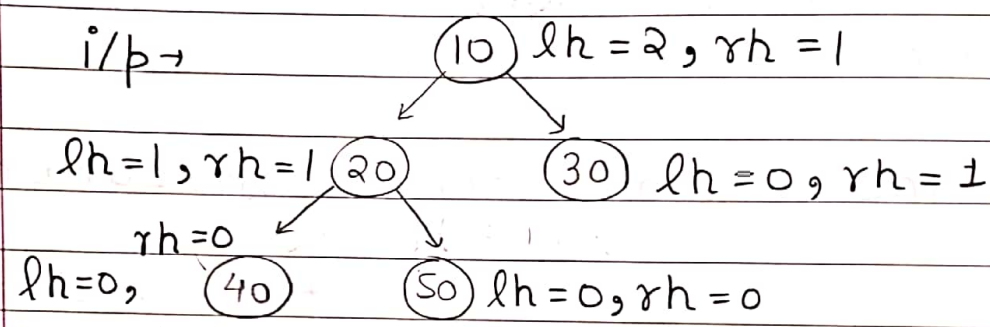
07/05/2023

Q1 Balanced Binary Tree.



Root (10) ← current node

lh − rh = 2 − 1 = 1 <= 1 } true ans1 = true.

Node (20) ← current node

lh − rh = 0 <= 1 } true ans1 = true

Node (40)

lh − rh = 0 − 0 = 0 <= 1 } true, ans1 = true

Hence leftAns for 20 is true. Similarly rightAns for 20 will also be true. Hence all 3 answers are true and hence leftAns for 10 is true.

Node (30)

Right ans for 10 is true and hence all ans1, leftAns and rightAns are true. Hence balanced

Code

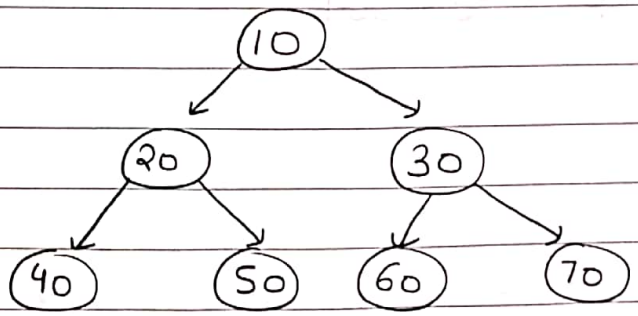
```
bool isbalanced (Treenode * root){
    //Base case → Tree empty
    if (root == NULL)
        return true;
    //Solve 1 case
    int lh = height (root → left);
    int rh = height (root → right);
    int diff = abs (lh - rh);
    bool ans1 = (diff <= 1);
    //Recursion will handle
```

```
bool leftAns = isBalanced(root->left);
bool rightAns = isBalanced(root->right);
//ans1, leftAns & rightAns all are true
if(ans1 && leftAns && rightAns)
        return true;
else
    return false; //Not balanced
}
```

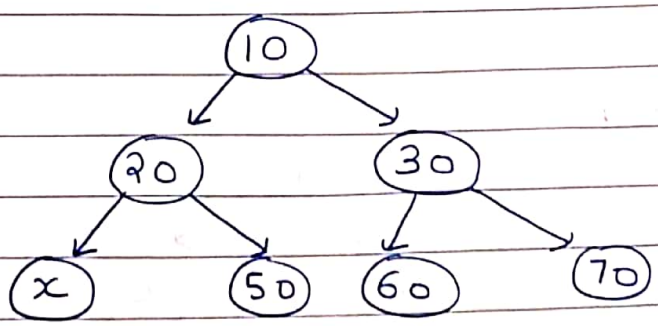
Definition of balanced binary tree
A balanced binary tree is the one in which for every node height of left & right tree for any node does not differ by more than 1.

Q2 Convert a tree to the sum tree
Sum tree is basically replacing the current node value with the data of leftChild + rightChild + current node.

i/p →

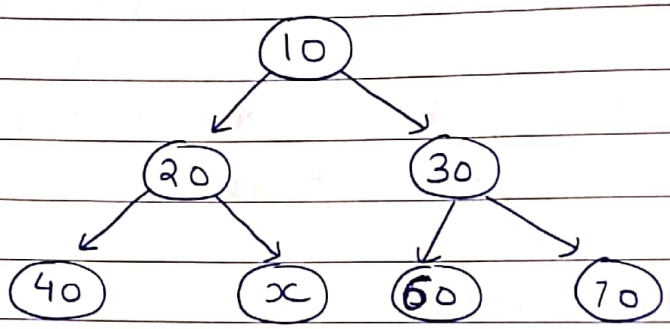


1st step



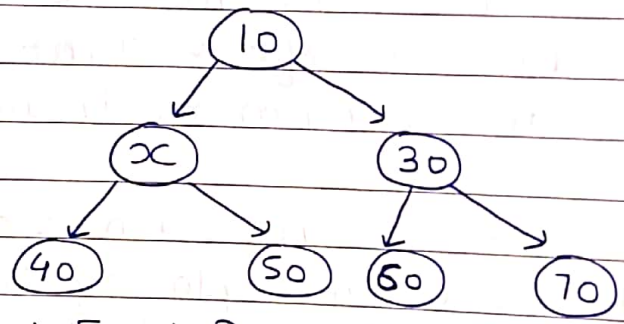
x = 40 + 0 + 0 = 40

2nd step



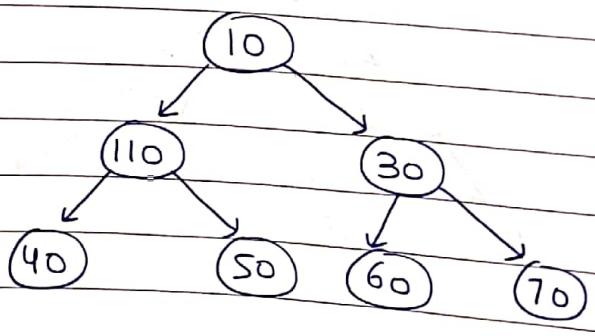
$x = 50 + 0 + 0 = 50$

3rd step

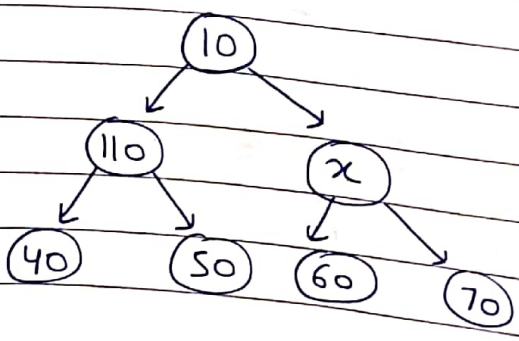


$x = 40 + 50 + 20 = 110$

4th and 5th step

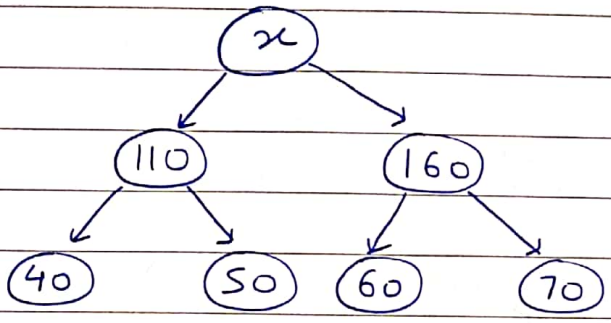


6th step



$x = 60 + 70 + 30 = 160$

7th Step



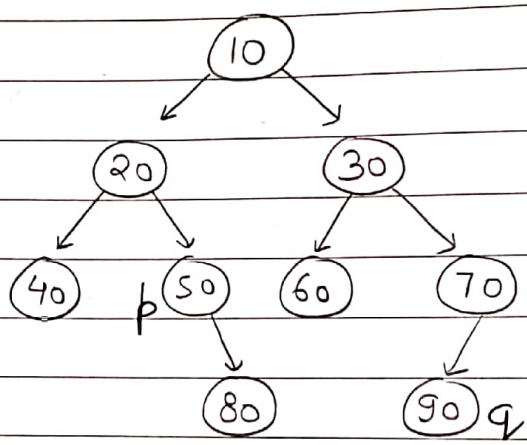
$x = 110 + 160 + 10 = 280$

Code

```
int convertToSumTree (Node* root){
    //Empty Tree
    if (root == NULL){
        return 0; //Like we do at leaf node
    }
    //Find left Ans        ToSumTree
    int leftAns = convert (root→    left);
    //Find rightAns
    int rightAns = convertToSumTree (root→right);
    //Update the node
    root→data = root→data + leftAns + rightAns;
    //Return root data
    return root→data;
}
```

Q3 Lowest common ancestor

i/p →

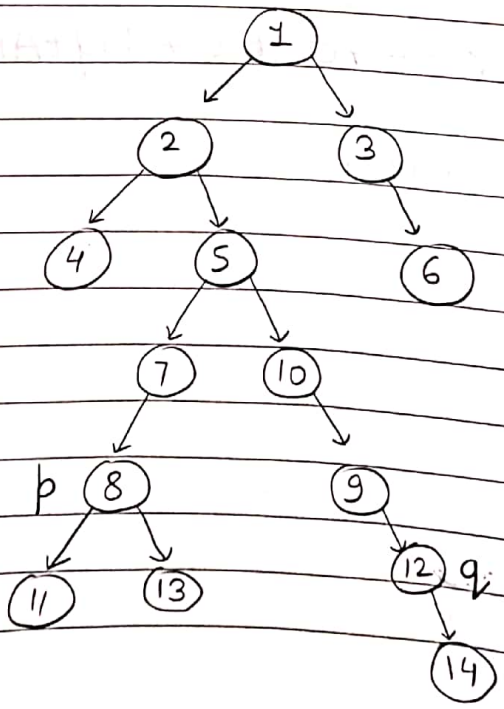


Ancestors of p i.e 50 → 20, 10
Ancestors of q i.e 90 → 70, 30, 10

Ans = 10

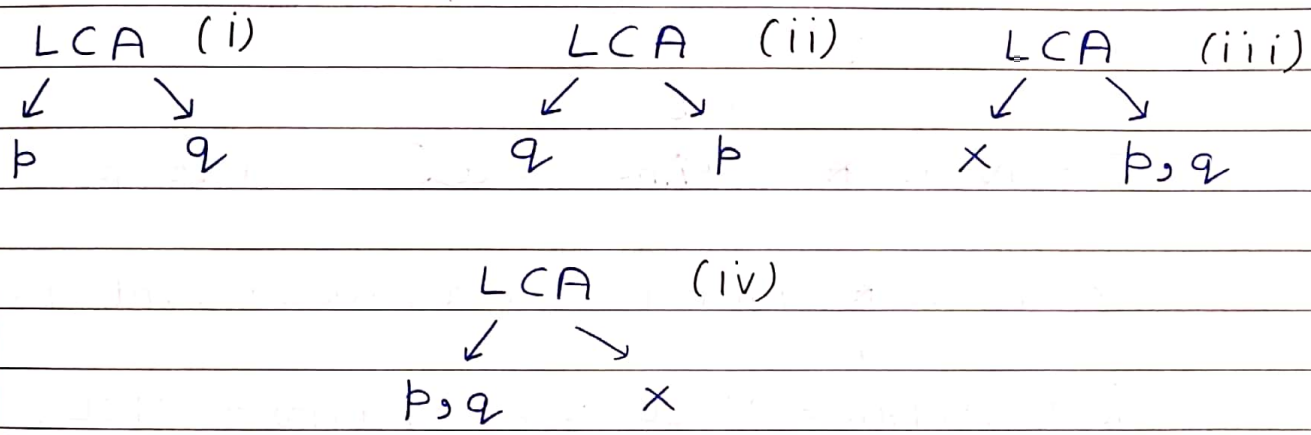
Common means same & lowest mean the one which has at higher level. In the given test case there was only one common ancestor and hence we have returned that as an answer.

Dry run / Algorithm

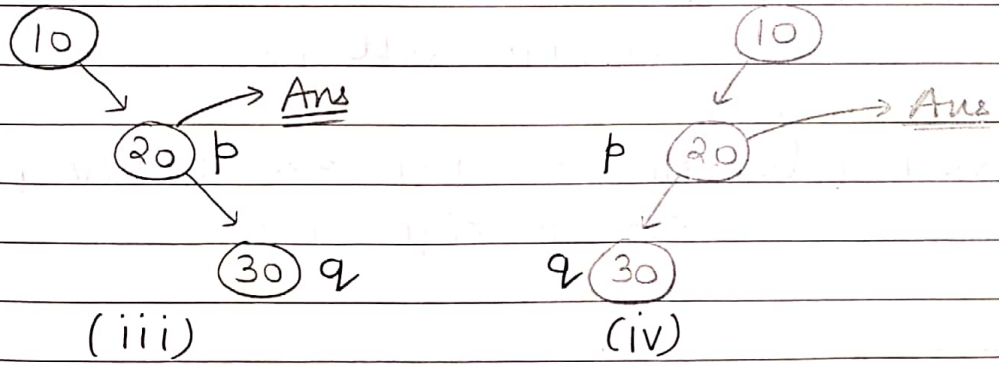


1) Search for p and q in the binary tree by first checking the current node, then left and then right, this is done via recursion.

2) There can be 4 cases i.e



The above test case which we have taken can be considered of case 1 and 2.



3) As we reach the point where we got either p or q, then we have to return the data of that particular node.

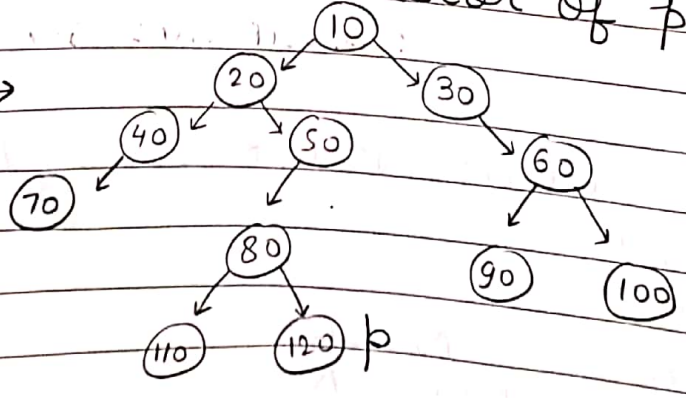
Code

```
TreeNode* lca (TreeNode* root, TreeNode* p,
TreeNode* q) {
        // Base case
        if (root == NULL) {
            return NULL;
```

```
//Search for p & q in tree
if (root → val == p → val) {
        return p; // Return p if found
}
if (root → val == q → val) {
        return q; // Return q if found
}
// Go to left subtree
TreeNode * leftAns = lca (root → left, p, q);
// Go to right subtree
TreeNode* rightAns = lca (root → right, p, q);
// Answer not found
if (leftAns == NULL && rightAns == NULL)
        return NULL;
// Ans found in left
else if (leftAns != NULL && rightAns == NULL)
        return leftAns;
// Ans found in right
else if (leftAns == NULL && rightAns != NULL)
        return rightAns;
// Ans found
else
        return root;
}
```

Q4 Find the kth ancestor of particular node.

i/p →



k = 2, p = 120

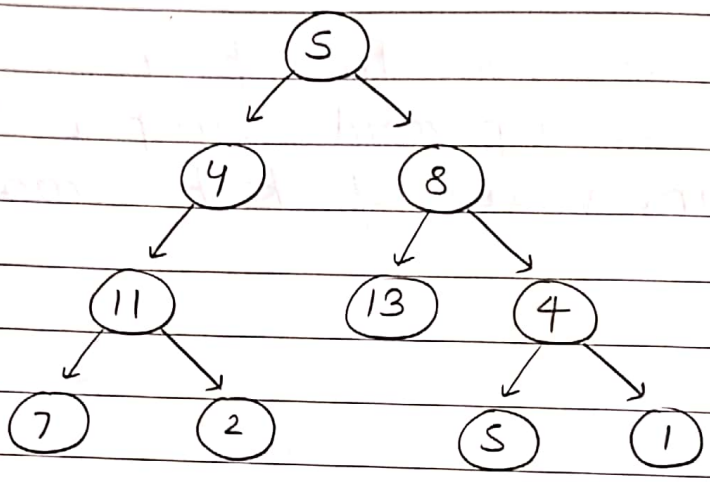
Here the 2nd ancestor of 120 is 50.

1) We need to find the p in our tree.
2) As we move upward, simply do k--.
3) Now as the value of k becomes 0, simply return.

Code

```
bool kthAncestor (Node * root, int &k, int p)
{
    //p not found case
    if (root == NULL) {
        return false;
    }
    //p found case
    if (root -> data == p)
        return true;
    //Find in left subtree then rightsubtree
    bool leftAns = kthAncestor (root->left, k, p);
    bool rightAns= kthAncestor (root->right, k, p);
    //Going upwards (Backtracking in recursion)
    if (leftAns || rightAns) {
        k--;
    }
    if (k == 0) { // Found kth ancestor
        cout << "Ans = " << root -> data << " ";
        k = -1; //Single answer print
    }
    return leftAns || rightAns;
}
```

Q5 Path sum - II



5 → 4 → 11 → 7 (27)
5 → 4 → 11 → 2 (22)
5 → 8 → 13 (26)
5 → 8 → 4 → 5 (22)
5 → 8 → 4 → 1 (18)

Ans ⇒ 5 → 4 → 11 → 2
5 → 8 → 4 → 5

Dry run / Algorithm

1) Check whether current node is NULL or not.
Root ⇒ Node = 5
not null

2) Now check whether it is a leaf node. It is not a leaf node.

3) Push the current node in path vector & update currSum by the value.
currSum += root→val

path → {5}
currSum = 0 + 5 = 5
* Now recursive call for left child will go
✓ Not null and not leaf node, so simply include in path and update currSum

path → {5, 4}
currSum = 5 + 4 = 9
* Now again recursive call for left child
✓ Not null & not leaf node, so simply include in path & currSum.
path → {5, 4, 11}
currSum = 9 + 11 = 20
* Now again recursive call for left child
✓ Not null but leaf node, so simply include in path & currSum.
path → {5, 4, 11, 7}
currSum = 20 + 7 = 27

Compare currSum with targetSum. Here not equal. Hence simply do pop from path & update currSum.
path → {5, 4, 11}
currSum = 27 - 7 = 20
Now simply return
* Now recursive call for right child.
Not null but it is a leaf node. Simply add data in path vector & update currSum.
path → {5, 4, 11, 2}
currSum = 20 + 2 = 22
Compare current sum and target sum.
Here equal hence push this path to the ans.

Similarly we can find all the paths.

Code

```
void solve (Treenode * root , int targetSum,
int currSum , vector<int> & path , vector<vector
<int>> & ans) {

        // Base case
        if (root == NULL)
                return ;
        // Leaf node
        if (root→left == NULL && root→right == NULL
                //Consider leaf node
                path . push_back (root→val);
                currSum += root→val ;
                // Compare currSum & targetSum
                if (currSum == target Sum) {
                        // Push in ans
                        ans.push_back (path);
                } // To consider other answers
                path . pop_back ();
                curr Sum -= root→val ;
                return ;
        }
        // include current node
        path . push_back (root→val);
        currSum += root→val ;
        // Call for left child
        solve (root→left , targetSum , currSum , path , ans);
        // Call for right child
        solve (root→right , targetSum , currSum , path , ans);
```

```
        // Backtracking to consider other answers
        path.pop_back();
        currSum -= root->val;
    }

    vector<vector<int>> pathSum(TreeNode *root,
    int targetSum) {

        vector<vector<int>> ans;
        int sum = 0;
        vector<int> target;
        solve(root, targetSum, sum, target, ans);
        return ans;
    }
```