Trees 4

Madhan Kumar M S
Abhishek Sharma
Akansh Nirmal
amit khandelwal
Balaji S K
Bhaveshkumar
Burhan
Gagan Kumar S
Gowtham
Ishan
Khushi Raj
Nikhil Pandey
Purusharth A
Rajat Sharma
Rajendra
Rathna
Sanket Giri
Saurabh Ruikar
Shani Jaiswal
sharath r
Subhashini
Sumit Adwani
Suyash Gupta
Vasanth
Vetrivel H M
Vimal Kumar
Yugesh v

AGENDA:

- k^{th} smallest element in BST
- Morris Inorder Traversal
- LCA in BT
- LCA in BST
- LCA using in-time & out-time

Imp Announcement

Saturday 9pm

Interactive Quiz based revision

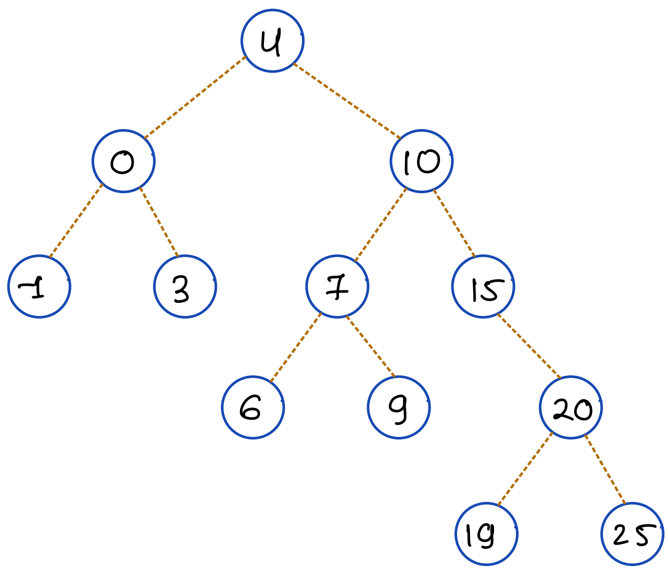
DSA 1 + DSA 2 Revision

Rules

→ Q → QT

→ any chat → private

K^{th} Smallest Element in BST



K = 3 ⟶ 3

K = 5 ⟶ 6

K = 10 ⟶ 19

Note ⟶ Inorder of a BST is sorted

Bruteforce

fill the inorder array.

return inorder[k-1]

A ⟶ stores the entire inorder

```
void inorder ( root ) {
    if ( root == null ) return
    inorder ( root.left )
    A.add( root.val )
    inorder ( root.right )
}
```

return in main A[k-1]

TC : O(N)

SC : O(N+H)

O(N)

Idea 2 maintain a global index

```
index = 0   // global index
ans = -∞

void inorder ( root ) {
    if ( root == null ) return
    inorder ( root.left )
    if (index == k-1) ans = root.data
    index += 1
    inorder ( root.right )
}

return in main ans
```

TC : O(N)

SC : O(H)

morris inorder

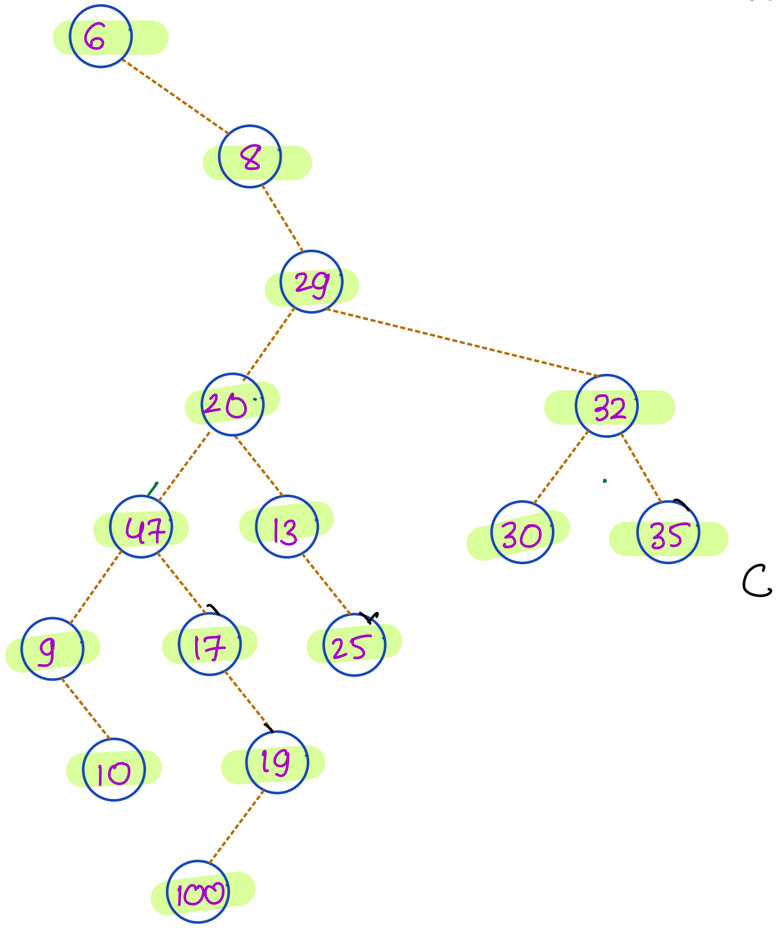
O(1)

Morris Inorder Traversal of Binary Tree

SC : O(1)

NO stack

NO recursion

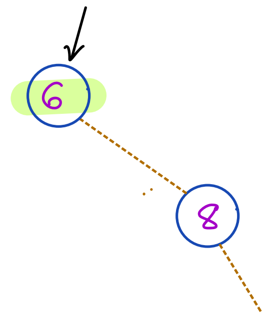


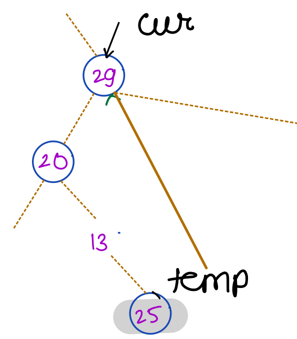
Node 25 will never have a right child

HW ⟶ code morris inorder, preorder, postorder

```
void  morris Inorder ( root ) {
   cur  =  root

   while ( cur != null) {
          // left is null
          if ( cur . left == null ) {
               print (cur . data )
               cur = cur . right
          }
          else {
                 temp = curr . left
                 while ( temp. right != null
                         && temp.right != cur )
                         temp = temp. right
                 }
                 // create link
                 if ( temp. right == null ) {
                       temp. right = curr
                       cur = cur . left
                 }
                 // delete link
              else ( temp. right == cur ) {
                       temp. right = null
                       print ( cur . data)
                       cur = cur. right
                 }
          }
   }
```

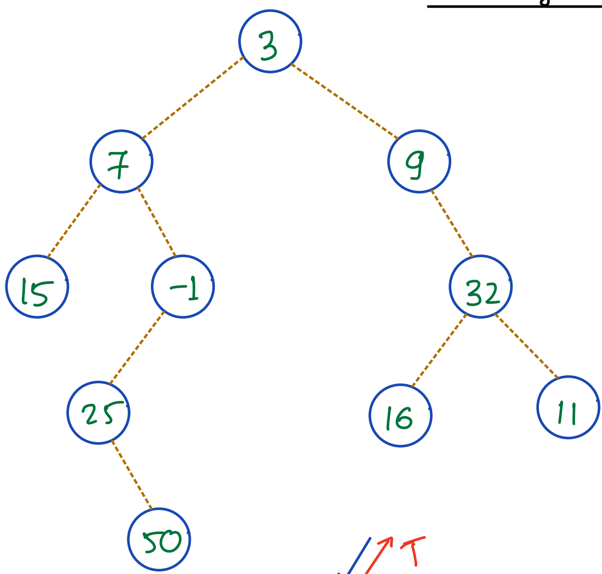




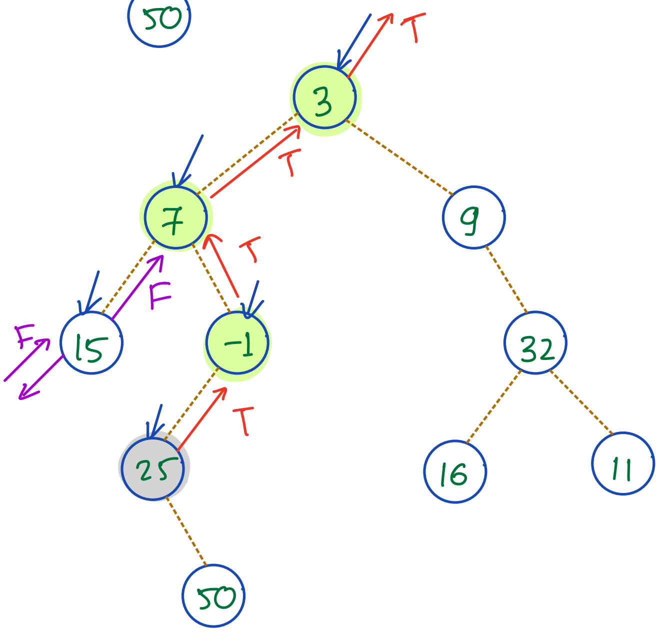
TC : O(N)

SC : O(1)

Find an element in a Binary Tree



Find (25)



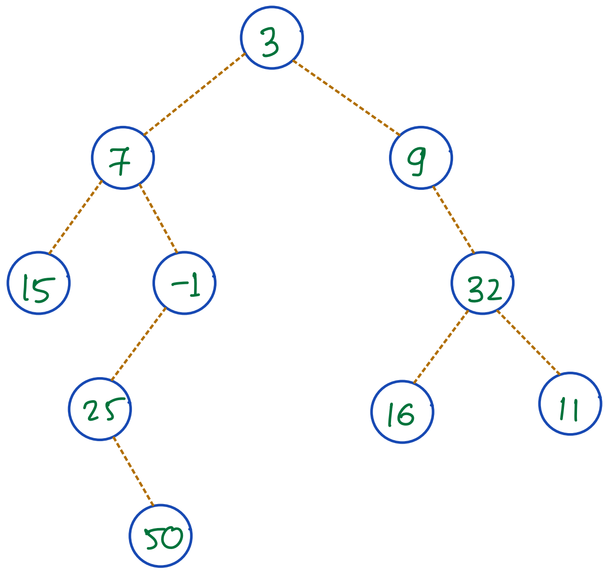

```
boolean search ( root , target ) {
        if ( root == null ) return false
        if ( root . val == target ) { return true }
        res = search ( root. left , target ) ||
                search ( root. right , target )

        return res
}
```

TC : O(N)

SC : O(H)

Path from root to node



Find (25)

$3 \rightarrow 7 \rightarrow -1 \rightarrow 25$

path =

Pseudocode

path = empty list

```
boolean rTNPath ( root , target ) {
    if ( root == null ) return false
    if ( root . val == target ) {
        path.add (root. data)
        return true
    }
    res = search ( root. left , target )  ||
          search ( root. right , target )
    if ( res ) path.add ( root. data )
    return res
}
```

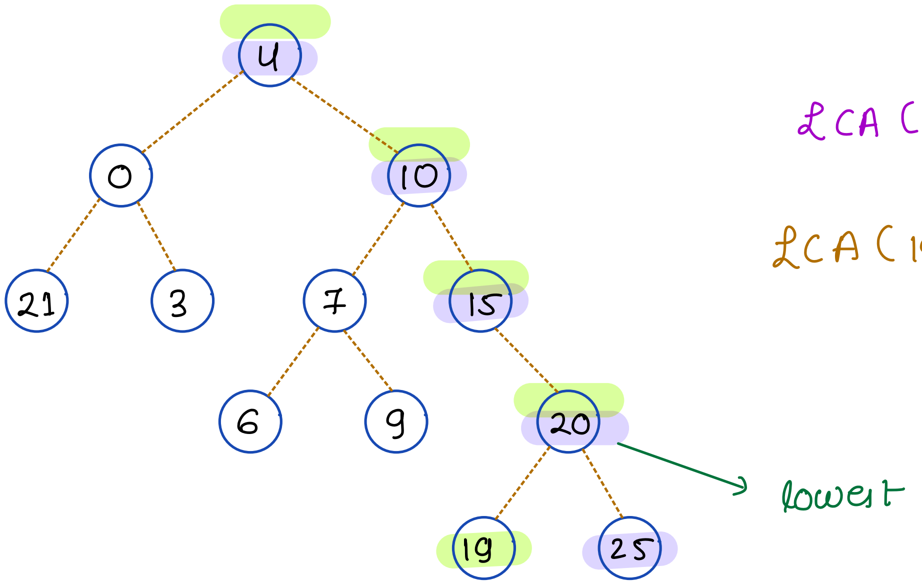
TC: O(N)

SC: O(H)

In the main reverse the path

Break : 22:40

Lowest Common Ancestor {LCA}



LCA(3, 6) = 4

LCA(19, 25) = 20

Ancestors(X) = All nodes from which node X can be reached

LCA(19, 25)

path root to 19 = 4 10 15 20 19

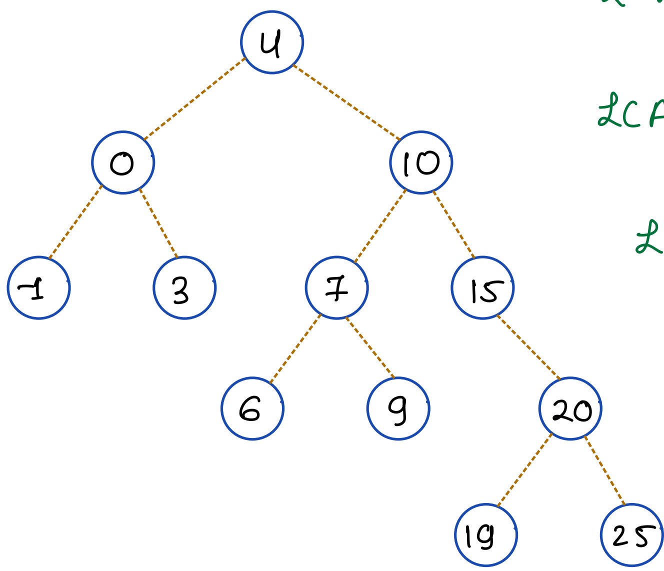
path root to 25 = 4 10 15 20 25

last common

TC: O(N)

SC: O(H)

LCA in a BST



LCA(6 -1) = 4

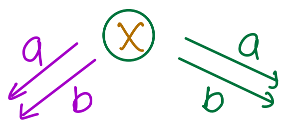
LCA(7, 20) = 10

LCA(100, 200) = null
-1
....

Prev approach works ∵ BST is also BT

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

Idea LCA(a, b)



move left move right

otherwise return X

Pseudocode

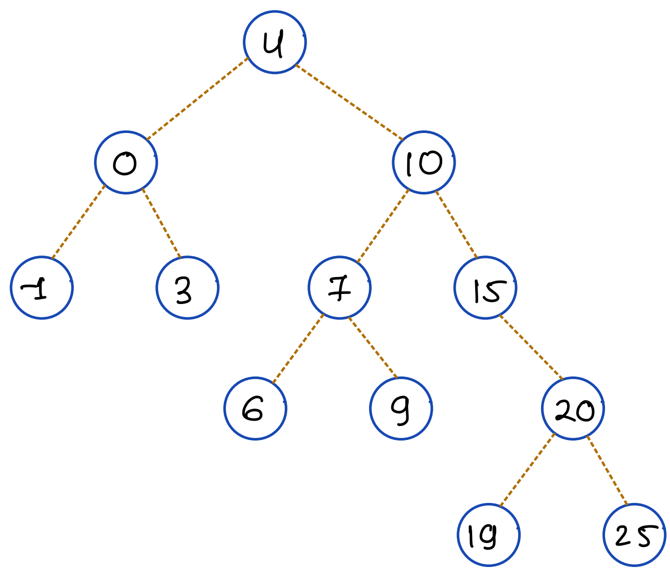
```
// ensure a & b are present.
TreeNode    LCABST ( TreeNode root, int a, int b) {
    if (root == null) return null
    node = root

    while ( node != null) {
            val = node.data

            if ( val > a && val > b ) {
                    node = node.left
            }
            else if ( val < a && val < b ) {
                    node = node.right
            }
            else {
                    return node
            }
    }
    return null
}
```

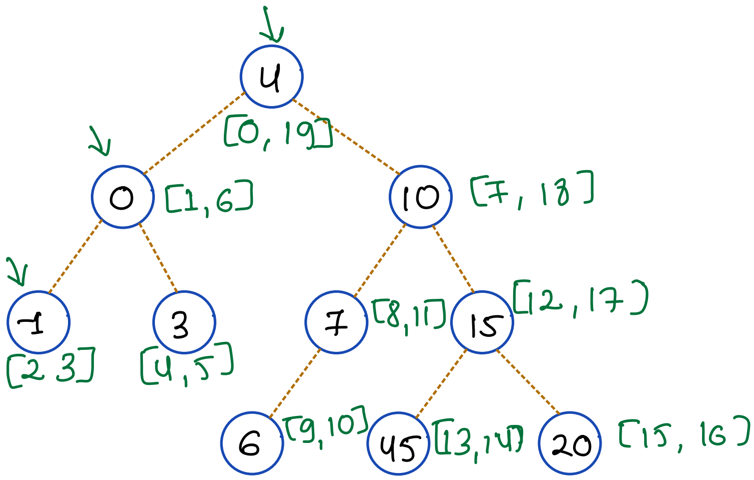
TC: O(H)

SC: O(1)



In - time Out - time

time = 0 // always unique

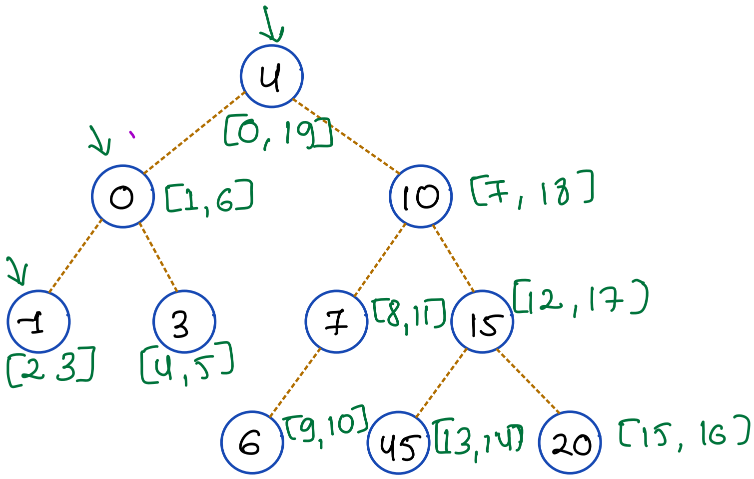



```
void  inOut ( root) {
      if (root == null)  return

      in[root] = time
      time ++
      inOut(root. left)
      inOut( root. right)
      out[root] = time
      time ++
}
```

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

map <TreeNode, Int>



LCA (-1 3)

How to find if node x is an ancestor of node y

In time of node x <= in time of node y

Out time of node x >= out time of node y

$\Longrightarrow$ Node x is an ancestor of node y

// calculate in and out time

```
TreeNode LCAIO ( TreeNode root, int a, int b) {
    if (root == null) return null
    node = root

    while ( node != null) {
            val = node.data

            if ( node.left is ancestor of both a & b) {
                    node = node.left
            }
        else if ( node.right is ancestor of both a & b) {
                    node = node.right
            }
            else {
                    return node
            }
    }
    return null
}
```

LCA

TC: O(H)
SC: O(1)
} per query

Overall code { forming in+out , LCA }

TC : O (N)

SC : O (N)

In out time approach is useful for multiple LCA query