Trees : Level 2
Lesson 1

[Node-at-distance]
[Nodes-at-given-distance-in-binary-tree]
[Print-nodes-having-k-leaves]
[Right-sibling-in-binary-tree]
[Check-mirror-in-n-ary-tree]
[cousins-of-a-given-node]

Node at distance

Given a Binary Tree and a positive integer **k**. The task is to count all distinct nodes that are distance k from a leaf node. A node is at k distance from a leaf if it is present k levels above the leaf and also, is a direct ancestor of this leaf node. If k is more than the height of the Binary Tree, then nothing should be counted.

Example 1:

Input:
```
      1
    /   \
   2     3
  / \   / \
 4   5 6   7
        \
         8
```
K = 2
Output: 2

Example 2:

Input:
```
        1
       /
      3
     /
    5
   /  \
  7    8
        \
         9
```
K = 4
Output: 1
Expected Time Complexity: O(N).

Expected Auxiliary Space: O(Height of the Tree).

Constraints:

1 <= N <= 105

Method 1:

```
void nodes(Node*node , int k ,unordered_set<Node*>&s , vector<Node*>v)
{
  if(!node)
     return;
  v.push_back(node);
  if(!node->left && !node->right && v.size()>k)
  {
     s.insert(v[v.size()-k-1]);
  }
  nodes(node->left,k,s,v);
  nodes(node->right,k,s,v);
}
int printKDistantfromLeaf(Node* node, int k)
{
  unordered_set<Node*>s;
  vector<Node*>v;
  nodes(node,k,s,v);
  return s.size();
}
```

Method 2:

```
int printKDistantfromLeaf(struct Node *node, int k)
```

```
{
 if (node == NULL)
   return -1;
 int lk = printKDistantfromLeaf(node->left, k);
 int rk = printKDistantfromLeaf(node->right, k);
 bool isLeaf = lk == -1 && lk == rk;
 if (lk == 0 || rk == 0 || (isLeaf && k == 0))
   cout<<(" " )<<( node->data);
 if (isLeaf && k > 0)
   return k - 1; // leaf node
 if (lk > 0 && lk < k)
   return lk - 1; // parent of left leaf
 if (rk > 0 && rk < k)
   return rk - 1; // parent of right leaf
 return -2;
}
```

Nodes at given distance in binary tree

Given a binary tree, a target node in the binary tree, and an integer value k, find all the nodes that are at a distance k from the given target node. No parent pointers are available.

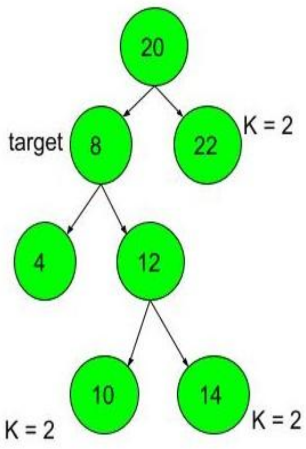
Example 1:

Input :

```
        20
       /   \
     8       22
   /   \
  4    12
      /   \
     10    14
```

Target Node = 8
K = 2

Output: 10 14 22



Example 2:

Input :

```
      20
    /   \
   7     24
 /   \
4     3
     /
    1
```

Target Node = 7
K = 2

Output: 1 24

Expected Time Complexity: O(N*logN)

Expected Auxiliary Space: O(Height of tree)

Constraints:

1 ≤ N ≤ 10^3

1 ≤ data of node ≤ 10000

1 ≤ target ≤ 10000

1 ≤ k ≤ 20

```
bool findNode(stack<TreeNode*> &st, TreeNode* root, TreeNode* Target)
{
    if(!root)
```

```
        return false;
    if(root==Target)
        return true;
    st.push(root);
    if(findNode(st,root->left,Target))  return true;
    if(findNode(st,root->right,Target)) return true;
    st.pop();
    return false;
}
set<TreeNode*> s;
vector<int> ans;
void subtree(int k, TreeNode *root)
{
    if(!root || s.count(root)==1)
        return;
    s.insert(root);
    if(k==0)
    {
        ans.push_back(root->val);
        return;
    }
    subtree(k-1,root->left);
    subtree(k-1,root->right);
}
class Solution {
public:
    vector<int> distanceK(TreeNode* root, TreeNode* target, int k) {
        stack<TreeNode*> st;
        findNode(st,root,target);
        st.push(target);
        ans.clear();
        s.clear();
        while(!st.empty() && k>=0)
        {
            subtree(k,st.top());
            st.pop();
```

```
                k--;
            }
            return ans;
        }
    };
```

Print Nodes having K leaves

Given a binary tree and an integer value **K**, the task is to **find all nodes** data in a given binary tree **having exactly K leaves** in the **subtree rooted with them**.

NOTE: Nodes should be printed in the order in which they appear in the postorder traversal.

Example 1:

Input:
K = 1
```
    0
  /   \
 1     2
```
Output: -1
Example 2:

Input:
K = 2
```
      0
    /   \
   1     2
        /
      4
    /   \
   5     9
```
Output: 4 2
Note: If no node is found the list returned should contain only one value -1.

Expected Time Complexity: O(N).

Expected Auxiliary Space: O(Height of the Tree).

Constraints:

1 <= N <= 1000

1 <= K <= 1000

1 <= value of nodes <= 10000

```
class Solution{
 public:
   /*You are required to complete below method */
   //returns the sum of leaves in left and right subtree
   int leaves(Node*node,int k,vector<int>&v)
   {
       if(!node)
           return 0;
       if(!node->left && !node->right)
       {
           return 1;
       }
       int sum = leaves(node->left,k,v)+leaves(node->right,k,v);
       if(sum==k)
           v.push_back(node->data);
       return sum;
   }
   vector<int> btWithKleaves(Node *ptr, int k)
   {
       vector<int>v;
       leaves(ptr,k,v);
       if(v.size()==0)
           v.push_back(-1);
       return v;
       //your code here.
   }

};
```

Right Sibling in Binary Tree

Given a binary tree, your task is to complete the function **findRightSibling()**, which should return the right sibling to a given node if it doesn't exist return **null**.

The structure of the node of the binary tree is as

```
struct Node
{
    int data;
    Node* left;
    Node* right;
    Node* parent;
};
```

Examples:

```
        1
       / \
      2   3
     /  \  \
    4    6  5
   /      \  \
  7        9  8
 /          \
10           12
```

Input : Given above tree with parent pointer and node 10
Output : 12

Constraints:

1<=T<=100

1<=N<=100

O(N)=queue level order traversal

O(1)=Recursion , parent pointer is given

```
Node* findRightSibling(Node* node, int level)
{
    if (node == NULL || node->parent == NULL)
        return NULL;

    // GET Parent pointer whose right child is not
    // a parent or itself of this node. There might
    // be case when parent has no right child, but,
    // current node is left child of the parent
    // (second condition is for that).
    while (node->parent->right == node
          || (node->parent->right == NULL
             && node->parent->left == node)) {
        if (node->parent == NULL
            || node->parent->parent == NULL)
            return NULL;

        node = node->parent;
        level--;
    }

    // Move to the required child, where right sibling
    // can be present
    node = node->parent->right;

    if (node == NULL)
        return NULL;
```

```
        // find right sibling in the given subtree(from current
        // node), when level will be 0
        while (level < 0) {

            // Iterate through subtree
            if (node->left != NULL)
                node = node->left;
            else if (node->right != NULL)
                node = node->right;
            else

                // if no child are there, we cannot have right
                // sibling in this path
                break;

            level++;
        }

        if (level == 0)
            return node;

        // This is the case when we reach 9 node in the tree,
        // where we need to again recursively find the right
        // sibling
        return findRightSibling(node, level);
    }
```

Check Mirror in N-ary tree

Given two **n**-ary trees. Check if they are mirror images of each other or not. You are also given **e** denoting the number of edges in both trees, and two arrays, **A[]** and **B[]**. Each array has 2*e space separated values u,v denoting an edge from u to v for the both trees.

Example 1:

Input:
n = 3, **e** = 2
A[] = {1, 2, 1, 3}
B[] = {1, 3, 1, 2}
Output:
1
Explanation:

```
   1          1
 / \        /  \
2   3      3    2
```

Example 2:

Input:
n = 3, **e** = 2
A[] = {1, 2, 1, 3}
B[] = {1, 2, 1, 3}
Output:
1
Explanation:

```
   1          1
 / \        /  \
2   3      2    3
```

Expected Time Complexity: O(n)

Expected Auxiliary Space: O(n)

Constraints:

1 <= n,e <= 105

```
class Solution {
 public:
   int checkMirrorTree(int n, int e, int A[], int B[])
   {
       // code here
unordered_map<int,stack<int>> mp;
for(int i = 0 ; i < 2*e ; i+=2 ){
mp[A[i]].push(A[i+1]);
}
for(int i = 0 ; i < 2*e ; i+=2){
if(!(mp[B[i]].top() == B[i+1])) return 0;
mp[B[i]].pop();
}
return 1;
   }
};
```

Cousins of a given node

Given a binary tree and a node, print all cousins of a given node. Note that siblings should not be printed.

Example 1:

Input :

```
        1
      /   \
     2     3
   /  \  /  \
  4    5 6   7
```

Given node : 5

Output : 6 7

Explanation :

Nodes 6 and 7 are on the same level
as 5 and have different parents.

Example 2 :

Input :

```
     9
    /
   5
```

Given node : 5

Output : -1

Explanation :

There are no other nodes than 5 in the same level.

Expected Time Complexity : O(n)

Expected Auxiliary Space : O(n)

Constraints :

1 <= n <=10^5

```
void printCousins(Node* root, Node* node_to_find)
{
  // if the given node is the root itself,
  // then no nodes would be printed
  if (root == node_to_find) {
    cout << "Cousin Nodes : None" << endl;
    return;
  }

  queue<Node*> q;
  bool found = false;
  int size_;
  Node* p;
  q.push(root);

  // the following loop runs until found is
  // not true, or q is not empty.
  // if found has become true => we have found
  // the level in which the node is present
  // and the present queue will contain all the
  // cousins of that node
  while (!q.empty() && !found) {

    size_ = q.size();
    while (size_) {
      p = q.front();
      q.pop();

      // if current node's left or right child
      // is the same as the node to find,
```

```
            // then make found = true, and don't push
            // any of them into the queue, as
            // we don't have to print the siblings
            if ((p->left == node_to_find ||
                p->right == node_to_find)) {
                found = true;
            }
            else {
                if (p->left)
                    q.push(p->left);
                if (p->right)
                    q.push(p->right);
            }

            size_--;
        }
    }

    // if found == true then the queue will contain the
    // cousins of the given node
    if (found) {
        cout << "Cousin Nodes : ";
        size_ = q.size();

        // size_ will be 0 when the node was at the
        // level just below the root node.
        if (size_ == 0)
            cout << "None";
        for (int i = 0; i < size_; i++) {
            p = q.front();
```

```
            q.pop();
            cout << p->data << " ";
        }
    }
    else {
        cout << "Node not found";
    }
    cout << endl;
    return;
```