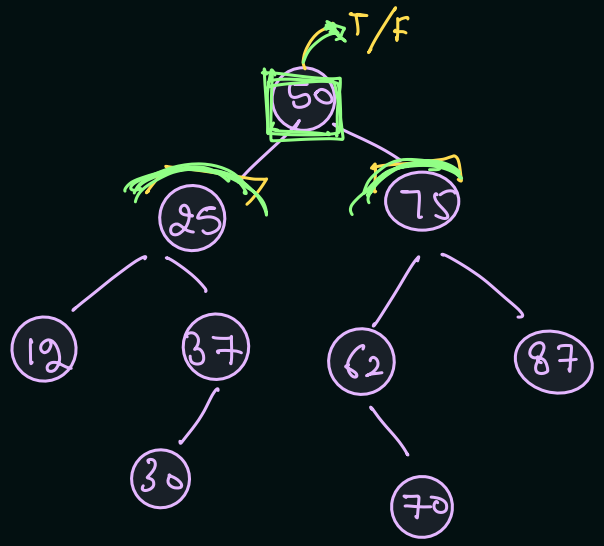
Find in a Binary Tree:

root, dtf → data to find

Find → 37 → dtf



* [Recording ✗]
* [Attend live class] ✓

Expectation: Root] → find ✓

faith :] children] → Guaranteed

Merging:] Merge faith & Expectation

Expectation.: find(root, 37] → True / False

faith →
- find(root.left, 37) → T/F
- find(root.right, 37) → T/F

Merging. → A1: Self check

A2 = left check

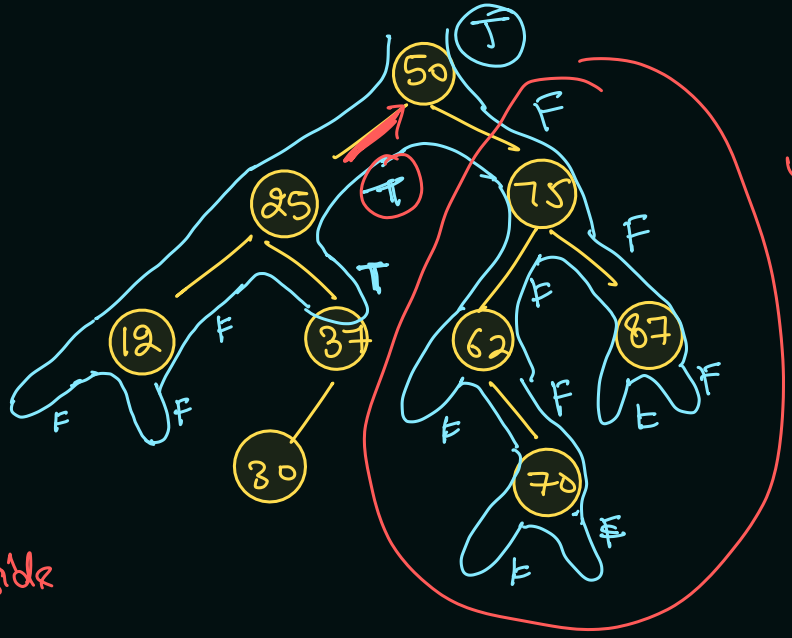
A3 Right check

A3 ↳ these are Boolean variable

, res = A1 || A2 || A3 ; (True.)

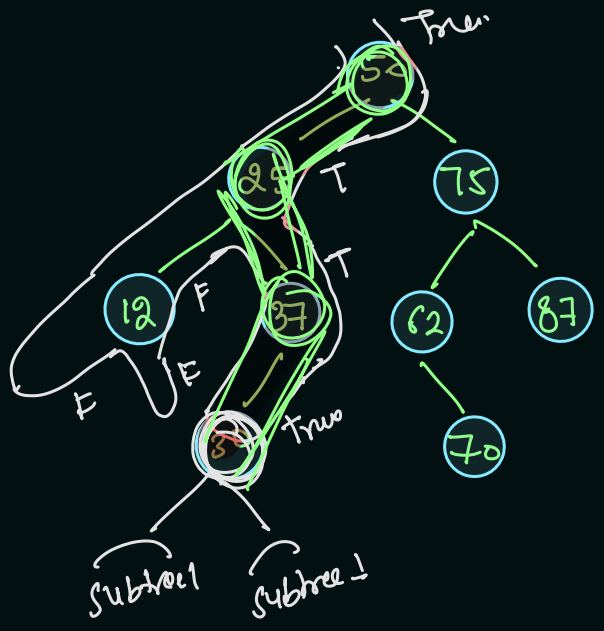
```java
public static boolean find(Node node, int data){
    if(node == null) return false;

    if(node.data == data) return true;      → self check

    boolean lres = find(node.left, data);   → left call
    boolean rres = find(node.right, data);  → right call
    return lres || rres;
}
```

dtf = 37



wastage of time,

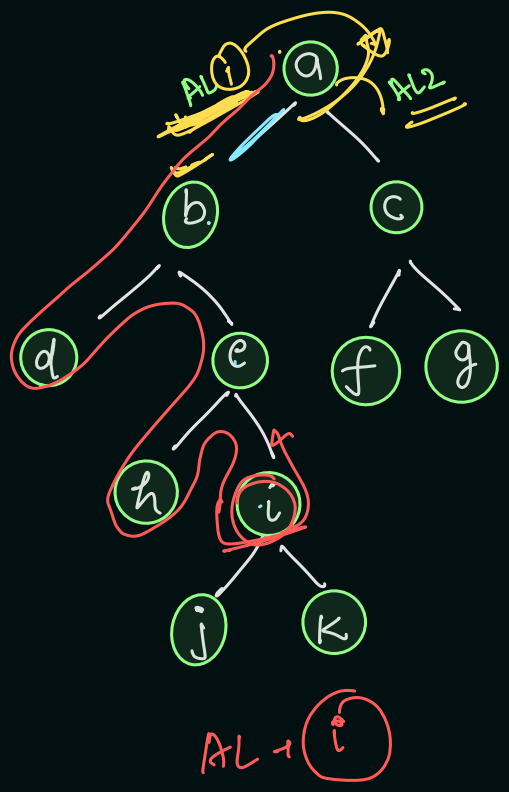
NOTE: if data is found in left side then no need to move Right side



dtf = 30

```java
public static boolean find(Node node, int data){
    if(node == null) return false;

    if(node.data == data) return true;

    boolean lres = find(node.left, data);
    if(lres == true) {
        return true;
    }
    return find(node.right, data);
}
```

Node to Root Path



Node to Root path:

data = i

i e b a → i e b a

Expectation → node to Root path (Root, data) → path in AL

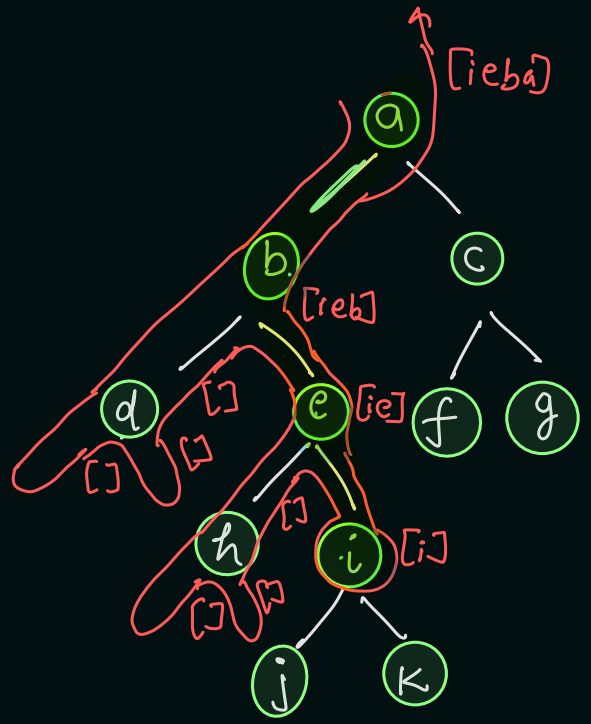
faith → node to Root path (root.left, data) → AL① → i, e, b

node to Root path (root.right, data) → AL② → blank List OR null

AL 1 → [i e b] → data is found in left subs

AL 2 → [blank List OR null]

data = i

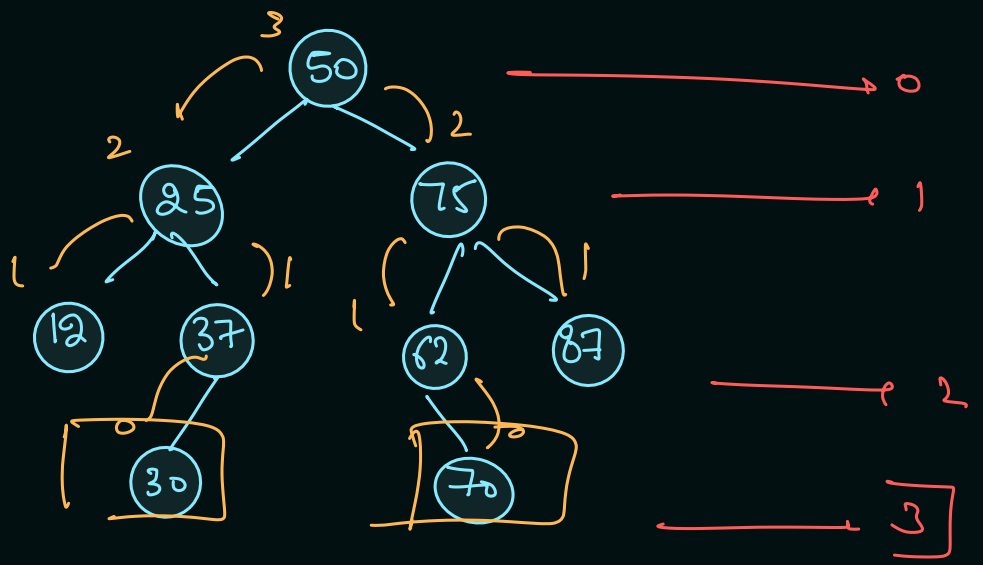



```java
public static ArrayList<Integer> nodeToRootPath(Node node, int data){
    if(node == null) return new ArrayList<>();

    if(node.data == data) {
        ArrayList<Integer> mres = new ArrayList<>();
        mres.add(node.data);
        return mres;
    }

    ArrayList<Integer> lres = nodeToRootPath(node.left, data);
    if(lres.size() > 0) {
        lres.add(node.data);
        return lres;
    }
    ArrayList<Integer> rres = nodeToRootPath(node.right, data);
    if(rres.size() > 0) {
        rres.add(node.data);
        return rres;
    }
    return new ArrayList<>();
}
```

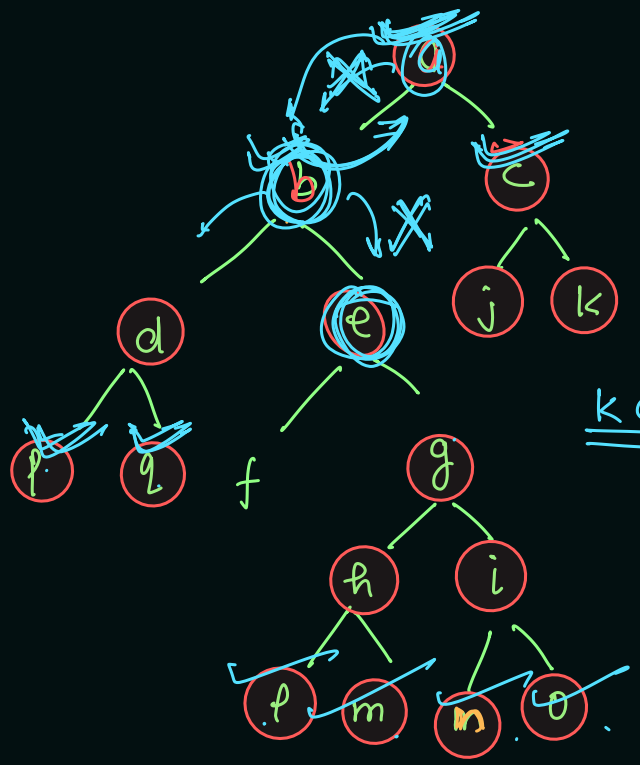
Print k level down

for k=3

Nodes → 30 , 70



Print k-distance away / k-far



k down.

data = e

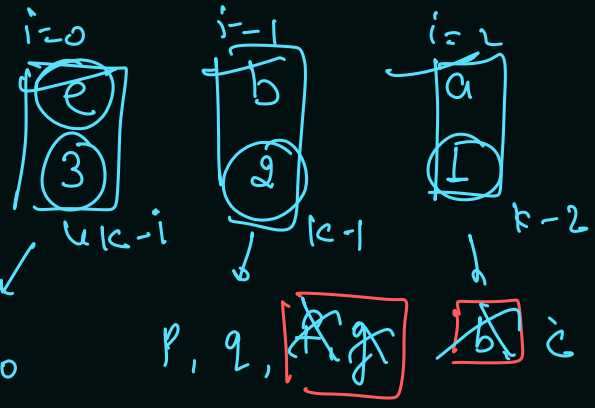
k = 3

k-far → l , m , n , o , p , q , c

node to Root path →

k down , k =

k-i =



l, m, n, o

p, q, ~~g~~

~~b~~ c

```java
public static void printKNodesFar(Node node, int data, int k) {
  ArrayList<Node> n2rp = nodeToRootPath2(node, data);
  Node blocker = null;
    for(int i = 0; i < n2rp.size() && k - i >= 0; i++) {
      Node root = n2rp.get(i);
        printKLevelsDown2(root, k - i, blocker);
        blocker = root;
    }
}
```

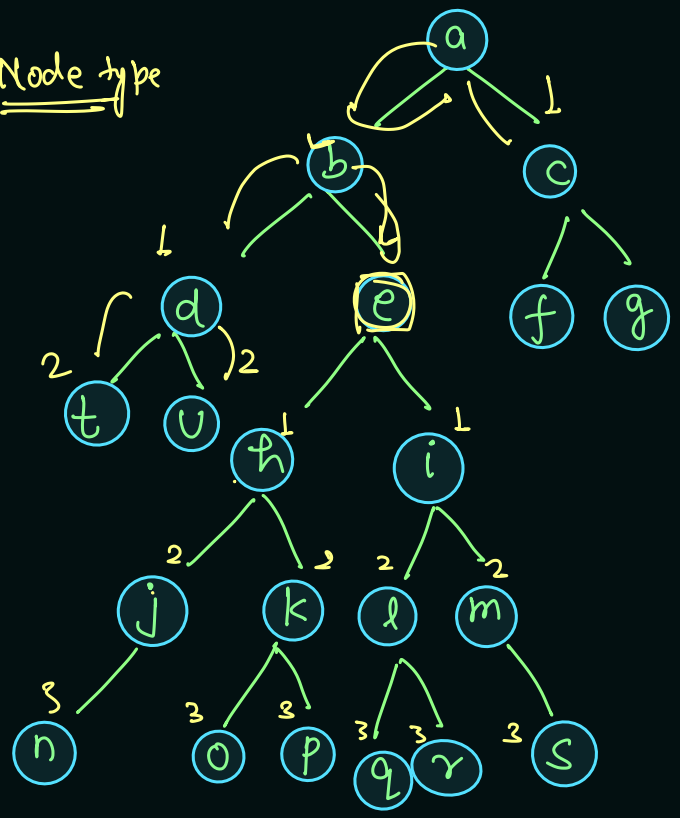
→ Return Node type

data = e, k = 3

e	b	a

i =	0	1	2	3
k-i =	3	2	1	
root =	e	b	a	

blocker = ~~null~~ e b a

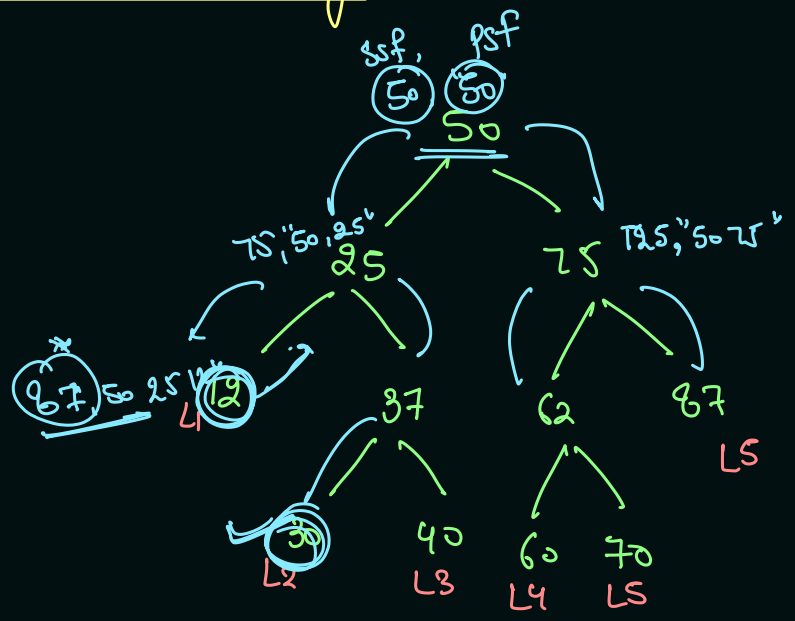


n o p q r s t u c

Path to Leaf from Root, In Range;

lo = 150

hi = 250



Variable → Int
- ssf → sum so far
- psf → path so far
- String.

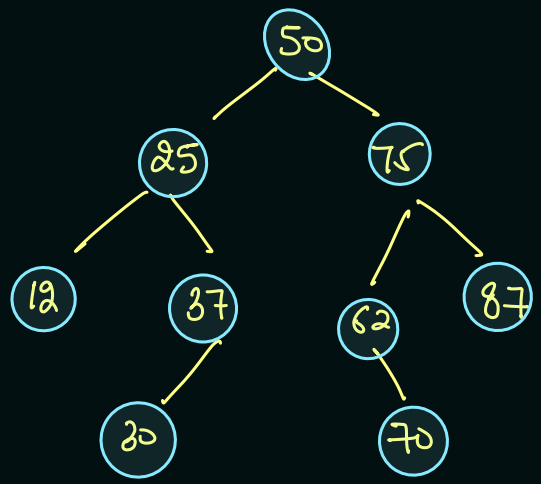
Root to Leaf paths → Path sum

P1	50 → 25 → 12	87	✗
P2	50 → 25 → 37 → 30	142	✗
P3	50 → 25 → 37 → 40	152	✓
P4	50 → 75 → 62 → 60	247	✓
P5	50 → 75 → 62 → 70	257	✗
P6	50 → 75 → 87	212	✓

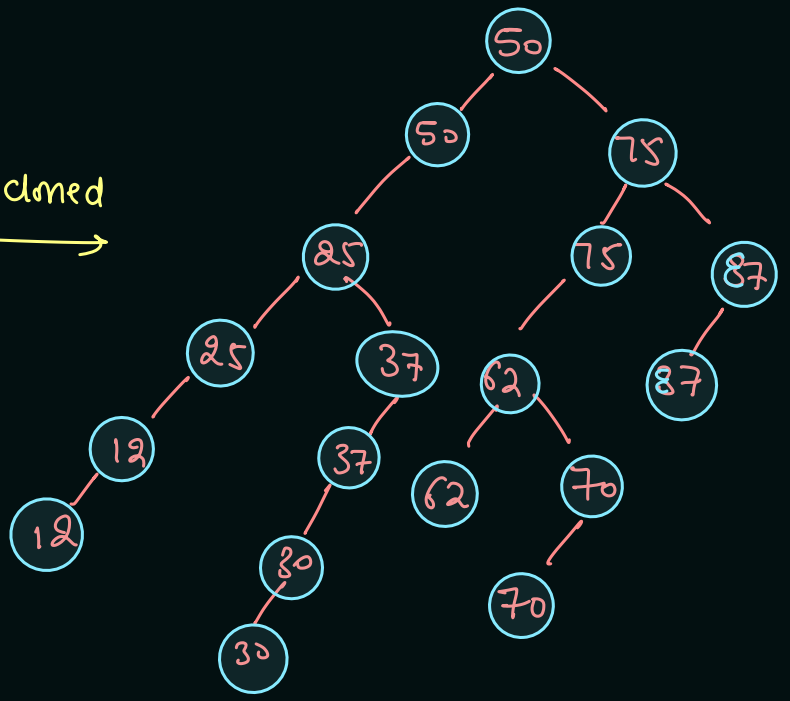
Valid Range: 150 ≤ path sum ≤ 253

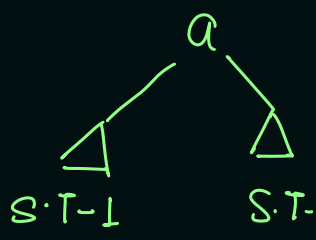
- P3 →
- P4 →
- P6

Transform to left cloned Tree:



left cloned



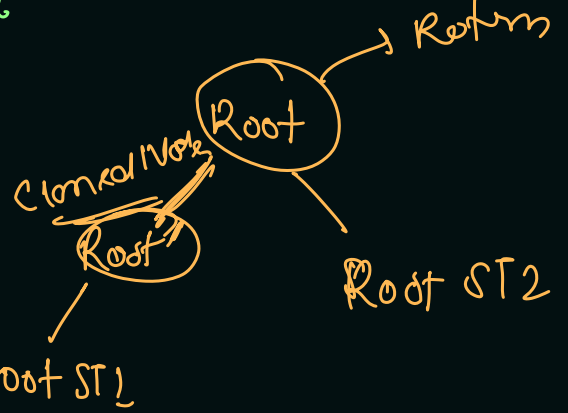


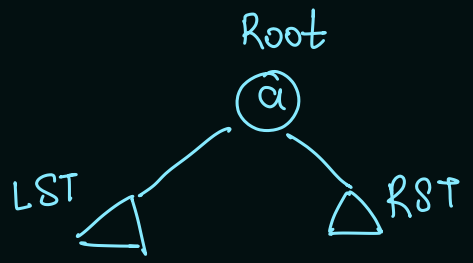
Expectation → leftclone(50)

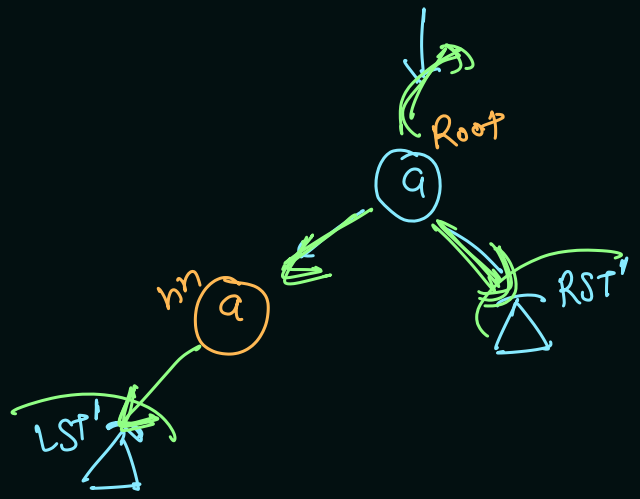
faith → root of ST1 = [left clone(ST1)

root of ST2 = [left clone(ST2)

Merging → cloned Root Node.





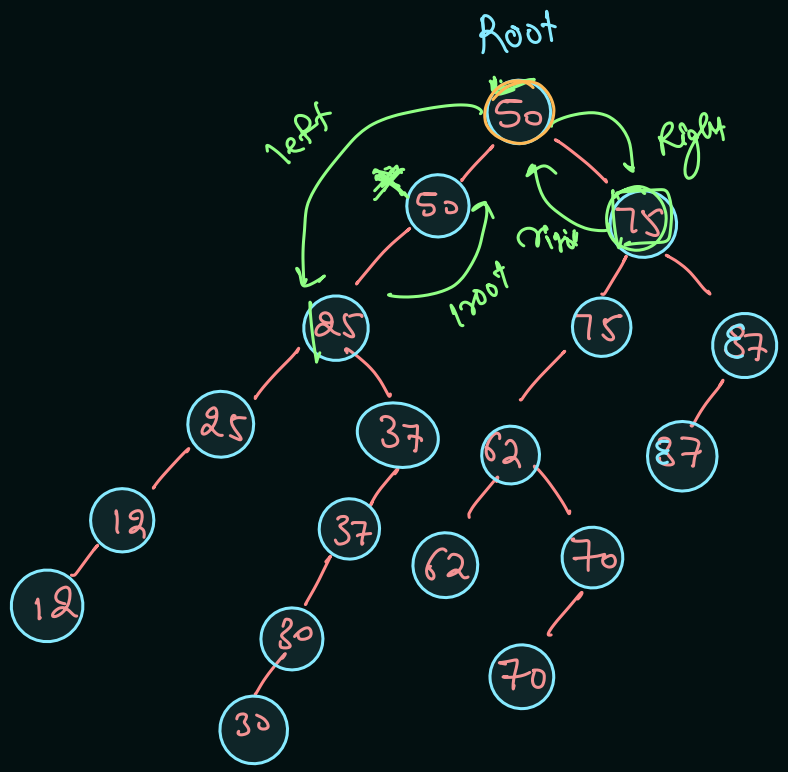



```java
public static Node createLeftCloneTree(Node root){
    if(root == null) return null;

    Node leftRoot = createLeftCloneTree(root.left);
    Node rightRoot = createLeftCloneTree(root.right);

    Node nn = new Node(root.data);

    nn.left = leftRoot;
    root.left = nn;
    root.right = rightRoot;

    return root;
}
```

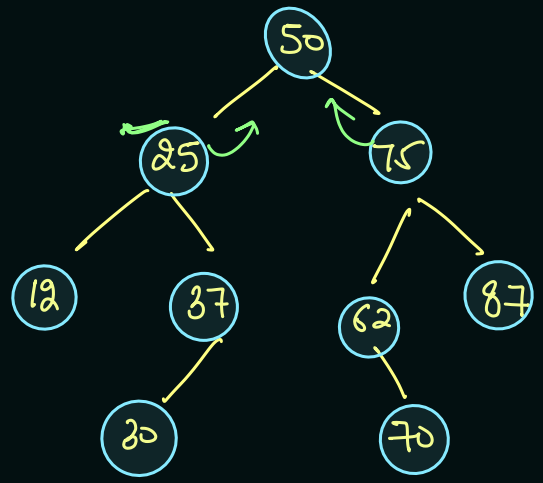
LST → left subtree

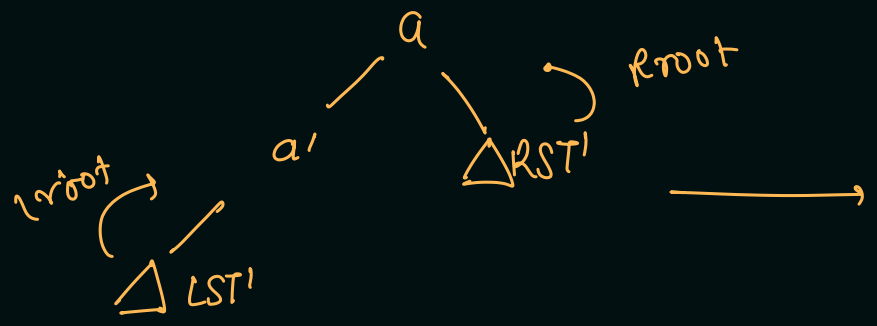
RST → Right subtree

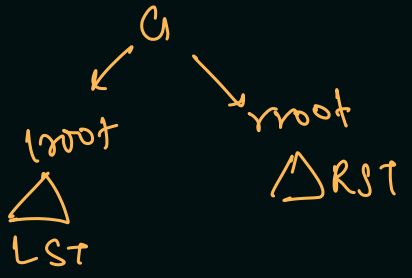
original tree from left cloned tree:



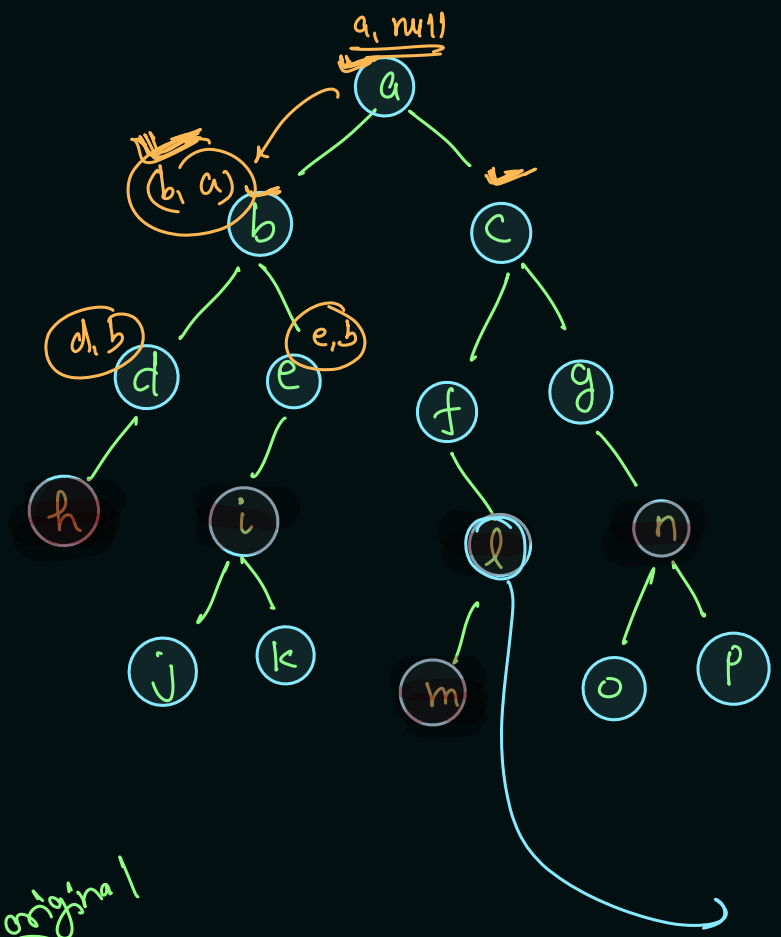
Convert to original tree







Print Single child :



Single child → h, i, l, n, m

next → Remove Leaves -

to verify single child → when we have address of parent

original

Root have no parent

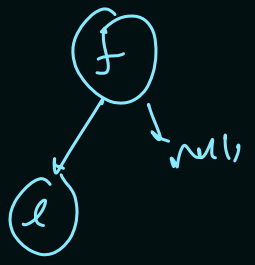
Root,

parent = null

parent != null

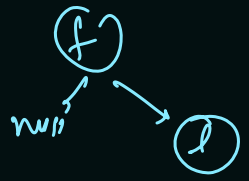
root = l, parent = f

(f.left != null && f.right == null) ||

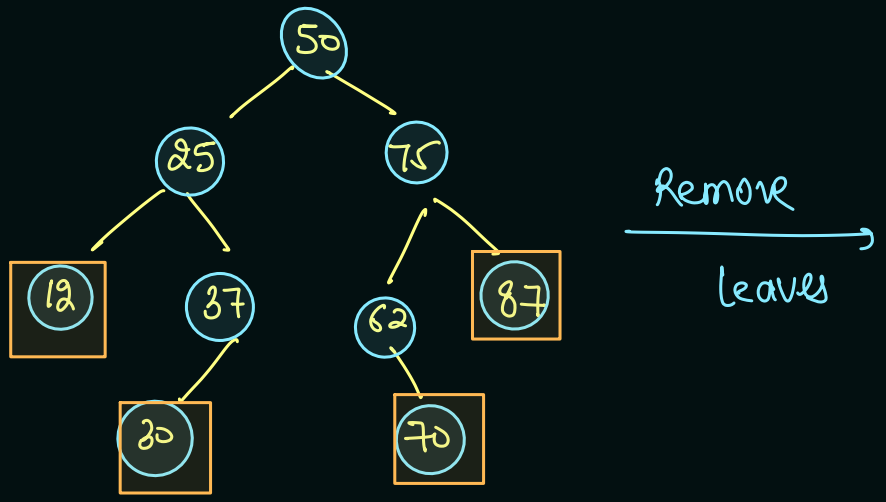


(f.right != null && f.left == null)

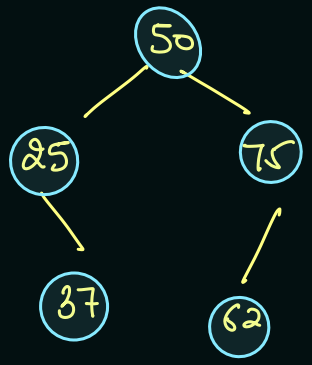
l is a single child



Remove leaves:→



Remove leaves



Traversal code of tree →

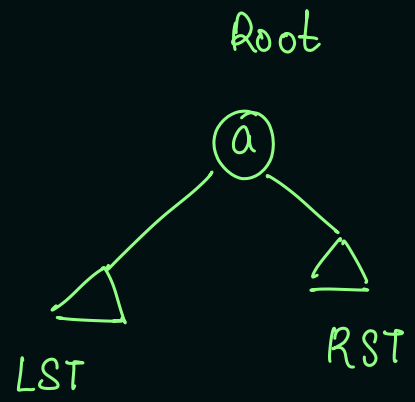
```java
// call -> foolish, base case-> smart
public static void traverse1(Node node) {
    if(node == null) return;

    traverse1(node.left);
    traverse1(node.right);
}
```

```java
// no base case, every call is smart
public static void traverse2(Node node) {
    if(node.left != null)
        traverse1(node.left);
    if(node.right != null)
        traverse1(node.right);
}
```

```java
public static void traverse3(Node node) {
    if(node.left != null && node.right != null) {
        // left + right both exist
        traverse3(node.left);
        traverse3(node.right);
    } else if(node.left != null) {
        // left exist
        traverse3(node.left);
    } else if(node.right != null) {
        // right exist
        traverse3(node.right);
    } else {
        // no child exist, yopu are at leaf
        // no call
    }
}
```

Remove Leaves -

Expectation → removeLeaves(a)] → Remove all leaves



faith → removeLeaves(a.left) → LST] remove.

removeLeaves(a.right) → RST]

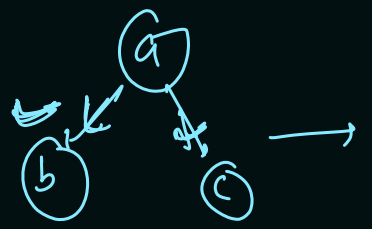
Merging → Self check] if iam leaf.

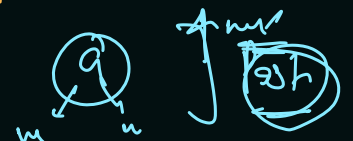
remove.

newRoot] return] null

Area of work] → Post Area] X

Pre Area] left level] do





Schedule → Mon - Fri → [10 to 2 -]