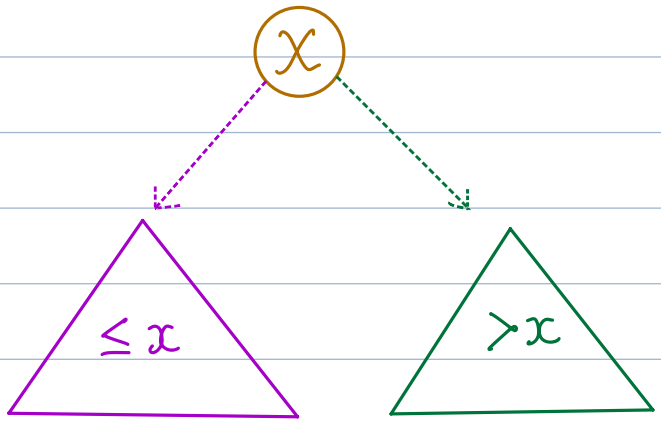
Trees – 3

Content

- BST Introduction
- Searching in BST.
- Insertion in BST.
- Deletion in BST.
- Construct balanced BST from sorted array
- Check if the given binary tree is a BST.

Binary Search Tree

Searching data in organised dataset using divide and conquer



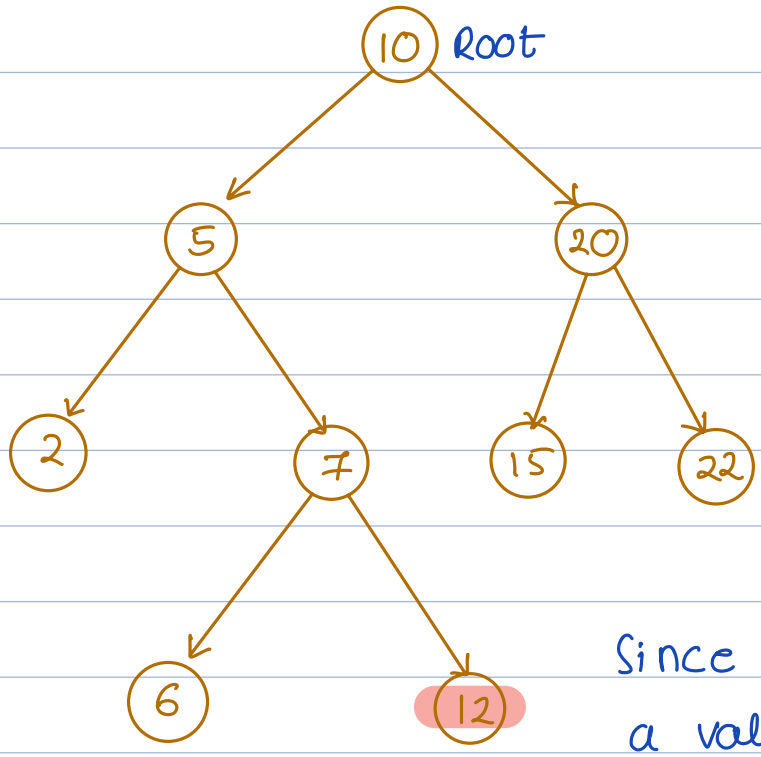
∀ nodes x

→ all the data in the left subtree <= x

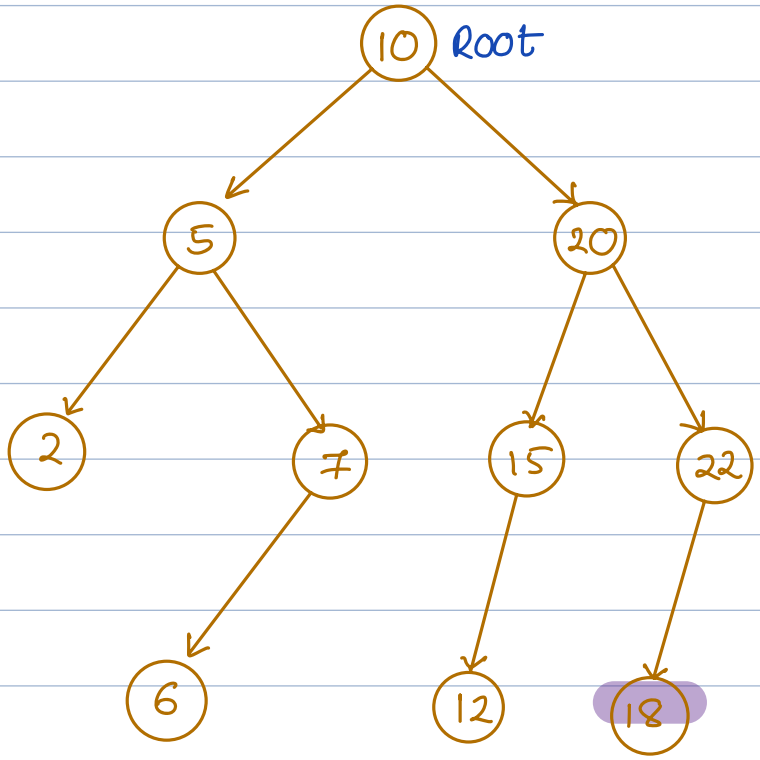
equality on the left side for this lecture

↓

all the data in the right subtree > x

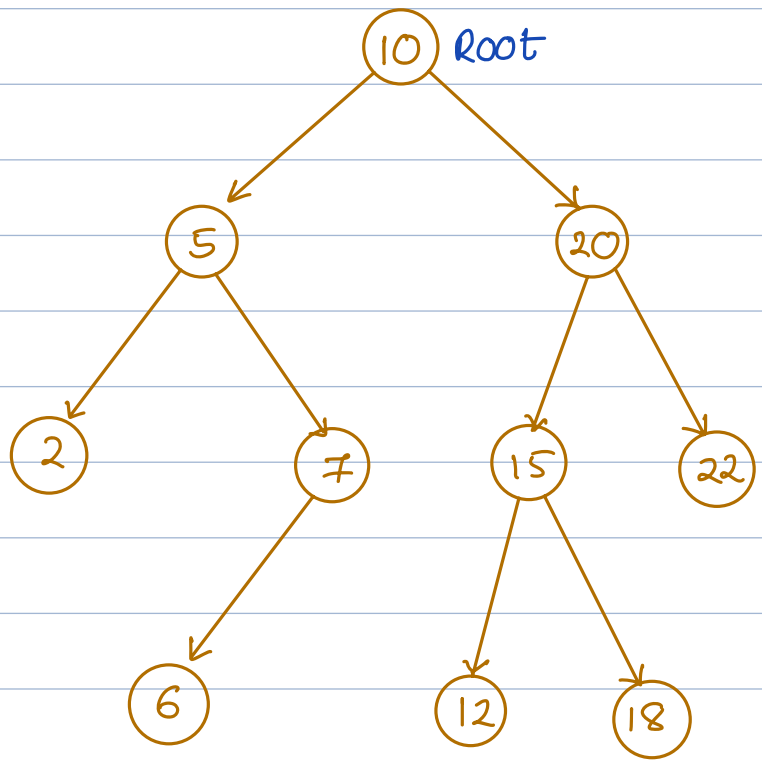


Since 12 > 10, this is not a valid BST.

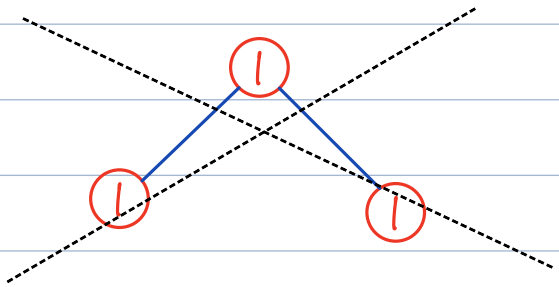


Since 18 < 20

This is not a valid BST.

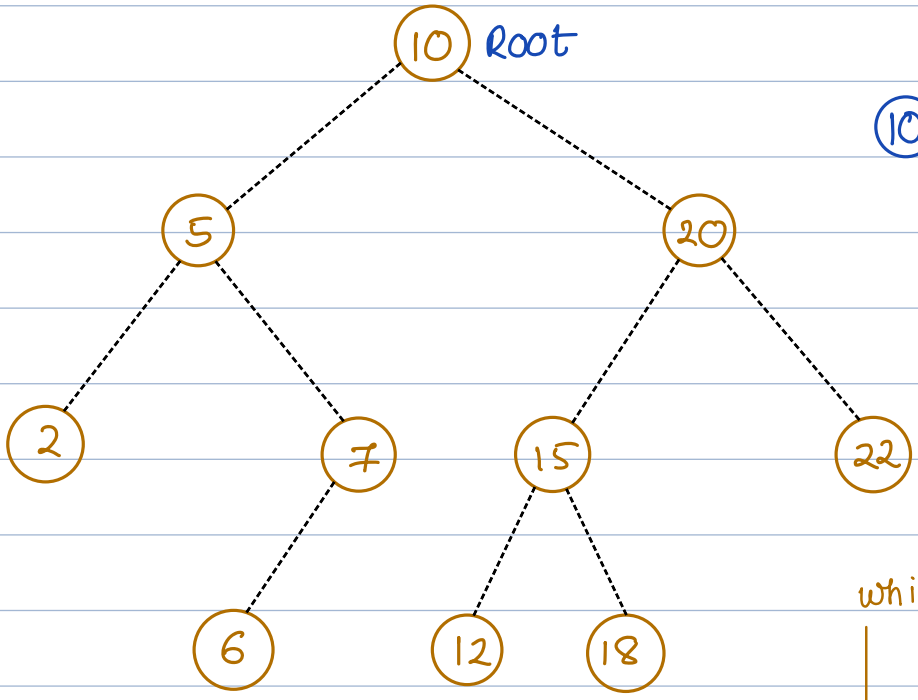


Valid BST.



Right side should be strictly greater than (1)

Searching in BST



Find 18

10 → 20 → 15 → 18

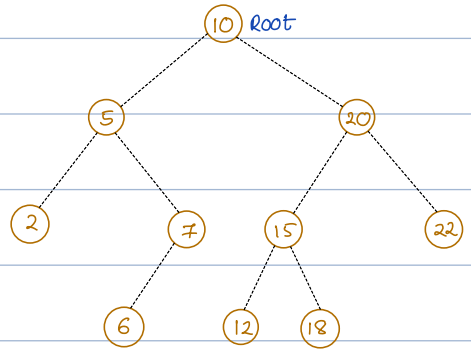
```
while (root != null) {
    if root.val == target
        return true
    if root.val < target
        root = root.right
    if root val > target
        root = root.left
}
return false.
```

TC : O(H)

SC : O(1)

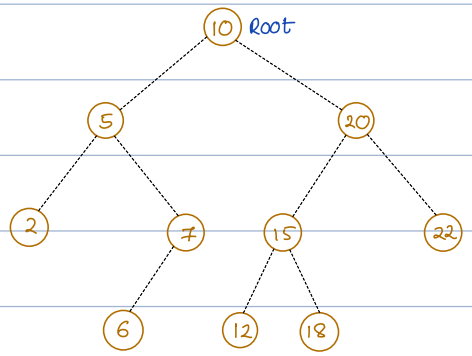
Q) Find the smallest element in a BST.

```
min = inf
while (root != null) {
    min = root.val
    root = root.left
}
print(min)
```

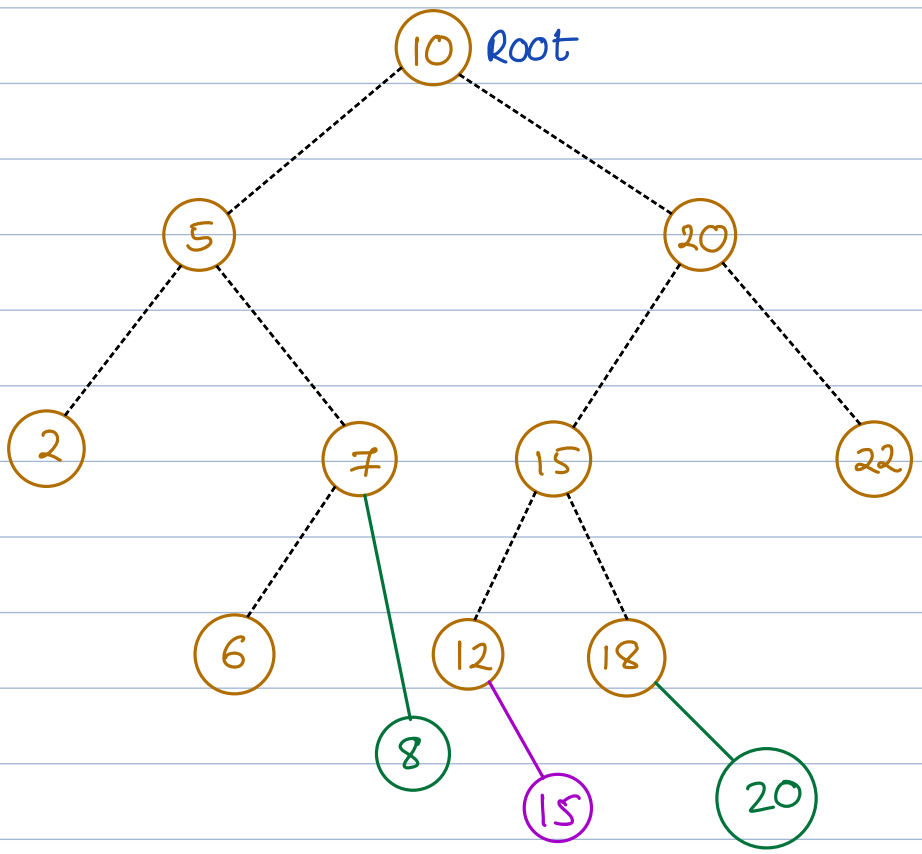


Q) Find the largest element in a BST

```
max = - inf
while (root != null) {
    max = root.val
    root = root.right
}
print(max)
```



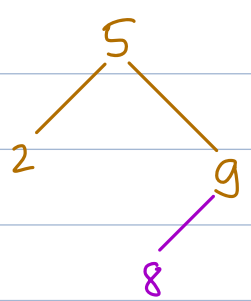
Insertion in a binary search tree



Insert (8)

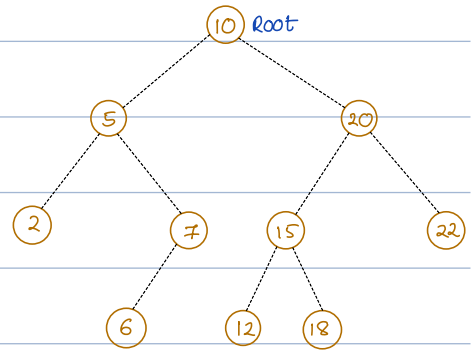
Insert 15

Insert 20



Pseudocode

```
// Insert a node with value X
xnode = Node(X)
temp = root

while ( temp != null ) {
    if ( temp.val < X ) {
        // insert as leaf if right is null
        if ( temp.right == null ) {
            temp.right = xnode
            return root
        }
        temp = temp.right
    }
    else {
        // insert as leaf if left is null
        if (temp.left == null) {
            temp.left = xnode
            return root
        }
        temp = temp.left
    }
}
```

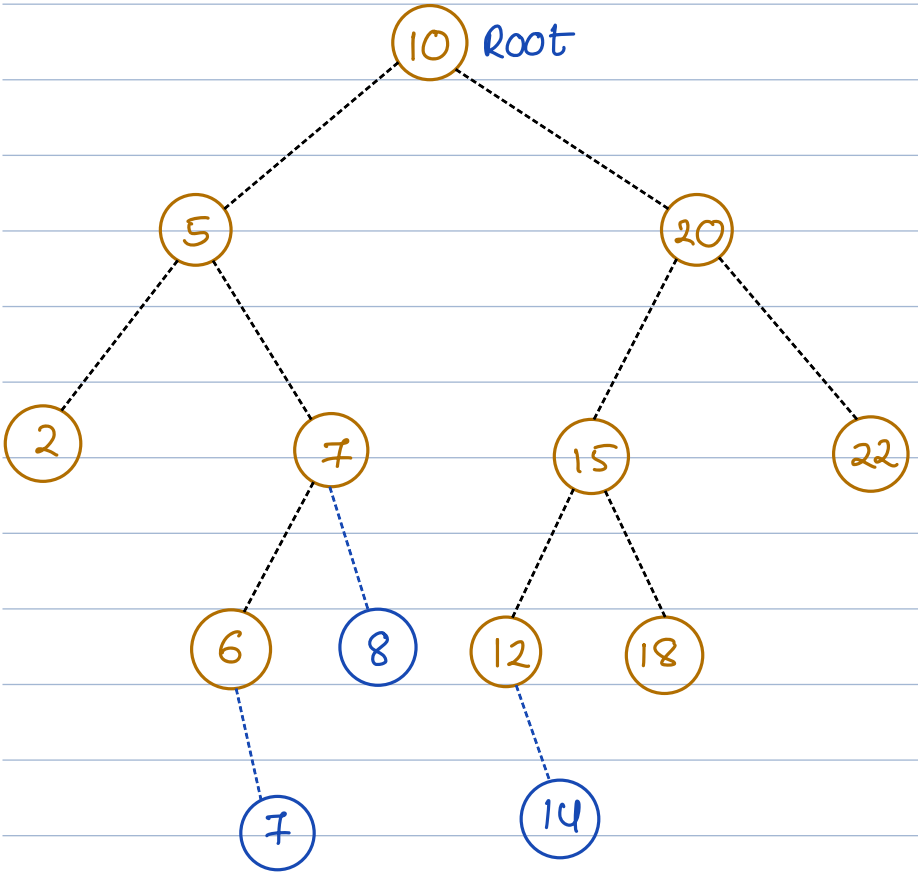


X = 20

TC : O(H)

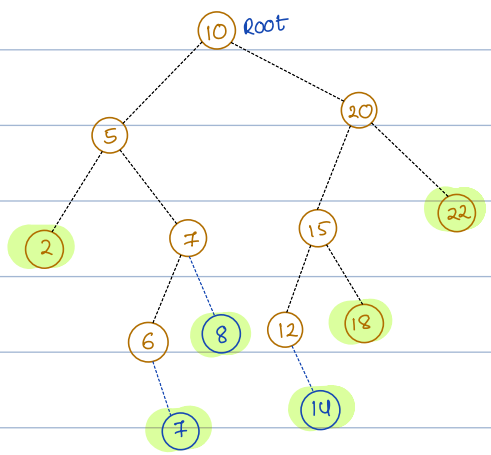
SC : O(1)

Deletion in BST



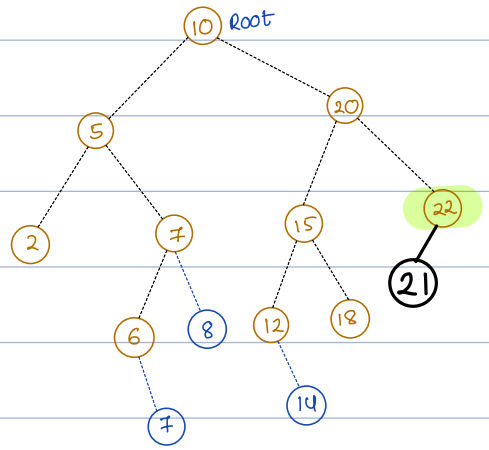
First we need to search for the node

Case A If node to be deleted {DN} is a leaf



Case B If node to be deleted {DN} 's right child is null

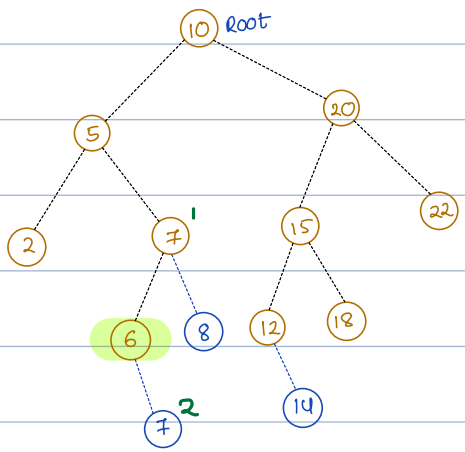
delete 22



Right child of 20 becomes 21

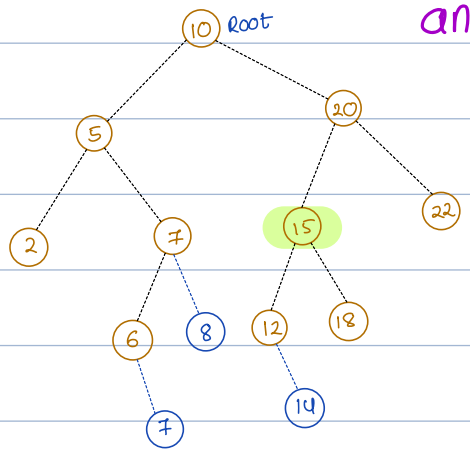
Case C If node to be deleted {DN} 's left child is null

delete 6



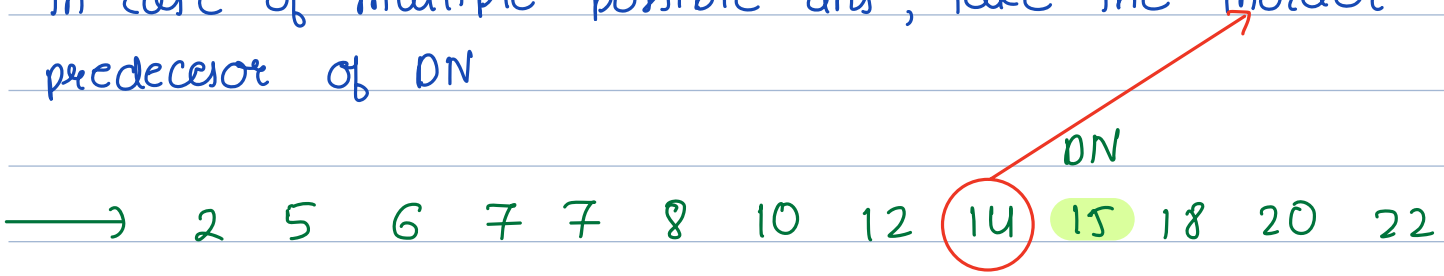
7^1 . left = 7^2

Case D If node to be deleted {DN}'s left child and right child is not null.



delete 15

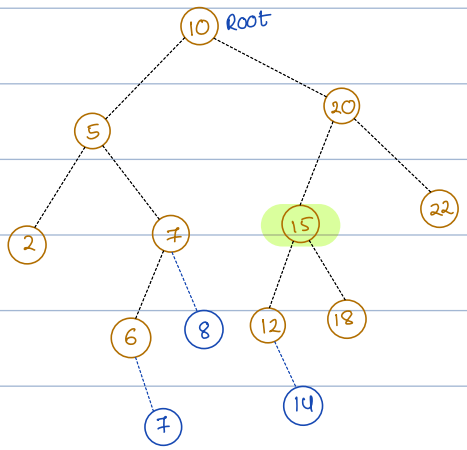
In case of multiple possible ans, take the inorder predecessor of DN

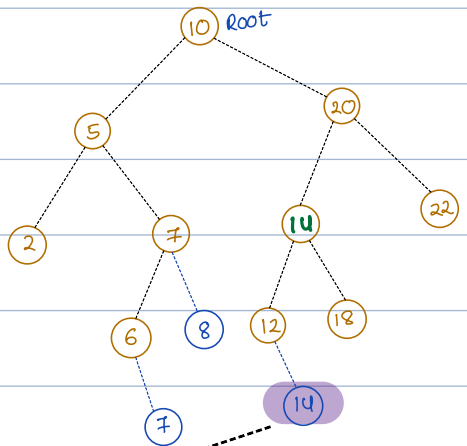


Inorder traversal of BST is always sorted

→ Find the largest value on the left subtree. {PN}

→ DN.val = PN.val





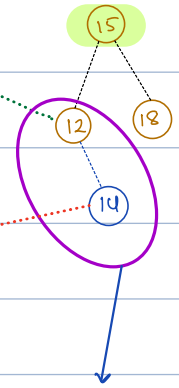
→ Delete PN

For PN the right child has to be null {case B}

Pseudocode

```
Node delete (root, target) {
    // if (root == null) {
    |     return null
    |}
    // case to go left
    if (root.val > target) {
    |     root.left = delete (root.left, target)
    |}
    // case to go right
    if (root.val < target) {
        root.right = delete (root.right, target)
    }
    // This node needs to be deleted
    if (root.val == target) {
        // case A leaf node
        if (root.left == null && root.right == null)
        |     return null
        |}
        // case B right child is null
        if (root.right == null) {
        |     return root.left
        |}
        // case C left child is null
        if (root.left == null) {
        |     return root.right.
        |}
```

```
// case 0
temp = root.left
while (temp.right != null ) {
        temp = temp.right
}
root.val = temp.val
root.left = delete ( root.left,
                     temp.val)
}
return root
}
```



Delete 14 from left subtree

TC : O(H)

SC : O(H)

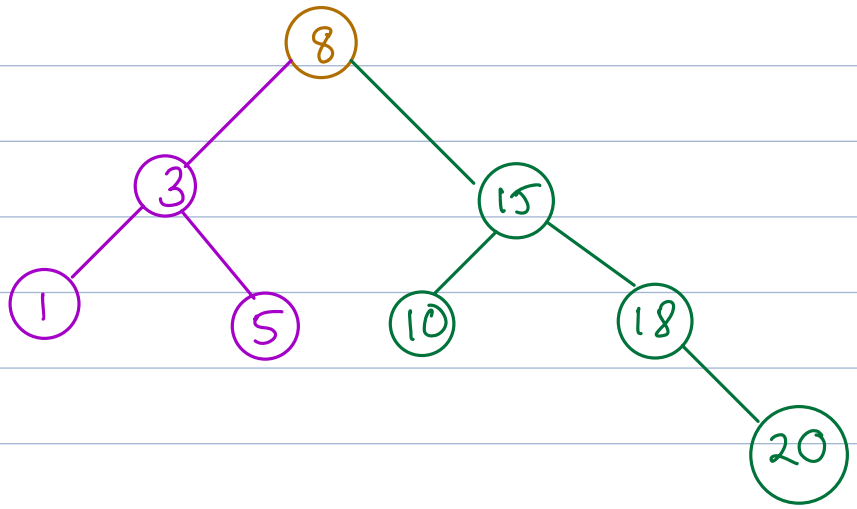
Break 8:54

Contruct Balanced BST from sorted array

all unique

A = 1 3 5 8 10 15 18 20

	0	1	2	3	4	5	6	7
A =	1	3	5	8	10	15	18	20



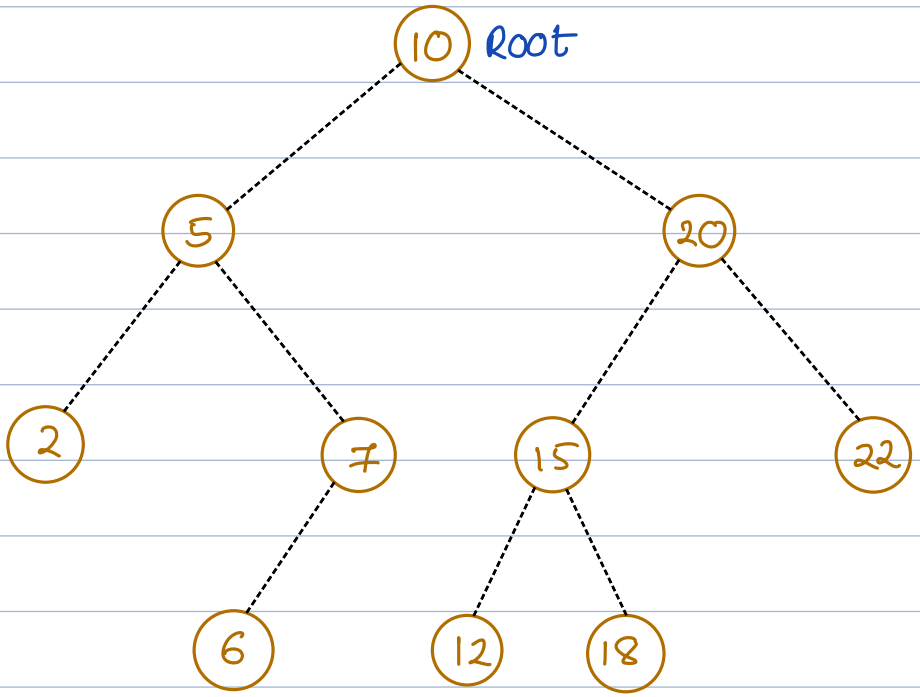
Approach : Choose mid element as the root recursively.

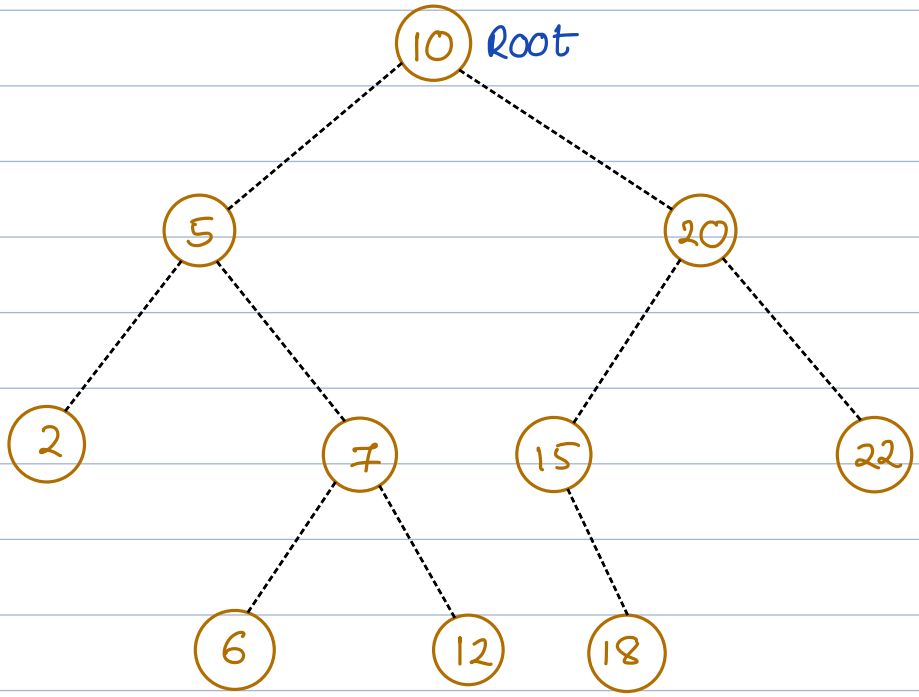
```
Node balanced (A[], left, right) {
    // Base condition                          TC : O(N)
    if (left > right) return null              SC : O(logN)

    mid = left + (right - left)/2
    root = new Node(A[mid])

    root.left = balanced(A, left, mid-1)
    root.right = balanced(A, mid+1, right)
    return root
}
```

(left → 0, right → N-1)

Check if the given binary tree is a BST

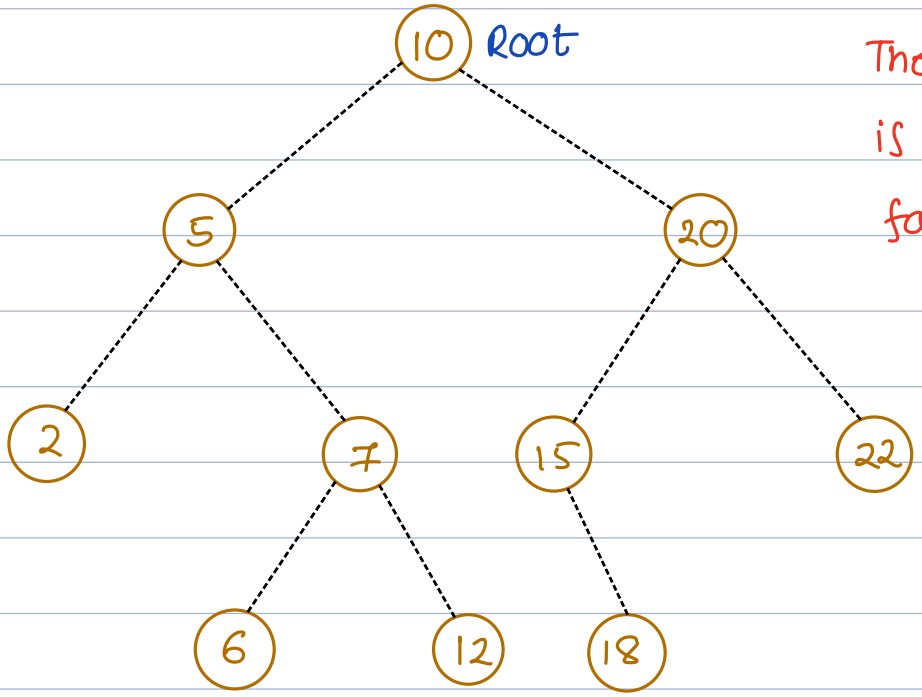
Microsoft





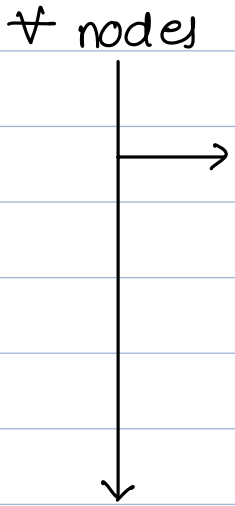
NOTE: Checking inorder of BT as sorted, will only work if the values are unique

Idea : Recursively check node val is greater than left child and smaller than right child.



The above idea is wrong as it fails for this tree

Idea 2>



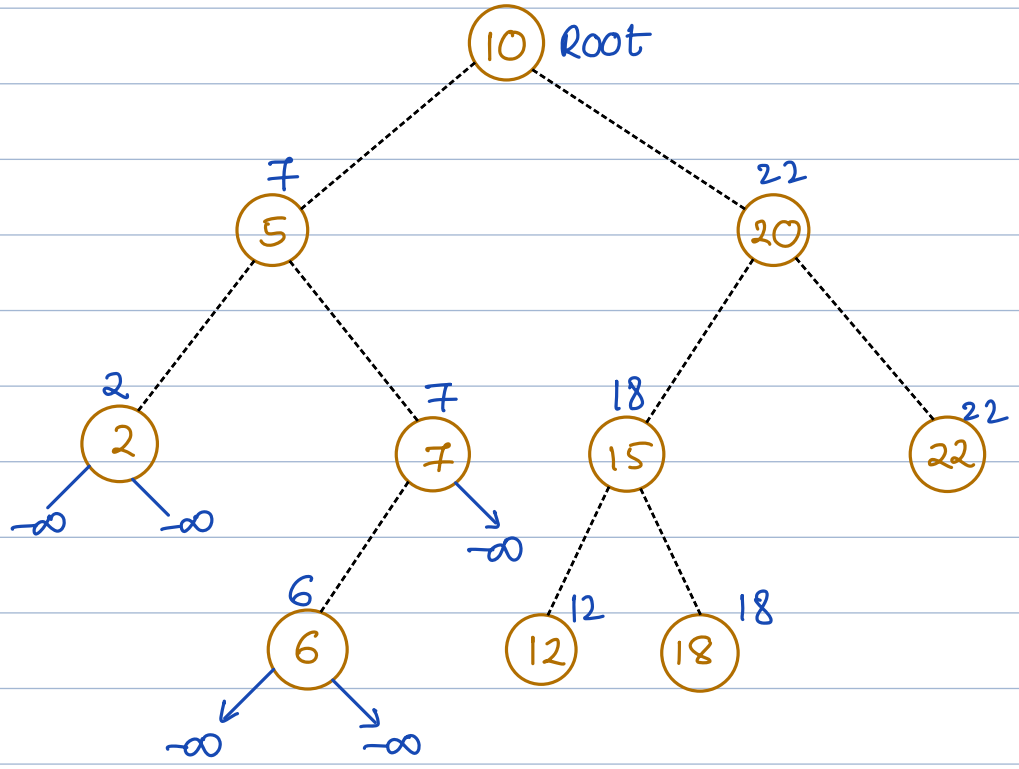
all the data in the left subtree <= x

all the data in the right subtree > x

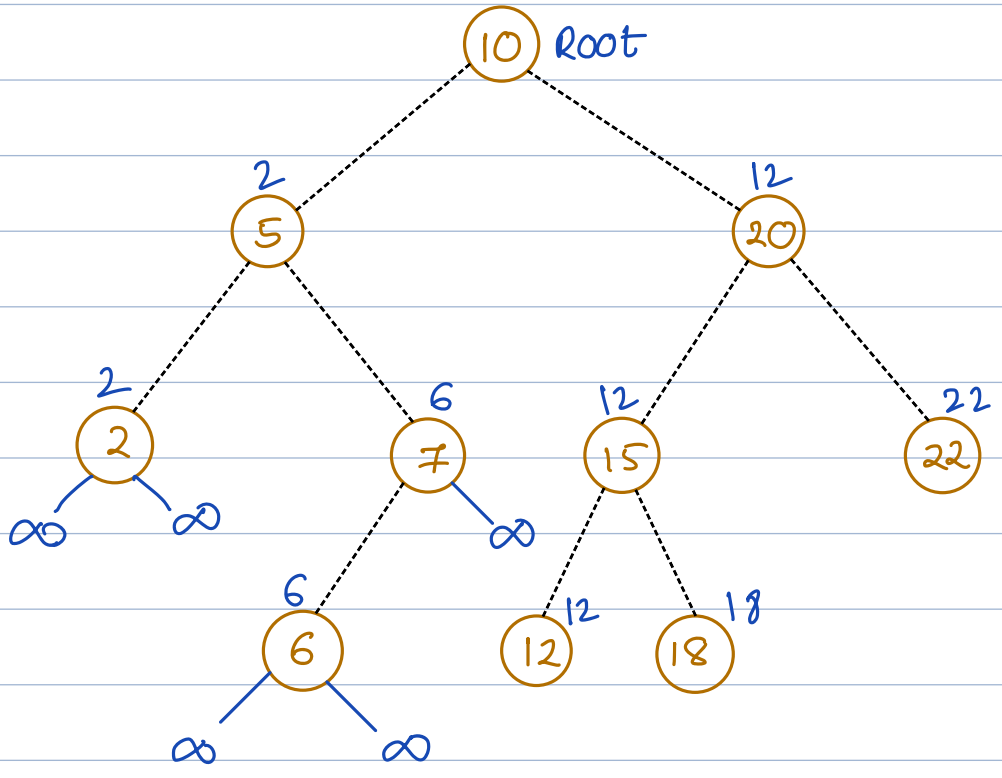
⟶ Max value from left subtree should <= X

⟶ Min value from right subtree > X

Max of each subtree.



Min for each subtree



→ max on left subtree <= X < min on right subtree

Pseudocode

```
class Pair {
    max
    min
}

isBst = true

Pair validBst (root) {
    if (root == null) {
        return Pair (-∞, ∞)
    }

    lp = validBst (root.left)
    rp = validBst (root.right)

    if (lp.max >= root.val ||
        rp.min <= root.val)
        isBst = false

    return Pair ( max (lp.max, root.val, rp.max),
                  min (lp.min, root.val, rp.min)
                )
}
```

TC : O(N)

SC : O(H)