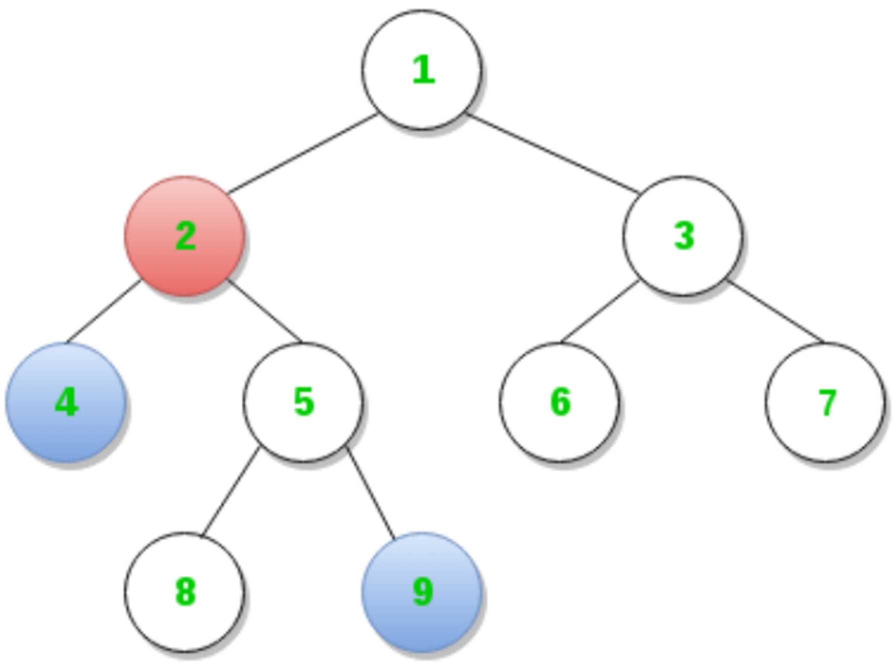
Find LCA in Binary Tree using RMQ

The article describes an approach to solving the problem of finding the LCA of two nodes in a tree by reducing it to a RMQ problem.

Lowest Common Ancestor (LCA) of two nodes u and v in a rooted tree T is defined as the node located farthest from the root that has both u and v as descendants.

For example, in below diagram, LCA of node 4 and node 9 is node 2.

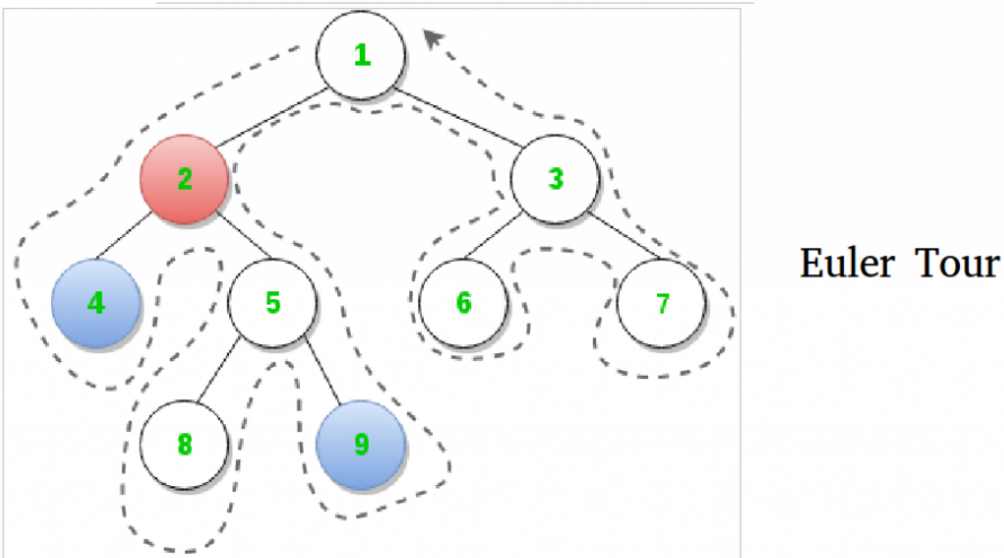


There can be many approaches to solve the LCA problem. The approaches differ in their time and space complexities. Here is a link to a couple of them (these do not involve reduction to RMQ).

Range Minimum Query (RMQ) is used on arrays to find the position of an element with the minimum value between two specified indices. Different approaches for solving RMQ have been discussed here and here. In this article, Segment Tree based approach is discussed. With segment tree, preprocessing time is O(n) and time to for range minimum query is O(Logn). The extra space required is O(n) to store the segment tree.

Reduction of LCA to RMQ:

The idea is to traverse the tree starting from root by an Euler tour (traversal without lifting pencil), which is a DFS-type traversal with preorder traversal characteristics.



An euler tour of the tree starting from node 1 will yield:

1	2	4	2	5	8	5	9	5	2	1	3	6	3	7	3	1

The corresponding levels for every node in Euler tour:

0	1	2	1	2	3	2	3	2	1	0	1	2	1	2	1	0

Observation: The LCA of nodes 4 and 9 is node 2, which happens to be the node closest to the root amongst all those encountered between the visits of 4 and 9 during a DFS of T. This observation is the key to the reduction. Let's rephrase: Our node is the node at the smallest level and the only node at that level amongst all the nodes that occur between consecutive occurrences (any) of u and v in the Euler tour of T.

We require three arrays for implementation:

1. Nodes visited in order of Euler tour of T
2. Level of each node visited in Euler tour of T
3. Index of the **first** occurrence of a node in Euler tour of T (since any occurrence would be good, let's track the first one)

The first occurrences corresponding to every node in Euler tour of T:

Node	1	2	3	4	5	6	7	8	9
First Occurrence	0	1	11	2	4	12	14	5	7

Algorithm:

1. Do a Euler tour on the tree, and fill the euler, level and first occurrence arrays.

2. Using the first occurrence array, get the indices corresponding to the two nodes which will be the corners of the range in the level array that is fed to the RMQ algorithm for the minimum value.
3. Once the algorithm return the index of the minimum level in the range, we use it to determine the LCA using Euler tour array.

Below is the implementation of above algorithm.

C++

```
/* C++ Program to find LCA of u and v by reducing the problem to RMQ */
#include<bits/stdc++.h>
#define V 9               // number of nodes in input tree

int euler[2*V - 1];       // For Euler tour sequence
int level[2*V - 1];       // Level of nodes in tour sequence
int firstOccurrence[V+1]; // First occurences of nodes in tour
int ind;                  // Variable to fill-in euler and level arrays

// A Binary 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;
}

// log base 2 of x
int Log2(int x)
{
    int ans = 0 ;
    while (x>>=1) ans++;
    return ans ;
}

/*  A recursive function to get the minimum value in a given range
     of array indexes. The following are parameters for this function.

    st    --> Pointer to segment tree
    index --> Index of current node in the segment tree. Initially
              0 is passed as root is always at index 0
    ss & se  --> Starting and ending indexes of the segment represented
                by current node, i.e., st[index]
    qs & qe  --> Starting and ending indexes of query range */
int RMQUtil(int index, int ss, int se, int qs, int qe, int *st)
{
    // If segment of this node is a part of given range, then return
    //  the min of the segment
    if (qs <= ss && qe >= se)
        return st[index];

    // If segment of this node is outside the given range
    else if (se < qs || ss > qe)
```

```
        return -1;

    // If a part of this segment overlaps with the given range
    int mid = (ss + se)/2;

    int q1 = RMQUtil(2*index+1, ss, mid, qs, qe, st);
    int q2 = RMQUtil(2*index+2, mid+1, se, qs, qe, st);

    if (q1==-1) return q2;

    else if (q2==-1) return q1;

    return (level[q1] < level[q2]) ? q1 : q2;
}

// Return minimum of elements in range from index qs (quey start) to
// qe (query end).  It mainly uses RMQUtil()
int RMQ(int *st, int n, int qs, int qe)
{
    // Check for erroneous input values
    if (qs < 0 || qe > n-1 || qs > qe)
    {
        printf("Invalid Input");
        return -1;
    }

    return RMQUtil(0, 0, n-1, qs, qe, st);
}

// A recursive function that constructs Segment Tree for array[ss..se].
// si is index of current node in segment tree st
void constructSTUtil(int si, int ss, int se, int arr[], int *st)
{
    // If there is one element in array, store it in current node of
    // segment tree and return
    if (ss == se)st[si] = ss;

    else
    {
        // If there are more than one elements, then recur for left and
        // right subtrees and store the minimum of two values in this node
        int mid = (ss + se)/2;
        constructSTUtil(si*2+1, ss, mid, arr, st);
        constructSTUtil(si*2+2, mid+1, se, arr, st);

        if (arr[st[2*si+1]] < arr[st[2*si+2]])
            st[si] = st[2*si+1];
        else
            st[si] = st[2*si+2];
    }
}

/* Function to construct segment tree from given array. This function
   allocates memory for segment tree and calls constructSTUtil() to
   fill the allocated memory */
int *constructST(int arr[], int n)
{
    // Allocate memory for segment tree

    // Height of segment tree
    int x = Log2(n)+1;

    // Maximum size of segment tree
    int max_size = 2*(1<<x) - 1;  //  2*pow(2,x) -1
```

```
    int *st = new int[max_size];

    // Fill the allocated memory st
    constructSTUtil(0, 0, n-1, arr, st);

    // Return the constructed segment tree
    return st;
}

// Recursive version of the Euler tour of T
void eulerTour(Node *root, int l)
{
    /* if the passed node exists */
    if (root)
    {
        euler[ind] = root->key; // insert in euler array
        level[ind] = l;         // insert l in level array
        ind++;                  // increment index

        /* if unvisited, mark first occurrence */
        if (firstOccurrence[root->key] == -1)
            firstOccurrence[root->key] = ind-1;

        /* tour left subtree if exists, and remark euler
           and level arrays for parent on return */
        if (root->left)
        {
            eulerTour(root->left, l+1);
            euler[ind]=root->key;
            level[ind] = l;
            ind++;
        }

        /* tour right subtree if exists, and remark euler
           and level arrays for parent on return */
        if (root->right)
        {
            eulerTour(root->right, l+1);
            euler[ind]=root->key;
            level[ind] = l;
            ind++;
        }
    }
}

// Returns LCA of nodes n1, n2 (assuming they are
//  present in the tree)
int findLCA(Node *root, int u, int v)
{
    /* Mark all nodes unvisited.  Note that the size of
        firstOccurrence is 1 as node values which vary from
        1 to 9 are used as indexes */
    memset(firstOccurrence, -1, sizeof(int)*(V+1));

    /* To start filling euler and level arrays from index 0 */
    ind = 0;

    /* Start Euler tour with root node on level 0 */
    eulerTour(root, 0);

    /* construct segment tree on level array */
    int *st = constructST(level, 2*V-1);

    /* If v before u in Euler tour.  For RMQ to work, first
       parameter 'u' must be smaller than second 'v' */
```

```
    if (firstOccurrence[u]>firstOccurrence[v])
       std::swap(u, v);

    // Starting and ending indexes of query range
    int qs = firstOccurrence[u];
    int qe = firstOccurrence[v];

    // query for index of LCA in tour
    int index = RMQ(st, 2*V-1, qs, qe);

    /* return LCA node */
    return euler[index];
}

// Driver program to test above functions
int main()
{
    // Let us create the Binary Tree as shown in the 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);
    root->left->right->left = newNode(8);
    root->left->right->right = newNode(9);

    int u = 4, v = 9;
    printf("The LCA of node %d and node %d is node %d.\n",
            u, v, findLCA(root, u, v));
    return 0;
}
```

Run on IDE

Java

```
// Java program to find LCA of u and v by reducing problem to RMQ

import java.util.*;

// A binary tree node
class Node {

    Node left, right;
    int data;

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

class St_class {

    int st;
    int stt[] = new int[10000];
}

class BinaryTree {
```

```
    static Node root;
    int v = 9; // v is the highest value of node in our tree
    int euler[] = new int[2 * v - 1]; // for euler tour sequence
    int level[] = new int[2 * v - 1]; // level of nodes in tour sequence
    int f_occur[] = new int[2 * v - 1]; // to store 1st occurance of nodes in tour
    int fill; // variable to fill euler and level arrays
    St_class sc = new St_class();

    // log base 2 of x
    int Log2(int x) {
        int ans = 0;
        int y = x >>= 1;
        while (y-- != 0) {
            ans++;
        }
        return ans;
    }

    int swap(int a, int b) {
        return a;
    }

    /*  A recursive function to get the minimum value in a given range
     of array indexes. The following are parameters for this function.

     st    --> Pointer to segment tree
     index --> Index of current node in the segment tree. Initially
     0 is passed as root is always at index 0
     ss & se  --> Starting and ending indexes of the segment represented
     by current node, i.e., st[index]
     qs & qe  --> Starting and ending indexes of query range */
    int RMQUtil(int index, int ss, int se, int qs, int qe, St_class st) {

        // If segment of this node is a part of given range, then return
        //  the min of the segment
        if (qs <= ss && qe >= se) {
            return st.stt[index];
        }

        // If segment of this node is outside the given range
        else if (se < qs || ss > qe) {
            return -1;
        }

        // If a part of this segment overlaps with the given range
        int mid = (ss + se) / 2;

        int q1 = RMQUtil(2 * index + 1, ss, mid, qs, qe, st);
        int q2 = RMQUtil(2 * index + 2, mid + 1, se, qs, qe, st);

        if (q1 == -1) {
            return q2;
        } else if (q2 == -1) {
            return q1;
        }

        return (level[q1] < level[q2]) ? q1 : q2;
    }

    // Return minimum of elements in range from index qs (quey start) to
    // qe (query end).  It mainly uses RMQUtil()
    int RMQ(St_class st, int n, int qs, int qe) {

        // Check for erroneous input values
        if (qs < 0 || qe > n - 1 || qs > qe) {
```

```
            System.out.println("Invalid input");
            return -1;
        }

        return RMQUtil(0, 0, n - 1, qs, qe, st);
    }

    // A recursive function that constructs Segment Tree for array[ss..se].
    // si is index of current node in segment tree st
    void constructSTUtil(int si, int ss, int se, int arr[], St_class st) {

        // If there is one element in array, store it in current node of
        // segment tree and return
        if (ss == se) {
            st.stt[si] = ss;
        } else {

            // If there are more than one elements, then recur for left and
            // right subtrees and store the minimum of two values in this node
            int mid = (ss + se) / 2;
            constructSTUtil(si * 2 + 1, ss, mid, arr, st);
            constructSTUtil(si * 2 + 2, mid + 1, se, arr, st);

            if (arr[st.stt[2 * si + 1]] < arr[st.stt[2 * si + 2]]) {
                st.stt[si] = st.stt[2 * si + 1];
            } else {
                st.stt[si] = st.stt[2 * si + 2];
            }
        }
    }

    /* Function to construct segment tree from given array. This function
     allocates memory for segment tree and calls constructSTUtil() to
     fill the allocated memory */
    int constructST(int arr[], int n) {

        // Allocate memory for segment tree
        // Height of segment tree
        int x = Log2(n) + 1;

        // Maximum size of segment tree
        int max_size = 2 * (1 << x) - 1;  //  2*pow(2,x) -1

        sc.stt = new int[max_size];

        // Fill the allocated memory st
        constructSTUtil(0, 0, n - 1, arr, sc);

        // Return the constructed segment tree
        return sc.st;
    }

    // Recursive version of the Euler tour of T
    void eulerTour(Node node, int l) {

        /* if the passed node exists */
        if (node != null) {
            euler[fill] = node.data; // insert in euler array
            level[fill] = l;         // insert l in level array
            fill++;                  // increment index

            /* if unvisited, mark first occurrence */
            if (f_occur[node.data] == -1) {
                f_occur[node.data] = fill - 1;
            }
```

```
        /* tour left subtree if exists, and remark euler
         and level arrays for parent on return */
        if (node.left != null) {
            eulerTour(node.left, l + 1);
            euler[fill] = node.data;
            level[fill] = l;
            fill++;
        }

        /* tour right subtree if exists, and remark euler
         and level arrays for parent on return */
        if (node.right != null) {
            eulerTour(node.right, l + 1);
            euler[fill] = node.data;
            level[fill] = l;
            fill++;
        }
    }
}

// returns LCA of node n1 and n2 assuming they are present in tree
int findLCA(Node node, int u, int v) {

    /* Mark all nodes unvisited.  Note that the size of
     firstOccurrence is 1 as node values which vary from
     1 to 9 are used as indexes */
    Arrays.fill(f_occur, -1);

    /* To start filling euler and level arrays from index 0 */
    fill = 0;

    /* Start Euler tour with root node on level 0 */
    eulerTour(root, 0);

    /* construct segment tree on level array */
    sc.st = constructST(level, 2 * v - 1);

    /* If v before u in Euler tour.  For RMQ to work, first
     parameter 'u' must be smaller than second 'v' */
    if (f_occur[u] > f_occur[v]) {
        u = swap(u, u = v);
    }

    // Starting and ending indexes of query range
    int qs = f_occur[u];
    int qe = f_occur[v];

    // query for index of LCA in tour
    int index = RMQ(sc, 2 * v - 1, qs, qe);

    /* return LCA node */
    return euler[index];

}

// Driver program to test above functions
public static void main(String args[]) {
    BinaryTree tree = new BinaryTree();

    // Let us create the Binary Tree as shown in the diagram.
    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);
        tree.root.left.right.left = new Node(8);
        tree.root.left.right.right = new Node(9);

        int u = 4, v = 9;
        System.out.println("The LCA of node " + u + " and " + v + " is "
                + tree.findLCA(root, u, v));
    }

}
```

Run on IDE

Output:

```
The LCA of node 4 and node 9 is node 2.
```

Note:

1. We assume that the nodes queried are present in the tree.
2. We also assumed that if there are V nodes in tree, then keys (or data) of these nodes are in range from 1 to V.

Time complexity:

1. Euler tour: Number of nodes is V. For a tree, E = V-1. Euler tour (DFS) will take O(V+E) which is O(2*V) which can be written as O(V).
2. Segment Tree construction : O(n) where n = V + E = 2*V – 1.
3. Range Minimum query: O(log(n))

Overall this method takes O(n) time for preprocssing, but takes O(Log n) time for query. Therefore, it can be useful when we have a single tree on which we want to perform large number of LCA queries (Note that LCA is useful for finding shortest path between two nodes of Binary Tree)

Auxiliary Space:

1. Euler tour array: O(n) where n = 2*V – 1
2. Node Levels array: O(n)
3. First Occurrences array: O(V)
4. Segment Tree: O(n)

Overall: O(n)

Another observation is that the adjacent elements in level array differ by 1. This can be used to convert a RMQ problem to a LCA problem.

This article is contributed by **Yash Varyani**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

4 Comments Category: Advanced Data Structure Trees

Related Posts:

- Centroid Decomposition of Tree
- Gomory-Hu Tree | Set 1 (Introduction)
- kasai's Algorithm for Construction of LCP array from Suffix Array
- Overview of Data Structures | Set 3 (Graph, Trie, Segment Tree and Suffix Tree)
- Heavy Light Decomposition | Set 2 (Implementation)
- Heavy Light Decomposition | Set 1 (Introduction)
- Count Inversions of size three in a give array
- Count inversions in an array | Set 3 (Using BIT)

(Login to Rate and Mark)

4.1 Average Difficulty : **4.1/5.0**
Based on **9** vote(s)

Add to TODO List

Mark as DONE

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

4 Comments **GeeksforGeeks** 1 Login

Recommend Share Sort by Newest

Join the discussion…

RacUnite • 2 months ago

excellent article

• Reply • Share ›

holdnet • 3 months ago

Very informative article written nicely!

• Reply • Share ›

Govind s Prabhu • 3 months ago

Size of firstOccurrence will be too large if we have data in the range is large? It is storing first occurrence of each element in taking the element as an index. Isn't it too inefficient?

• Reply • Share ›

Yash → Govind s Prabhu • 3 months ago

In the above scenario, we have the data of nodes and the nodes themselves identical. You can have your own way of asking the user of which two nodes to get the LCA of. That way, the size of firstOccurence will be number of vertices. Read the space complexity analysis just in case you missed.

• Reply • Share ›

Subscribe Add Disqus to your site Add Disqus Add Privacy