Today's content:

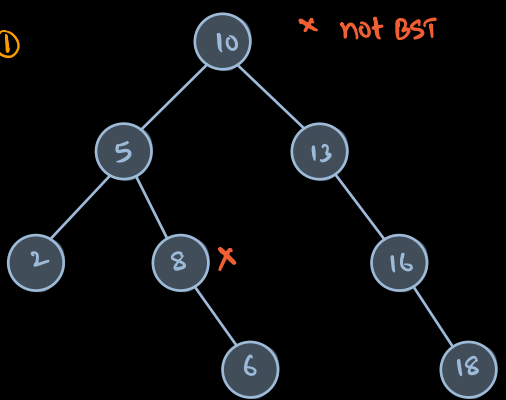
- → Basics of binary search trees
- → Insert / search
- → isBST()
- → Recover BST
- → Max in a BST.

Binary Search Tree (BST).

A Binary tree is said to be BST if

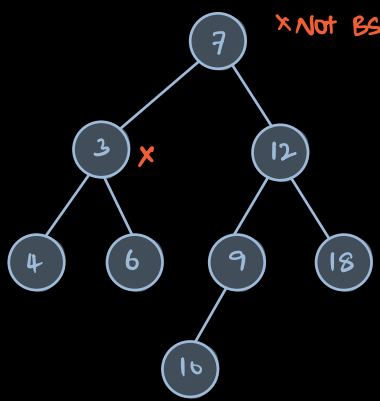
For all nodes { All nodes in LST < node.data < All nodes in RST }.

Ex: ①



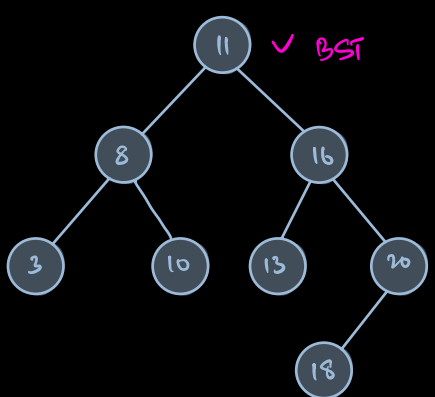
× not BST

②



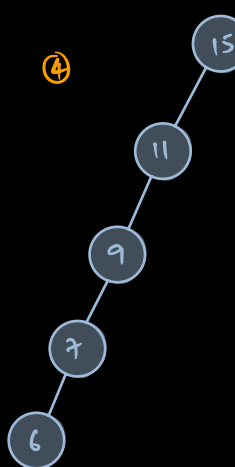
× Not BST

③



✓ BST

④

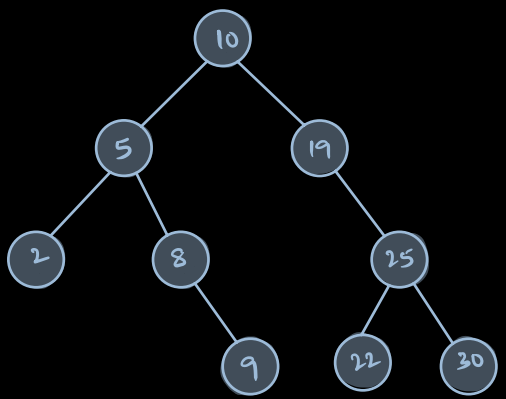


✓ BST

Note: A null node holds the property of BST.

In BST all values are distinct

BST property.



LDR

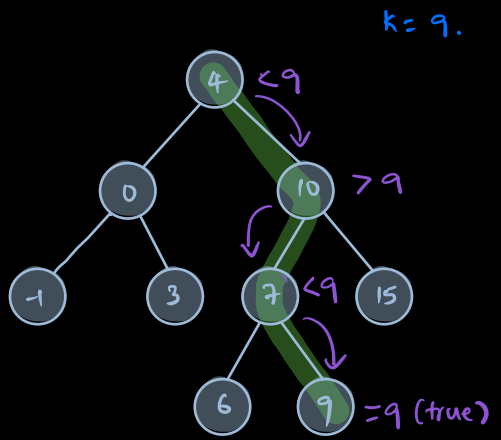
Inorder : { 2, 5, 8, 9, 10, 19, 22, 25, 30 }.

Sorted data in increasing order.

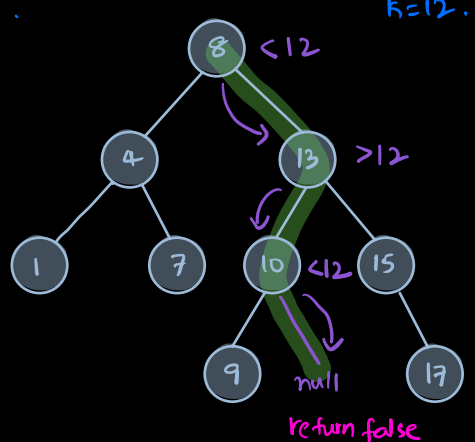
LST < D < RST

Search K in BST.

①



②.



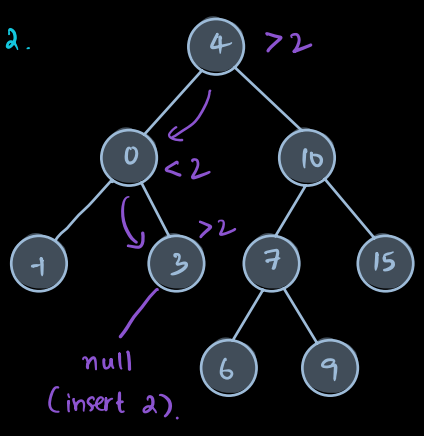

```
bool search (Node root, int k)
    While (root != null)
        if (root.data == k)  {return true}
        if (root.data > k)   { root = root.left }
        else  { root = root.right }
    return false.
```

TC : O(H)

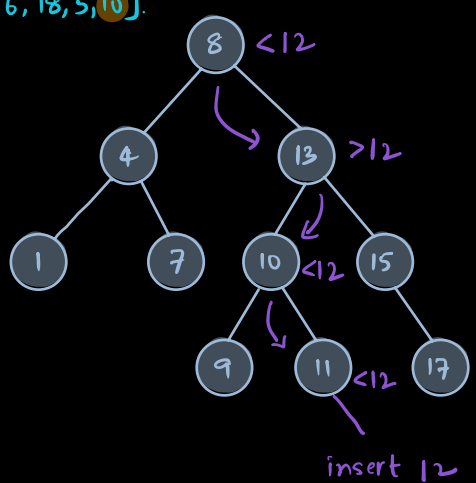
SC : O(1)

Insertion in BST.

k = 2.



k = [12, 6, 18, 5, 10].



Note:

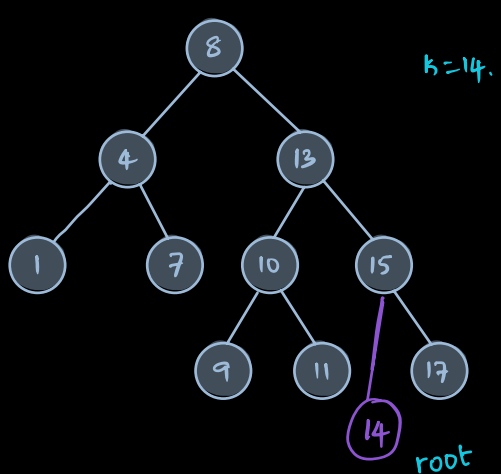
i) Try inserting the new node as a leaf node

ii) Search as if 'k' is present => This will lead to correct leaf node.

```
Node insert(Node root, int k)
{
    if (root == null) { return new Node(k) }
    Node temp = root.
    while(root != null)
        if (root.data > k) // left.
            if(root.left == null)
                root.left = new Node(k)
                break
            root = root.left
        else
            if(root.right == null)
                root.right = new Node(k)
                break
            root = root.right.
    return temp
}
```

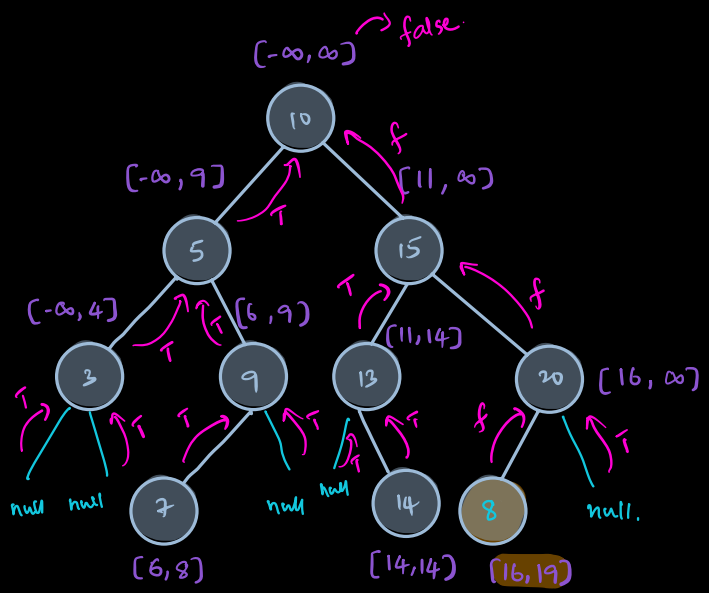
inserting new node.

Tree tracing



k = 14.

38: Check if given tree is BST or not.



idea: in-order -> sorted. (Check)

Recursively calling for left & right by checking root.data with only left & right data will not work. ✗.

Steps:

(i) root can be any value

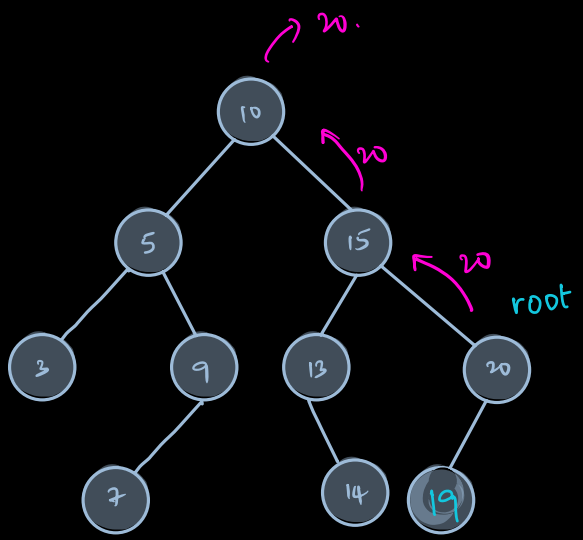
(ii) moving left
- -> update upper bound
- -> lower bound remains same

(iii) moving right
- -> update lower bound
- -> upper bound remains same.

```
boolean isBST (Node root, int s, int e)
    if(root==null)
    |   return true
    if ( root.data >=s && root.data <= e)
    |   // check if LST & RST are in range
    |   l= isBST(root.left , s, root.data-1)
    |   r= isBST (root.right, root.data+1, e)
    |   return l && r ;
    return false
```

TC : O(N),
SC: O(H).

Q: Max in BST: Given root node of a BST, return max of BST.



TODO: min of a BST.

48. Recover sorted arr[].

Given an arr[], which is formed by swapping two distinct index positions in a sorted arr[], get original sorted arr[].

Ex1:

	0	1	2	3	4	5	6	7
arr :	4	10	19	14	18	12	25	28
Sorted :	4	10	12	14	18	19	25	28

0	1	2	3	4	5	6	7
4	10	19	14	18	12	25	28

P_1 P_2

first wrong ele → 19. (take it from P_1)

second wrong ele → 12 (take it from P_2)

idea:

Iterate on arr[], Compare 2 adj elements,

(i) 1st time comp. fails → consider 1st ele as first, consider 2nd ele as second.

(ii) 2nd time comp. fails → consider 2nd ele as second

Swap first & second

Ex 2:

0	1	2	3	4	5	6	7	8
2	6	23	10	14	19	8	40	51

P_1 P_2

Ex 3:

0	1	2	3	4	5	6	7
3	6	10	15	12	17	20	33

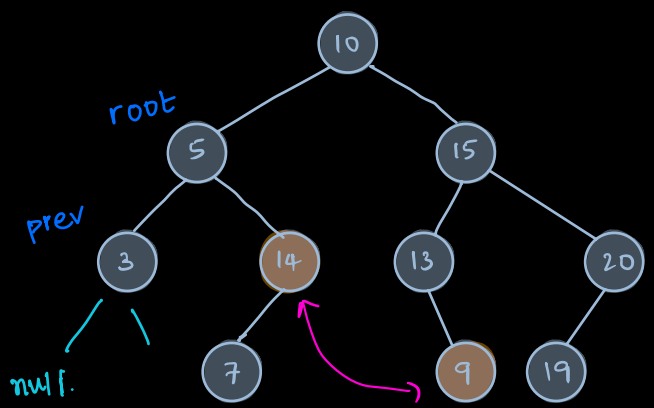
P_1 P_2

Ex:

0	1	2	3	4	5	6	7	8	9
3	6	9	31	14	18	24	12	35	40

P_1 P_2

Sol: Given a BST which is formed by swapping 2 distinct nodes, recover original BST.



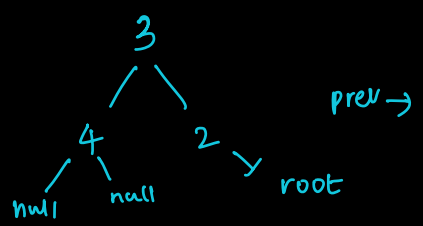
idea: Do inorder traversal, and find nodes which needs to be swapped.

Use above question to swap.

idea 2: Modify inorder traversal.

```
Node prev = null
Node first = null, second = null.
void inorder ( Node root)
    if (root == null) { return }
    inorder(root.left)
    if (prev != null && prev.data > root.data && first == null)
        first = prev, sec = root
    else if (prev != null && prev.data > root.data)
        sec = root
    prev = root.
    inorder(root.right)
// Swap content of first & second.
```

first = 4
second = 2



prev → P_1

root → P_2.

↓

Same as last question.

50:

Delete node.

(i) delete node where there's no child. ✓

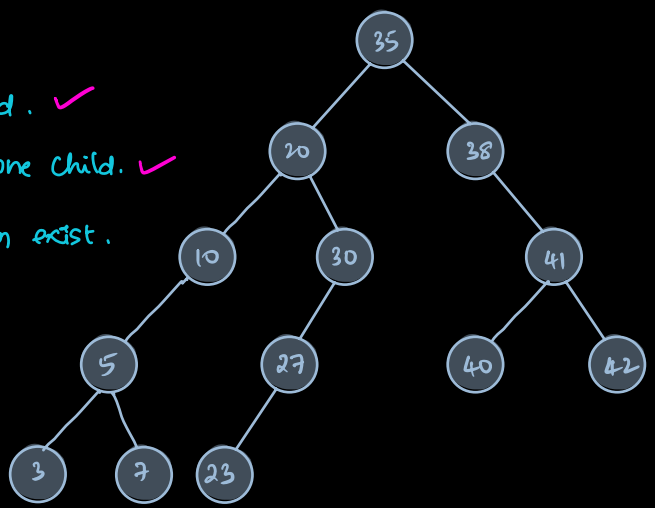
(ii) delete node when there's only one child. ✓

(iii) delete node when both children exist.

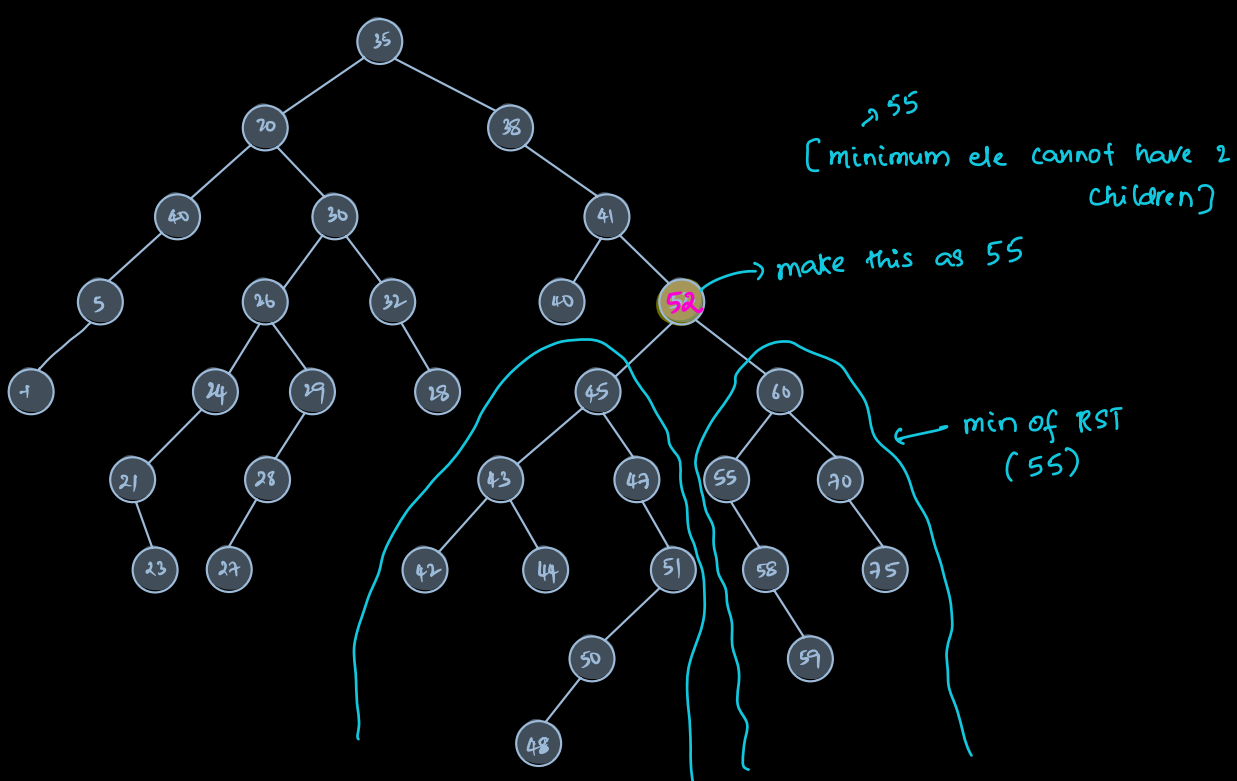
(i) find min on RST

(ii) make node.data = min

(iii) delete min from RST
(same as step 2, since it can only have 1/0 child).



for 2 child.



→ 55
[minimum ele cannot have 2 children]

→ make this as 55

← min of RST (55)

```
Node delete ( Node root, int k).
{
    if (root.data == k)
    {
        // case -1 . Leaf node.
        if (root.left == null && root.right == null)
        {
            return null
        }
        // case - 2 , only one child.
        if (root.left == null)
            return root.right
        if (root.right == null)
            return root.left.

        // case -3 , both children are there.
        int max = findMax(root.left).   ---> needs to be implemented.
        root.data = max.
        root.left = delete ( root.left , max)
        return root
    }
    if (root.data > k)
    {
        root.left = delete(root.left, k)
    }
    else {
        root.right = delete (root.right , k)
    }
    return root
}
```