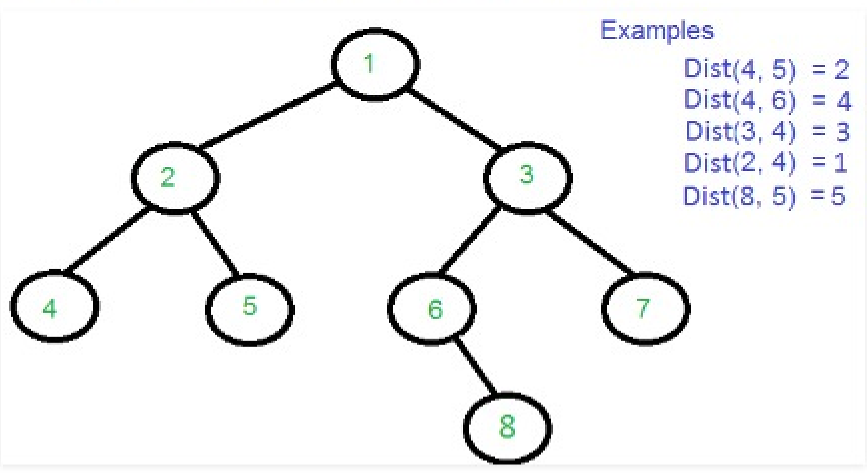
Find distance between two given keys of a Binary Tree

Find the distance between two keys in a binary tree, no parent pointers are given. Distance between two nodes is the minimum number of edges to be traversed to reach one node from other.



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

The distance between two nodes can be obtained in terms of lowest common ancestor. Following is the formula.

```
Dist(n1, n2) = Dist(root, n1) + Dist(root, n2) - 2*Dist(root, lca)
'n1' and 'n2' are the two given keys
'root' is root of given Binary Tree.
'lca' is lowest common ancestor of n1 and n2
Dist(n1, n2) is the distance between n1 and n2.
```

Following is the implementation of above approach. The implementation is adopted from last code provided in Lowest Common Ancestor Post.

C++

```cpp
/* Program to find distance between n1 and n2 using one traversal */
#include <iostream>
using namespace std;

// A Binary Tree Node
struct Node
{
    struct Node *left, *right;
    int key;
};

// Utility function to create a new tree Node
Node* newNode(int key)
{
    Node *temp = new Node;
    temp->key = key;
    temp->left = temp->right = NULL;
    return temp;
}

// Returns level of key k if it is present in tree, otherwise returns -1
int findLevel(Node *root, int k, int level)
{
    // Base Case
    if (root == NULL)
        return -1;

    // If key is present at root, or in left subtree or right subtree,
    // return true;
    if (root->key == k)
        return level;

    int l = findLevel(root->left, k, level+1);
    return (l != -1)? l : findLevel(root->right, k, level+1);
}

// This function returns pointer to LCA of two given values n1 and n2.
// It also sets d1, d2 and dist if one key is not ancestor of other
// d1 --> To store distance of n1 from root
// d2 --> To store distance of n2 from root
// lvl --> Level (or distance from root) of current node
// dist --> To store distance between n1 and n2
Node *findDistUtil(Node* root, int n1, int n2, int &d1, int &d2,
                   int &dist, int lvl)
{
    // Base case
    if (root == NULL) return NULL;

    // If either n1 or n2 matches with root's key, report
    // the presence by returning root (Note that if a key is
    // ancestor of other, then the ancestor key becomes LCA
    if (root->key == n1)
    {
         d1 = lvl;
         return root;
    }
    if (root->key == n2)
    {
         d2 = lvl;
         return root;
    }

    // Look for n1 and n2 in left and right subtrees
    Node *left_lca  = findDistUtil(root->left, n1, n2, d1, d2, dist, lvl+1);
    Node *right_lca = findDistUtil(root->right, n1, n2, d1, d2, dist, lvl+1);

    // If both of the above calls return Non-NULL, then one key
    // is present in once subtree and other is present in other,
    // So this node is the LCA
    if (left_lca && right_lca)
    {
        dist = d1 + d2 - 2*lvl;
        return root;
    }

    // Otherwise check if left subtree or right subtree is LCA
    return (left_lca != NULL)? left_lca: right_lca;
}

// The main function that returns distance between n1 and n2
// This function returns -1 if either n1 or n2 is not present in
// Binary Tree.
int findDistance(Node *root, int n1, int n2)
{
    // Initialize d1 (distance of n1 from root), d2 (distance of n2
    // from root) and dist(distance between n1 and n2)
    int d1 = -1, d2 = -1, dist;
    Node *lca = findDistUtil(root, n1, n2, d1, d2, dist, 1);

    // If both n1 and n2 were present in Binary Tree, return dist
    if (d1 != -1 && d2 != -1)
        return dist;

    // If n1 is ancestor of n2, consider n1 as root and find level
    // of n2 in subtree rooted with n1
    if (d1 != -1)
    {
        dist = findLevel(lca, n2, 0);
        return dist;
    }

    // If n2 is ancestor of n1, consider n2 as root and find level
    // of n1 in subtree rooted with n2
    if (d2 != -1)
    {
        dist = findLevel(lca, n1, 0);
        return dist;
    }

    return -1;
}

// 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 << "Dist(4, 5) = " << findDistance(root, 4, 5);
    cout << "\nDist(4, 6) = " << findDistance(root, 4, 6);
    cout << "\nDist(3, 4) = " << findDistance(root, 3, 4);
    cout << "\nDist(2, 4) = " << findDistance(root, 2, 4);
    cout << "\nDist(8, 5) = " << findDistance(root, 8, 5);
    return 0;
}
```

Run on IDE

Output:

```
Dist(4, 5) = 2
Dist(4, 6) = 4
Dist(3, 4) = 3
Dist(2, 4) = 1
Dist(8, 5) = 5
```

Time Complexity: Time complexity of the above solution is O(n) as the method does a single tree traversal.

Thanks to **Atul Singh** for providing the initial solution for this post.