Trees

A **tree** is an undirected connected graph with no cycles.

Equivalently, a **tree** is a connected graph with N nodes and N - 1 edges.

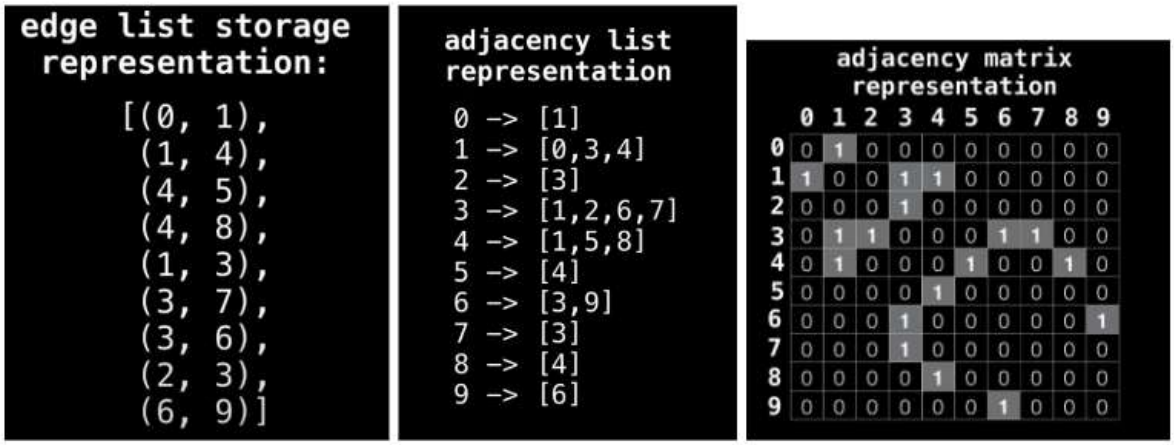
Equivalently, a **tree** is a graph in which any tw vertices are connected by exactly one path.

The **height** of a tree is a number of edges from the root to the lowest leaf.

The **degree** of a node is the number of nodes it is connected to.

Tree representations:

- Edge list (fast to iterate over, cheap to store BUT storing a tree as a list lacks the structure to efficiently quarry all the neighbors of a node)
- Adjacency list
- Adjacency matrix (not efficient in terms of memory - O(n^2))



Depth-first Traversal/Search:

A depth-first traversal starts at the root, and **always visits the children of a visited node before its siblings**. It does not make any guarantees about the order in which nodes on a particular level are visited.

Pseudocode:

```
Algorithm depthFirstTraversal(N)
Parameters:
    N is a tree node

visit node N
for each child T of N   // Assume children processed left-to-right
    depthFirstTraversal(T)
```

Iterative implementation with stack (js):

```
23
24 const depthFirstValues = (root) => {
25   if (root === null) return [];
26
27   const result = [];
28   const stack = [ root ]; // [ null ]
29   while (stack.length > 0) {
30     const current = stack.pop();
31     result.push(current.val);
32
33     if (current.right) stack.push(current.right);
34     if (current.left) stack.push(current.left);
35   }
36   return result;
37 };
```

We can change the order of lines 33 and 34 to get a desired order of depth-first traversal.

Recursive implementation (js):

```
24 const depthFirstValues = (root) => {
25   if (root === null) return [];
26   const leftValues = depthFirstValues(root.left); // [b, d, e]
27   const rightValues = depthFirstValues(root.right); // [c, f]
28   return [ root.val, ...leftValues, ...rightValues ];
29 };
```

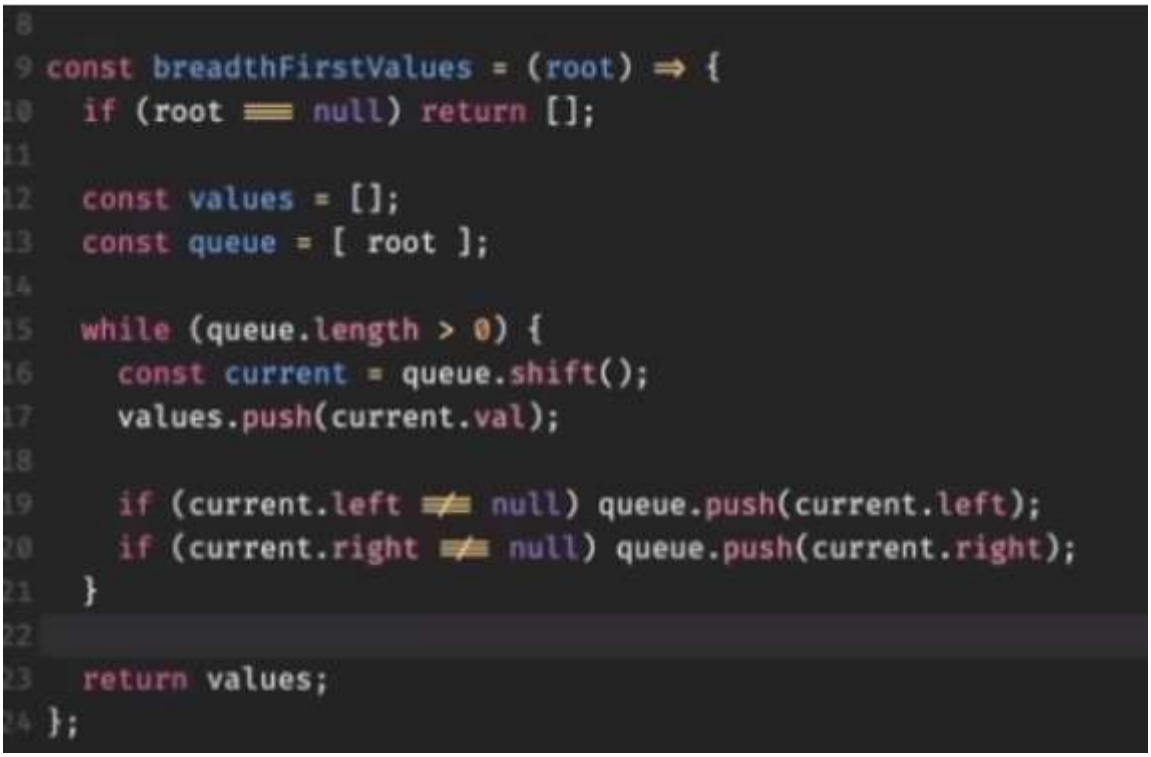
Complexity: O(n)

Space: O(n)

Breadth-first Traversal/Search:

A breadth-first traversal starts at the root, and **always visits all the nodes on the current level of the tree before visiting nodes on the next level**. It does not make any guarantees about the order in which nodes on a particular level are visited.

Iterative implementation with queue (js):



```
const breadthFirstValues = (root) => {
  if (root === null) return [];

  const values = [];
  const queue = [ root ];

  while (queue.length > 0) {
    const current = queue.shift();
    values.push(current.val);

    if (current.left !== null) queue.push(current.left);
    if (current.right !== null) queue.push(current.right);
  }

  return values;
};
```

Complexity: O(n)

Space: O(n)

Special cases:

1. V: visit the node
2. L: traverse the left subtree of the node
3. R: traverse the right subtree of the node

Pre-order Traversal: The pre-order traversal is a special case of the **depth-first traversal**. In the pre-order traversal, we start at the root, and then perform the above operations in the order: **VLR**. That is, every node is visited before both of its subtrees, and then we always traverse the left subtree before the right subtree.

In-order Traversal: The in-order traversal performs the traversal operations in the sequence: **LVR**, starting at the root. That is, the left subtree of a node is always traversed before the node is visited, and the right subtree of a node is always traversed after the node is visited. **In-order traversal is neither breadth-first, nor depth-first traversal**.

Post-order Traversal: The post-order traversal starts at the root and performs the traversal operations in the sequence: **LRV**. That is, both the left and right subtrees of a node t are traversed before t is visited. Again, this is **neither a breadth-first, nor depth-first traversal**.

These traversals can be extended to n-ary trees, not just binary.

Binary Search Trees (BST)

Binary Search Trees are **trees** which satisfy the **BST invariant** which states that for every node x:

x.left.value <= x.value <= x.right.value

Storing Rooted Trees

Maintain a pointer reference to the **root node**.

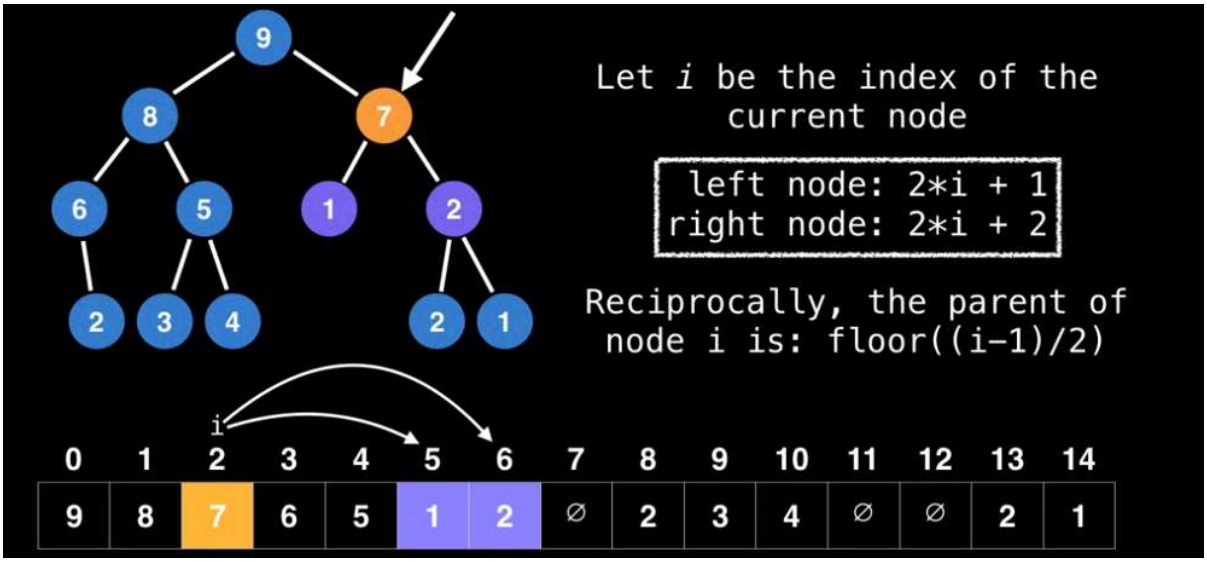
Each node has access to a list of its **children**.

Sometimes it is useful to maintain a pointer to a node's **parent node** effectively making edges **bidirectional**.

Flattened array

If your tree is a **binary tree**, you can store it in a **flattened array** where each node has an assigned index position based on where it is in the tree.

The root node is always at index 0 and the children of the current node i are accessed relative to position i.



Theoretical Problems

Problem 1: Leaf Node Sum

What is the sum of all leaf node values in a tree?

```
# Sums up leaf node values in a tree.
# Call function like: leafSum(root)
function leafSum(node):
  # Handle empty tree case
  if node == null:
    return 0
  if isLeaf(node):
    return node.getValue()
  total = 0
  for child in node.getChildNodes():
    total += leafSum(child)
  return total

function isLeaf(node):
  return node.getChildNodes().size() == 0
```

Problem 2: Tree Height

Find the height of a binary tree.

```
# The height of a tree is the number of
# edges from the root to the lowest leaf.
function treeHeight(node):
  # Handle empty tree case
  if node == null:
    return -1

  # Identify leaf nodes and return zero
  if node.left == null and node.right == null:
    return 0

  return max(treeHeight(node.left),
             treeHeight(node.right)) + 1
```

OR

```
# The height of a tree is the number of
# edges from the root to the lowest leaf.
function treeHeight(node):
  # Return -1 when we hit a null node
  # to correct for the right height.
  if node == null:
      return -1

  return max(treeHeight(node.left),
             treeHeight(node.right)) + 1
```

Problem 3: Rooting a tree

Sometimes it is useful to root an undirected tree to add structure to the problem you are trying to solve.

You can root a tree using any of its nodes. However, not every node will give you a well-balanced tree.

In some situations, it is also useful to keep a reference to the parent node in order to walk up the tree.

Rooting a tree is easily done with depth-first search (during the traversal).

```
# TreeNode object structure.
class TreeNode:
  # Unique integer id to identify this node.
  int id;

  # Pointer to parent TreeNode reference. Only the
  # root node has a null parent TreeNode reference.
  TreeNode parent;

  # List of pointers to child TreeNodes.
  TreeNode[] children;
```

```
# g is the graph/tree represented as an adjacency
# list with undirected edges. If there's an edge between
# (u, v) there's also an edge between (v, u).
# rootId is the id of the node to root the tree from.
function rootTree(g, rootId = 0):
  root = TreeNode(rootId, null, [])
  return buildTree(g, root, null)

# Build tree recursively depth first.
function buildTree(g, node, parent):
  for childId in g[node.id]:
    # Avoid adding an edge pointing back to the parent.
    if parent != null and childId == parent.id:
      continue
    child = TreeNode(childId, node, [])
    node.children.add(child)
    buildTree(g, child, node)
  return node
```

Problem 4: Tree center(s)

There can be more than one center. But no more than 2.

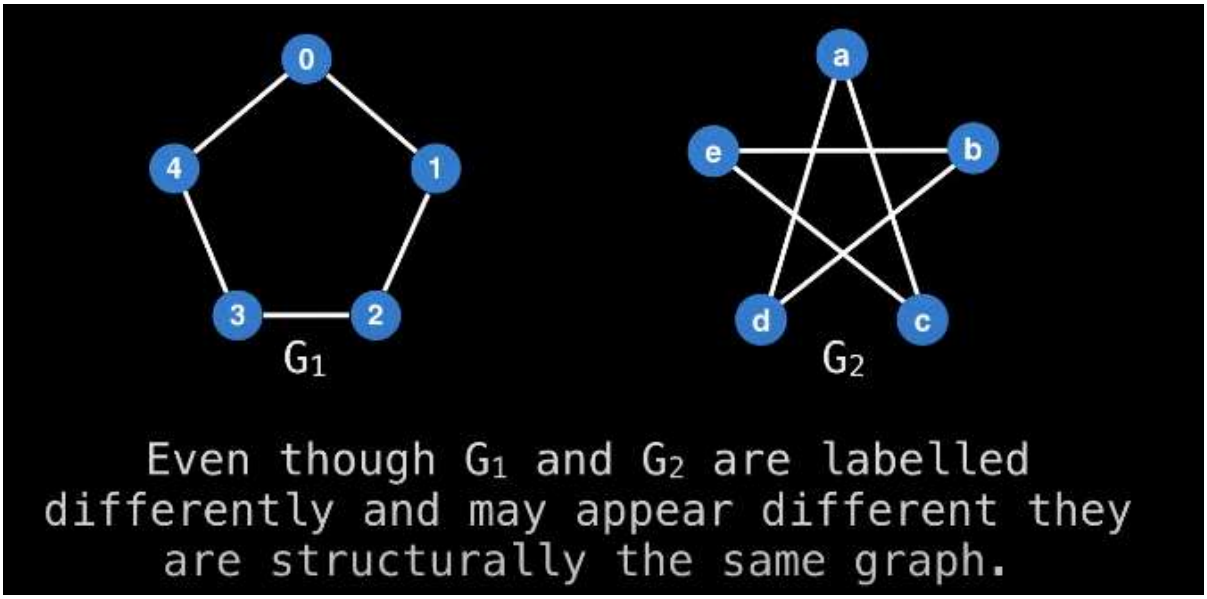
Notice that the center is always the middle vertex or middle two vertices in every longest path along the tree.

Another approach to find the center is to iteratively pick off each leaf node layer like we are peeling an onion --> identify the leaf nodes by counting the degree of each node and prune them.

```
# g = tree represented as an undirected graph
function treeCenters(g):
  n = g.numberOfNodes()
  degree = [0, 0, …, 0] # size n
  leaves = []
  for (i = 0; i < n; i++):
    degree[i] = g[i].size()
    if degree[i] == 0 or degree[i] == 1:
      leaves.add(i)
      degree[i] = 0
  count = leaves.size()
  while count < n:
    new_leaves = []
    for (node : leaves):
      for (neighbor : g[node]):
        degree[neighbor] = degree[neighbor] - 1
        if degree[neighbor] == 1:
          new_leaves.add(neighbor)
      degree[node] = 0
    count += new_leaves.size()
    leaves = new_leaves
  return leaves # center(s)
```

Problem 5: Identifying Isomorphic Trees

The question of asking whether two graphs G1 and G2 are **isomorphic** is asking whether they are structurally the same.



We can also define the notion of isomorphism more rigorously:

G1(V1, E1) and G2(V2, E2) are isomorphic if there exists a **bijection** φ between the sets V1 -> V2 such that:

$$\forall\ u,v \in V_1,\ (u,v) \in E_1 \iff (\varphi(u),\varphi(v)) \in E_2$$

In simple terms, for an isomorphism to exist, there needs to be a function φ which can map all the nodes/edges in G1 to G2 and vice-versa.

The method presented here involves **serializing** a tree into a **unique encoding**. This unique encoding is simply a unique string that represents a tree. If another tree has the same encoding, then they are isomorphic.

For rooted trees: one caveat to watch out is to ensure that the same root node is selected in both trees when rooting them (Problem 3) before serializing/encoding the trees. Otherwise, you will get two different encodings for isomorphic trees.

To select a common node between both trees, we can use what we learned from finding the center(s) of a tree (Problem 4).

Thus, the steps are:

1. Find the centers of trees
2. Root the trees at their center nodes
3. Generate the encoding for each tree and compare the serialized trees for equality.

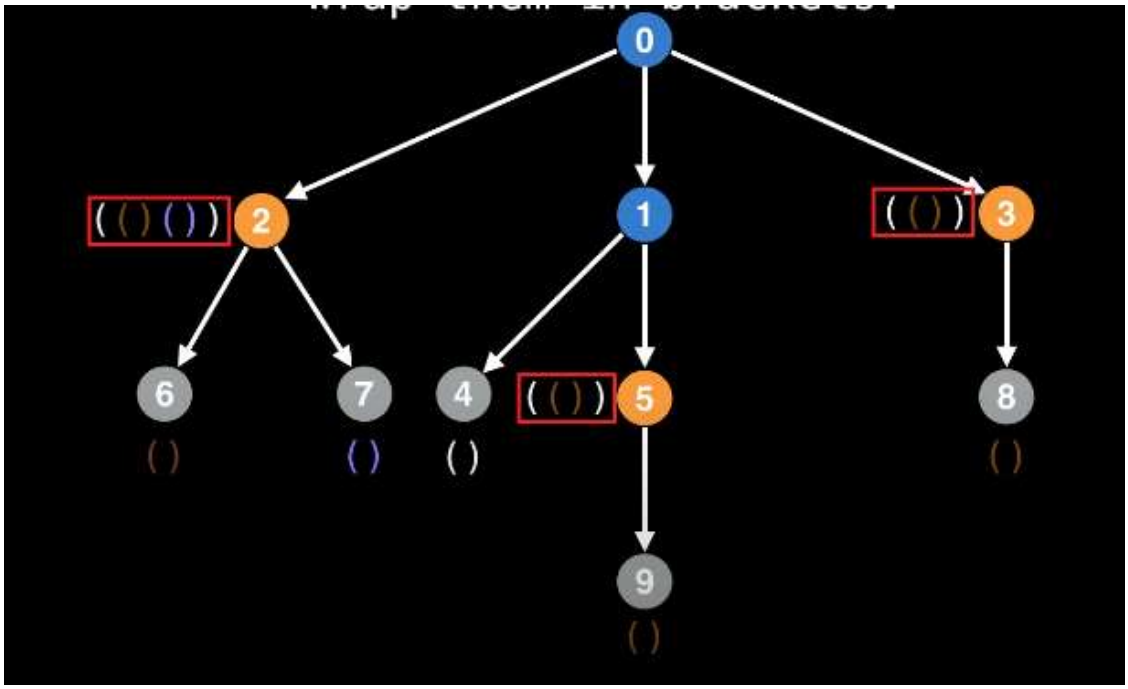
The tree encoding is simply a sequence of left and right brackets. However, you can also think of them as 1's and 0's.

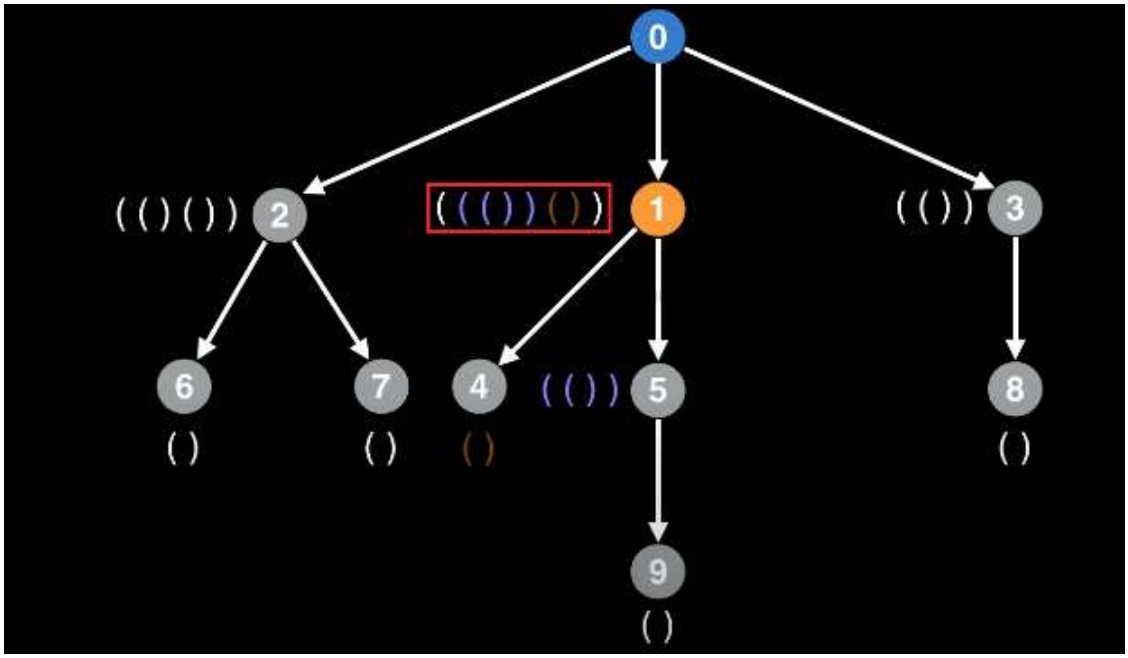
It should also be possible to reconstruct the tree solely from the encoding.

The **AHU** (Aho, Hopcroft, Ullman) algorithm is a clever serialization technique for representing a tree as a unique string. AHU is able to capture a complete history of a tree's **degree spectrum** and structure, ensuring a deterministic method of checking for tree isomorphism.

AHU:

1. Assign all leaf nodes Knuth tuples: '()'
2. Process all nodes with grayed out children, combine labels of their child nodes and wrap them in brackets. Notice that labels are **sorted** lexicographically when combined before wrapping them in brackets, this is important.





3. You can not process a node until you have processed all its children.
4. Repeat till the root

Pseudocode

```
        Rooted trees are stored recursively in
                  TreeNode objects:

# TreeNode object structure.
class TreeNode:
  # Unique integer id to identify this node.
  int id;

  # Pointer to parent TreeNode reference. Only the
  # root node has a null parent TreeNode reference.
  TreeNode parent;

  # List of pointers to child TreeNodes.
  TreeNode[] children;
```

```
# Returns whether two trees are isomorphic.
# Parameters tree1 and tree2 are undirected trees
# stored as adjacency lists.
function treesAreIsomorphic(tree1, tree2):
  tree1_centers = treeCenters(tree1)
  tree2_centers = treeCenters(tree2)

  tree1_rooted = rootTree(tree1, tree1_centers[0])
  tree1_encoded = encode(tree1_rooted)

  for center in tree2_centers:
    tree2_rooted = rootTree(tree2, center)
    tree2_encoded = encode(tree2_rooted)
    # Two trees are isomorphic if their encoded
    # canonical forms are equal.
    if tree1_encoded == tree2_encoded:
      return True
  return False
```

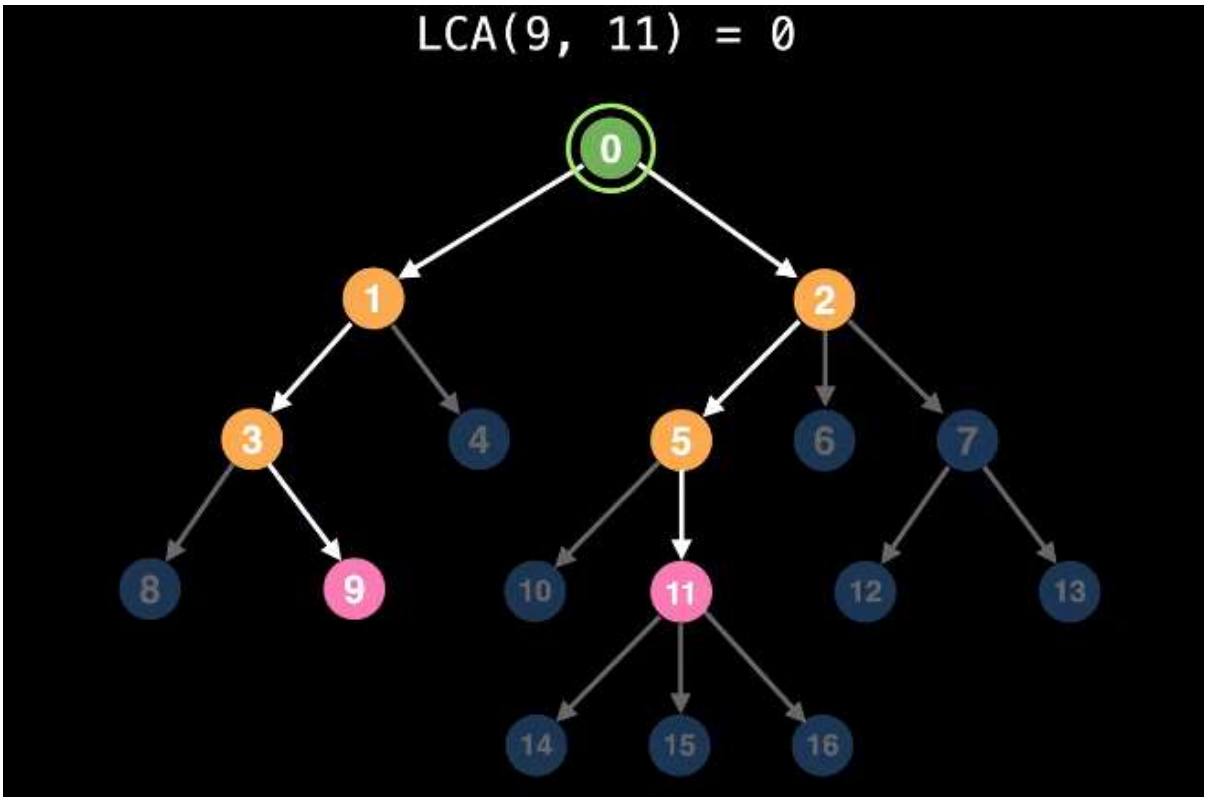
In regards to the second tree's loop: we do not know what center to choose in case the second tree has two centers. Therefore, we try both and compare them to the encoding of the first tree.

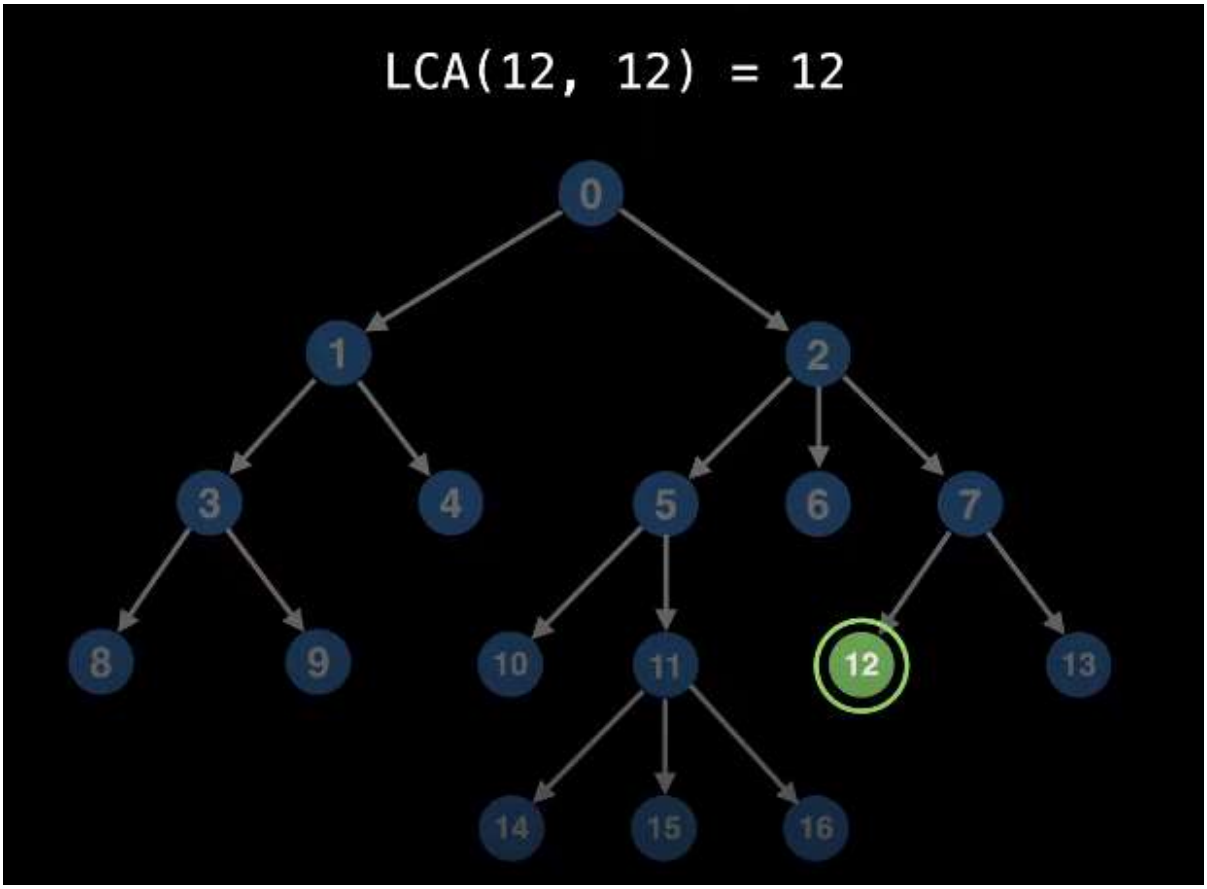
```
function encode(node):
  if node == null:
    return ""

  labels = []
  for child in node.children():
    labels.add(encode(child))

  # Regular lexicographic sort
  sort(labels)

  result = ""
  for label in labels:
    result += label

  return "(" + result + ")"
```

Problem 6: Lowest Common Ancestor (LCA) Problem | Eulerian path method

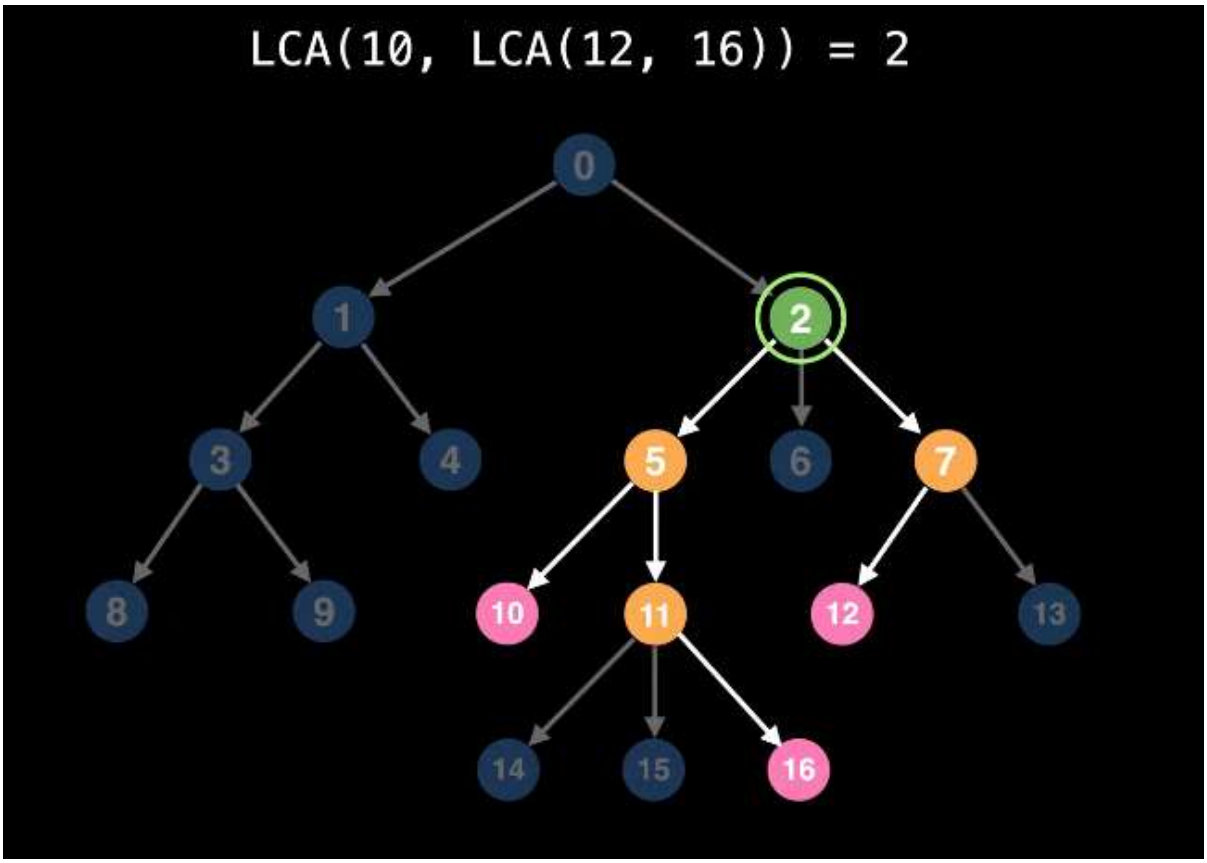
The **Lowest Common Ancestor (LCA)** of two nodes a and b in a **rooted tree** is the *deepest* node c that has both a and b as descendants (where a node can be a descendant of itself).

Note: the notion of LCA also exists for Directed Acyclic Graphs (DAGs) but here we are only looking at LCA in the context of trees.





You can also find LCA of more than 2 nodes.



Eulerian tour + Range Minimum Query (RMQ) method

This method can answer LCA in O(1) time with O(nlogn) pre-processing when using a **Sparse Table** to do the RMQs.

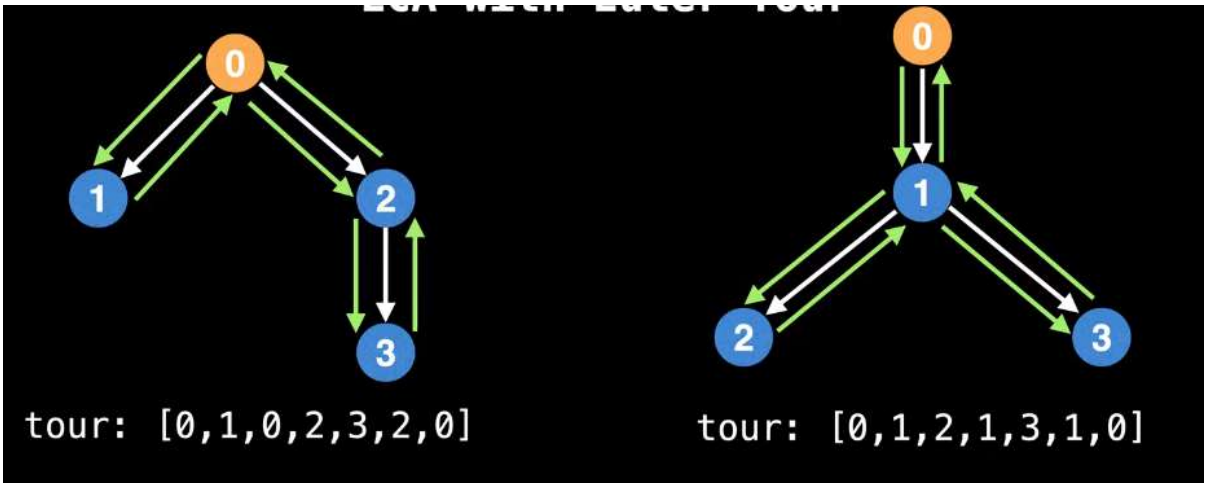
However, the pre-processing time can be improved to O(n) with **Farach-Colton and Bender optimization**.

Given a tree we want to do LCA queries on, first:

1. Make sure the tree is **rooted**.
2. Ensure that all nodes are **uniquely indexed** in some way, so that we can reference them later. One easy way - assigning each node a unique id between [0, n-1].

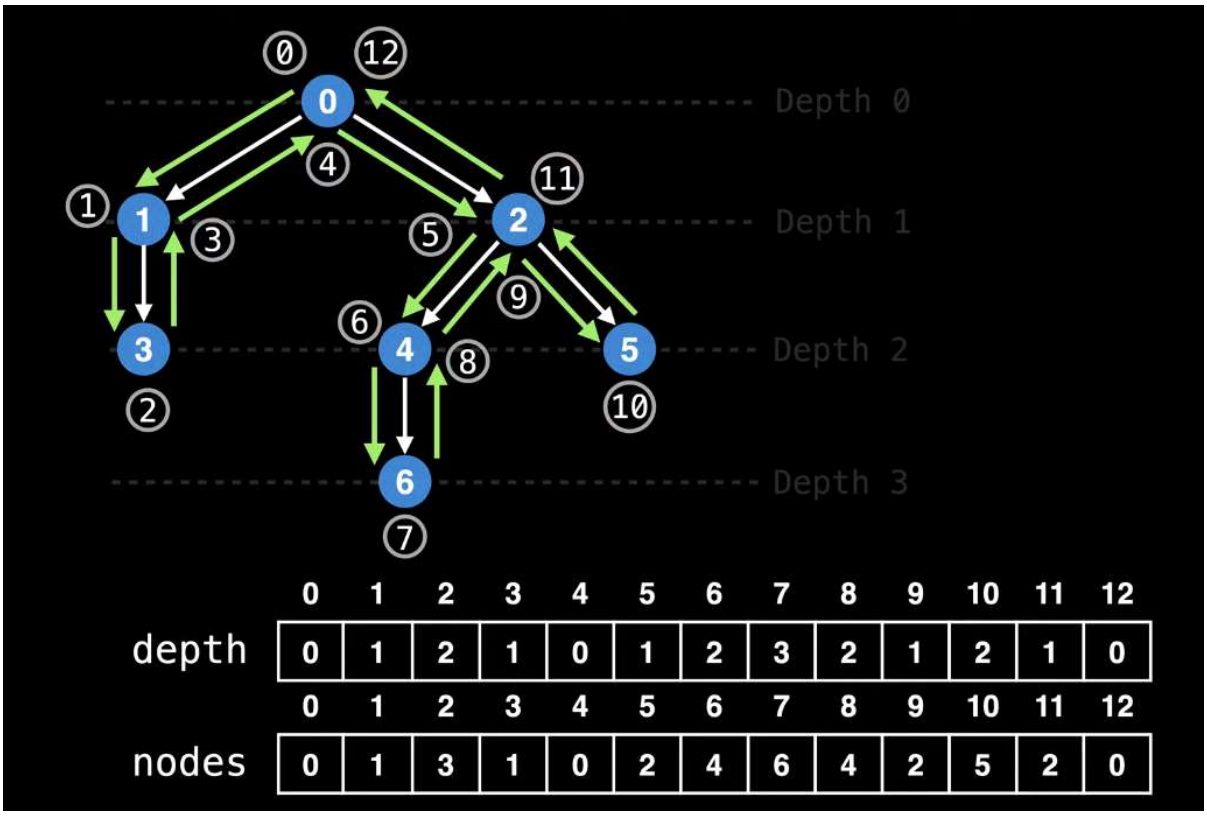
Eulerian Tour (Eulerian circuit):

This method begins by finding an Eulerian tour of the edges in a rooted tree. Rather than doing an Eulerian tour on the white edges of our tree, we are going to do the Euler tour on a new set of imaginary **green edges** which wrap around the tree. This ensures that our tour visits every node in the tree.



Start at the root node, traverse all green edges, and finally return to the root node. As you do this, keep track of which nodes you visit and this will be your Euler tour.

We also need to keep track of depths while doing the Euler tour:



	0	1	2	3	4	5	6	7	8	9	10	11	12
depth	0	1	2	1	0	1	2	3	2	1	2	1	0
nodes	0	1	3	1	0	2	4	6	4	2	5	2	0

What is **LCA(6, 5)** for this example?

1. Find the index position value for the nodes a and b (5 and 6 respectively). In this example, a is at index 10 and b is at index 7.
2. Using the depth array, find the index of the minimum value in the range of the indices obtained in step 1. I.e. query the range [7, 10] in the depth array to find the index of the minimum value. This can be done in O(1) with Sparse Table. For this example, the index is 9 with a value of 1.
3. Using the index obtained in step 2, find the LCA of a and b in the nodes array -> with index 9 found in step 2, retrieve the LCA at `nodes[9]`.

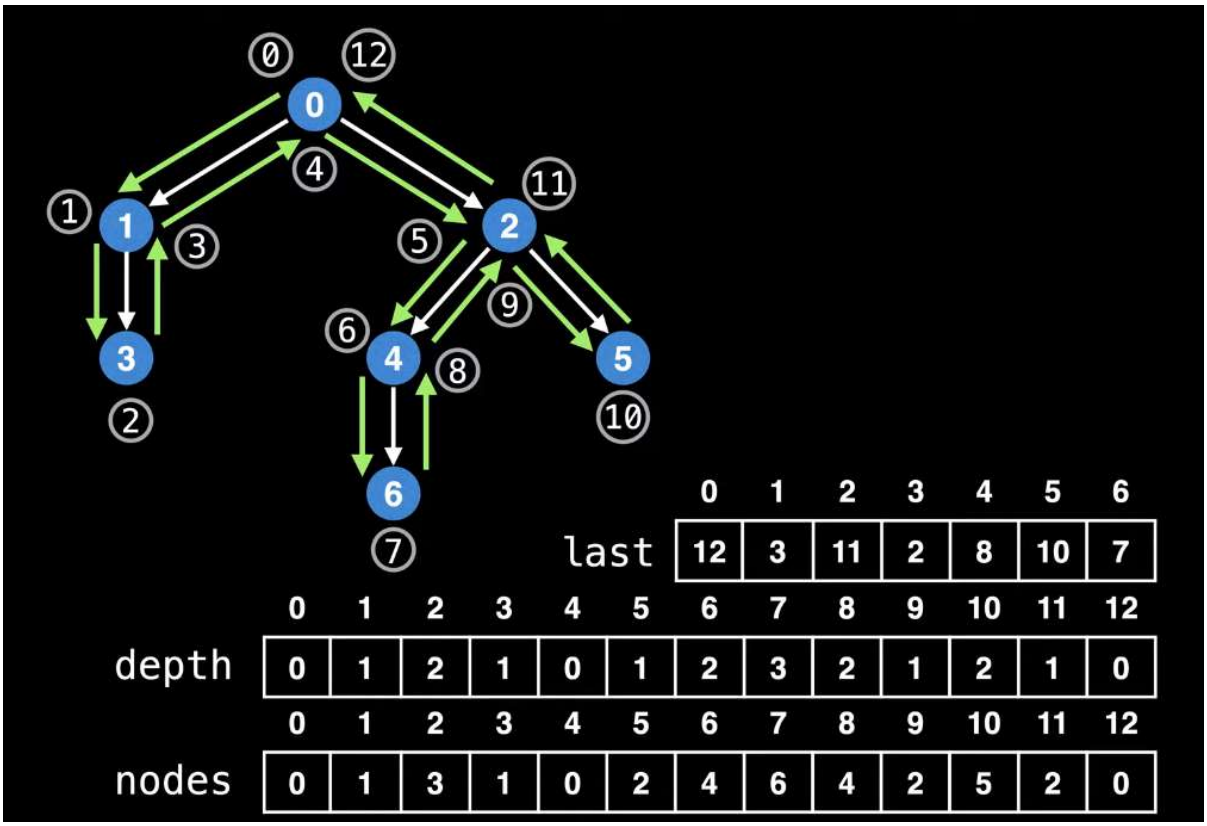
For this example, the answer will be 2.

Issue: what do we do with nodes that have more than 1 index in the Euler tour? For instance, node with value `1` from the previous example has two indices: 1 and 3.

There are $2n - 1$ node index position in the Euler tour and only n nodes in total, so a perfect 1 to 1 **inverse mapping** is impossible.

The answer is that it **does not matter**. Any of the inverse index values will do. However, in practice, it is easier to select the **last encountered index** while doing the Euler tour.

To maintain the **inverse mapping**, we are going to keep track of some additional information, namely an inverse map called `map`. We are going to be saving the last encountered index position.



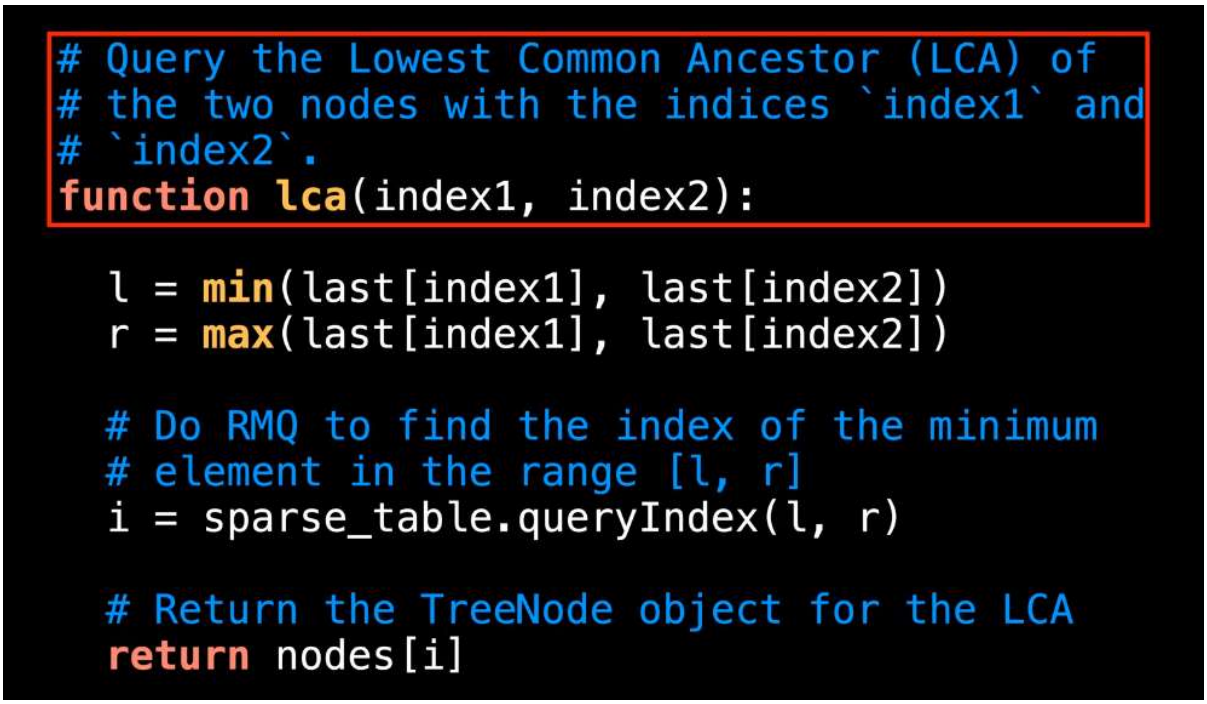
Pseudocode

```
class TreeNode:

  # A unique index (id) associated with this
  # TreeNode.
  int index;

  # List of pointers to child TreeNodes.
  TreeNode children[];
```

```
function setup(n, root):

  nodes = … # array of nodes of size 2n - 1
  depth = … # array of integers of size 2n - 1

  last = … # node index -> Euler tour index

  # Do Eulerian Tour around the tree
  dfs(root)

  # Initialize sparse table data structure to
  # do Range Minimum Queries (RMQs) on the
  # `depth` array. Sparse tables take O(nlogn)
  # time to construct and do RMQs in O(1)
  sparse_table = CreateMinSparseTable(depth)
```

```
# Eulerian tour index position
tour_index = 0

# Do an Eulerian Tour of all the nodes using
# a DFS traversal.
function dfs(node, node_depth = 0):
  if node == null:
    return

  visit(node, node_depth)
  for (TreeNode child in node.children):
    dfs(child, node_depth + 1)
    visit(node, node_depth)

# Save a node's depth, inverse mapping and
# position in the Euler tour
function visit(node, node_depth):
  nodes[tour_index] = node
  depth[tour_index] = node_depth
  last[node.index] = tour_index
  tour_index = tour_index + 1
```

Pay attention to the second `visit` call after calling `dfs` on the subtree. We do this, since we want to maintain proper tour_index, last, etc numbers.



Video: Lowest Common Ancestor (LCA) Problem | Eulerian path method