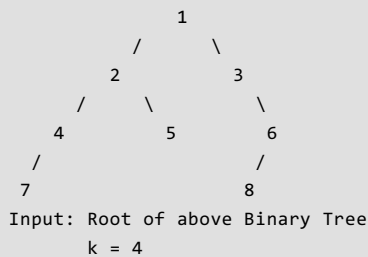


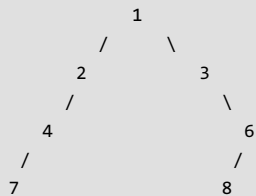
Remove nodes on root to leaf paths of length < K

Given a Binary Tree and a number k, remove all nodes that lie only on root to leaf path(s) of length smaller than k. If a node X lies on multiple root-to-leaf paths and if any of the paths has path length $\geq k$, then X is not deleted from Binary Tree. In other words a node is deleted if all paths going through it have lengths smaller than k.

Consider the following example Binary Tree



Output: The tree should be changed to following



There are 3 paths

- i) 1->2->4->7 path length = 4
- ii) 1->2->5 path length = 3
- iii) 1->3->6->8 path length = 4

There is only one path " 1->2->5 " of length smaller than 4.

The node 5 is the only node that lies only on this path, so node 5 is removed.

Nodes 2 and 1 are not removed as they are parts of other paths of length 4 as well.

If k is 5 or greater than 5, then whole tree is deleted.

If k is 3 or less than 3, then nothing is deleted.

We strongly recommend to minimize your browser and try this yourself first

The idea here is to use post order traversal of the tree. Before removing a node we need to check that all the children of that node in the shorter path are already removed.

There are 2 cases:

- i) This node becomes a leaf node in which case it needs to be deleted.
- ii) This node has other child on a path with path length $\geq k$. In that case it needs not to be deleted.

The implementation of above approach is as below :

C/C++

```
// C++ program to remove nodes on root to leaf paths of length < K
#include<iostream>
using namespace std;

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

```

//New node of a tree
Node *newNode(int data)
{
    Node *node = new Node;
    node->data = data;
    node->left = node->right = NULL;
    return node;
}

// Utility method that actually removes the nodes which are not
// on the pathLen >= k. This method can change the root as well.
Node *removeShortPathNodesUtil(Node *root, int level, int k)
{
    //Base condition
    if (root == NULL)
        return NULL;

    // Traverse the tree in postorder fashion so that if a leaf
    // node path length is shorter than k, then that node and
    // all of its descendants till the node which are not
    // on some other path are removed.
    root->left = removeShortPathNodesUtil(root->left, level + 1, k);
    root->right = removeShortPathNodesUtil(root->right, level + 1, k);

    // If root is a leaf node and it's level is less than k then
    // remove this node.
    // This goes up and check for the ancestor nodes also for the
    // same condition till it finds a node which is a part of other
    // path(s) too.
    if (root->left == NULL && root->right == NULL && level < k)
    {
        delete root;
        return NULL;
    }

    // Return root;
    return root;
}

// Method which calls the utility method to remove the short path
// nodes.
Node *removeShortPathNodes(Node *root, int k)
{
    int pathLen = 0;
    return removeShortPathNodesUtil(root, 1, k);
}

//Method to print the tree in inorder fashion.
void printInorder(Node *root)
{
    if (root)
    {
        printInorder(root->left);
        cout << root->data << " ";
        printInorder(root->right);
    }
}

// Driver method.
int main()
{
    int k = 4;
    Node *root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    root->left->left->left = newNode(7);
    root->right->right = newNode(6);
    root->right->right->left = newNode(8);
    cout << "Inorder Traversal of Original tree" << endl;
    printInorder(root);
    cout << endl;
}

```

```

cout << "Inorder Traversal of Modified tree" << endl;
Node *res = removeShortPathNodes(root, k);
printInorder(res);
return 0;
}

```

Java

```

// Java program to remove nodes on root to leaf paths of length < k

/* Class containing left and right child of current
   node and key value*/
class Node
{
    int data;
    Node left, right;

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

class BinaryTree
{
    Node root;

    // Utility method that actually removes the nodes which are not
    // on the pathLen >= k. This method can change the root as well.
    Node removeShortPathNodesUtil(Node node, int level, int k)
    {
        //Base condition
        if (node == null)
            return null;

        // Traverse the tree in postorder fashion so that if a leaf
        // node path length is shorter than k, then that node and
        // all of its descendants till the node which are not
        // on some other path are removed.
        node.left = removeShortPathNodesUtil(node.left, level + 1, k);
        node.right = removeShortPathNodesUtil(node.right, level + 1, k);

        // If root is a leaf node and it's level is less than k then
        // remove this node.
        // This goes up and check for the ancestor nodes also for the
        // same condition till it finds a node which is a part of other
        // path(s) too.
        if (node.left == null && node.right == null && level < k)
            return null;

        // Return root;
        return node;
    }

    // Method which calls the utility method to remove the short path
    // nodes.
    Node removeShortPathNodes(Node node, int k)
    {
        int pathLen = 0;
        return removeShortPathNodesUtil(node, 1, k);
    }

    //Method to print the tree in inorder fashion.
    void printInorder(Node node)
    {
        if (node != null)
        {
            printInorder(node.left);
            System.out.print(node.data + " ");
        }
    }
}

```

```

        printInorder(node.right);
    }
}

// Driver program to test for samples
public static void main(String args[])
{
    BinaryTree tree = new BinaryTree();
    int k = 4;
    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.left.left.left = new Node(7);
    tree.root.right.right = new Node(6);
    tree.root.right.right.left = new Node(8);
    System.out.println("The inorder traversal of original tree is : ");
    tree.printInorder(tree.root);
    Node res = tree.removeShortPathNodes(tree.root, k);
    System.out.println("");
    System.out.println("The inorder traversal of modified tree is : ");
    tree.printInorder(res);
}
}

// This code has been contributed by Mayank Jaiswal

```

Output:

```

Inorder Traversal of Original tree
7 4 2 5 1 3 8 6
Inorder Traversal of Modified tree
7 4 2 1 3 8 6

```

Time complexity of the above solution is $O(n)$ where n is number of nodes in given Binary Tree.