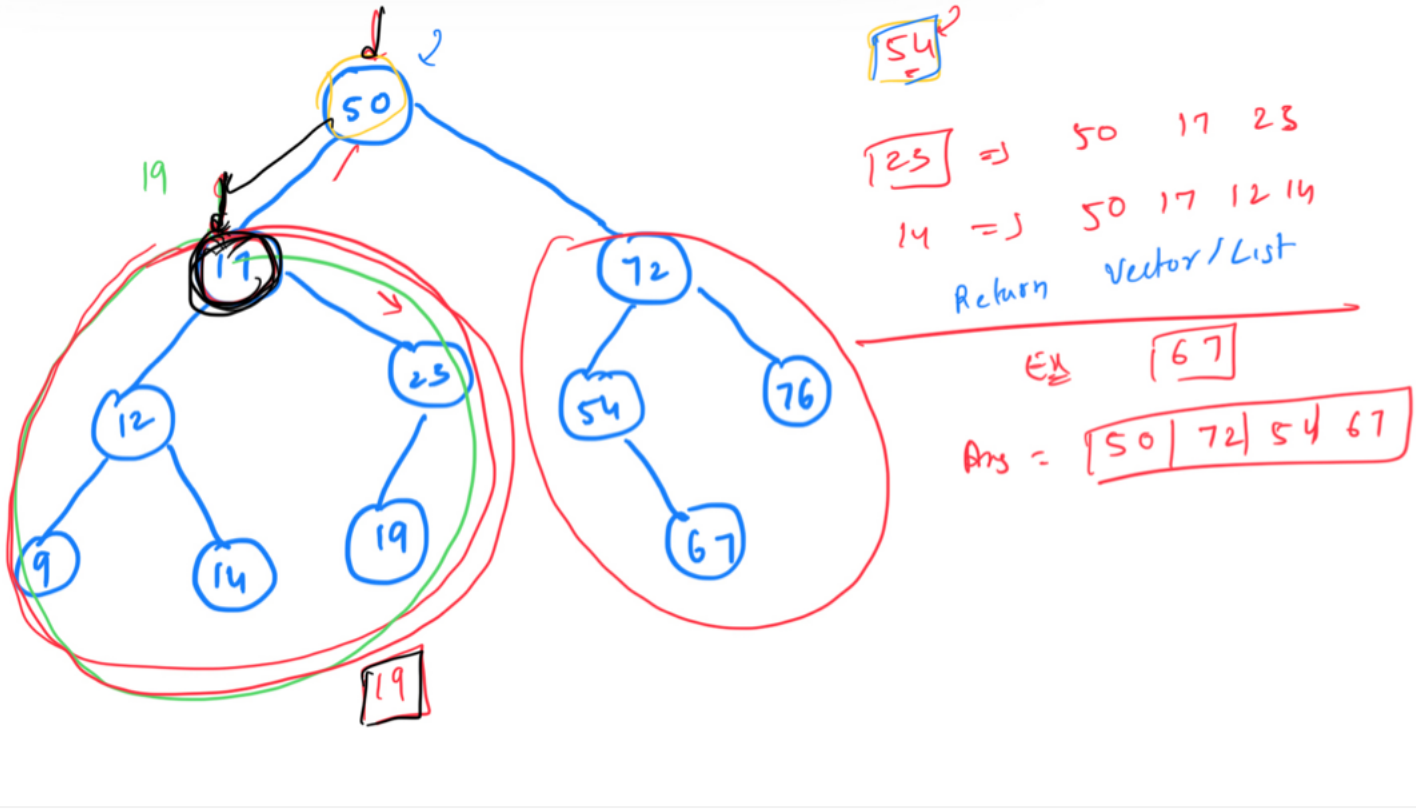
LCA + Problems on Tree

Question: search for an element



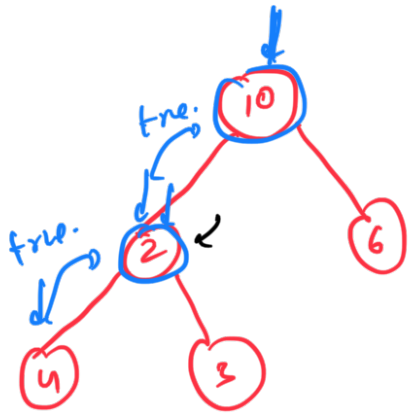
```
bool search(root, x) {
    if(root == NULL) return false;
    if(root.data == x) {
        ans.push(root.data);
        return true;
    }
    left = search(root.left, x);
    right = search(root.right, x);
    if(left || right) {
        ans.push(root.data);
        return true;
    }
    else return false;
}
```

T.C: O(H) ? x → BST

O(N) → B.T

S.C: O(H) [Height of Tree]

Question: Print the path from root to that node

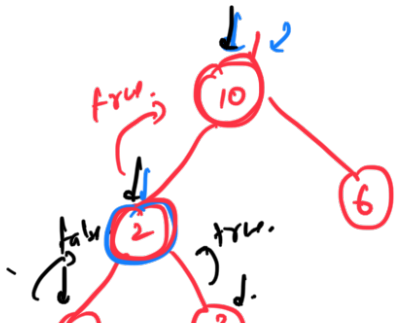


Path(4)

Ans = [4 | 2 | 1 | 10]

Reverse the array

x = 3



T.C: O(n)

S.C.

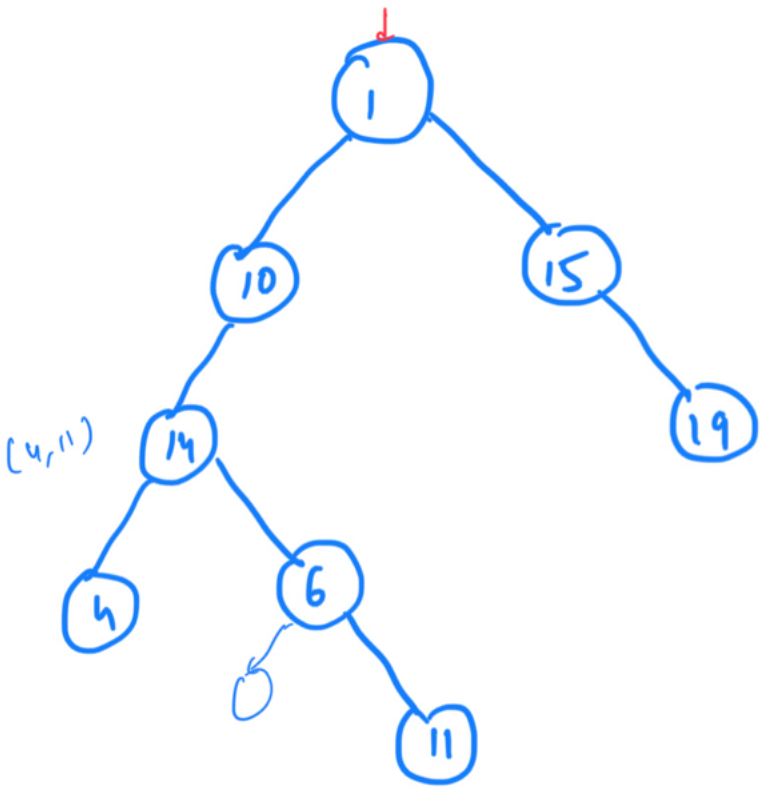
Space Complexity:- Extra space Apart from Input & Output

Tree. List

Recursion stack. : O(H)

Recursion stack

Question: Find path between 2 nodes



(6,19) =

6, 14, 10, 1, 15, 19

path(4,11) = 4, 14, 6, 11

path(14,11) = 14, 6, 11

path(6,19)

path(6) = 1 10 14 6

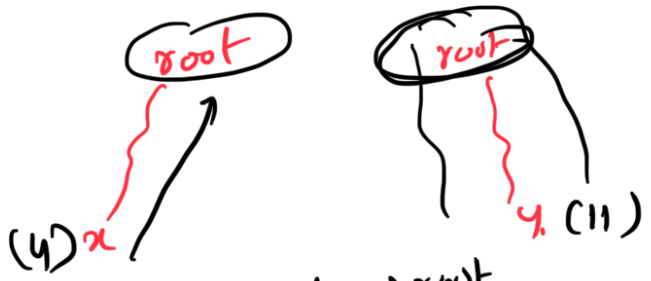
path(19) = 1 15 19

6, 14, 10, 1, 15, 19

path(4, 11) = 4 14 6 11

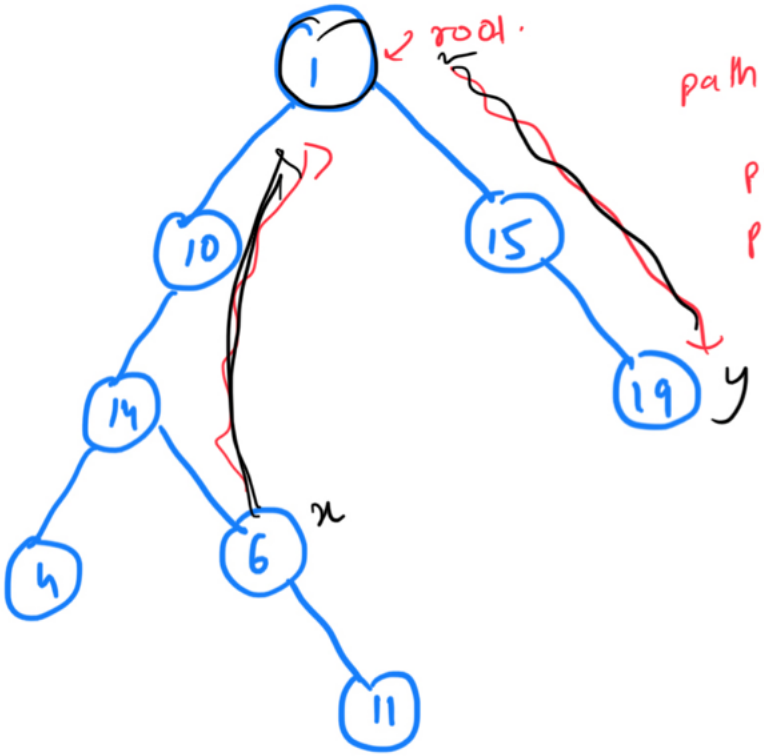
→ path(4) = 1 10 14 4

path(11) = 1 10 14 6 11



4 → root

{ 4 14 10 1 } 10 14 6 11

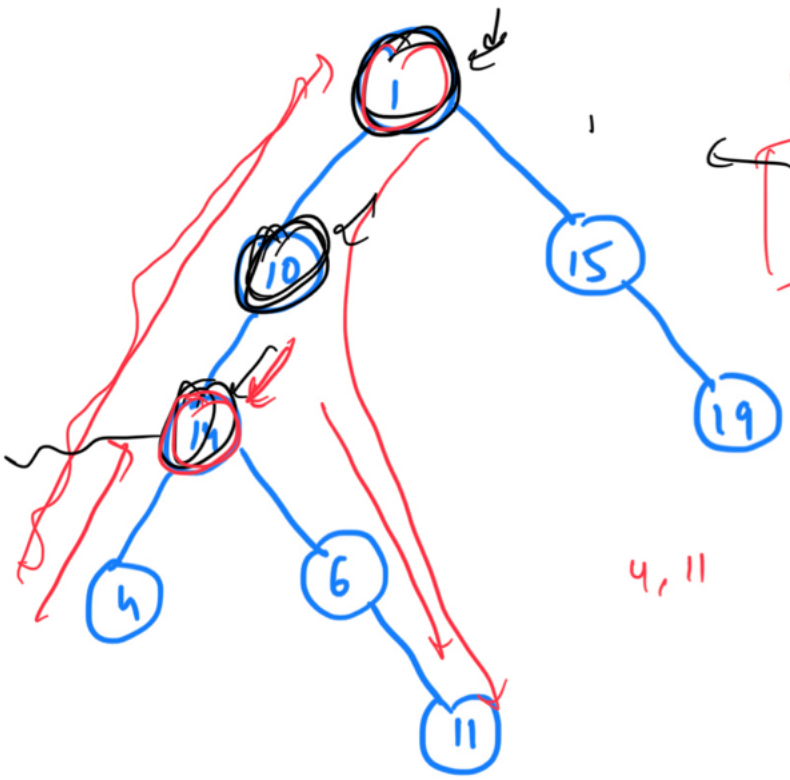


path(6, 19) = 6, 14, 10, ?, 15, 19

path(6) = 1 10 14 6 11

path(19) = 1 15 19.

6, 14, 10, 1, 15, 19.



path(4, 11) = 4, 14, 6, 11

path(4) = 1 10 14 4

path(11) = 1 10 14 6 11

i+1

4, 14, 6, 1

4, 11

LCA

Lowest Common Ancestor

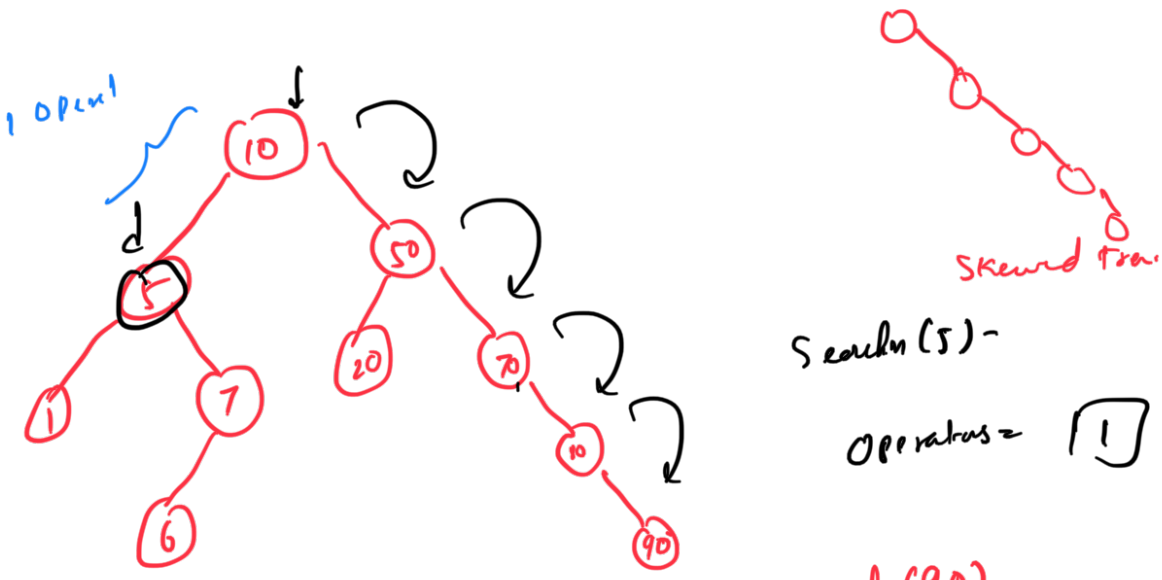
T.C : O(n)

S.C : O(H) + O(H) + O(H) = O(H)

2 arrays

Recursion stack

worst case : O(N)

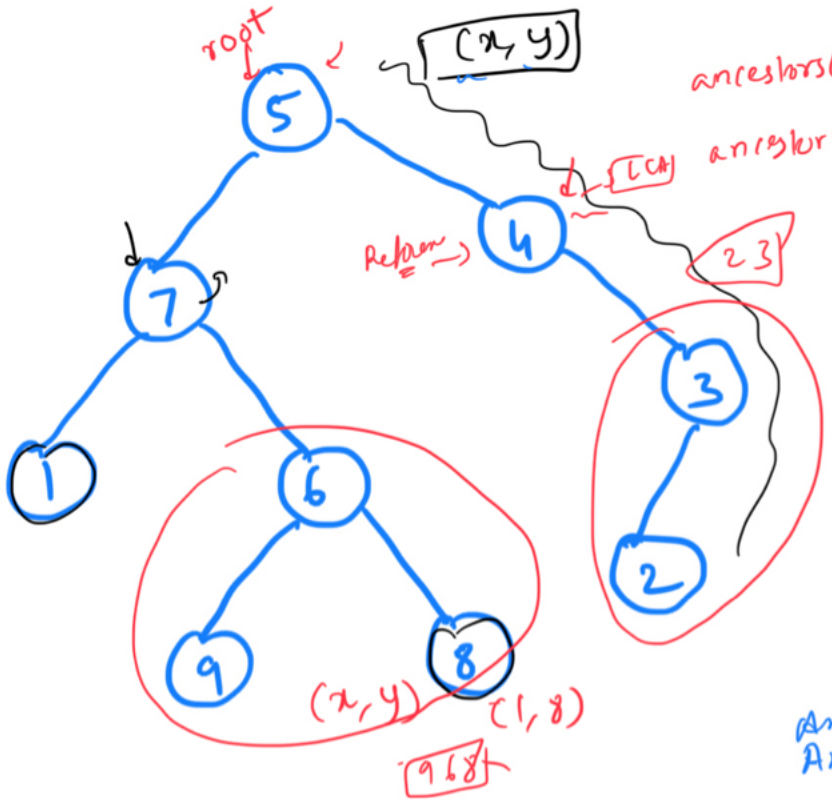


O(N)

Search (90)

4 operations

Question: Lowest Common Ancestor of 2 nodes (LCA)



ancestors(2) = 5, 4, 3, 2

ancestors(6) = 5, 7, 6. (6, 5)

LCA(2, 6) = 5.

LCA(1, 8) = 7

Anc(1) = 5, 7, 1

Anc(8) = 5, 7, 6, 8

Last Common Node of their paths

LCA(7, 2) = 5.

LCA(4, 2) = 4.

Anc(4) = 5, 4

Anc(2) = 5, 4, 3, 2

Approach 1 :

1) Find paths from root to x and y
2) Common Node at highest depth

T.C : O(n)

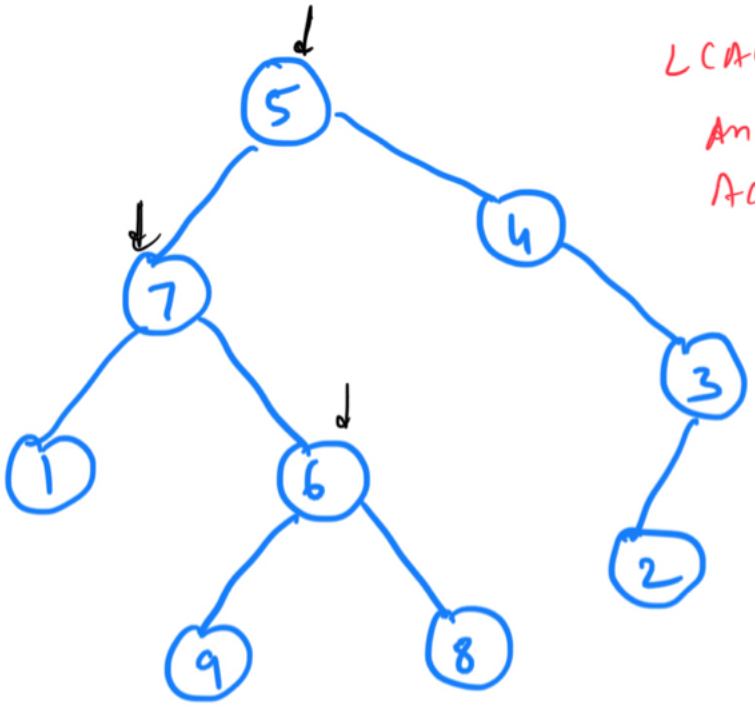
S.C : O(H) + O(H) + O(H)

O(H) : Arrays

O(H) : Recursion stack

Approach 2 :

1) If one node is ancestor of other, then it is the LCA
2) LCA is the only node/ancestor where x is in one subtree & y in other.



LCA(9,8) = 6

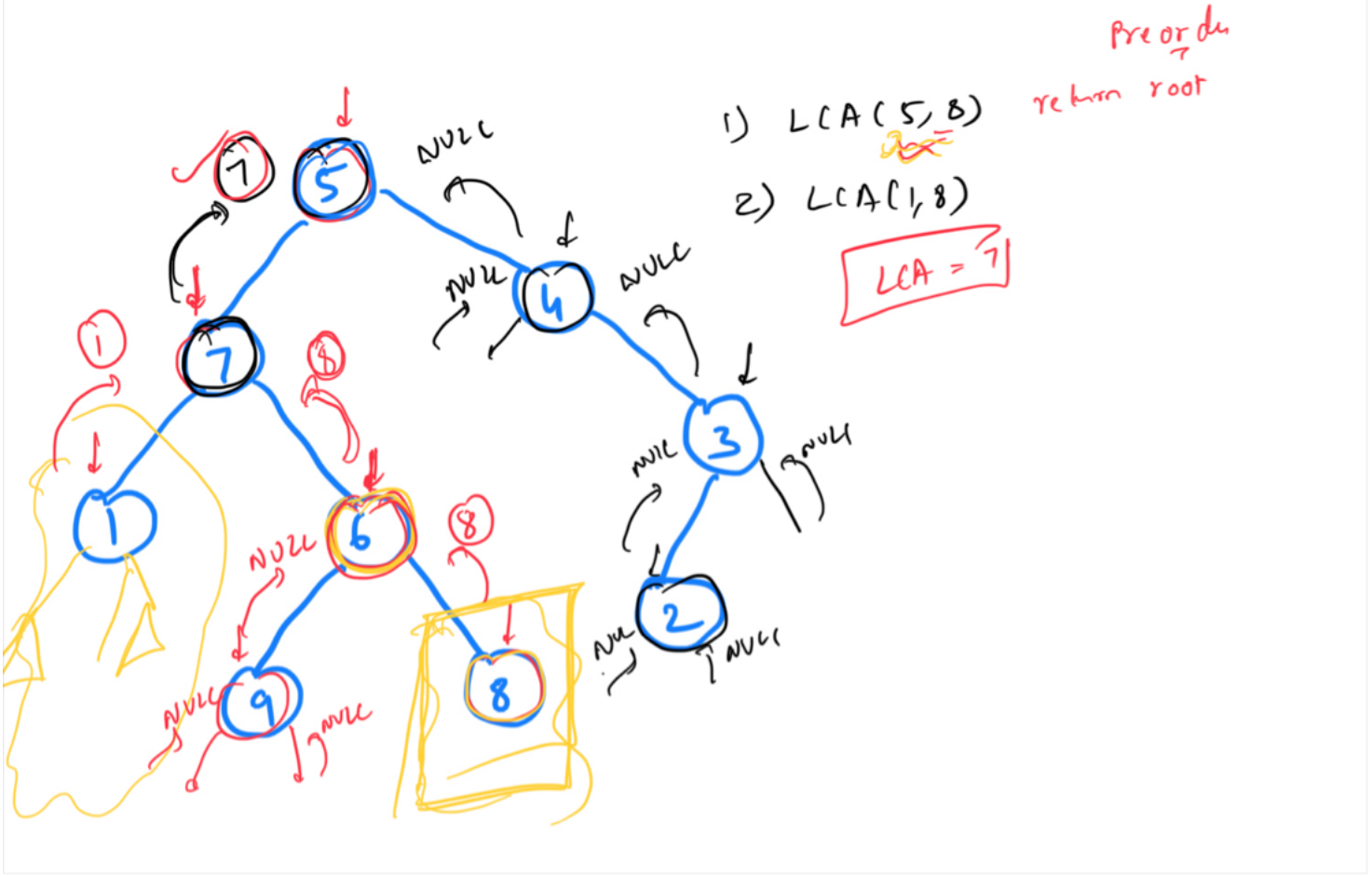
(9,8)

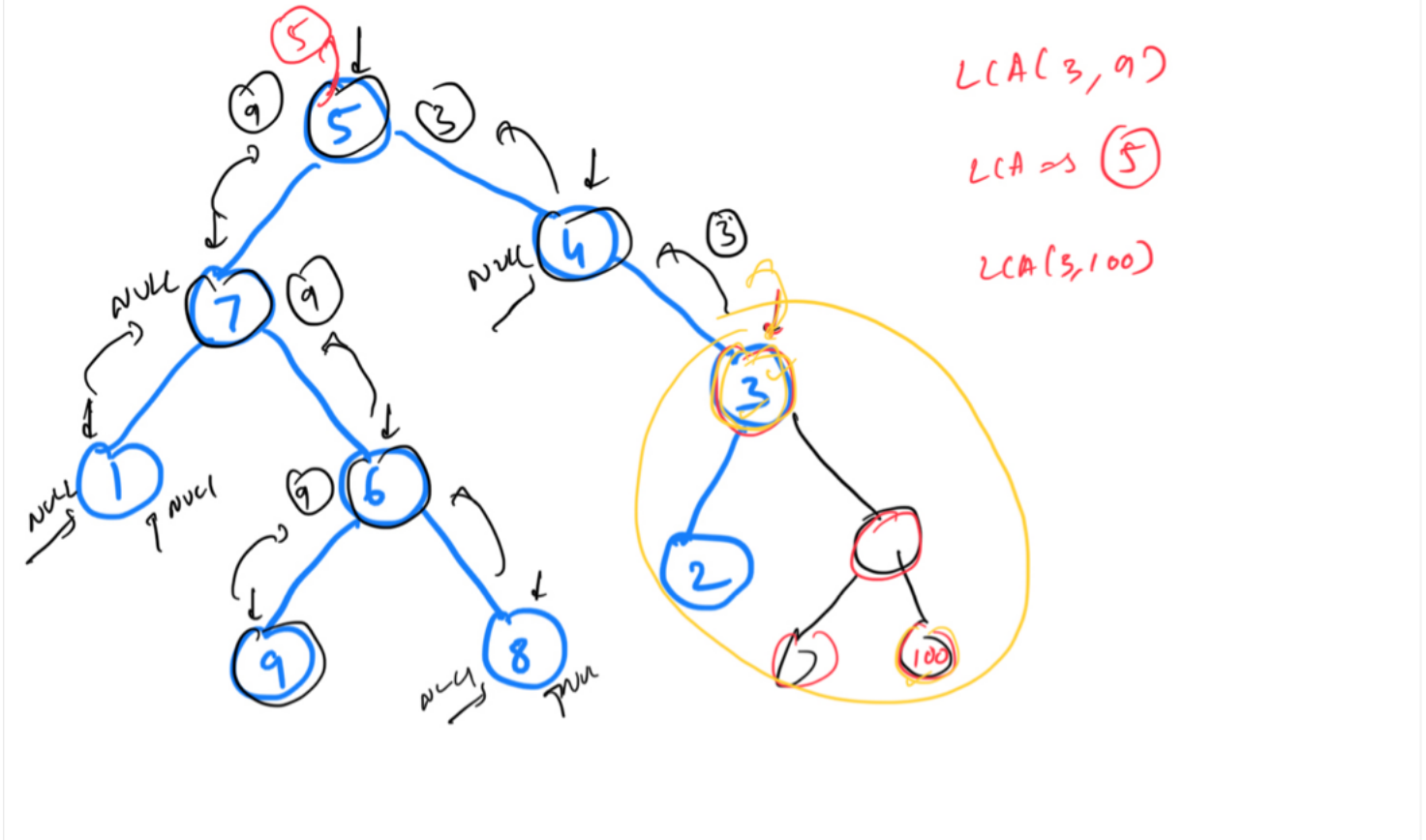
Ances(9) = 5, 7, 6, 9

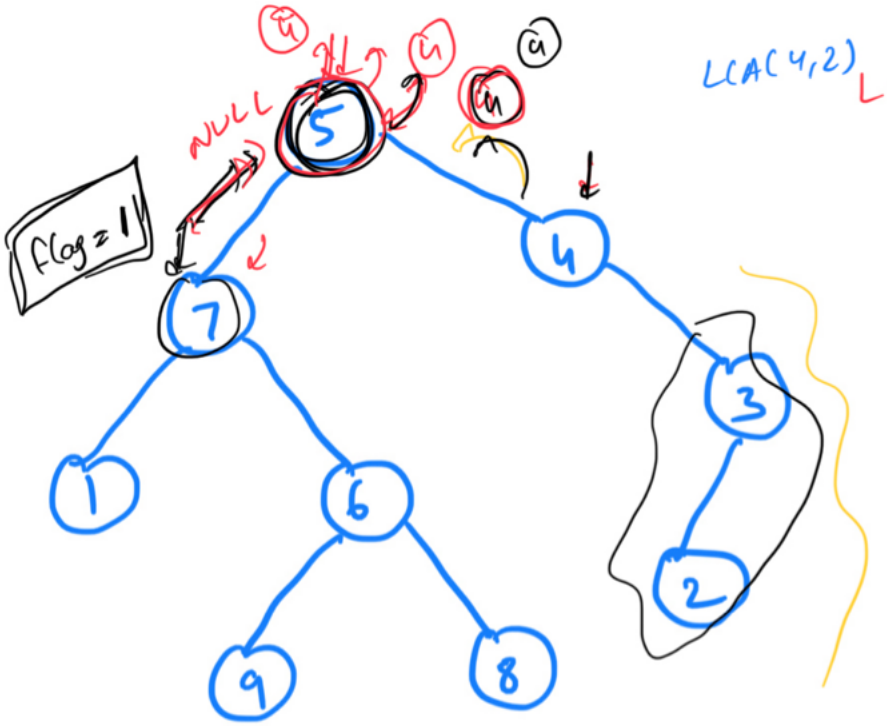
Ans(8) = 5, 7, 6, 8

LCA is the only ancestor node where x lies in one subtree and y lies on other subtree

LCA(4,5)


Preorder
1) LCA(5, 8) return root
2) LCA(1, 8)
LCA = 7
NULL
NULL
NULL
NULL
NULL
NULL
NULL
NULL
NULL
NULL


LCA(3, 9)
LCA = 5
LCA(3, 100)
NULL
NULL
NULL
NULL
NULL
NULL



LCA(4,2) LCA(4,2)

Stop the process

Recursion

Least Amount of Control.

if (flag = =0)
visit Right child

```
func (root) {
        visit root
if(flg==0) visit LST
if(flag==2) visit RST
}
```

Good Practice

```
Node lca ( root, x, y) {
    if(root == NULL) return NULL;

    if(root.data == x || root.data == y)
        return root;

    Node left_lca = lca(root.left, x, y);
    Node right_lca = lca(root.right, x, y);

    if(left_lca != NULL && right_lca != NULL)
        return root;

    if(left_lca == NULL) return right_lca.
    if(right_lca == NULL) return left_lca;
}
```

lca_left || lca.right

Boolean

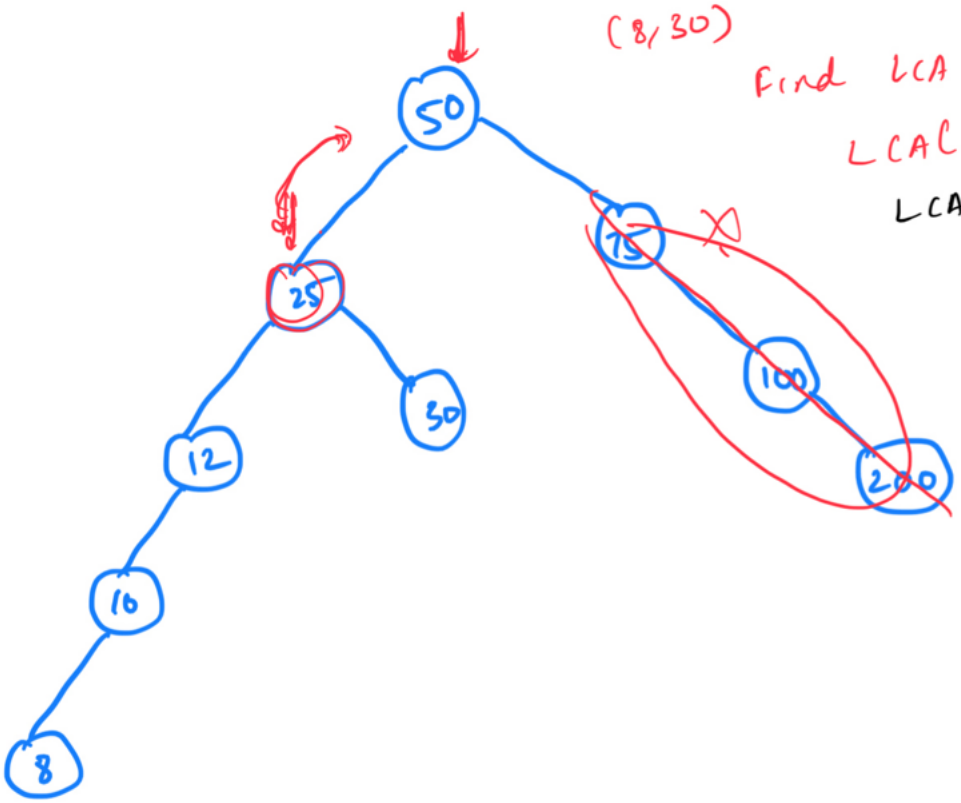
T.C: O(N)

S.C: O(H)

search(x) == T && search(y) == T.

→

Question:



(8, 30)

Find LCA of nodes in BST.

LCA(8, 30) = 25

LCA(75, 200) = 75.

(x, y)

x < y

(8, 50)

```
Node LCA(root, x, y) {
    if(root == NULL) return NULL;
    if(root.data == x || root.data == y)
        return root;
    if(root.data > x && root.data < y)
        return root;
    else if(root.data > y)
        return LCA(root.left, x, y);
    else  // root.data < x
        return LCA(root.right, x, y);
}
```

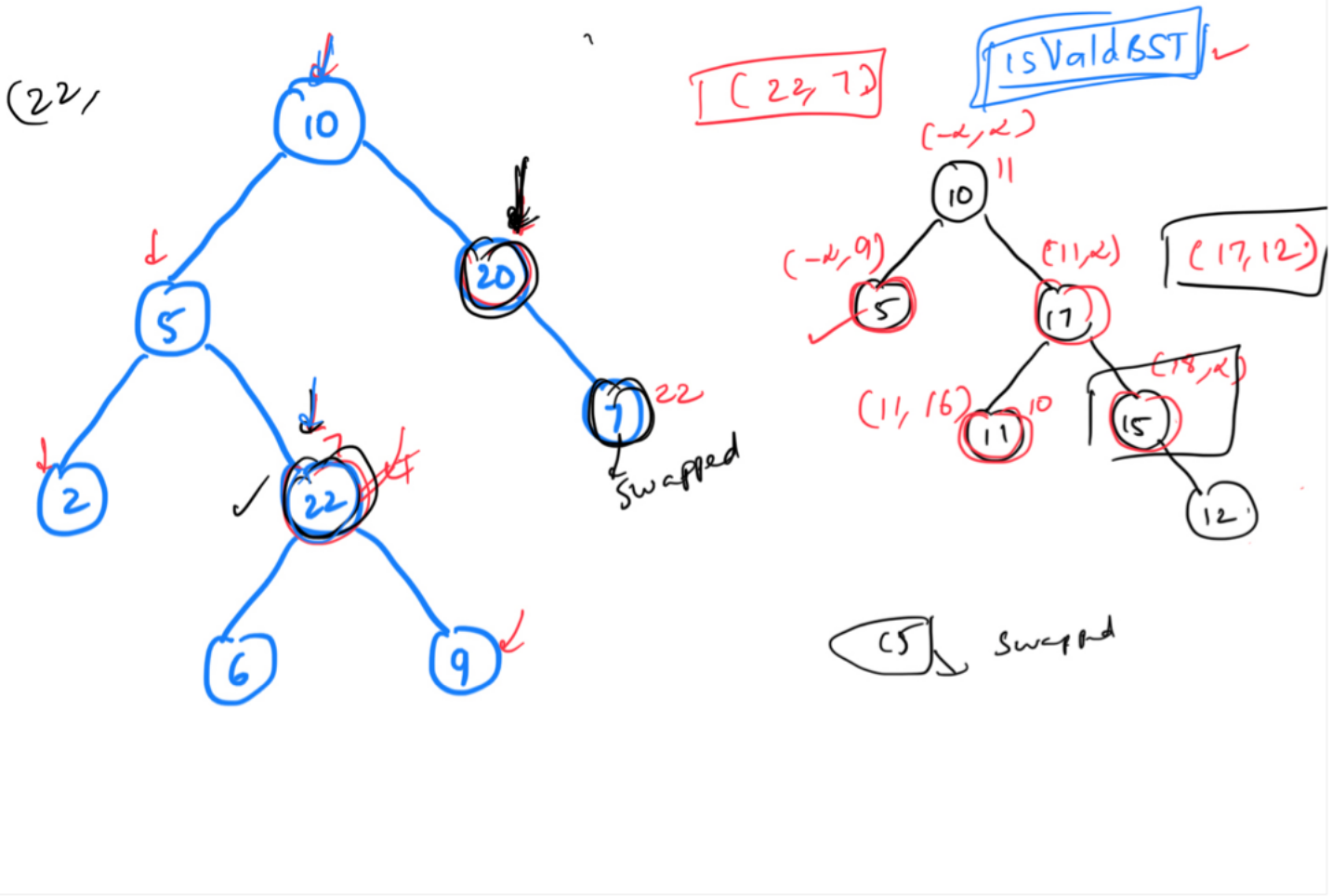
x < y

=> Search is: O(H) for B.S.T.

T.C: O(H)

S.C: O(H) -> Recursion stack.

Question: Recover B.S.T.


(22,
10
5
20
2
22
7
22
swapped
6
9
(22, 7)
isValidBST
(-∞, ∞)
10
11
(-∞, 9)
(11, ∞)
(17, 12)
5
17
(11, 16)
11
10
(18, ∞)
15
12
(5
swapped

Approach1 :

O(n) → Inorder: 2 5 6 22 9 10 20 7

=> sort(Inorder): 2 5 6 7 9 10 20 22

=> O(n)

T.C : O(nlogn)

S.C : O(n)

Approach2 :

Inorder: 2 5 6 22 9 10 20 7

0 1 2 3 4 5 6 7

P1 (22, 9) P2 (20, 7)

a[i] > a[i+1]

Inversion Pairs

if (a[i] > a[i+1]) => (a[i], a[i+1])

Inversion pair

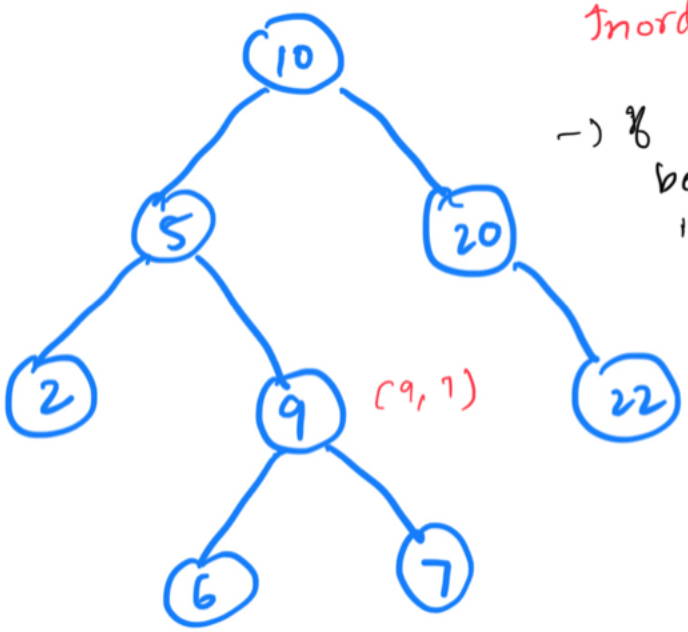
Claim

1) First element of P1

2) Second of P2

map(P1) :

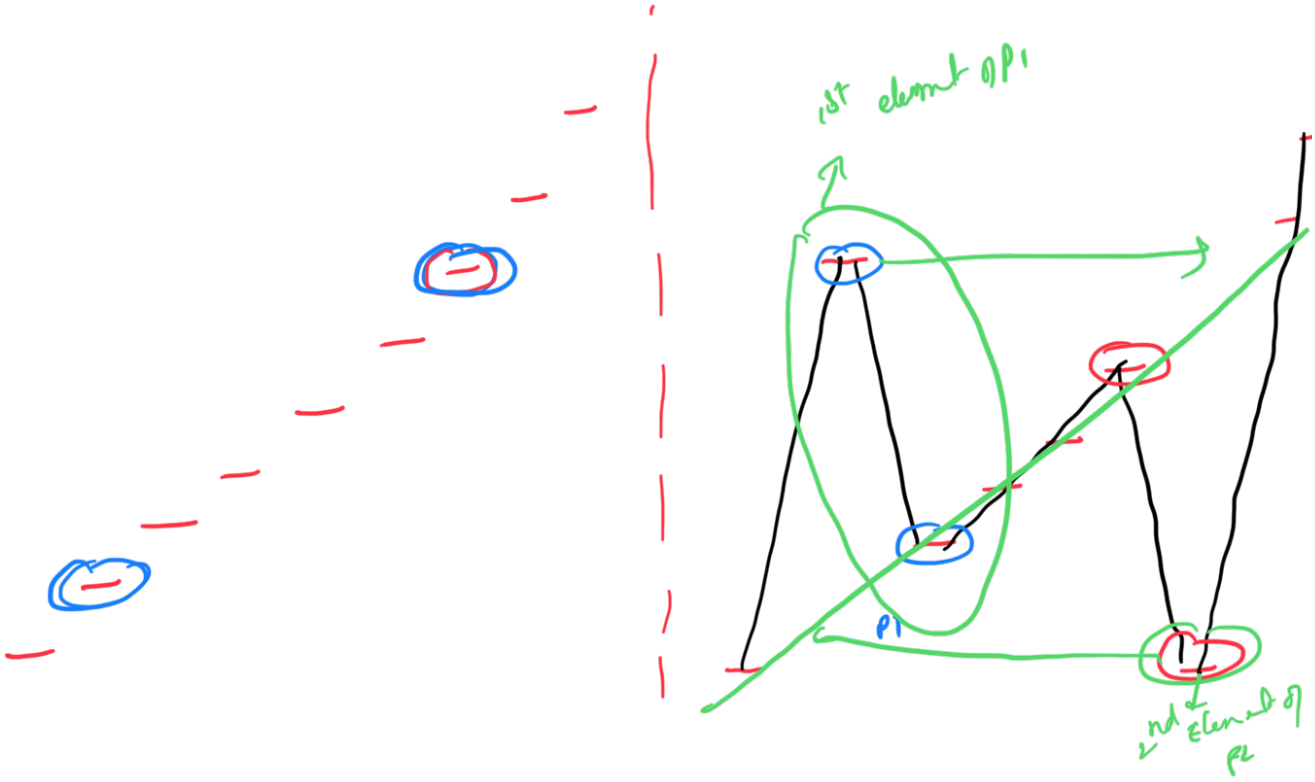
min(P2) :



(9,7) => 1 swapped

Inorder: 2 5 6 9 7 10 20 22

-) If the 2 elements which have been swapped are adjacent in Inorder traversal, then only 1 inversion pair



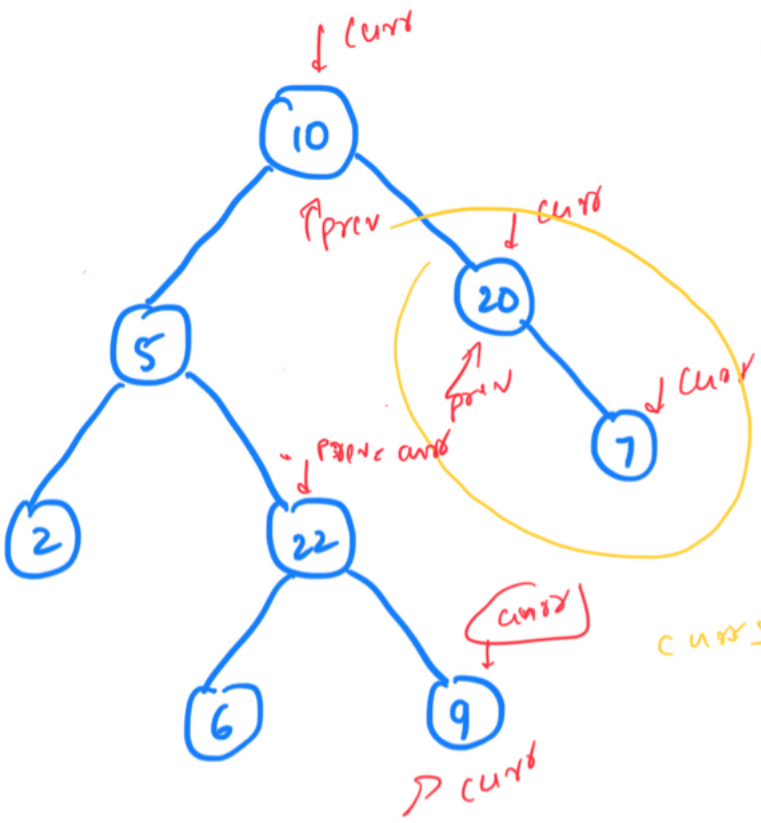
T.C: $O(n)$

S.C: $O(n) + O(H) \Rightarrow O(N)$

$O(n)$ ↓ inorder Traversal

$O(H)$ ↓ Recursion

Approach 3:



maxDiff = prev.data = 22

minDiff = curr.data = 7

(22 > 7)

flag = 1

maxDiff - minDiff = - INF, INF.

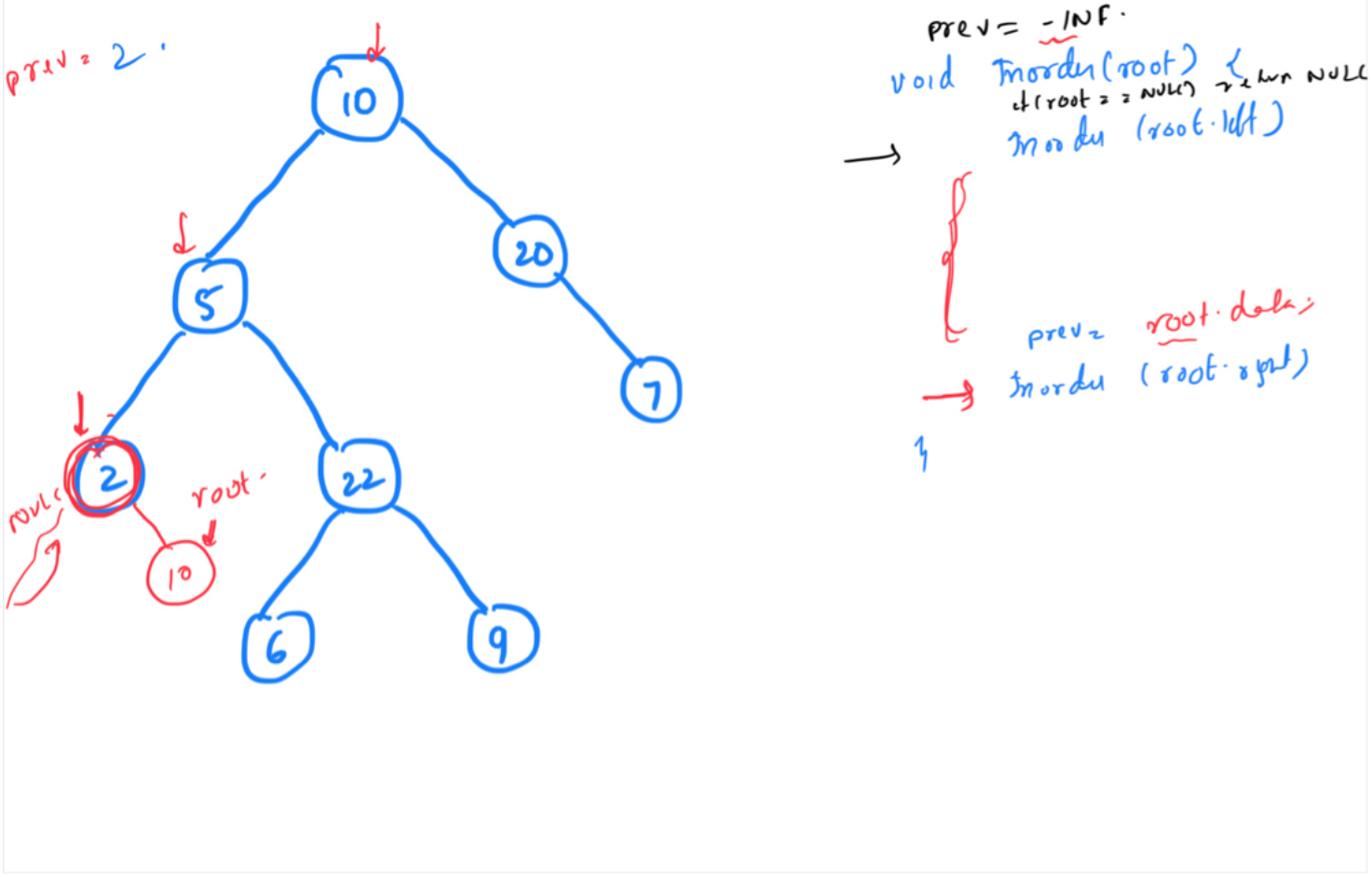
```
int maxDiP, minDiP
int prev = -INF;
flag = 1;

void inorder(root){
    if(root == NULL) return;
    inorder(root.left);
    if(curr.data < prev && flag == 1){
        maxDiP = prev;
        minDiP = curr.data;
        flag = 0
    }
    else if(curr.data < prev && flag == 0){
        minDiP = curr;
    }
    prev = curr.data;
    inorder(root.right);
}

Ans = [maxDiP, minDiP]
```

T.C : O(n)

S.C : O(H) → Recursion stack



LCA

```
Node* lca(Node* root, int x, int y){
      if(root == NULL) return NULL;
      if(root->val == x || root->val == y)
            return root;

      left_lca = lca(root->left, x, y);
      right_lca = lca(root->right, x, y);

      if(left_lca && right_lca) return root;

      if(left_lca != NULL)
            return left_lca;
      else
            return right_lca;
}
```

LCA in BST

```
Node LCA(root, x, y){
      if(root.data == x || root.data == y)
            return root;

      if(root.data > x && root.data < y)
            return root;
      else if(root.data > y)
            return LCA(root.left, x, y);
      else
            return LCA(root.right, x, y);
}
```

Recover a BST

```
int maxdip, mindip,  prev = INT_MIN, flag = true;
void inorder(Node* root){
      if(root == NULL) return;

      inorder(root->left);

      // 1st Inversion pair
      if(prev > root->val && flag){
            maxdip = prev;
            mindip = root->val;
            flag = false;
      }
      // Second pair
      else if(root->val < prev)
                  mindip = root->val;

      prev = root->val;
      inorder(root->right);
}
return [maxDip, minDip]
```