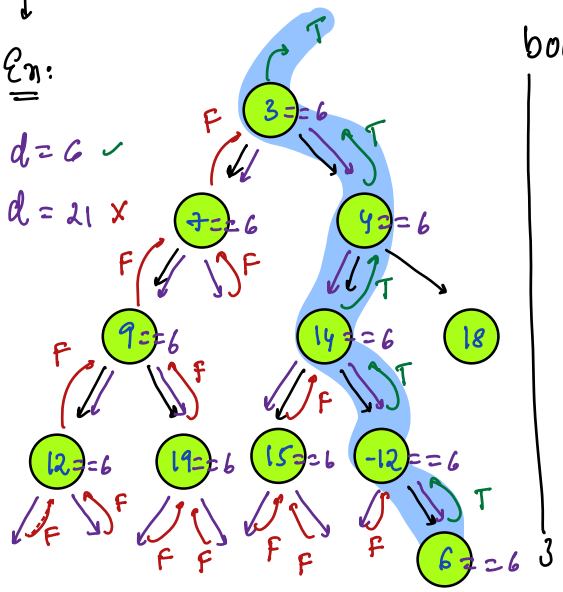
Todays Content:

- → Path from root to given node ✓
- → LCA in Binary Tree ✓ [9:50 → 10 pm
- → No: of Nodes at k from given node ✓
- → Recover BST ✓

Q1: Given a B.T which contains all unique values, Search if there exists a d in B.T

Ass: Given tree, search for d return T/F




```
bool check(Node root, int d){ TC:O(N) SC:O(H)
    if(root == null){ return false}
    if(root.data == d){ return True}
    if( check(root.left,d) ||
                check(root.right, d))
        return true
    }
    return false
}
```

Obs: If we store all nodes, which returns true, it gives path between source & root node.

```
list<Node> path; // path between root & source (d)

bool getPath(Node root, int d)  TC:O(N) SC:O(H)
    if(root == null){ return false}
    if(root.data == d){ path.add(root), return True}
    if( check(root.left,d) || check(root.right, d))
        path.add(root) // this node is returning true hence it
        return true      belongs in path
    }
    return false
}
```

Note: Path we get in above example:

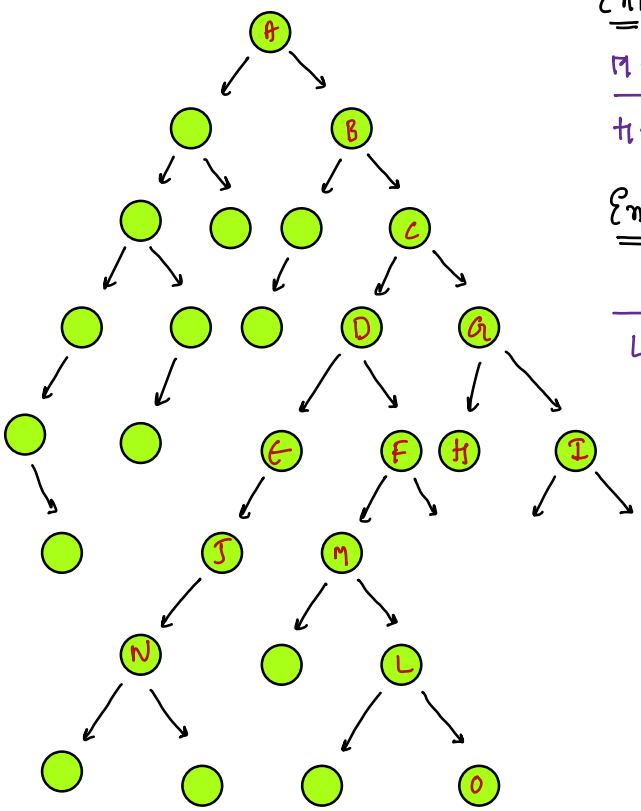
6 -12 4 4 3

↳ // path is stored from source to root

↳ If root is source is need: Reverse the list

LCA: **Lowest common ancestor**: lowest common ancestor is defined between 2 nodes p & q as the lowest node in Tree, which is an ancestor to both p & q.

Note: We allow a node to be ancestor to itself



Ex1: LCA (M, H): C

M: M F D C B A

H: H G C B A

Ex2: LCA (J, L): D

J: J E D C B A

L: L M F D C B A

Ex3: LCA (D, C): C

D: D C B A

C: C B A

All Node are distinct

Node LCA (Node root, int p, int q) { TC: O(N + N + H) → TC: O(N+H)

SC: O(H + H) → O(H)

Step1: Get path from

a) p → root : stored in list<Node> Path p

b) q → root : stored in list<Node> Path q

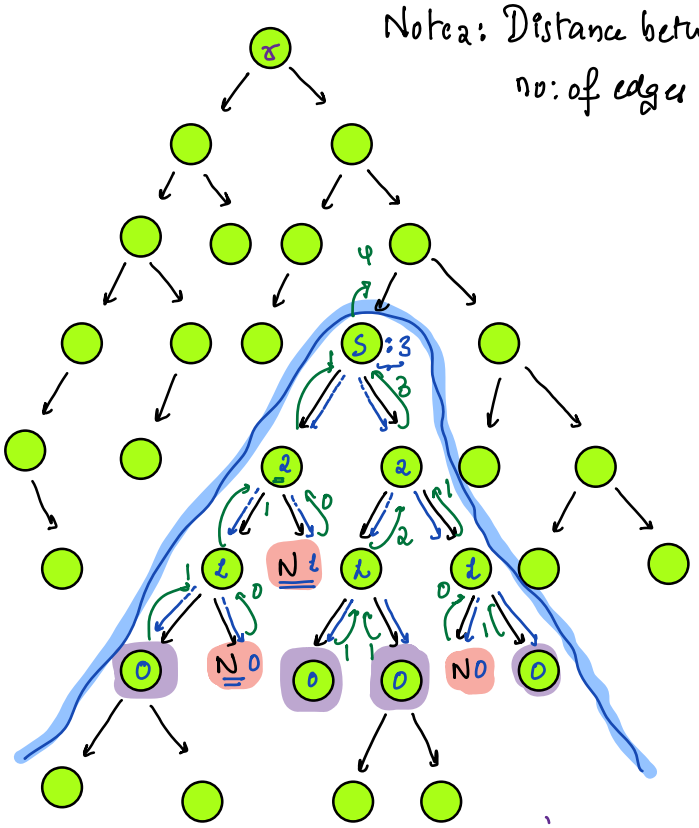
Step2: From both paths iterate from back & get last node which is matching.

}

Q2: Given a sourcenode (address of node) how many nodes are there at distance = k,

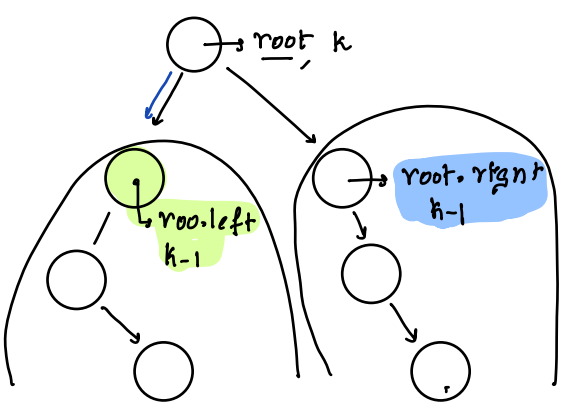
Note1: All nodes should be below source

Note2: Distance between nodes calculate no: of edges between them




```
int below(Node s, int k){
    if(s==null) {return 0;}
    if(k==0) {return 1;}
    int l = below(s.left, k-1)
    int r = below(s.right, k-1)
    return l+r
}
```

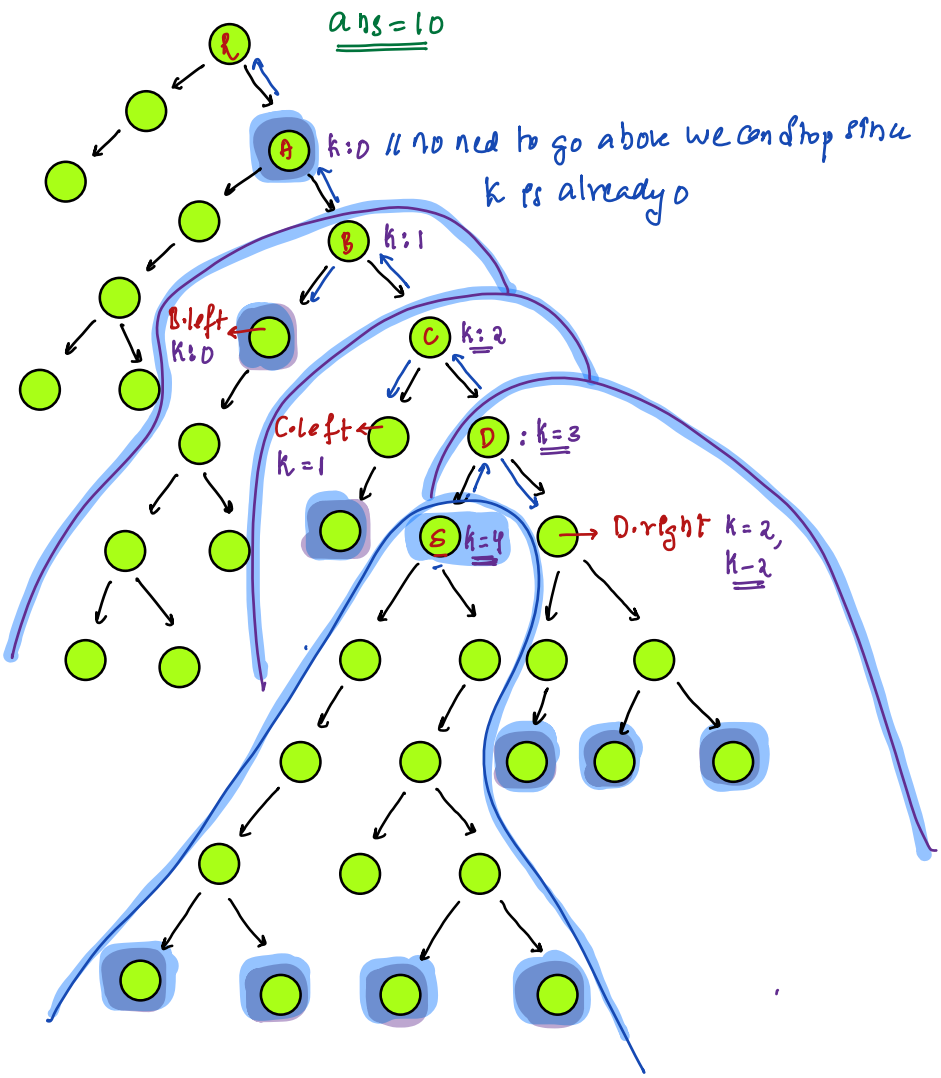
TC: O(N) SC: O(H)



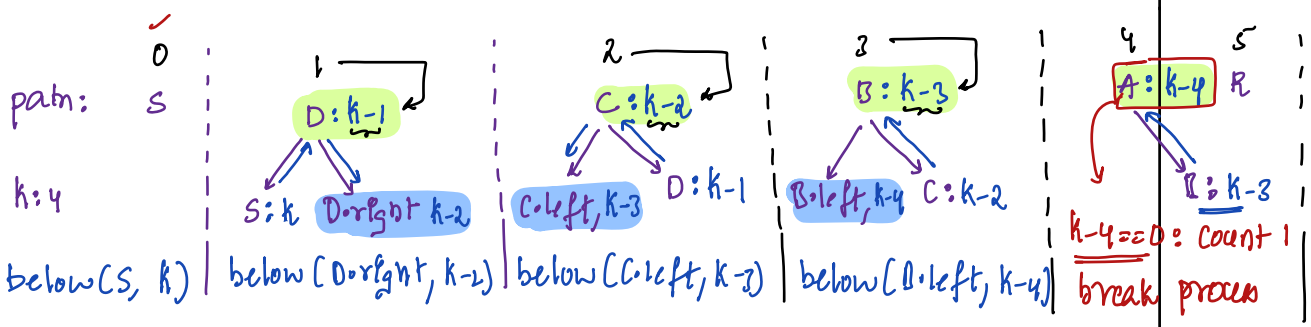
38) Calculate no: of nodes are at a distance k from Source

→ Note: Only Source node value is given, we need to search it first

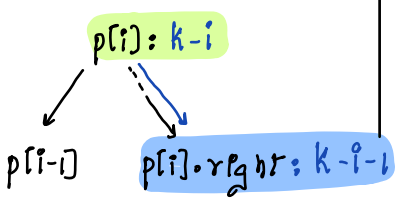
→ Note2: Binary Tree contains only distinct values



Step1: Get path from src → root, we store address



```
int countNodes(Node r, int s, int k){   TC: O(N)  SC: O(H)
    //Step1: Path from src -> root
    List<Node> p
    getPath(r, s)// We have path from src to root in p
    int c = below(p[0], k);
    for(i=1; i<p.size(); i++){
        //node at p[i], distance of node = k-i
        if(k-i==0){ c = c+1, break}
        //node at p[i] child is at p[i-1]
        if(p[i].left == p[i-1]){//goto right & search
            c = c+below(p[i].right, k-i-1)
        }
        else
            c = c+below(p[i].left, k-i-1)
        }
    }
    return c;
}
```

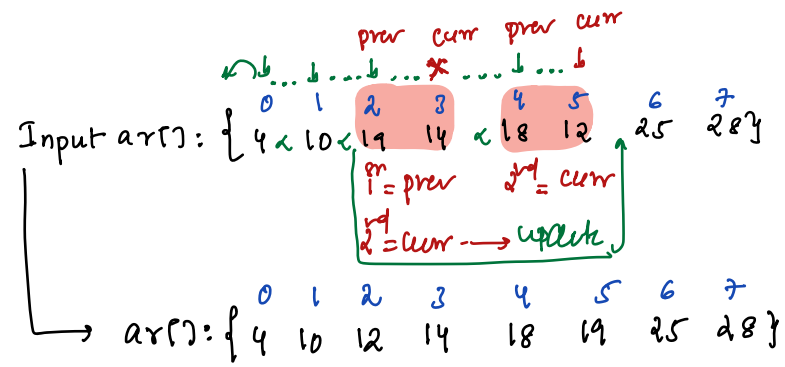


Recover Sorted arr[]

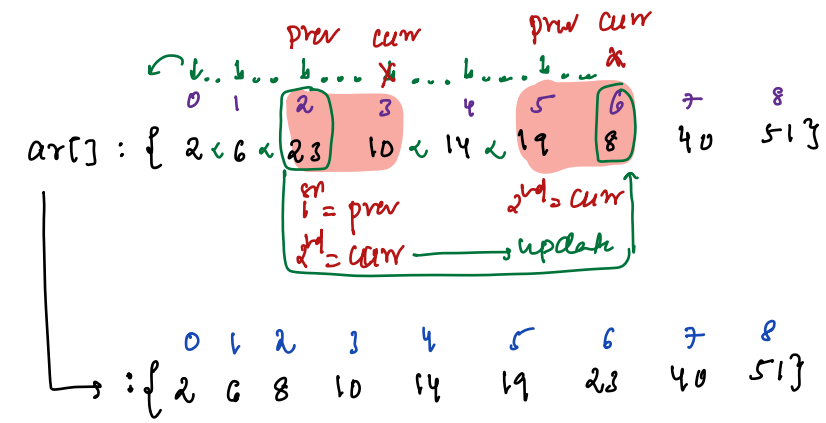
Given an arr[], which is formed by swapping 2 distinct index positions in a sorted {inc} arr[] : {1 swap}

get original sorted arr[] → TC: O(N)

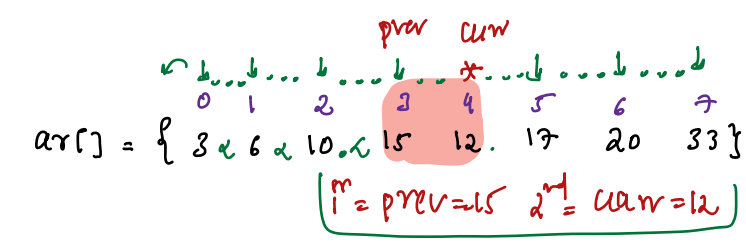
Eg1:



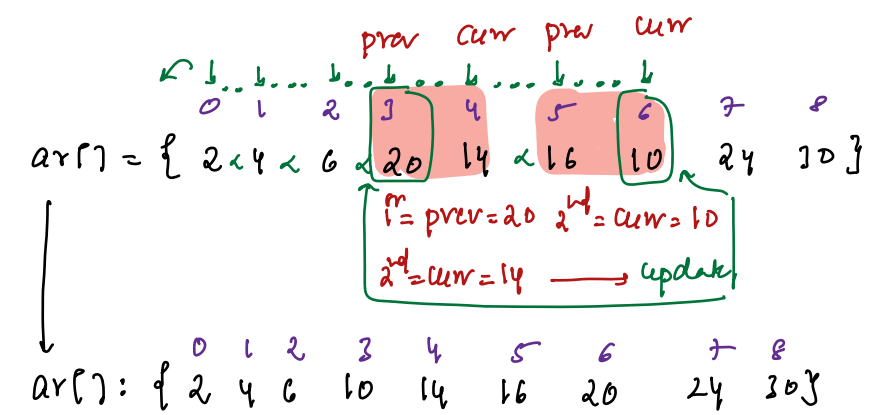
Eg2:



Eg3:

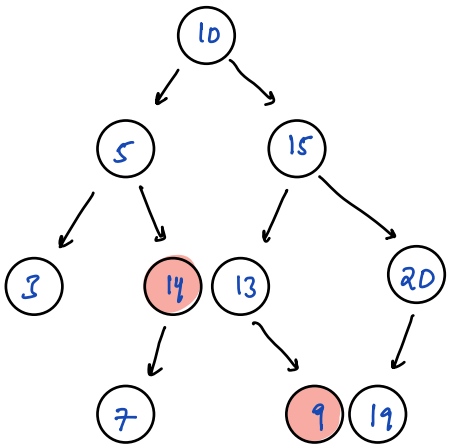


Eg4:

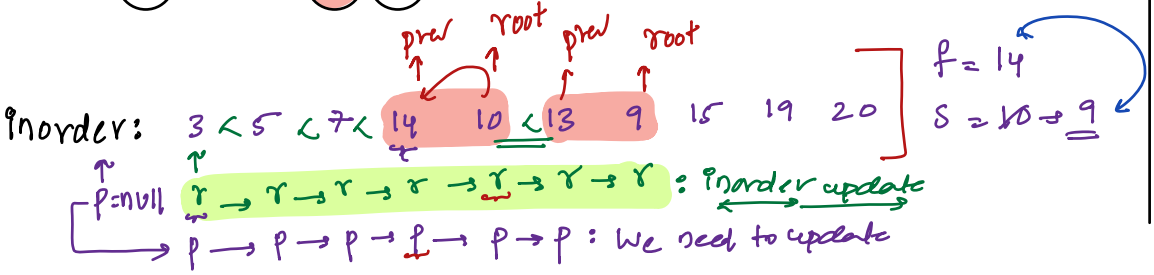


Recover BST :

Given a BST, which is formed by swapping 2 distinct nodes, recover original BST



Idea: Find both nodes for which we need to swap data



Node f = null s = null p = null

void inorder(Node root){ TC: O(N) SC: O(H)

```
if(root==null){return}
inorder(root.left)
if( p!=null && p.data > root.data){
    if( f==null){// it happened 1st time
        f=p, s=r
    }
    else {// it happened 2nd time s= r}
}
p = root // updating previous
inorder(root.right)
}
Swap(f.data & s.data)
```