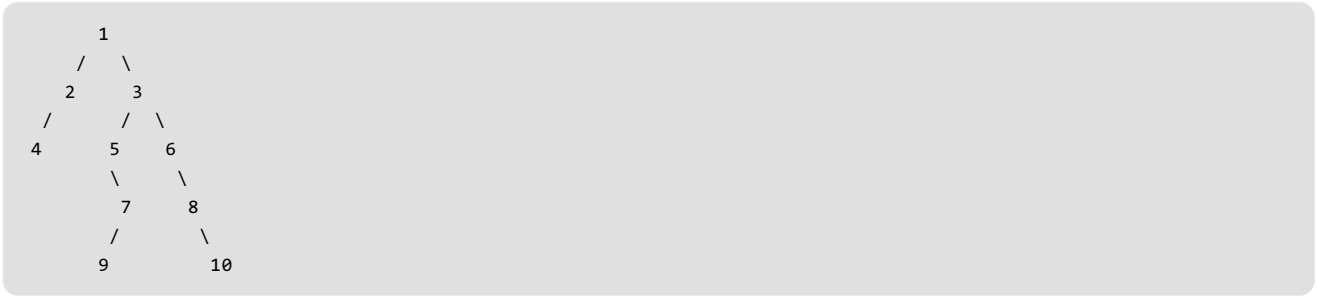
Deepest left leaf node in a binary tree

Given a Binary Tree, find the deepest leaf node that is left child of its parent. For example, consider the following tree. The deepest left leaf node is the node with value 9.



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

The idea is to recursively traverse the given binary tree and while traversing, maintain "level" which will store the current node's level in the tree. If current node is left leaf, then check if its level is more than the level of deepest left leaf seen so far. If level is more then update the result. If current node is not leaf, then recursively find maximum depth in left and right subtrees, and return maximum of the two depths. Thanks to Coder011 for suggesting this approach.

C/C++

```
// A C++ program to find the deepest left leaf in a given binary tree
#include <stdio.h>
#include <iostream>
using namespace std;

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

Node *newNode(int data)
{
    Node *temp = new Node;
    temp->val = data;
    temp->left = temp->right =  NULL;
    return temp;
}

// A utility function to find deepest leaf node.
// lvl:  level of current node.
// maxlvl: pointer to the deepest left leaf node found so far
// isLeft: A bool indicate that this node is left child of its parent
// resPtr: Pointer to the result
void deepestLeftLeafUtil(Node *root, int lvl, int *maxlvl,
                         bool isLeft, Node **resPtr)
{
    // Base case
    if (root == NULL)
        return;

    // Update result if this node is left leaf and its level is more
    // than the maxl level of the current result
    if (isLeft && !root->left && !root->right && lvl > *maxlvl)
    {
        *resPtr = root;
        *maxlvl = lvl;
        return;
```

```
    }

    // Recur for left and right subtrees
    deepestLeftLeafUtil(root->left, lvl+1, maxlvl, true, resPtr);
    deepestLeftLeafUtil(root->right, lvl+1, maxlvl, false, resPtr);
}

// A wrapper over deepestLeftLeafUtil().
Node* deepestLeftLeaf(Node *root)
{
    int maxlevel = 0;
    Node *result = NULL;
    deepestLeftLeafUtil(root, 0, &maxlevel, false, &result);
    return result;
}

// Driver program to test above function
int main()
{
    Node* root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->right->left = newNode(5);
    root->right->right = newNode(6);
    root->right->left->right = newNode(7);
    root->right->right->right = newNode(8);
    root->right->left->right->left = newNode(9);
    root->right->right->right->right = newNode(10);

    Node *result = deepestLeftLeaf(root);
    if (result)
        cout << "The deepest left child is " << result->val;
    else
        cout << "There is no left leaf in the given tree";

    return 0;
}
```

Java

```
// A java program to find the deepest left leaf in a binary tree

// A Binary Tree node
class Node
{
    int data;
    Node left, right;

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

// Class to evaluate pass by reference
class Level
{
    // maxlevel: gives the value of level of maximum left leaf
    int maxlevel = 0;
}

class BinaryTree
{
    Node root;

    // Node to store resultant node after left traversal
    Node result;
```

```
    // A utility function to find deepest leaf node.
    // lvl:  level of current node.
    // isLeft: A bool indicate that this node is left child
    void deepestLeftLeafUtil(Node node, int lvl, Level level,
            boolean isLeft)
    {
        // Base case
        if (node == null)
            return;

        // Update result if this node is left leaf and its level is more
        // than the maxl level of the current result
        if (isLeft != false && node.left == null && node.right == null
                && lvl > level.maxlevel)
        {
            result = node;
            level.maxlevel = lvl;
        }

        // Recur for left and right subtrees
        deepestLeftLeafUtil(node.left, lvl + 1, level, true);
        deepestLeftLeafUtil(node.right, lvl + 1, level, false);
    }

    // A wrapper over deepestLeftLeafUtil().
    void deepestLeftLeaf(Node node)
    {
        Level level = new Level();
        deepestLeftLeafUtil(node, 0, level, false);
    }

    // Driver program to test above functions
    public static void main(String[] args)
    {
        BinaryTree tree = new BinaryTree();
        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.right.left = new Node(5);
        tree.root.right.right = new Node(6);
        tree.root.right.left.right = new Node(7);
        tree.root.right.right.right = new Node(8);
        tree.root.right.left.right.left = new Node(9);
        tree.root.right.right.right.right = new Node(10);

        tree.deepestLeftLeaf(tree.root);
        if (tree.result != null)
            System.out.println("The deepest left child is " + tree.result.data);
         else
            System.out.println("There is no left leaf in the given tree");
    }
}

// This code has been contributed by Mayank Jaiswal(mayank_24)
```

Python

```
# Python program to find the deepest left leaf in a given
# Binary tree

# A binary tree node
class Node:

    # Constructor to create a new node
    def __init__(self, val):
        self.val = val
        self.left = None
        self.right = None

# A utility function to find deepest leaf node.
# lvl:  level of current node.
# maxlvl: pointer to the deepest left leaf node found so far
# isLeft: A bool indicate that this node is left child
# of its parent
# resPtr: Pointer to the result
def deepestLeftLeafUtil(root, lvl, maxlvl, isLeft):

    # Base CAse
    if root is None:
        return

    # Update result if this node is left leaf and its
    # level is more than the max level of the current result
    if(isLeft is True):
        if (root.left == None and root.right == None):
            if lvl > maxlvl[0] :
                deepestLeftLeafUtil.resPtr = root
                maxlvl[0] = lvl
                return

    # Recur for left and right subtrees
    deepestLeftLeafUtil(root.left, lvl+1, maxlvl, True)
    deepestLeftLeafUtil(root.right, lvl+1, maxlvl, False)

# A wrapper for left and right subtree
def deepestLeftLeaf(root):
    maxlvl = [0]
    deepestLeftLeafUtil.resPtr = None
    deepestLeftLeafUtil(root, 0, maxlvl, False)
    return deepestLeftLeafUtil.resPtr


# Driver program to test above function
root = Node(1)
root.left = Node(2)
root.right = Node(3)
root.left.left = Node(4)
root.right.left = Node(5)
root.right.right = Node(6)
root.right.left.right = Node(7)
root.right.right.right = Node(8)
root.right.left.right.left = Node(9)
root.right.right.right.right= Node(10)

result = deepestLeftLeaf(root)

if result is None:
    print "There is not left leaf in the given tree"
else:
    print "The deepst left child is", result.val

# This code is contributed by Nikhil Kumar Singh(nickzuck_007)
```

Output:

```
The deepest left child is 9
```

Time Complexity: The function does a simple traversal of the tree, so the complexity is O(n).