null
r.val == 1
null


r, 14
⇒ 10.right = delete(14, 14)
14.right
r
14
qc

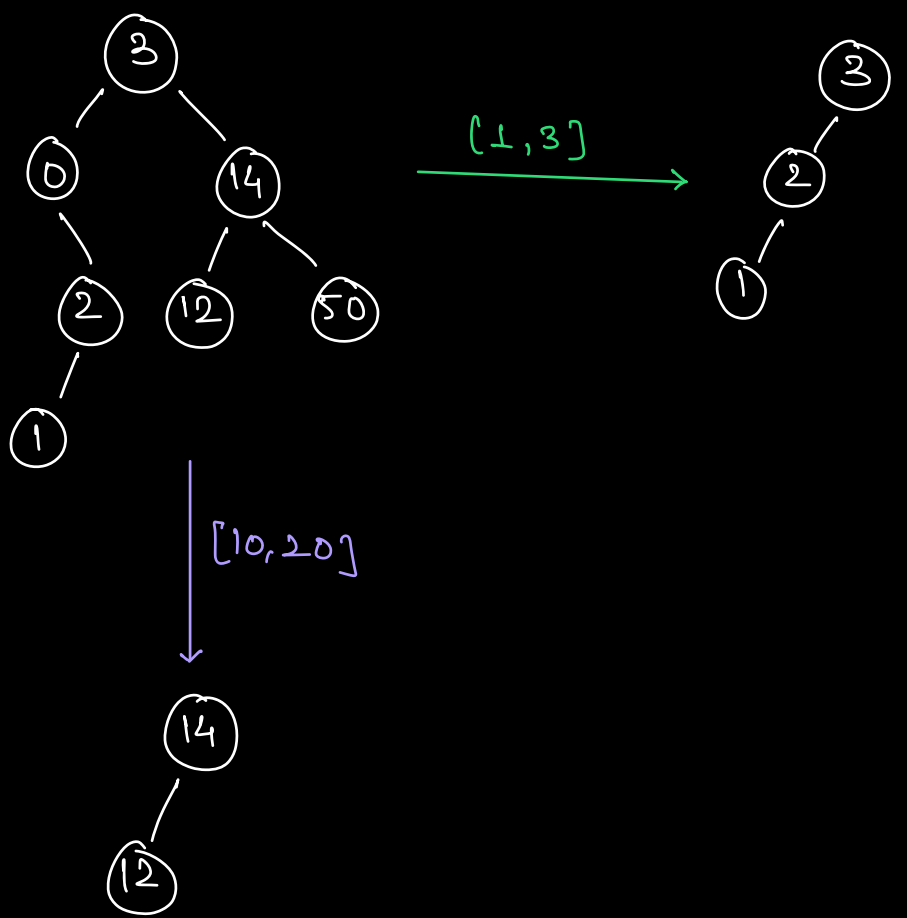

r, 6
r
r1
r
5
10.left = delete(6, 6)
max ← getMax(3)
6.val = 5
max
leaf
6.left = delete(6.left, 5)
4.right = NULL

TC : O(N)

SC : O(N)

⇒ Height Balanced BST } Next Class.

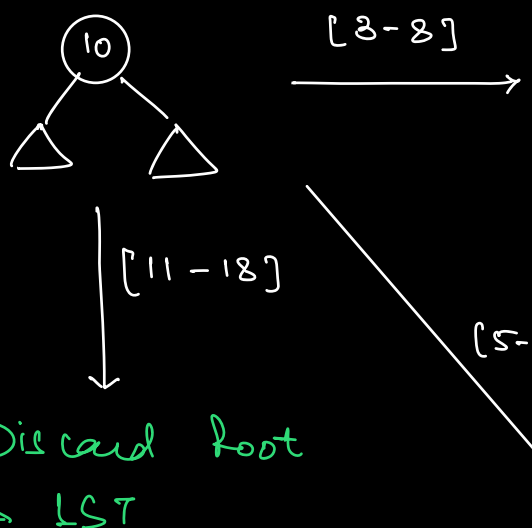
Q: Given a BST & a range l to h. Delete every node in BST outside this range. [l, h]



Approach 1 :

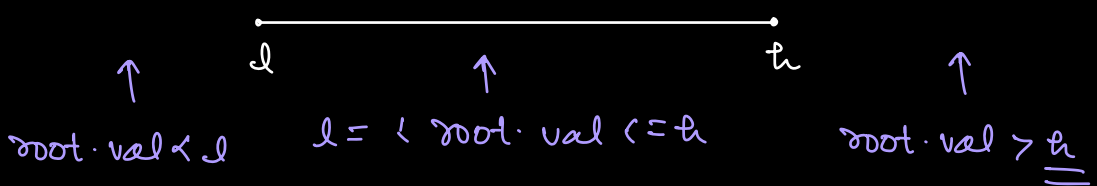
Iterate over all the Nodes & call the deleteNode (root, K) fun^ if root's value is Outside the range [l, h]

TC: $O(N^2)$




Root will be deleted & entire RST will also be delete.

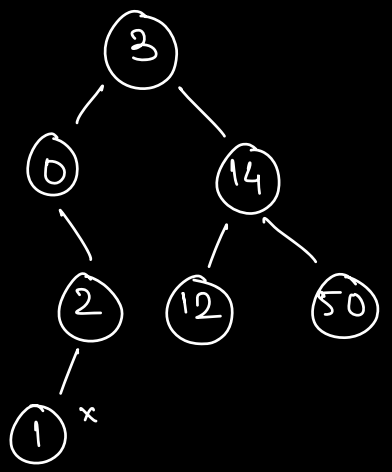
Discard Root & LST



```
TreeNode trimBST(root, l, h) {
    if (root == NULL)
        return NULL;
    if (root.val < l)
        return trimBST(root.right, l, h);
    if (root.val > h)
        return trimBST(root.left, l, h);
    root.left = trimBST(root.left, l, h);
    root.right = trimBST(root.right, l, h);
    return root;
}
```

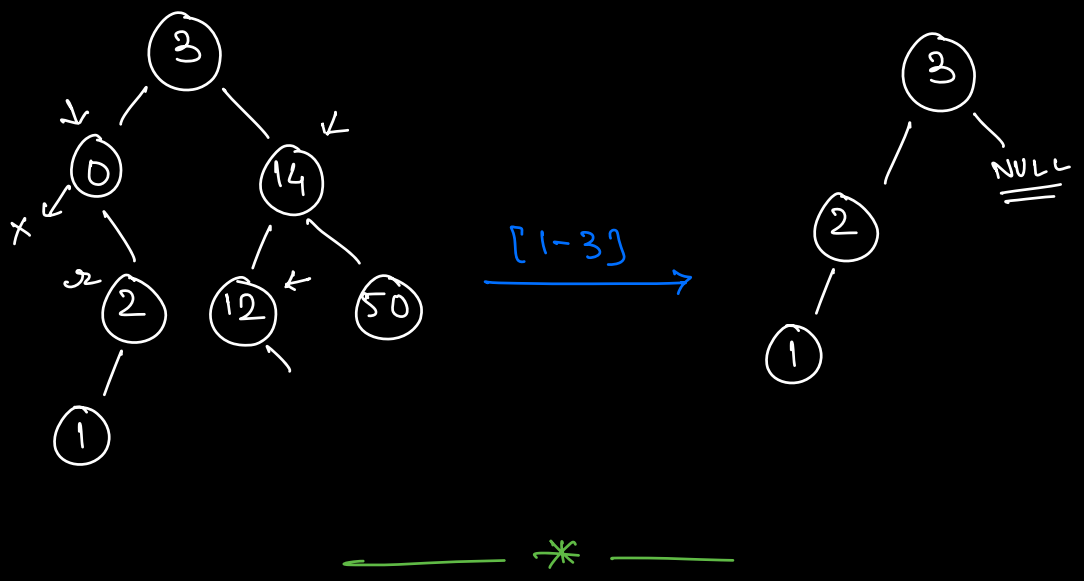
TC : O(N)
SC : O(N)

—— * ——




[3-8]

[1-3]


3
0
14
2
12
50
1
[1-3]
3
NULL
2
1