GeeksforGeeks

A computer science portal for geeks

GeeksQuiz

Login

Home Algorithms DS GATE Interview Corner Q&A C C++ Java Books Contribute Ask a Q About

Array Bit Magic C/C++ Articles GFacts Linked List MCQ Misc Output String Tree Graph

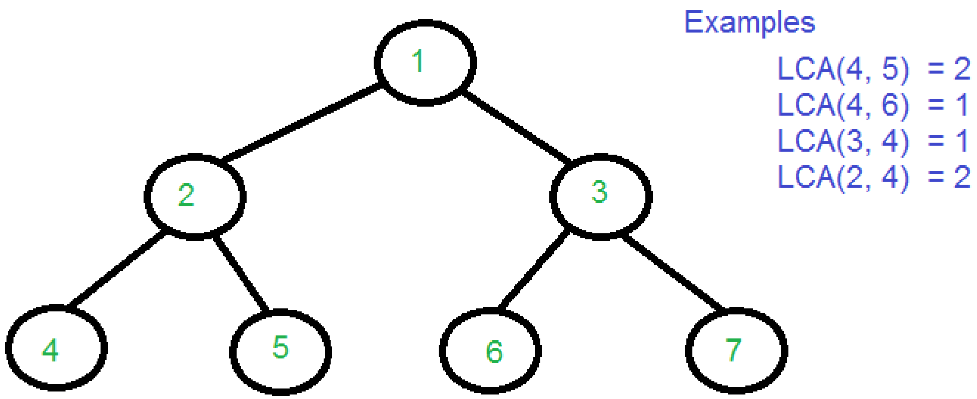
Lowest Common Ancestor in a Binary Tree | Set 1

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).

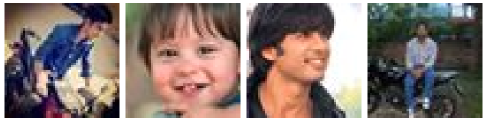
The LCA of n1 and n2 in T is the shared ancestor of n1 and n2 that is located farthest from the root. Computation of lowest common ancestors may be useful, for instance, as part of a procedure for determining the distance between pairs of nodes in a tree: the distance from n1 to n2 can be computed as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



Examples

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

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



Interview Experiences
Advanced Data Structures
Dynamic Programming
Greedy Algorithms
Backtracking
Pattern Searching
Divide & Conquer
Mathematical Algorithms
Recursion
Geometric Algorithms

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.

```
// 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 fals
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);
```

```
public float GetSaturation()
{
    float r = (float)this.R / 255f;
    float g = (float)this.G / 255f;
    float b = (float)this.B / 255f;
    float single = 0f;
    float single1 = r;
    float single2 = r;
```

Popular Posts

All permutations of a given string
Memory Layout of C Programs
Understanding "extern" keyword in C
Median of two sorted arrays
Tree traversal without recursion and without stack!
Structure Member Alignment, Padding and Data Packing
Intersection point of two Linked Lists
Lowest Common Ancestor in a BST.
Check if a binary tree is BST or not
Sorted Linked List to Balanced BST

```
    // 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;
}
```

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.

```
/* 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)
```

Recent Comments

affiszerv Your example has two 4s on row 3, that's why it...

Backtracking | Set 7 (Sudoku) · 27 minutes ago

RVM Can someone please elaborate this Qs from above...

Flipkart Interview | Set 6 · 47 minutes ago

Vishal Gupta I talked about as an Interviewer in general,...

Software Engineering Lab, Samsung Interview | Set 2 · 47 minutes ago

@meya Working solution for question 2 of 4f2f round....

Amazon Interview | Set 53 (For SDE-1) · 1 hour ago

sandeep void rearrange(struct node *head) {...

Given a linked list, reverse alternate nodes and append at the end · 2 hours ago

Neha I think that is what it should return as, in...

Find depth of the deepest odd level leaf node · 2 hours ago

```
{
    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;
}
```

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.

```
/* 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,
{
    // 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 p
// 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;
}
```

Output:

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

Thanks to Dhruv for suggesting this extended solution.

We will soon be discussing more solutions to this problem. Solutions considering the following.
1) If there are many LCA queries and we can take some extra preprocessing time to reduce the time taken to find LCA.
2) If parent pointer is given with every node.

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

Related Tpoics:

- Print a Binary Tree in Vertical Order | Set 2 (Hashmap based Method)
- Print Right View of a Binary Tree
- Red-Black Tree | Set 3 (Delete)
- Construct a tree from Inorder and Level order traversals
- Print all nodes at distance k from a given node

- Print a Binary Tree in Vertical Order | Set 1
- Interval Tree
- Check if a given Binary Tree is height balanced like a Red-Black Tree

39 Tweet 3 1

Writing code in comment? Please use **ideone.com** and share the link here.

18 Comments **GeeksforGeeks**

Sort by Newest ▾

Join the discussion…

bhopu • 8 days ago

this code handle all cases:

```
#include<stdio.h>

#include<stdlib.h>

struct tree {

int data;

struct tree *lchild;

struct tree *rchild;

};

struct tree *getnode(int data){

struct tree *newnode=(struct tree *)malloc(sizeof(struct tree));
```

see more

^ | ⌄ • Reply • Share ›

sandy • a month ago

```
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 ;

}
```

^ | ⌄ • Reply • Share ›

techomaniac • 2 months ago

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

^ | ⌄ • Reply • Share ›

wliao ➜ techomaniac • 2 months ago

I think this method would work iff assuming the level orders of n1 and r
be a problem, like finding lca of 4, 7.

^ | ⌄ • Reply • Share ›

wliao ➦ wliao • 2 months ago

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

⌃ | ⌄ • Reply • Share ›

Kartik ➦ techomaniac • 2 months ago

How can we do level order traversal in O(n) time and O(1) space, it is

⌃ | ⌄ • Reply • Share ›

Rahul • 2 months ago

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

⌃ | ⌄ • Reply • Share ›

Guest • 2 months ago

I do not think these 3 methods work well if there exist duplicates in the tree. Pl

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 improv
is that these three methods are trying to find LCA in top down fashion. I believe
fashion in order to eliminate fictitious LCA. Please correct me if I am wrong.

⌃ | ⌄ • Reply • Share ›

micintosh ➦ Guest • 2 months ago

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

⌃ | ⌄ • Reply • Share ›

bani • 2 months ago

Suppose we have only 1 of the 2 nodes present in our tree ... how to come to
our binary tree.....???

1 ^ | ˅ • Reply • Share ›

Dhruv ➦ bani • 2 months ago

You can take 2 booleans indicating whether n1 and n2 are present or n
them at the end to make decisions.

2 ^ | ˅ • Reply • Share ›

GeeksforGeeks Mod ➦ Dhruv • 2 months ago

Dhruv, thanks for sharing your thoughts. We have added exten

1 ^ | ˅ • Reply • Share ›

Gopal Shankar ➦ GeeksforGeeks • 2 months ago

Isn't single Boolean serve the purpose ? I see that exce
return path does not speak that both keys are present.

```
// If both of the above calls return Non-NULL, then one k
// is present in once subtree and other is present in othe
// So this node is the LCA
if (left_lca && right_lca)
{
found= true; // new flag ?
return root;
}
```

^ | ˅ • Reply • Share ›

bani ➦ GeeksforGeeks • 2 months ago

GeeksforGeeks ,@Dhruv , i dont feel taking 2 variables
Suppose that in the given figure (at the top of this page)
per the 3rd solution since node with 2 will be encountere
that time 4 has not been discovered so no chance of se

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

after finding 2 we will return ...since 4 lies one level dow
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 bo
the answer would be NULL in that case....
we need some work to be done to solve this edge case
what do u say??

⌃ | ⌄ • Reply • Share ›

bani ➦ bani • 2 months ago

okk i got the essence of your code..its too awesome ...b
single traversal ??? I think in the worst case the 3rd me
better approach would be 2 initially search if both the no
not and then answer out....

⌃ | ⌄ • Reply • Share ›

GeeksforGeeks Mod ➦ bani • 2 months ago

bani, please take a closer look. When we find either n1
findLCAUtil().
In findLCA(), we traverse only only the subtree rooted w
case, we visit every node once, except one node that ha

1 ⌃ | ⌄ • Reply • Share ›

Kartik ➦ bani • 2 months ago

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

^ | ˅ • Reply • Share ›

Swarup Mallick ➦ Kartik • 2 months ago

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

^ | ˅ • Reply • Share ›

✉ Subscribe   Add Disqus to your site