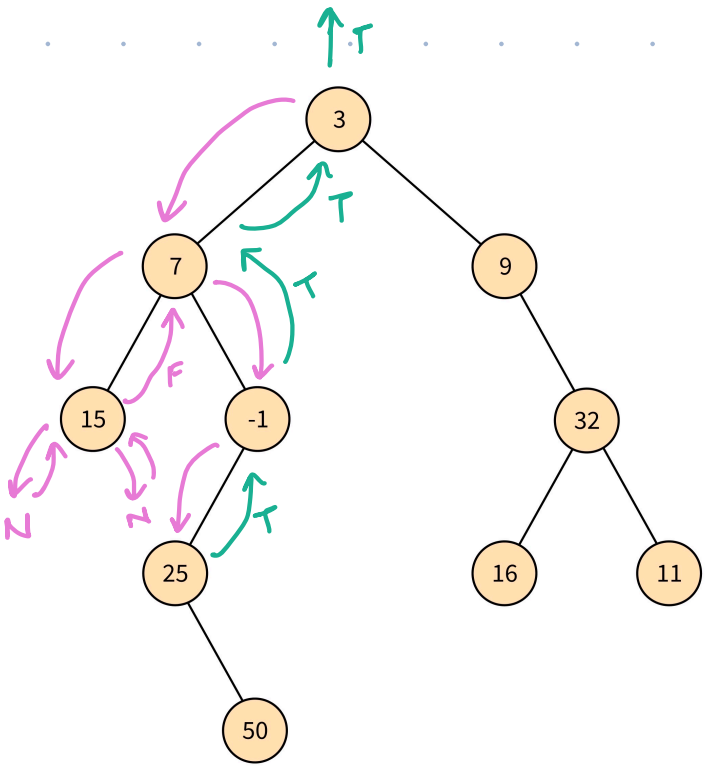
SCALER

# Trees - 4

TABLE OF CONTENTS



# Search an element in Binary Tree




K = 25 → true

Preorder

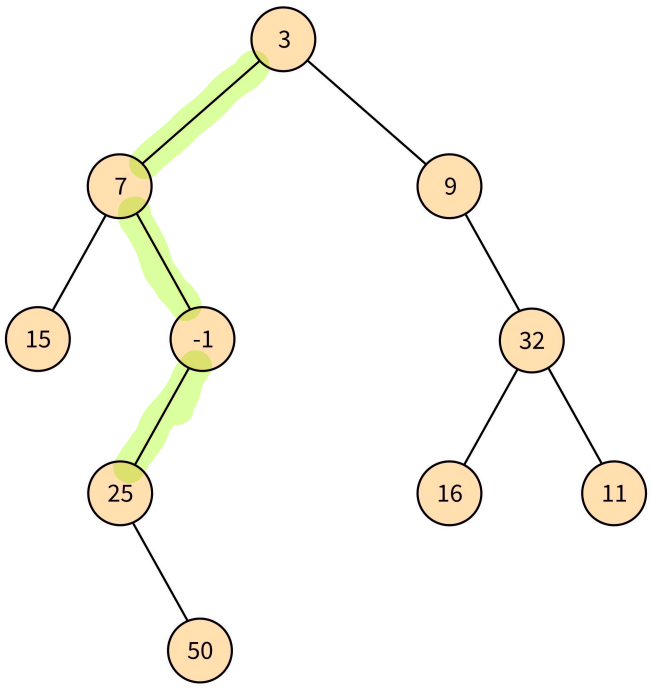
N L R

```
boolean search (Node root, int K) {
    // Base Case
    if (root == NULL)    return false;

    if (root.val == K)    return true;

    if ( search(root.left, K) == true)
            return true;

    if (search(root.right, K) == true)
            return true;

    return false;
}
```

T.C : O(N)
S.C : O(ht of tree)

return (search(root.left, K) || search(root.right, K));

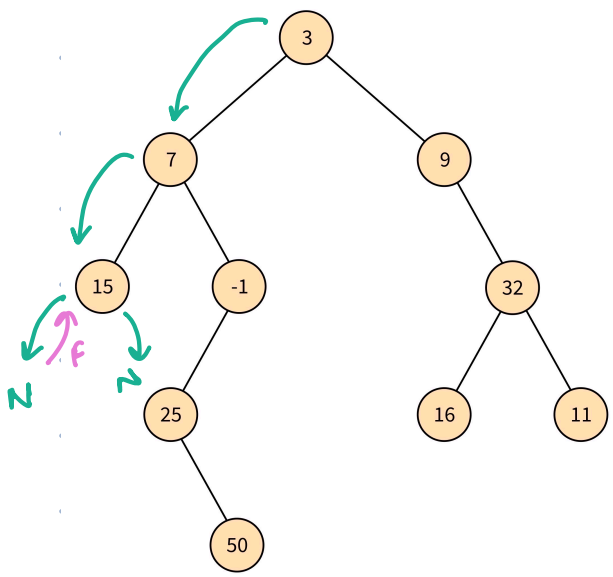
# Path from Root to Node




K = 25

[3, 7, -1, 25]

Approach: We search for an element & when the element is found we will return back & on returning back we will also add elements in an arraylist. Then Reverse the list.




K = 25

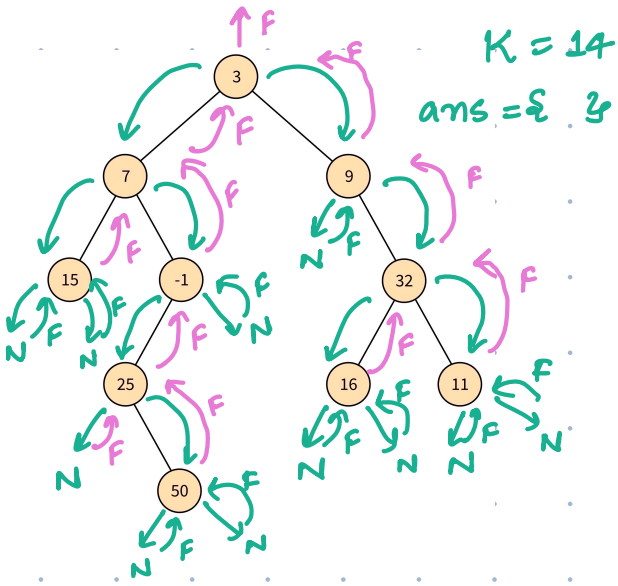
ans = { }

< / > Code

```
boolean search (Node root, int K, vector<int> ans) {
    if (root == NULL)  returns false;

    if (root.val == K) {
        ans.push(root.val);
        return true;
    }

    bool r = search(root.left, K, ans) || search(root.right, K, ans);

    if (r == true) {
        ans.push(root.val);
        return true;
    }

    return false;
}
```

T.C : O(N)
S.C : O(ht of tree)

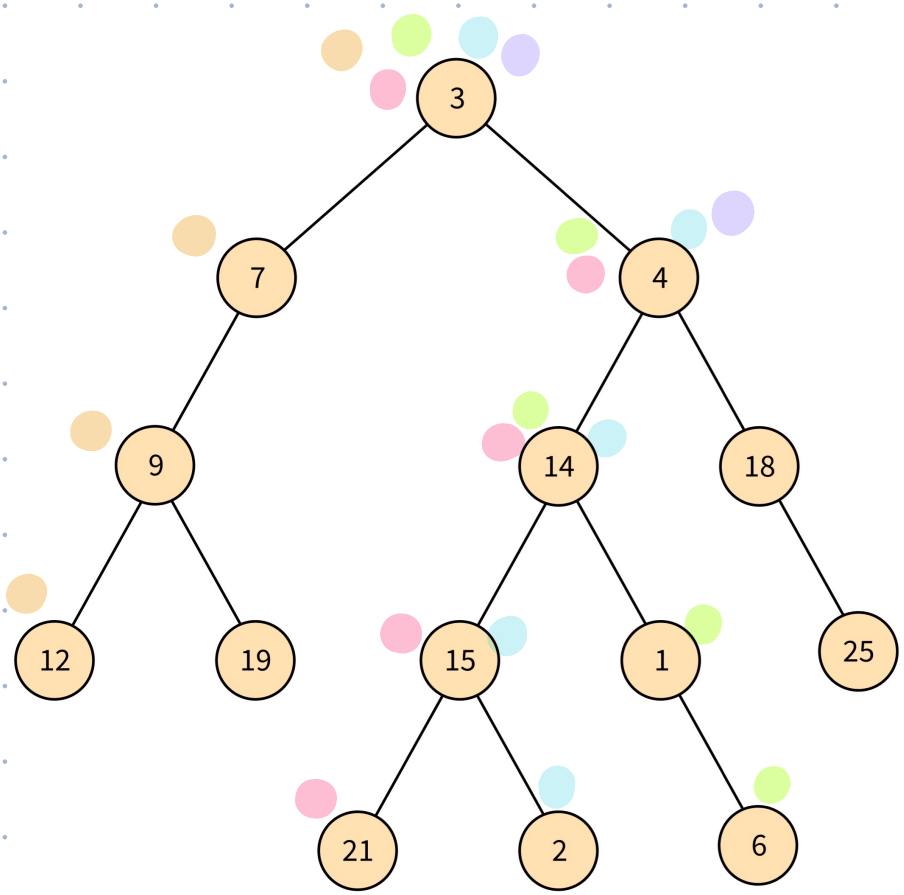
```
vector<int> RootToNodePath (Node root, int K) {
    vector<int> ans;
    search (root, K, ans);
    reverse(ans);
    return ans;
}
```



# Lowest Common Ancestor (LCA)

Flipkart | Acollite | Amazon | Microsoft | OYO Rooms | Snapdeal | MakeMyTrip | PayU | Times Internet | Cisco | PayPal | Expedia | Twitter | American Express

Note : All values are distinct. Both v1 & v2 will exist.




L.C.A( 21 , 6) → 14

L.C.A( 2 , 6) → 14

L.C.A( 21 , 4) → 4

L.C.A( 12 , 6) → 3

21 → [ 3 4 14 15 21 ]

6 → [ 3 4 14 1 6 ]

Root → Node

[21 15 14 4 3 ]

[6 1 14 4 3]

4 [4 3]

## Approach :

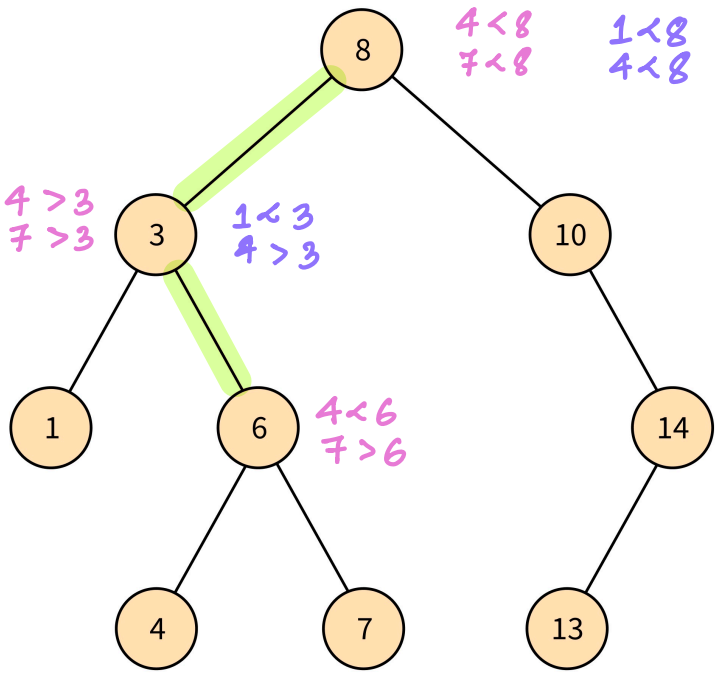
Find Root → Node Path for both $v_1$ & $v_2$.

Iterate on both arrays & check for last common value.

T.C : O(N) S.C : O(N)

# L.C.A in B.S.T

Both $v_1$ & $v_2$ will exist.




LCA (4, 7) → 6

LCA (1, 4) → 3

Idea 1 : Same approach as BT.

Idea 2 : Make use of BST property for searching & find the point where diversion pt.

## < / > Code

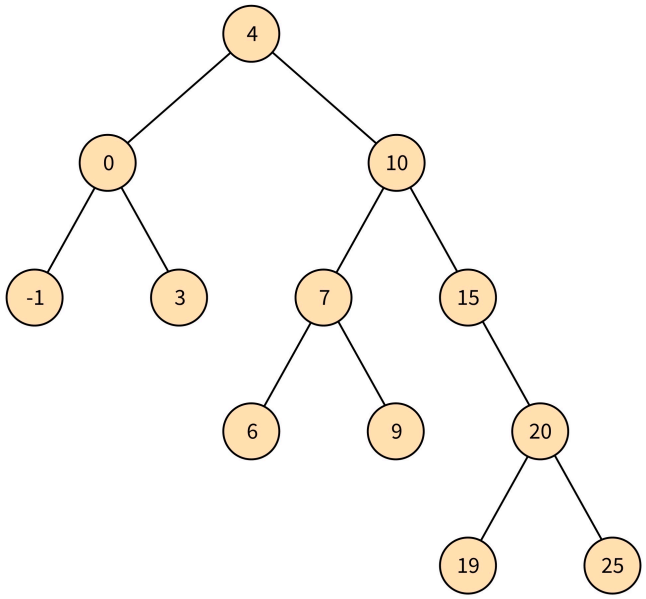
```
int LCA (Node root , int v1 , int v2) {
    Node curr = root ;
    while ( true) {
        if (v1 < curr.val && v2 < curr.val)
            curr = curr.left ;
        else if (v1 > curr.val && v2 > curr.val)
            curr = curr.right ;
        else
            return curr.val ;
    }
}
```

T.C : O (ht of tree)
S.C : O(1)

**< Question > :** Find Kth smallest element in B.S.T

K = 3

3rd smallest element : 3




Inorder of BST is sorted in Ascending Order

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Inorder : | -1 | 0 | 3 | 4 | 6 | 7 | 9 | 10 | 15 | 14 | 20 | 25 |

## BF Idea

Find Inorder of the BST & store it in Arraylist.

value at $(K-1)^{th}$ index of this arraylist is $K^{th}$ smallest value.

T.C : O(N)

S.C : O(N)

## Idea -2

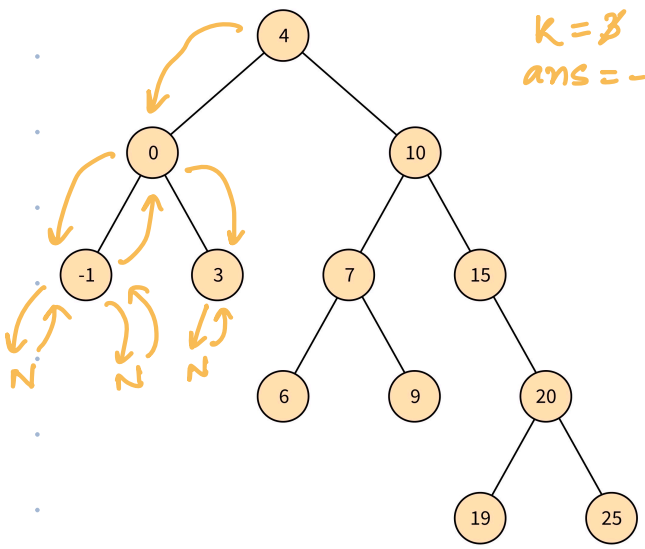
While doing Inorder traversal, keep track of count of nodes.

L N R

< / > Code

```
void Inorder(Node root, int &K, int &ans) {
    if (root == NULL) return;

    Inorder(root.left, K, ans);

    K--;
    if (K == 0) {
        ans = root.val;
        return;
    }

    Inorder(root.right, K, ans);
}
```

T.C : O(N)
S.C : O(Ht of tree)




K = 3 2 1 0
ans = -1
3

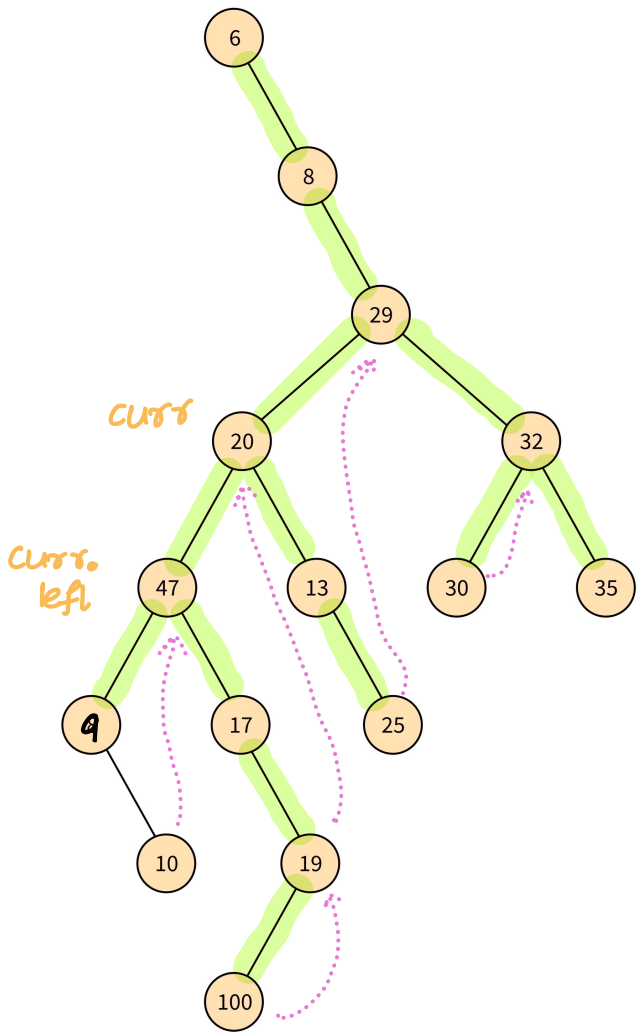
```
int Kth Smallest Element (Node root, int K) {
    int ans = -1;
    Inorder(root, K, ans);
    return ans;
}
```

# Morris Traversal

S.C of Recursive : O(Ht of tree)
" " Iterative : O(Ht of tree)

## In-Order Traversal of a Binary Tree

Expected S.C → O(1)




Can we use Recursion? ✗

Iteration ✓

Inorder : [6 8 9 10 47 17 100 19 20 13 25 29 30 32 35]

L N R

Inorder Predecessor : Node that comes just before the given node in its Inorder.

10 is IP of 47

100 " 19

19 " 20

25 " 29

30 " 32

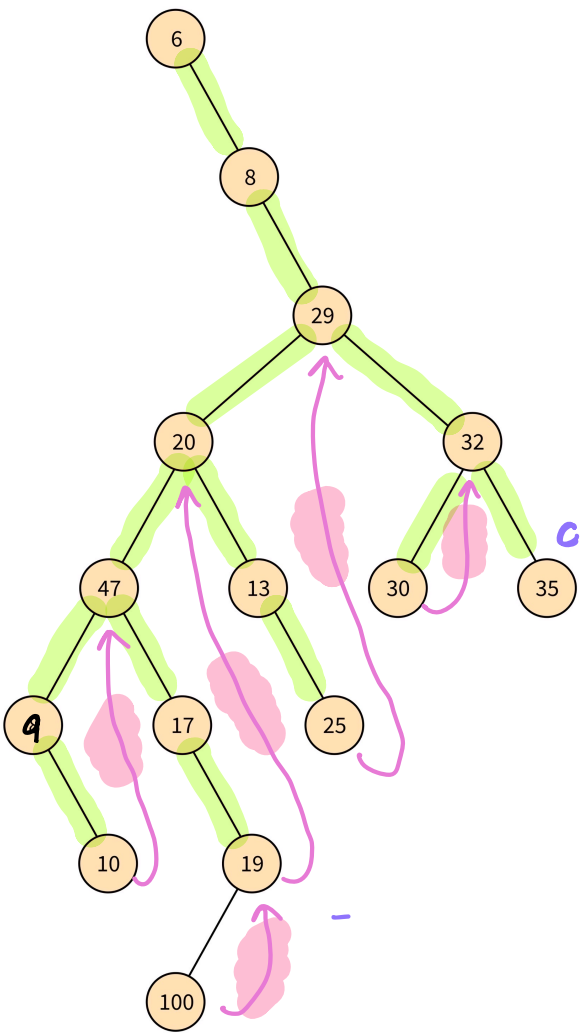
## Observations :

**For Nodes with LEFT CHILD**

- Inorder Predecessor of a node = Rightmost child of curr.left
- Inorder Predecessor of a node doesn't have RIGHT CHILD

Right child = NULL

We can use this to make connection to the next node in its Inorder.




6 8 9 10 47 17 100
19 20 13 25 29
30 32 35

```
Node temp = curr.left
while (temp.right != NULL)
    temp = temp.right ;
```

## Approach :

If Node has LEFT Child

yes

Find IP of curr Node
(Rightmost child of curr.left)

Check if the connection exists b/w IP & curr node

yes

- Break the connection
- Print the curr node value.
- Go to RIGHT child
  curr = curr.right

No

- Make Connection
- curr = curr.left

No

- Print Node
- Go to RIGHT child.

< / > Code

```
void MorrisInorder(Node root) {

    Node curr = root;
    while (curr != NULL) {

        if (curr.left == NULL) {
            print(curr.val);
            curr = curr.right;
        }
        else {          // Curr has left child
            // Find IP
            Node temp = curr.left;
            while (temp.right != NULL && temp.right != curr)
                temp = temp.right;

            // Connection doesn't exist
            if (temp.right == NULL) {
                temp.right = curr;
                curr = curr.left;
            }
            // Connection already exist
            else if (temp.right == curr) {
                temp.right = NULL;
                print(curr.val);
                curr = curr.right;
            }
        }
    }
}
```

T.C : O(N)
S.C : O(1)