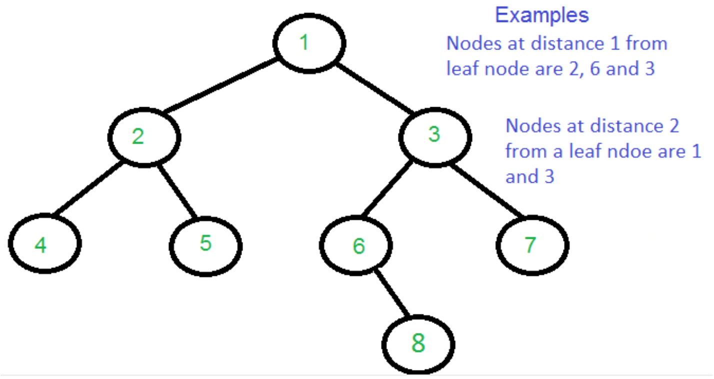
Print all nodes that are at distance k from a leaf node

Given a Binary Tree and a positive integer k, print all nodes that are distance k from a leaf node.

Here the meaning of distance is different from previous post. Here k distance from a leaf means k levels higher than a leaf node. For example if k is more than height of Binary Tree, then nothing should be printed. Expected time complexity is O(n) where n is the number nodes in the given Binary Tree.



Examples
Nodes at distance 1 from leaf node are 2, 6 and 3

Nodes at distance 2 from a leaf ndoe are 1 and 3

We strongly recommend to minimize the browser and try this yourself first.

The idea is to traverse the tree. Keep storing all ancestors till we hit a leaf node. When we reach a leaf node, we print the ancestor at distance k. We also need to keep track of nodes that are already printed as output. For that we use a boolean array visited[].

C++

```
/* Program to print all nodes which are at distance k from a leaf */
#include <iostream>
using namespace std;
#define MAX_HEIGHT 10000

struct Node
{
    int key;
    Node *left, *right;
};

/* utility that allocates a new Node with the given key  */
Node* newNode(int key)
{
    Node* node = new Node;
    node->key = key;
    node->left = node->right = NULL;
    return (node);
}

/* This function prints all nodes that are distance k from a leaf node
   path[] --> Store ancestors of a node
   visited[] --> Stores true if a node is printed as output.  A node may be k
                 distance away from many leaves, we want to print it once */
void kDistantFromLeafUtil(Node* node, int path[], bool visited[],
                          int pathLen, int k)
{
    // Base case
    if (node==NULL) return;

    /* append this Node to the path array */
    path[pathLen] = node->key;
    visited[pathLen] = false;
    pathLen++;
    /* it's a leaf, so print the ancestor at distance k only
       if the ancestor is not already printed  */
    if (node->left == NULL && node->right == NULL &&
        pathLen-k-1 >= 0 && visited[pathLen-k-1] == false)
    {
        cout << path[pathLen-k-1] << " ";
        visited[pathLen-k-1] = true;
        return;
    }

    /* If not leaf node, recur for left and right subtrees */
    kDistantFromLeafUtil(node->left, path, visited, pathLen, k);
    kDistantFromLeafUtil(node->right, path, visited, pathLen, k);
}

/* Given a binary tree and a nuber k, print all nodes that are k
   distant from a leaf*/
void printKDistantfromLeaf(Node* node, int k)
{
    int path[MAX_HEIGHT];
    bool visited[MAX_HEIGHT] = {false};
    kDistantFromLeafUtil(node, path, visited, 0, k);
}

/* Driver program to test above functions*/
int main()
{
    // Let us create binary tree given in the above example
    Node * root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    root->right->left = newNode(6);
    root->right->right = newNode(7);
    root->right->left->right = newNode(8);

    cout << "Nodes at distance 2 are: ";
    printKDistantfromLeaf(root, 2);

    return 0;
}
```

Java

```
// Java program to print all nodes at a distance k from leaf
// A binary tree node
class Node
{
    int data;
    Node left, right;

    Node(int item)
    {
        data = item;
        left = right = null;
    }
}

class BinaryTree
{
    Node root;

    /* This function prints all nodes that are distance k from a leaf node
     path[] --> Store ancestors of a node
     visited[] --> Stores true if a node is printed as output.  A node may
     be k distance away from many leaves, we want to print it once */
    void kDistantFromLeafUtil(Node node, int path[], boolean visited[],
                              int pathLen, int k)
    {
```

```
    {
        // Base case
        if (node == null)
            return;

        /* append this Node to the path array */
        path[pathLen] = node.data;
        visited[pathLen] = false;
        pathLen++;

        /* it's a leaf, so print the ancestor at distance k only
         if the ancestor is not already printed  */
        if (node.left == null && node.right == null
           && pathLen - k - 1 >= 0 && visited[pathLen - k - 1] == false)
        {
            System.out.print(path[pathLen - k - 1] + " ");
            visited[pathLen - k - 1] = true;
            return;
        }

        /* If not leaf node, recur for left and right subtrees */
        kDistantFromLeafUtil(node.left, path, visited, pathLen, k);
        kDistantFromLeafUtil(node.right, path, visited, pathLen, k);
    }

    /* Given a binary tree and a nuber k, print all nodes that are k
     distant from a leaf*/
    void printKDistantfromLeaf(Node node, int k)
    {
        int path[] = new int[1000];
        boolean visited[] = new boolean[1000];
        kDistantFromLeafUtil(node, path, visited, 0, k);
    }

    // Driver program to test the above functions
    public static void main(String args[])
    {
        BinaryTree tree = new BinaryTree();

        /* Let us construct the tree shown in above diagram */
        tree.root = new Node(1);
        tree.root.left = new Node(2);
        tree.root.right = new Node(3);
        tree.root.left.left = new Node(4);
        tree.root.left.right = new Node(5);
        tree.root.right.left = new Node(6);
        tree.root.right.right = new Node(7);
        tree.root.right.left.right = new Node(8);

        System.out.println(" Nodes at distance 2 are :");
        tree.printKDistantfromLeaf(tree.root, 2);
    }
}

// This code has been contributed by Mayank Jaiswal
```

Output:

```
Nodes at distance 2 are: 3 1
```

Time Complexity: Time Complexity of above code is O(n) as the code does a simple tree traversal.