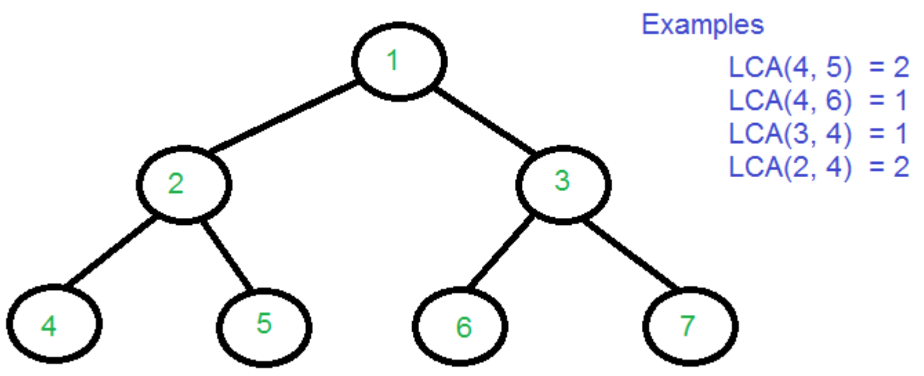
Lowest Common Ancestor in a Binary Tree | Set 1

Given a binary tree (not a binary search tree) and two values say n1 and n2, write a program to find the least common ancestor.

Following is definition of LCA from Wikipedia:

Let T be a rooted tree. The lowest common ancestor between two nodes n1 and n2 is defined as the lowest node in T that has both n1 and n2 as descendants (where we allow a node to be a descendant of itself).

The LCA of n1 and n2 in T is the shared ancestor of n1 and n2 that is located farthest from the root. Computation of lowest common ancestors may be useful, for instance, as part of a procedure for determining the distance between pairs of nodes in a tree: the distance from n1 to n2 can be computed as the distance from the root to n1, plus the distance from the root to n2, minus twice the distance from the root to their lowest common ancestor. (Source Wiki)



We have discussed an efficient solution to find LCA in Binary Search Tree. In Binary Search Tree, using BST properties, we can find LCA in O(h) time where h is height of tree. Such an implementation is not possible in Binary Tree as keys Binary Tree nodes don't follow any order. Following are different approaches to find LCA in Binary Tree.

Method 1 (By Storing root to n1 and root to n2 paths):

Following is simple O(n) algorithm to find LCA of n1 and n2.

1) Find path from root to n1 and store it in a vector or array.

2) Find path from root to n2 and store it in another vector or array.

3) Traverse both paths till the values in arrays are same. Return the common element just before the mismatch.

Following is C++ implementation of above algorithm.

C++

```
// A O(n) solution to find LCA of two given values n1 and n2
#include <iostream>
#include <vector>
using namespace std;

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

// Utility function creates a new binary tree node with given key
Node * newNode(int k)
{
    Node *temp = new Node;
    temp->key = k;
    temp->left = temp->right = NULL;
    return temp;
}

// Finds the path from root node to given root of the tree, Stores the
// path in a vector path[], returns true if path exists otherwise false
bool findPath(Node *root, vector<int> &path, int k)
{
    // base case
    if (root == NULL) return false;

    // Store this node in path vector. The node will be removed if
    // not in path from root to k
    path.push_back(root->key);

    // See if the k is same as root's key
    if (root->key == k)
        return true;

    // Check if k is found in left or right sub-tree
    if ( (root->left && findPath(root->left, path, k)) ||
         (root->right && findPath(root->right, path, k)) )
        return true;

    // If not present in subtree rooted with root, remove root from
    // path[] and return false
    path.pop_back();
    return false;
}
```

```
// Returns LCA if node n1, n2 are present in the given binary tree,
// otherwise return -1
int findLCA(Node *root, int n1, int n2)
{
    // to store paths to n1 and n2 from the root
    vector<int> path1, path2;

    // Find paths from root to n1 and root to n1. If either n1 or n2
    // is not present, return -1
    if ( !findPath(root, path1, n1) || !findPath(root, path2, n2))
          return -1;

    /* Compare the paths to get the first different value */
    int i;
    for (i = 0; i < path1.size() && i < path2.size() ; i++)
        if (path1[i] != path2[i])
            break;
    return path1[i-1];
}

// Driver program to test above functions
int main()
{
    // Let us create the Binary Tree shown in above diagram.
    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);
    cout << "LCA(4, 5) = " << findLCA(root, 4, 5);
    cout << "\nLCA(4, 6) = " << findLCA(root, 4, 6);
    cout << "\nLCA(3, 4) = " << findLCA(root, 3, 4);
    cout << "\nLCA(2, 4) = " << findLCA(root, 2, 4);
    return 0;
}
```

Python

```
# O(n) solution to find LCS of two given values n1 and n2

# A binary tree node
class Node:
    # Constructor to create a new binary node
    def __init__(self, key):
        self.key =  key
        self.left = None
        self.right = None

# Finds the path from root node to given root of the tree.
# Stores the path in a list path[], returns true if path
# exists otherwise false
def findPath( root, path, k):

    # Baes Case
    if root is None:
        return False

    # Store this node is path vector. The node will be
    # removed if not in path from root to k
    path.append(root.key)

    # See if the k is same as root's key
    if root.key == k :
        return True

    # Check if k is found in left or right sub-tree
    if ((root.left != None and findPath(root.left, path, k)) or
            (root.right!= None and findPath(root.right, path, k))):
        return True

    # If not present in subtree rooted with root, remove
    # root from path and return False

    path.pop()
    return False

# Returns LCA if node n1 , n2 are present in the given
# binary tre otherwise return -1
def findLCA(root, n1, n2):

    # To store paths to n1 and n2 fromthe root
    path1 = []
    path2 = []

    # Find paths from root to n1 and root to n2.
    # If either n1 or n2 is not present , return -1
    if (not findPath(root, path1, n1) or not findPath(root, path2, n2)):
        return -1

    # Compare the paths to get the first different value
    i = 0
    while(i < len(path1) and i < len(path2)):
        if path1[i] != path2[i]:
            break
        i += 1
    return path1[i-1]
```

```
# Driver program to test above function
# Let's create the Binary Tree shown in above diagram
root = Node(1)
root.left = Node(2)
root.right = Node(3)
root.left.left = Node(4)
root.left.right = Node(5)
root.right.left = Node(6)
root.right.right = Node(7)

print "LCA(4, 5) = %d" %(findLCA(root, 4, 5,))
print "LCA(4, 6) = %d" %(findLCA(root, 4, 6))
print "LCA(3, 4) = %d" %(findLCA(root,3,4))
print "LCA(2, 4) = %d" %(findLCA(root,2, 4))

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

Output:

```
LCA(4, 5) = 2
LCA(4, 6) = 1
LCA(3, 4) = 1
LCA(2, 4) = 2
```

Time Complexity: Time complexity of the above solution is O(n). The tree is traversed twice, and then path arrays are compared.
Thanks to *Ravi Chandra Enaganti* for suggesting the initial solution based on this method.

Method 2 (Using Single Traversal)
The method 1 finds LCA in O(n) time, but requires three tree traversals plus extra spaces for path arrays. If we assume that the keys n1 and n2 are present in Binary Tree, we can find LCA using single traversal of Binary Tree and without extra storage for path arrays.
The idea is to traverse the tree starting from root. If any of the given keys (n1 and n2) matches with root, then root is LCA (assuming that both keys are present). If root doesn't match with any of the keys, we recur for left and right subtree. The node which has one key present in its left subtree and the other key present in right subtree is the LCA. If both keys lie in left subtree, then left subtree has LCA also, otherwise LCA lies in right subtree.

C

```
/* Program to find LCA of n1 and n2 using one traversal of Binary Tree */
```

```
#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;
}

// This function returns pointer to LCA of two given values n1 and n2.
// This function assumes that n1 and n2 are present in Binary Tree
struct Node *findLCA(struct Node* root, int n1, int n2)
{
    // 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 || root->key == n2)
        return root;

    // Look for keys in left and right subtrees
    Node *left_lca  = findLCA(root->left, n1, n2);
    Node *right_lca = findLCA(root->right, n1, n2);

    // 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)  return root;

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

// 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);
    cout << "LCA(4, 5) = " << findLCA(root, 4, 5)->key;
    cout << "\nLCA(4, 6) = " << findLCA(root, 4, 6)->key;
    cout << "\nLCA(3, 4) = " << findLCA(root, 3, 4)->key;
    cout << "\nLCA(2, 4) = " << findLCA(root, 2, 4)->key;
    return 0;
}
```

Java

```
//Java implementation to find lowest common ancestor of
// n1 and n2 using one traversal of binary tree

/* 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;
    }
}

public class BinaryTree
{
    //Root of the Binary Tree
    Node root;

    Node findLCA(int n1, int n2)
    {
        return findLCA(root, n1, n2);
    }

    // This function returns pointer to LCA of two given
    // values n1 and n2. This function assumes that n1 and
    // n2 are present in Binary Tree
    Node findLCA(Node node, int n1, int n2)
    {
        // Base case
        if (node == 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 (node.data == n1 || node.data == n2)
            return node;

        // Look for keys in left and right subtrees
        Node left_lca = findLCA(node.left, n1, n2);
        Node right_lca = findLCA(node.right, n1, n2);

        // 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!=null && right_lca!=null)
            return node;

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

    /* 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.left.right = new Node(5);
        tree.root.right.left = new Node(6);
        tree.root.right.right = new Node(7);
        System.out.println("LCA(4, 5) = " +
                            tree.findLCA(4, 5).data);
        System.out.println("LCA(4, 6) = " +
                            tree.findLCA(4, 6).data);
        System.out.println("LCA(3, 4) = " +
                            tree.findLCA(3, 4).data);
        System.out.println("LCA(2, 4) = " +
                            tree.findLCA(2, 4).data);
    }
}
```

Python

```
# Python program to find LCA of n1 and n2 using one
# traversal of Binary tree
# A binary tree node
class Node:

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

# This function returns pointer to LCA of two given
# values n1 and n2
# This function assumes that n1 and n2 are present in
# Binary Tree
def findLCA(root, n1, n2):

    # Base Case
    if root is None:
        return None

    # 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 or root.key == n2:
        return root

    # Look for keys in left and right subtrees
    left_lca = findLCA(root.left, n1, n2)
    right_lca = findLCA(root.right, n1, n2)

    # 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 and right_lca:
        return root

    # Otherwise check if left subtree or right subtree is LCA
    return left_lca if left_lca is not None else right_lca


# Driver program to test above function

# Let us create a binary tree given in the above example
root = Node(1)
root.left = Node(2)
root.right = Node(3)
root.left.left = Node(4)
root.left.right = Node(5)
root.right.left = Node(6)
root.right.right = Node(7)
```

```
print "LCA(4,5) = ", findLCA(root, 4, 5).key
print "LCA(4,6) = ", findLCA(root, 4, 6).key
print "LCA(3,4) = ", findLCA(root, 3, 4).key
print "LCA(2,4) = ", findLCA(root, 2, 4).key

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

Output:

```
LCA(4, 5) = 2
LCA(4, 6) = 1
LCA(3, 4) = 1
LCA(2, 4) = 2
```

Thanks to *Atul Singh* for suggesting this solution.

Time Complexity: Time complexity of the above solution is O(n) as the method does a simple tree traversal in bottom up fashion.
Note that the above method assumes that keys are present in Binary Tree. If one key is present and other is absent, then it returns the present key as LCA (Ideally should have returned NULL).
We can extend this method to handle all cases by passing two boolean variables v1 and v2. v1 is set as true when n1 is present in tree and v2 is set as true if n2 is present in tree.

C

```
/* Program to find LCA of n1 and n2 using one traversal of Binary Tree.
   It handles all cases even when n1 or n2 is not there in Binary Tree */
#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;
}
```

```
// This function returns pointer to LCA of two given values n1 and n2.
// v1 is set as true by this function if n1 is found
// v2 is set as true by this function if n2 is found
struct Node *findLCAUtil(struct Node* root, int n1, int n2, bool &v1, bool &v2)
{
    // Base case
    if (root == NULL) return NULL;

    // If either n1 or n2 matches with root's key, report the presence
    // by setting v1 or v2 as true and return root (Note that if a key
    // is ancestor of other, then the ancestor key becomes LCA)
    if (root->key == n1)
    {
        v1 = true;
        return root;
    }
    if (root->key == n2)
    {
        v2 = true;
        return root;
    }

    // Look for keys in left and right subtrees
    Node *left_lca  = findLCAUtil(root->left, n1, n2, v1, v2);
    Node *right_lca = findLCAUtil(root->right, n1, n2, v1, v2);

    // 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)  return root;

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

// Returns true if key k is present in tree rooted with root
bool find(Node *root, int k)
{
    // Base Case
    if (root == NULL)
        return false;

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

    // Else return false
    return false;
}

// This function returns LCA of n1 and n2 only if both n1 and n2 are present
// in tree, otherwise returns NULL;
Node *findLCA(Node *root, int n1, int n2)
{
    // Initialize n1 and n2 as not visited
    bool v1 = false, v2 = false;

    // Find lca of n1 and n2 using the technique discussed above
    Node *lca = findLCAUtil(root, n1, n2, v1, v2);

    // Return LCA only if both n1 and n2 are present in tree
    if (v1 && v2 || v1 && find(lca, n2) || v2 && find(lca, n1))
        return lca;

    // Else return NULL
    return NULL;
}

// 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);
    Node *lca =  findLCA(root, 4, 5);
    if (lca != NULL)
       cout << "LCA(4, 5) = " << lca->key;
    else
       cout << "Keys are not present ";

    lca =  findLCA(root, 4, 10);
    if (lca != NULL)
       cout << "\nLCA(4, 10) = " << lca->key;
    else
       cout << "\nKeys are not present ";

    return 0;
}
```

Java

```
// Java implementation to find lowest common ancestor of
// n1 and n2 using one traversal of binary tree
// It also handles cases even when n1 and n2 are not there in Tree

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

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

public class BinaryTree
{
    // Root of the Binary Tree
    Node root;
    static boolean v1 = false, v2 = false;

    // This function returns pointer to LCA of two given
    // values n1 and n2.
    // v1 is set as true by this function if n1 is found
    // v2 is set as true by this function if n2 is found
    Node findLCAUtil(Node node, int n1, int n2)
    {
        // Base case
        if (node == null)
            return null;

        // If either n1 or n2 matches with root's key, report the presence
        // by setting v1 or v2 as true and return root (Note that if a key
        // is ancestor of other, then the ancestor key becomes LCA)
        if (node.data == n1)
        {
            v1 = true;
            return node;
        }
        if (node.data == n2)
        {
            v2 = true;
            return node;
        }

        // Look for keys in left and right subtrees
        Node left_lca = findLCAUtil(node.left, n1, n2);
        Node right_lca = findLCAUtil(node.right, n1, n2);
        // 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 != null && right_lca != null)
            return node;

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

    // Finds lca of n1 and n2 under the subtree rooted with 'node'
    Node findLCA(int n1, int n2)
    {
        // Initialize n1 and n2 as not visited
        v1 = false;
        v2 = false;

        // Find lca of n1 and n2 using the technique discussed above
        Node lca = findLCAUtil(root, n1, n2);

        // Return LCA only if both n1 and n2 are present in tree
        if (v1 && v2)
            return lca;

        // Else return NULL
        return null;
    }

    /* 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.left.right = new Node(5);
        tree.root.right.left = new Node(6);
        tree.root.right.right = new Node(7);

        Node lca = tree.findLCA(4, 5);
        if (lca != null)
            System.out.println("LCA(4, 5) = " + lca.data);
        else
            System.out.println("Keys are not present");

        lca = tree.findLCA(4, 10);
        if (lca != null)
            System.out.println("LCA(4, 10) = " + lca.data);
        else
```

```
            System.out.println("Keys are not present");
    }
}
```

Python

```
""" Program to find LCA of n1 and n2 using one traversal of
 Binary tree
It handles all cases even when n1 or n2 is not there in tree
"""

# A binary tree node
class Node:

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

# This function retturn pointer to LCA of two given values
# n1 and n2
# v1 is set as true by this function if n1 is found
# v2 is set as true by this function if n2 is found
def findLCAUtil(root, n1, n2, v):

    # Base Case
    if root is None:
        return None

    # IF either n1 or n2 matches ith root's key, report
    # the presence by setting v1 or v2 as true and return
    # root (Note that if a key is ancestor of other, then
    # the ancestor key becomes LCA)
    if root.key == n1 :
        v[0] = True
        return root

    if root.key == n2:
        v[1] = True
        return root

    # Look for keys in left and right subtree
    left_lca = findLCAUtil(root.left, n1, n2, v)
    right_lca = findLCAUtil(root.right, n1, n2, v)

    # 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 and right_lca:
        return root

    # Otherwise check if left subtree or right subtree is LCA
    return left_lca if left_lca is not None else right_lca


def find(root, k):

    # Base Case
    if root is None:
        return False

    # If key is present at root, or if left subtree or right
    # subtree , return true
    if (root.key == k or find(root.left, k) or
        find(root.right, k)):
        return True

    # Else return false
    return False

# This function returns LCA of n1 and n2 onlue if both
# n1 and n2 are present in tree, otherwise returns None
def findLCA(root, n1, n2):

    # Initialize n1 and n2 as not visited
    v = [False, False]

    # Find lac of n1 and n2 using the technique discussed above
    lca = findLCAUtil(root, n1, n2, v)

    # Returns LCA only if both n1 and n2 are present in tree
    if (v[0] and v[1] or v[0] and find(lca, n2) or v[1] and
        find(lca, n1)):
        return lca

    # Else return None
    return None

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

```
lca = findLCA(root, 4, 5)

if lca is not None:
    print "LCA(4, 5) = ", lca.key
else :
    print "Keys are not present"

lca = findLCA(root, 4, 10)
if lca is not None:
    print "LCA(4,10) = ", lca.key
else:
    print "Keys are not present"

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

Output:

```
LCA(4, 5) = 2
Keys are not present
```

Thanks to Dhruv for suggesting this extended solution.

You may like to see below articles as well :

LCA using Parent Pointer

Lowest Common Ancestor in a Binary Search Tree.

Find LCA in Binary Tree using RMQ

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

18 Comments Category: Trees

Related Posts:

- Check if removing an edge can divide a Binary Tree in two halves
- Check sum of Covered and Uncovered nodes of Binary Tree
- Lowest Common Ancestor in a Binary Tree | Set 2 (Using Parent Pointer)
- Construct a Binary Search Tree from given postorder
- BFS vs DFS for Binary Tree
- Maximum difference between node and its ancestor in Binary Tree
- Inorder Non-threaded Binary Tree Traversal without Recursion or Stack
- Check if leaf traversal of two Binary Trees is same?

Like Share 46 people like this.

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

bhopu

this code handle all cases:

```
#include
#include
struct tree {
int data;
struct tree *lchild;
struct tree *rchild;
};
struct tree *getnode(int data){
struct tree *newnode=(struct tree *)malloc(sizeof(struct tree));
newnode->data=data;
newnode->lchild=NULL;
newnode->rchild=NULL;
return newnode;
}
traverse(struct tree *ptr){
if(ptr){
printf(" %d",ptr->data);
traverse(ptr->lchild);
traverse(ptr->rchild);
}
}
lca(struct tree *ptr,int n1,int n2){
int l=0,r=0,lr=0;
if(!ptr)
return (l||r);
l =lca(ptr->lchild,n1,n2);
r=lca(ptr->rchild,n1,n2);
if(l&&r||((ptr->data==n1||ptr->data==n2)&&(l||r)))
printf(" nn lca: %dn ",ptr->data);
if(ptr->data==n1||ptr->data==n2)
return 1;
return (l||r);
}
main(){
struct tree *ptr=NULL,*root=NULL;
root=getnode(1);
root->lchild=getnode(2);
root->rchild=getnode(3);
root->lchild->lchild=getnode(4);
root->lchild->rchild=getnode(5);
root->rchild->lchild=getnode(6);
root->rchild->rchild=getnode(7);
traverse(root);
lca(root,3,8);
}
```

sandy

```
ListNode LCA(BinaryTreeNode root , BinaryTreeNode a , BinaryTreeNode b ){
if(root==null) return ;
if(root ==a || root==b) return root;
ListNode left = LCA(root.getleft(),a,b);
ListNode right = LCA(root.getright(),a,b);
if(left && right) return root;
else return left?left:right ;
}
```

techomaniac

Guys I think I have a different solution for this. You can do this by using level order traversal. The time complexity wll be O(n) and space complexity will be O(1). First check with level order traversal which of n1 or n2 is higher level. Now say if n1 is higher then check whether n2 lies in the same subtree from the root. If it is in diffrent subtree from the root then root node is LCA else parent of n1 is LCA.

Kartik

How can we do level order traversal in O(n) time and O(1) space, it is not possible as far as I know

wliao

I think this method would work iff assuming the level orders of n1 and n2 are the same. If they aren't there will be a problem, like finding lca of 4, 7.

wliao

Oops, I meant iff the level orders are NOT the same.

Rahul

3rd method is awesome, can get the answer in just one traversal 🙂

Guest

I do not think these 3 methods work well if there exist duplicates in the tree. Please consider the following tree:

```
Node * root = newNode(1);
root->left = newNode(11);
root->right = newNode(11);
root->left->left = newNode(11);
root->left->right = newNode(12);
root->left->right->left = newNode(6);
root->left->right->right = newNode(11);
```

LCA of 6 and 11 should be 12, but method 1 returns 11, method 2 and improved method 2 return 1. I think the reason is that these three methods are trying to find LCA in top down fashion. I believe we need to find LCA in bottom up fashion in order to eliminate fictitious LCA. Please correct me if I am wrong.

micintosh

I made the comment above, and it is wrong; sorry about that.

bani

Suppose we have only 1 of the 2 nodes present in our tree … how to come to a solution in just a single traversal of our binary tree…..???

Kartik

Doesn't look possible to find lca if one of the keys is absent.

Swarup Mallick

I think the assumption is both n1 and n2 must present.

Dhruv

You can take 2 booleans indicating whether n1 and n2 are present or not . And finally after the traversal ,use them at the end to make decisions.

GeeksforGeeks

Dhruv, thanks for sharing your thoughts. We have added extended solution to the original post.

bani

GeeksforGeeks ,@Dhruv , i dont feel taking 2 variables would work ..
Suppose that in the given figure (at the top of this page)we need to find the lca of 2 and 4 … as per the 3rd solution since node with 2 will be encountered first the variable v1 will be set…. at that time 4 has not been discovered so no chance of setting v2… according to this condition

```
if (root->key == n1)
{
v1 = true;
return root;
}
```

after finding 2 we will return …since 4 lies one level down 2 so its v2 will never be set in this case….
I think the 3rd solution will also be flawing in this case…
this flaw arises even if 2 nodes lie in the tree but their boolean variables cant be set…
the answer would be NULL in that case….
we need some work to be done to solve this edge case….
what do u say??

bani

okk i got the essence of your code..its too awesome …but still cant it be done in a single traversal ??? I think in the worst case the 3rd method would take 3 traversals …a better approach would be 2 initially search if both the nodes are present in the tree or not and then answer out….

GeeksforGeeks

bani, please take a closer look. When we find either n1 or n2, we stop at that node in findLCAUtil().
In findLCA(), we traverse only only the subtree rooted with either n1 or n2. So in worst case, we visit every node once, except one node that has n1 or n2.

Gopal Shankar

Isn't single Boolean serve the purpose ? I see that except for following return path, other two return path does not speak that both keys are present.

```
// 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)
{
found= true; // new flag ?
return root;
}
```