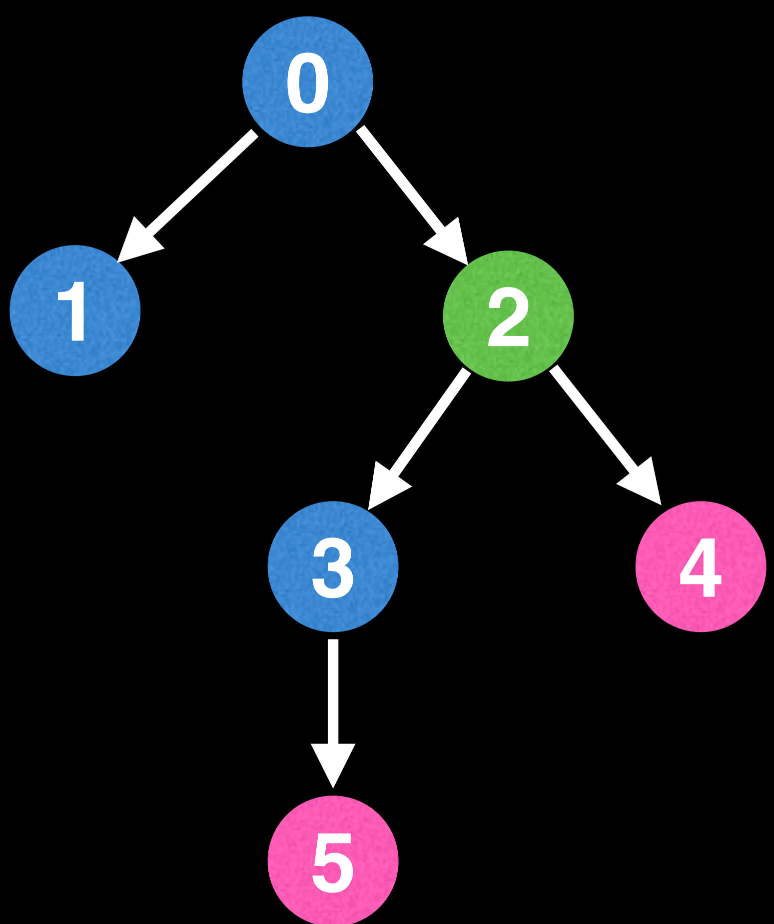
# Lowest Common Ancestor

Eulerian tour + range minimum query method

William Fiset

# Definition

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)

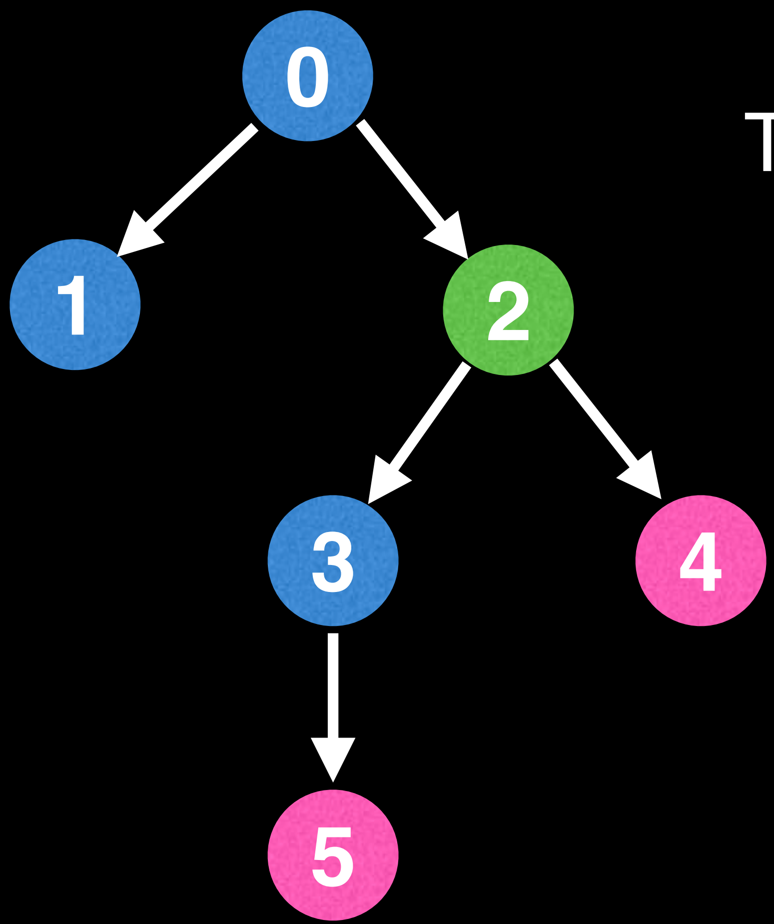



LCA(5, 4) = 2

NOTE: The notion of a LCA also exists for Directed Acyclic Graphs (DAGs), but today we’re only looking at the LCA in the context of trees.

# Definition

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)



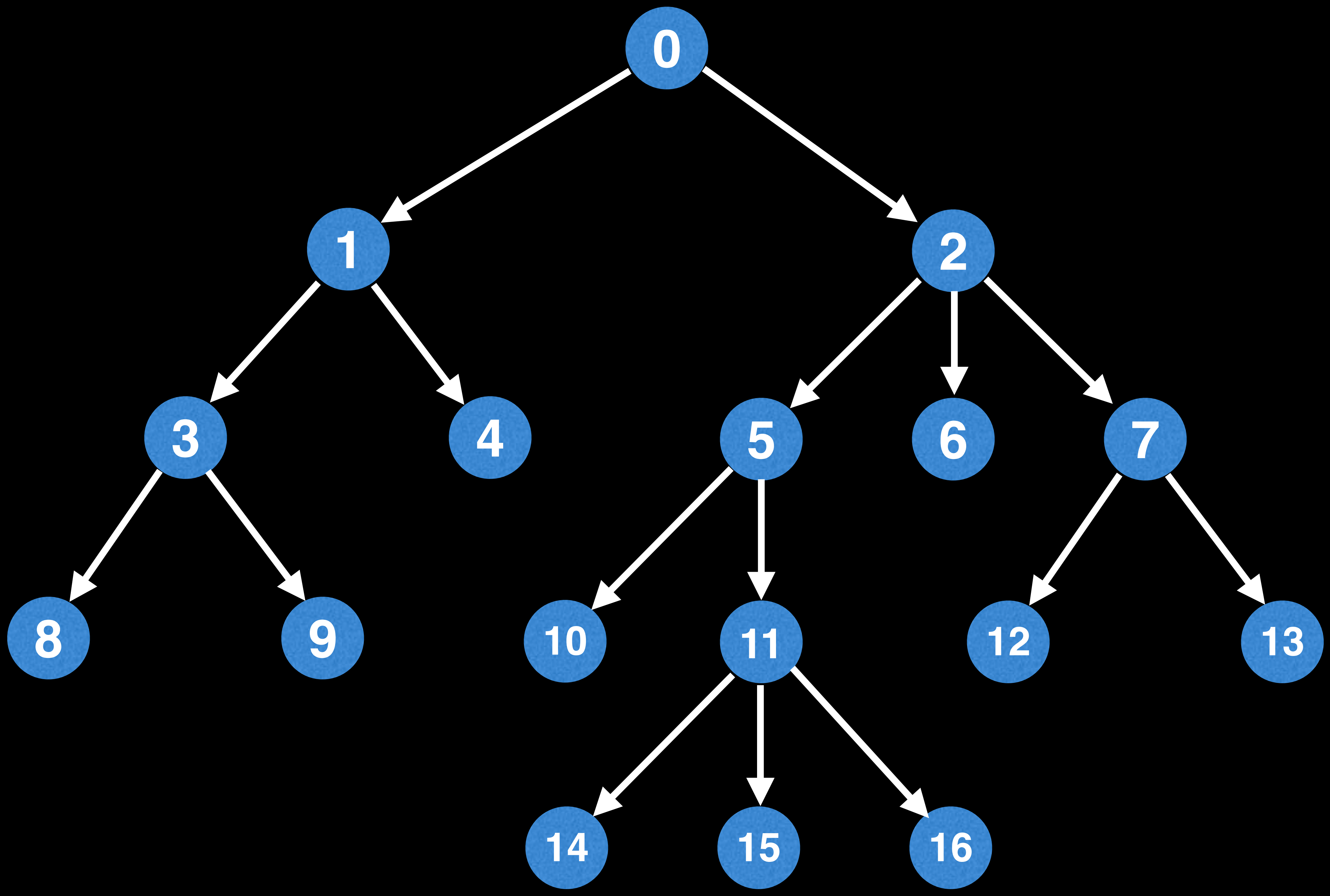

LCA(5, 4) = 2

The LCA problem has several applications in Computer Science, notably:

- Finding the distance between two nodes
- Inheritance hierarchies in OOP
- As a subroutine in several advanced algorithms and data structures
- etc…

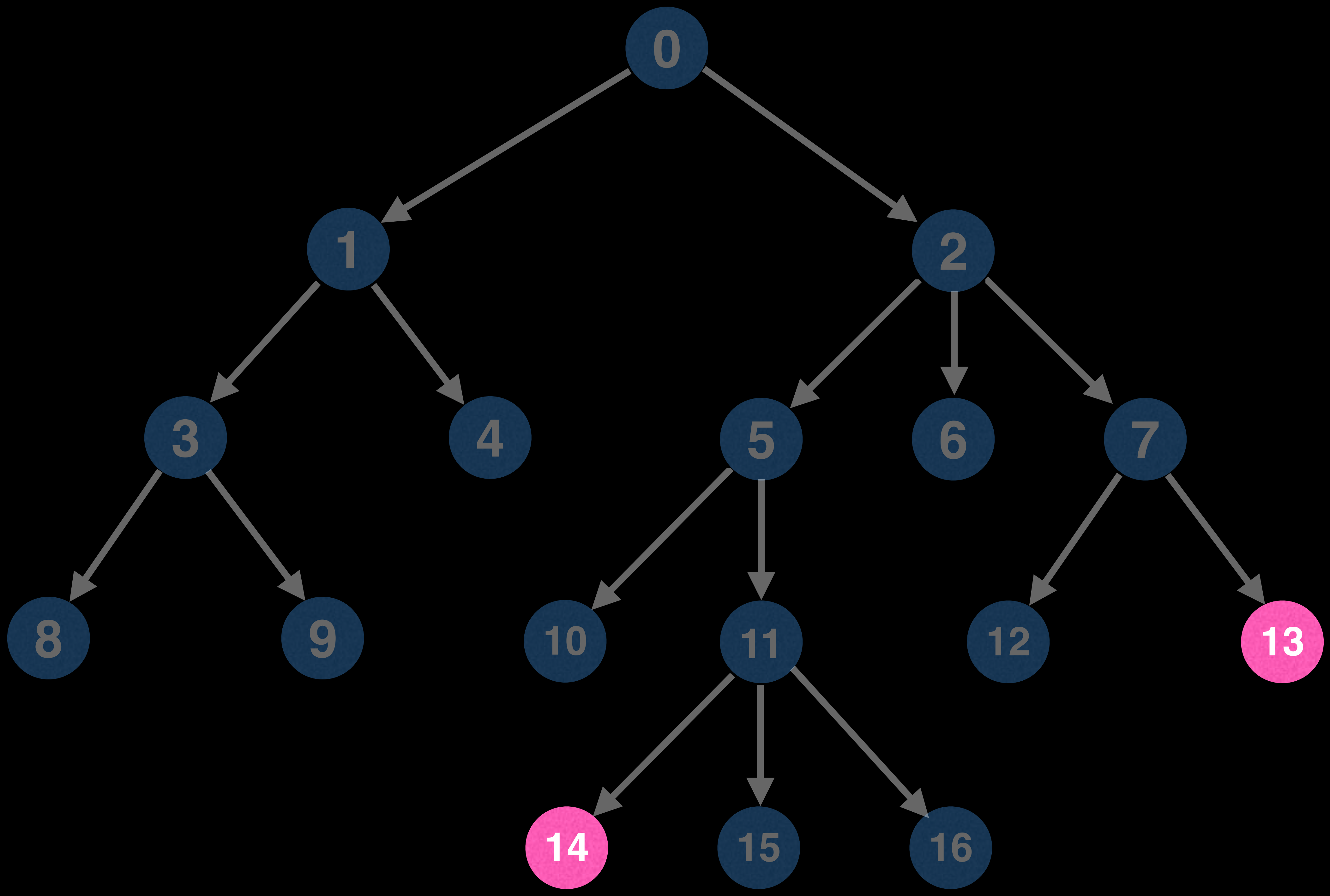
NOTE: The notion of a LCA also exists for Directed Acyclic Graphs (DAGs), but today we're only looking at the LCA in the context of trees.

# Understanding LCA



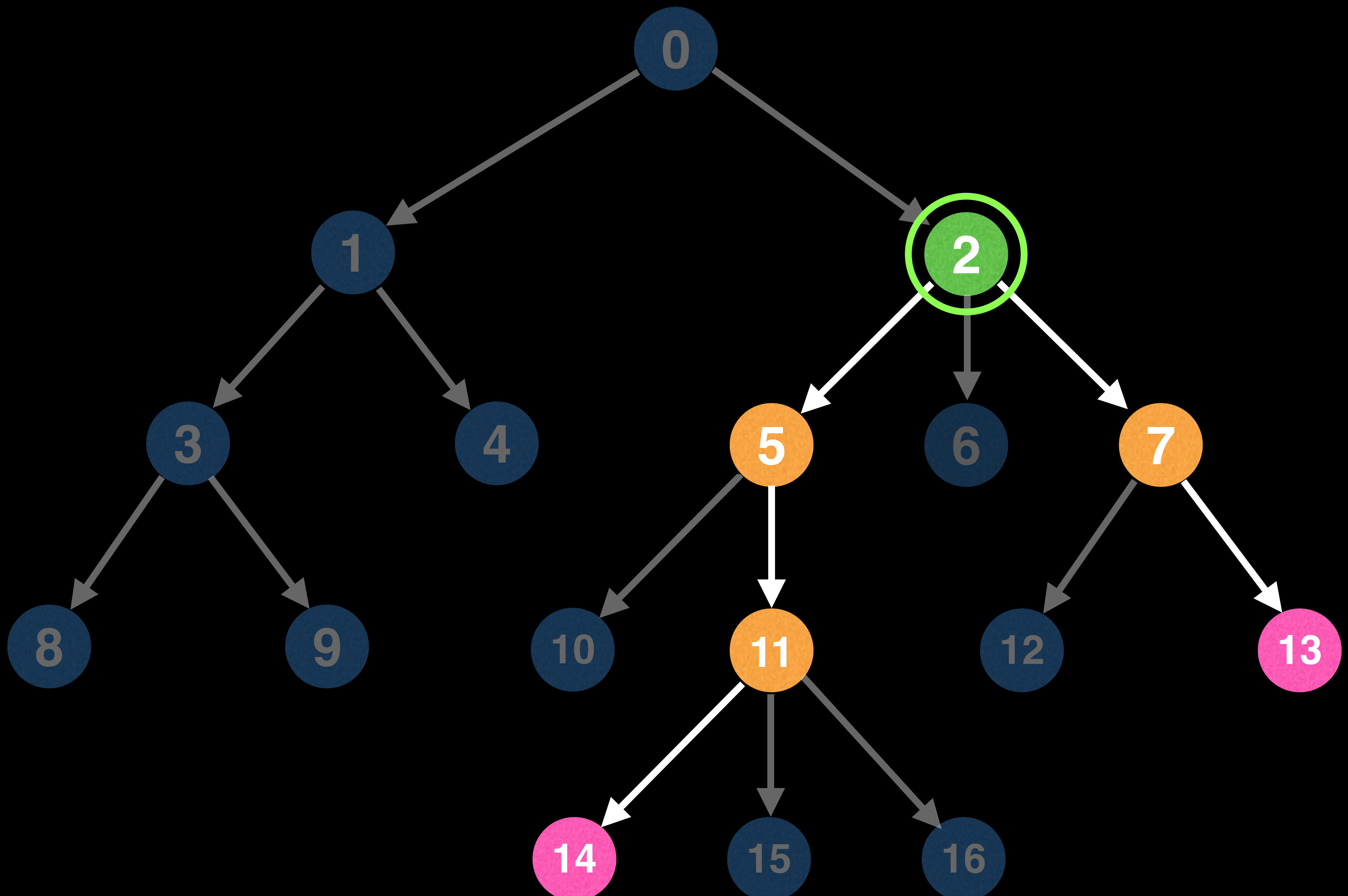
# Understanding LCA

LCA(13, 14)



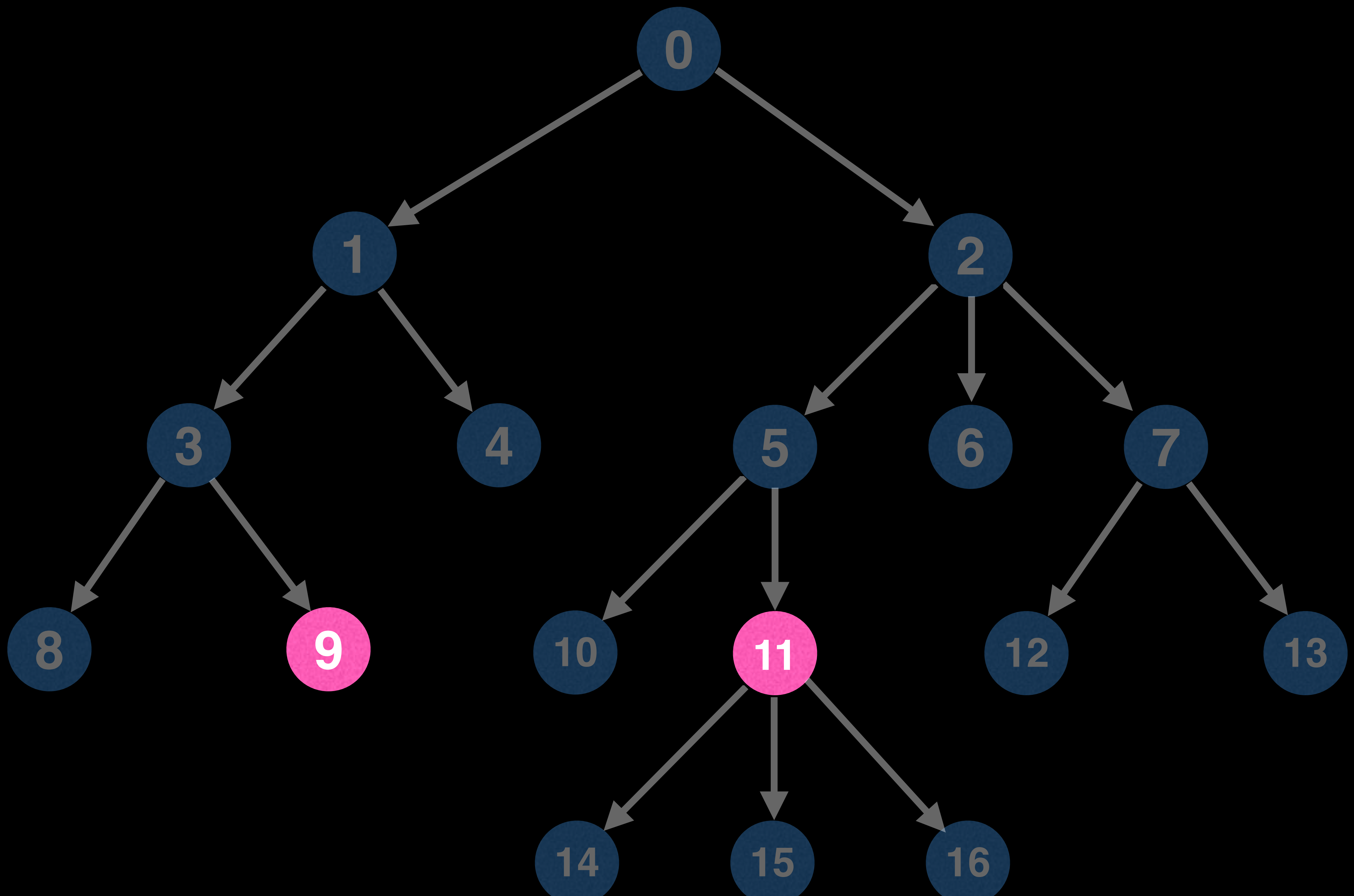
# Understanding LCA

LCA(13, 14) = 2



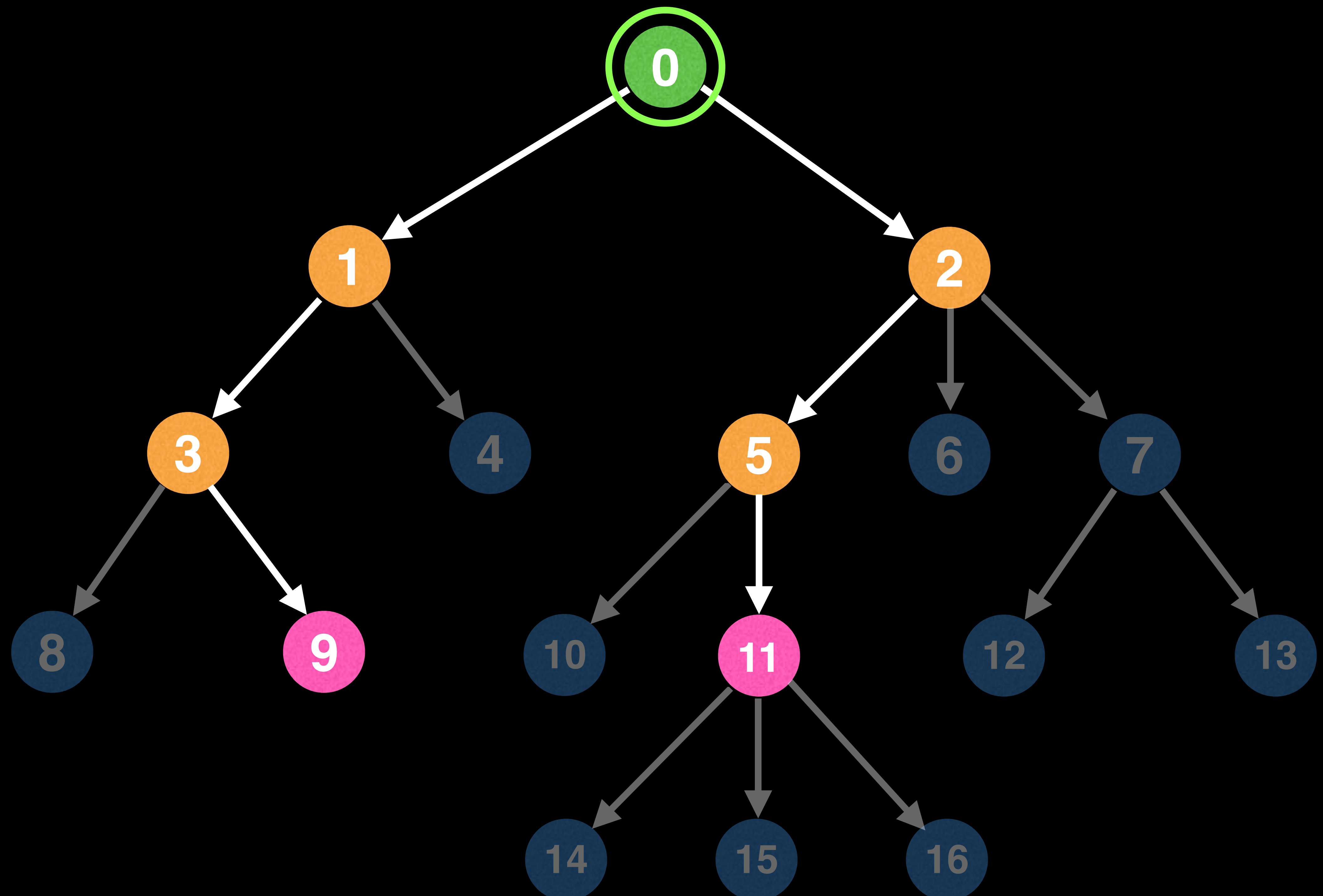
# Understanding LCA

LCA(9, 11)



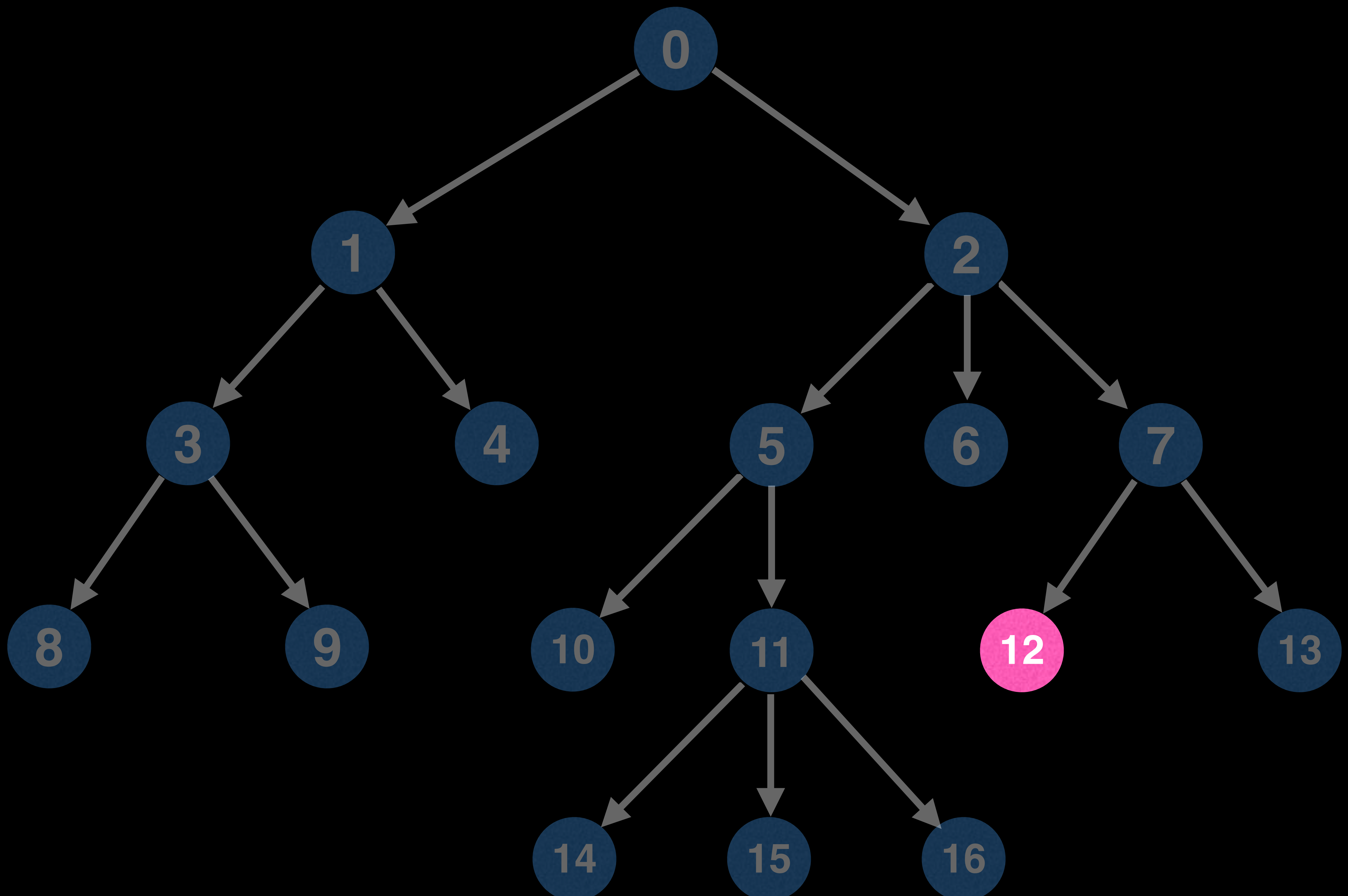
# Understanding LCA

LCA(9, 11) = 0



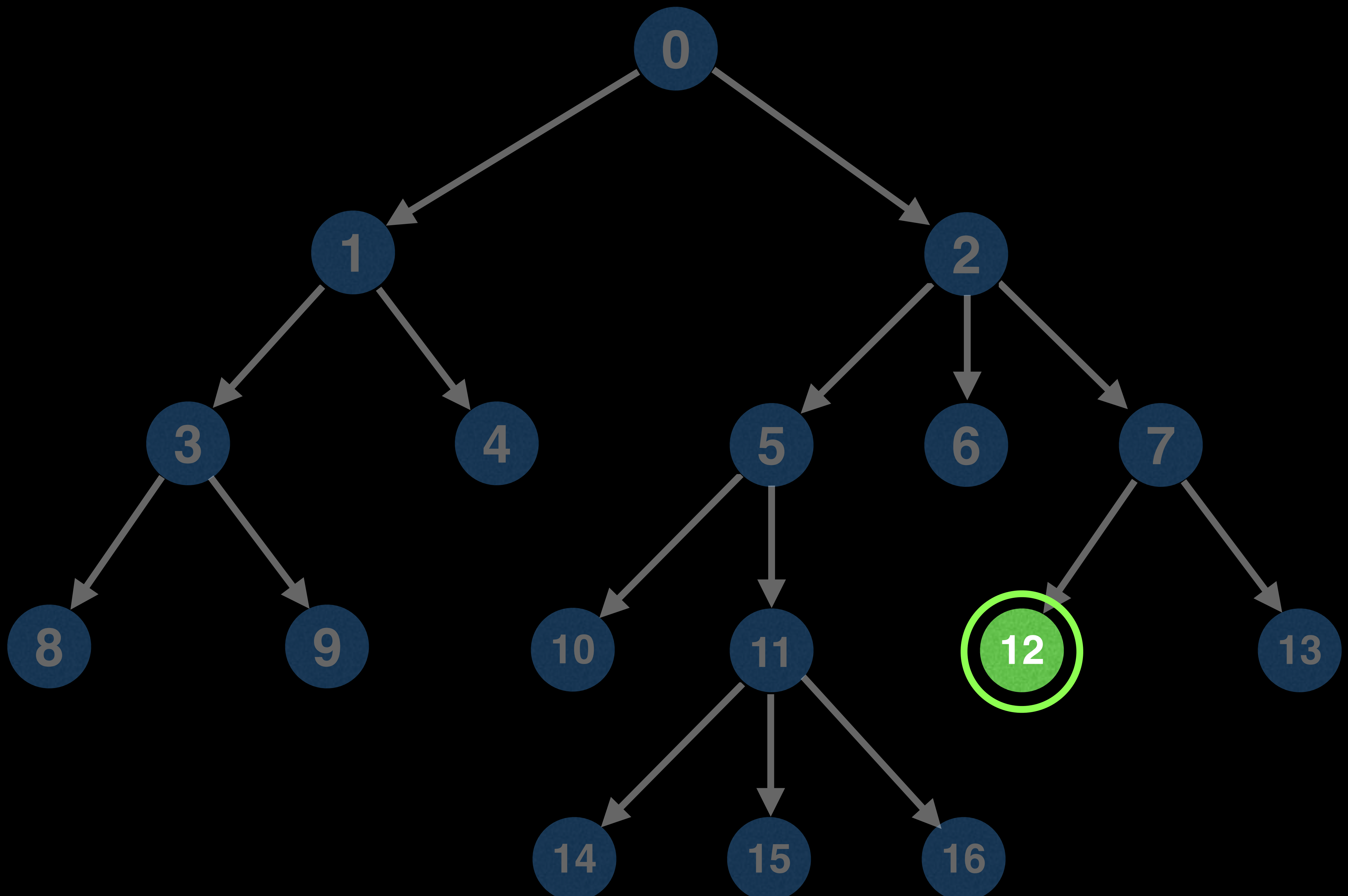
# Understanding LCA

LCA(12, 12)



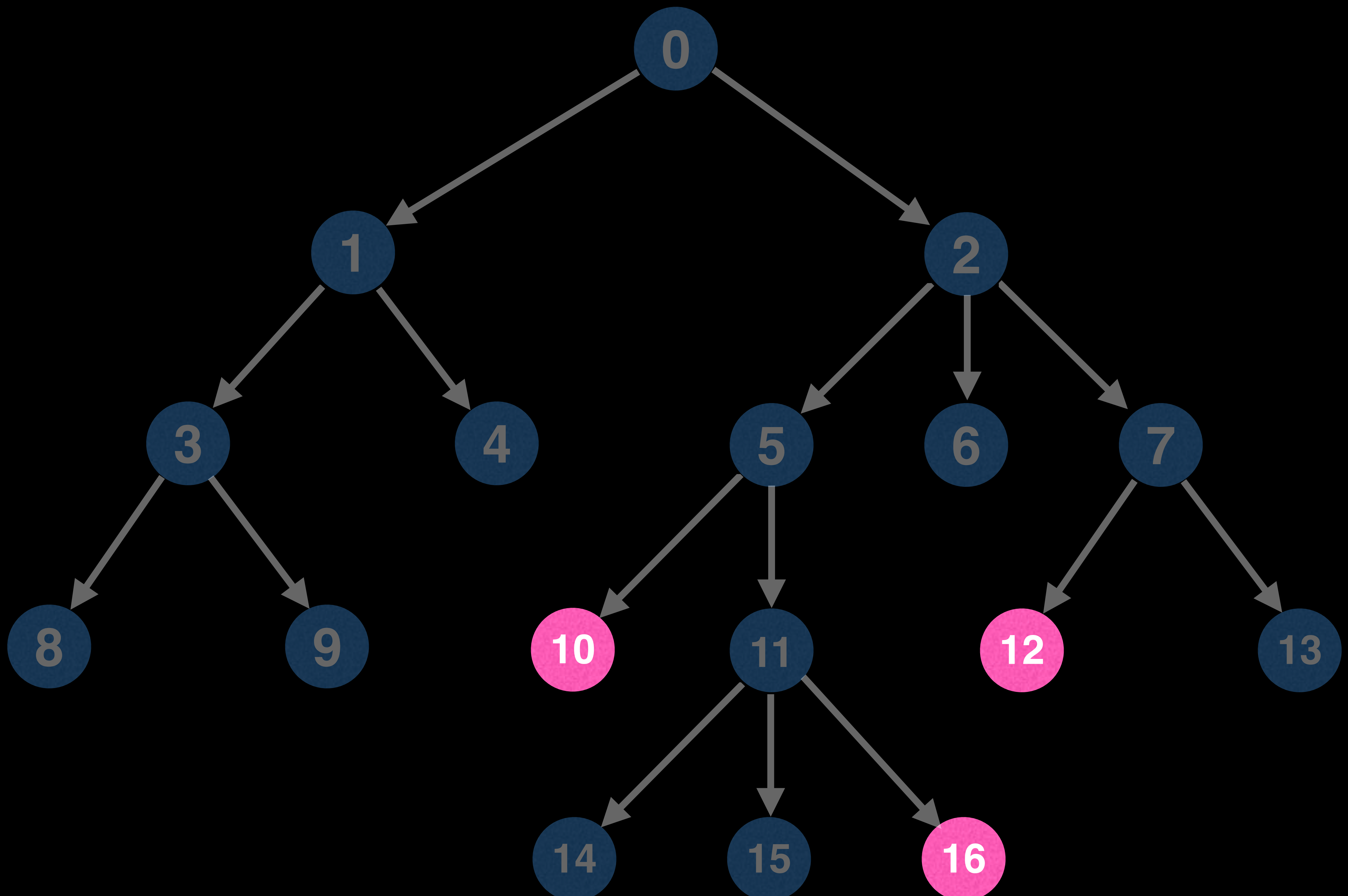
# Understanding LCA

LCA(12, 12) = 12



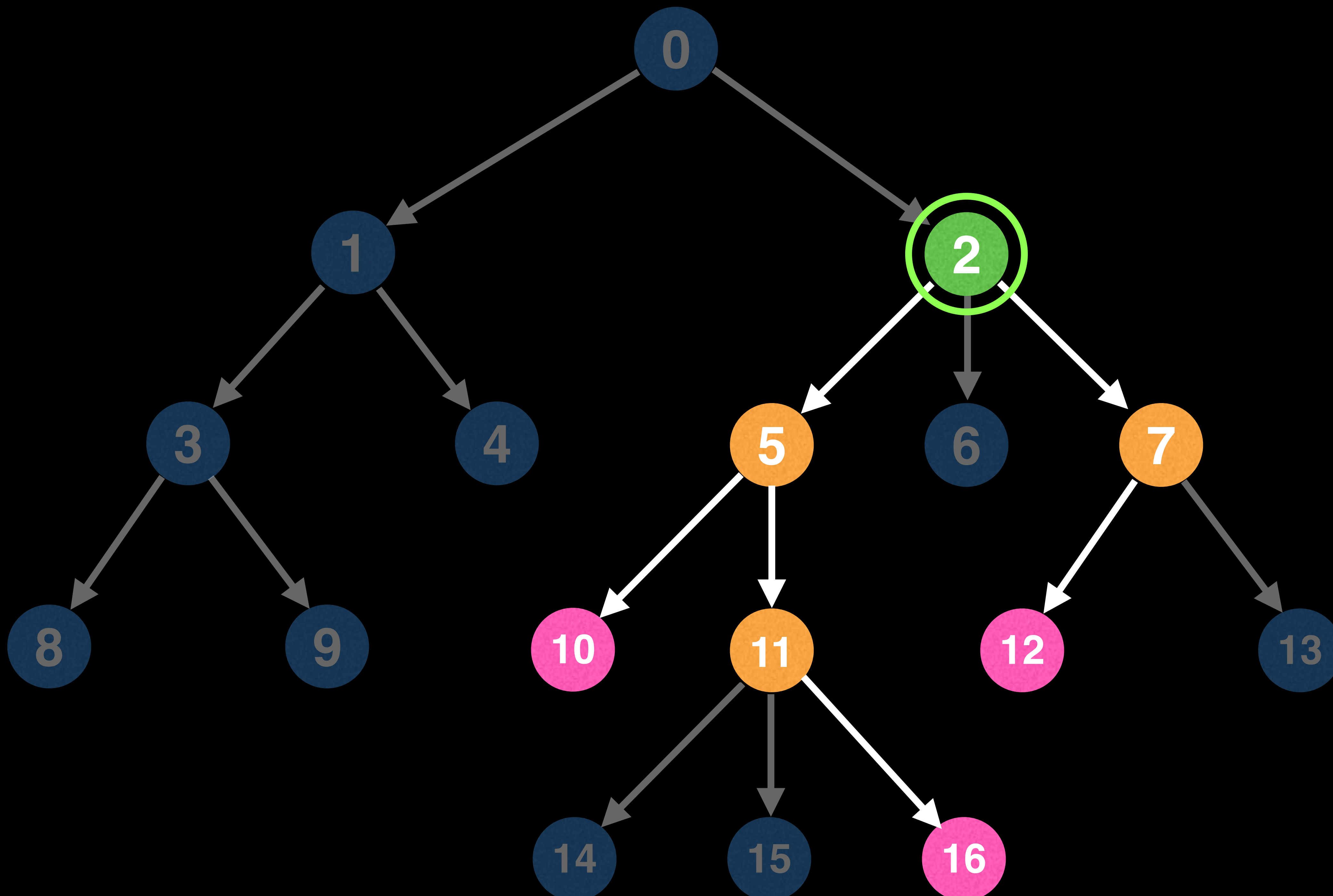
# Understanding LCA

You can also find the LCA of more than 2 nodes



# Understanding LCA

LCA(10, LCA(12, 16)) = 2



# LCA Algorithms

There are a diverse number of popular algorithms for finding the LCA of two nodes in a tree including:

- Tarjan’s offline LCA algorithm
- Heavy-Light decomposition
- Binary Lifting
- etc…

Today, we’re going to cover how to find the LCA using the **Eulerian tour** + **Range Minimum Query (RMQ)** method.

# LCA Algorithms

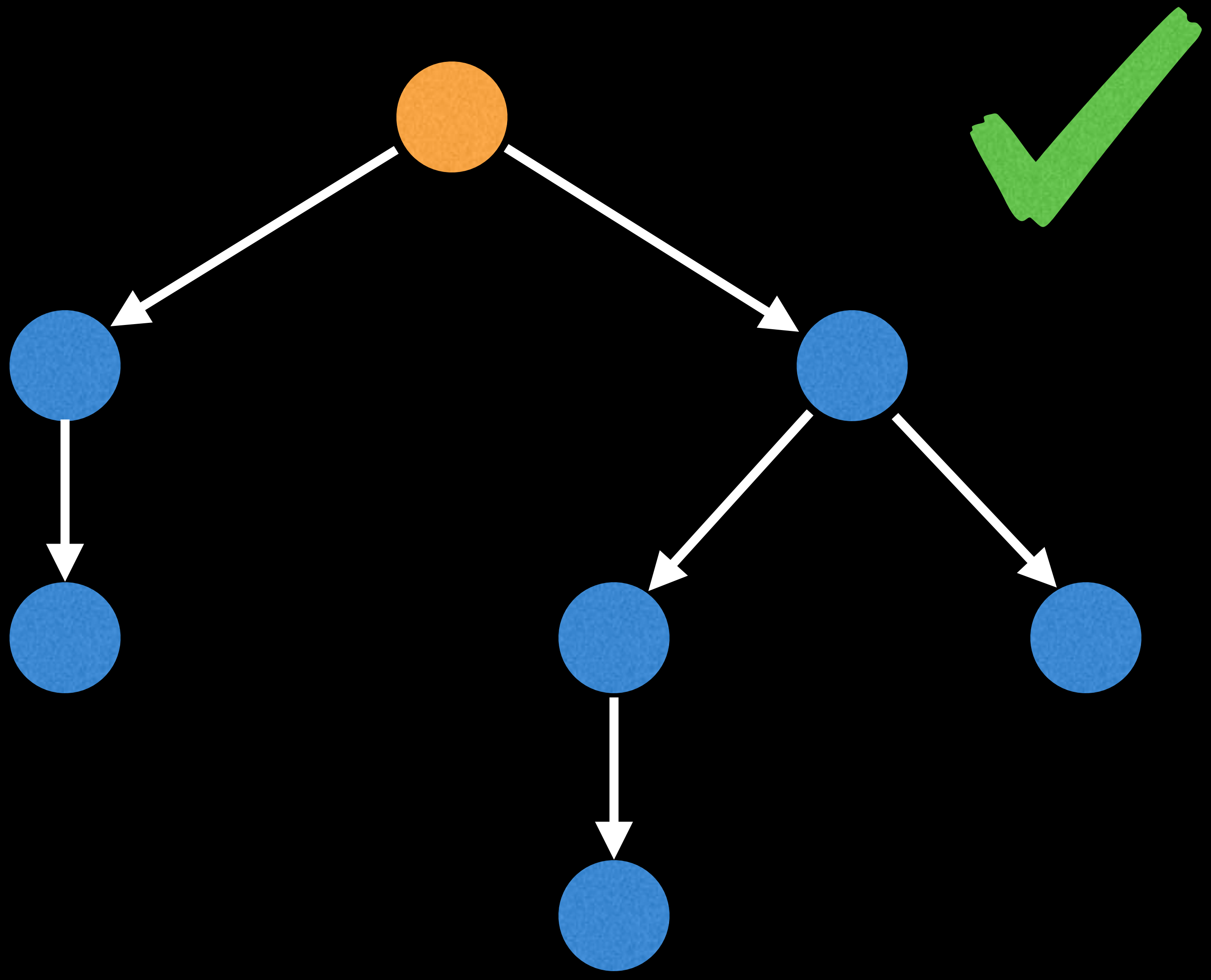
There are a diverse number of popular algorithms for finding the LCA of two nodes in a tree including:

- Tarjan’s offline LCA algorithm
- Heavy-Light decomposition
- Binary Lifting
- etc…

Today, we’re going to cover how to find the LCA using the **Eulerian tour** + **Range Minimum Query (RMQ)** method.

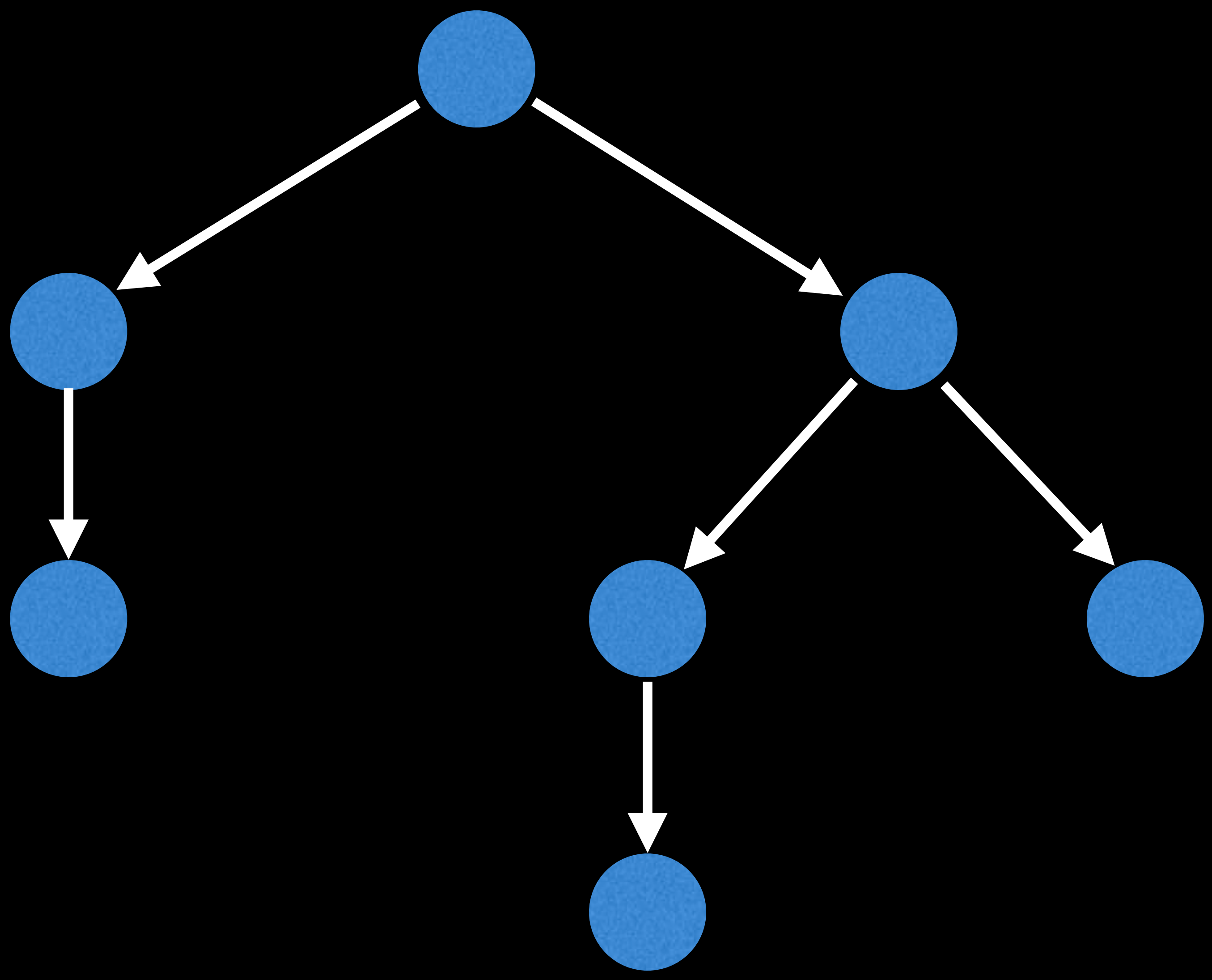
This method can answer LCA queries in **O(1)** time with **O(nlogn)** pre-processing when using a **Sparse Table** to do the RMQs.

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



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

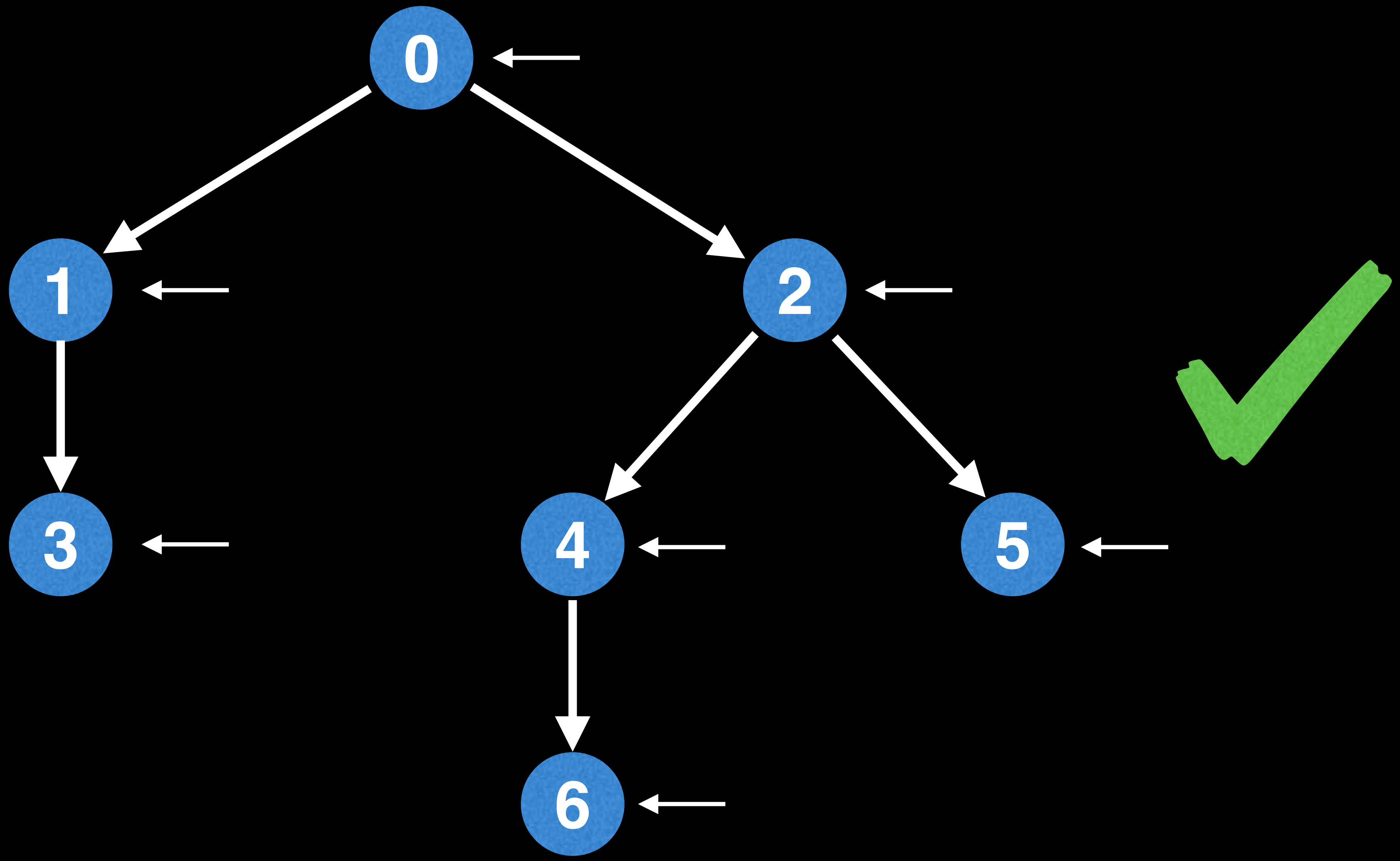
1. Make sure the tree is **rooted**



Tree is not labelled atm…

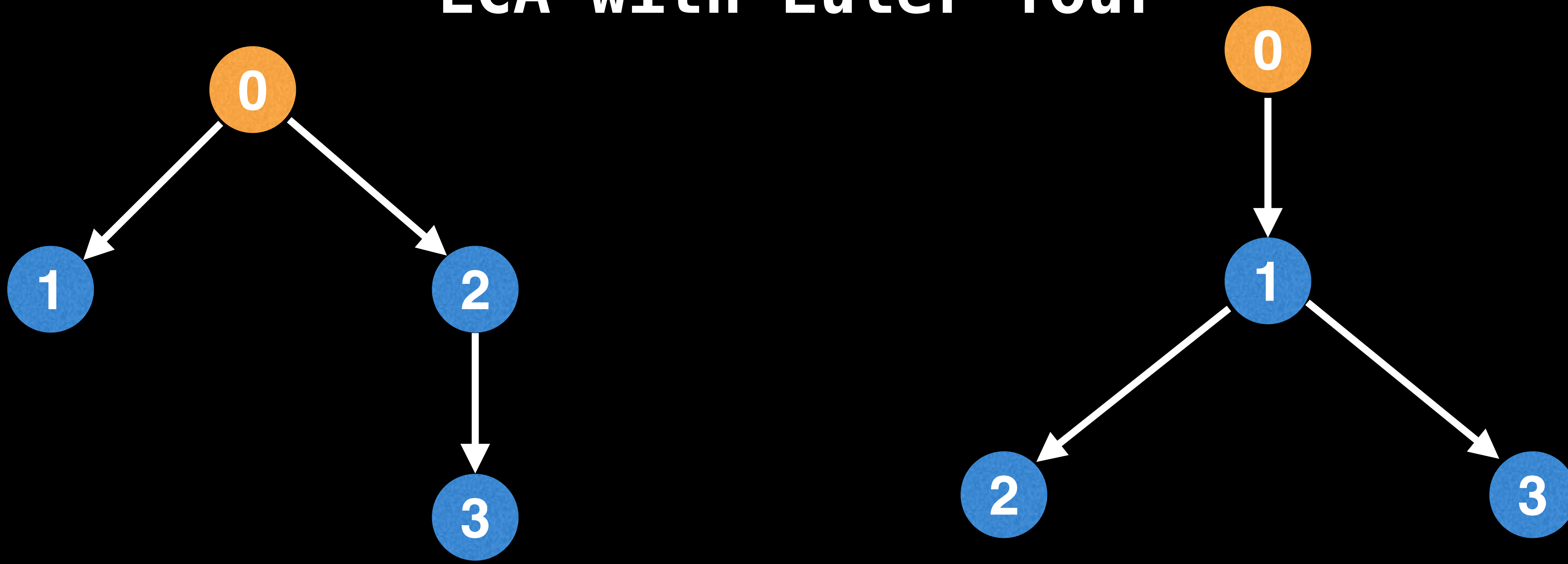
Given a tree we want to do LCA queries on, first:

2. Ensure that all nodes are **uniquely indexed** in some way so that we can reference them later.



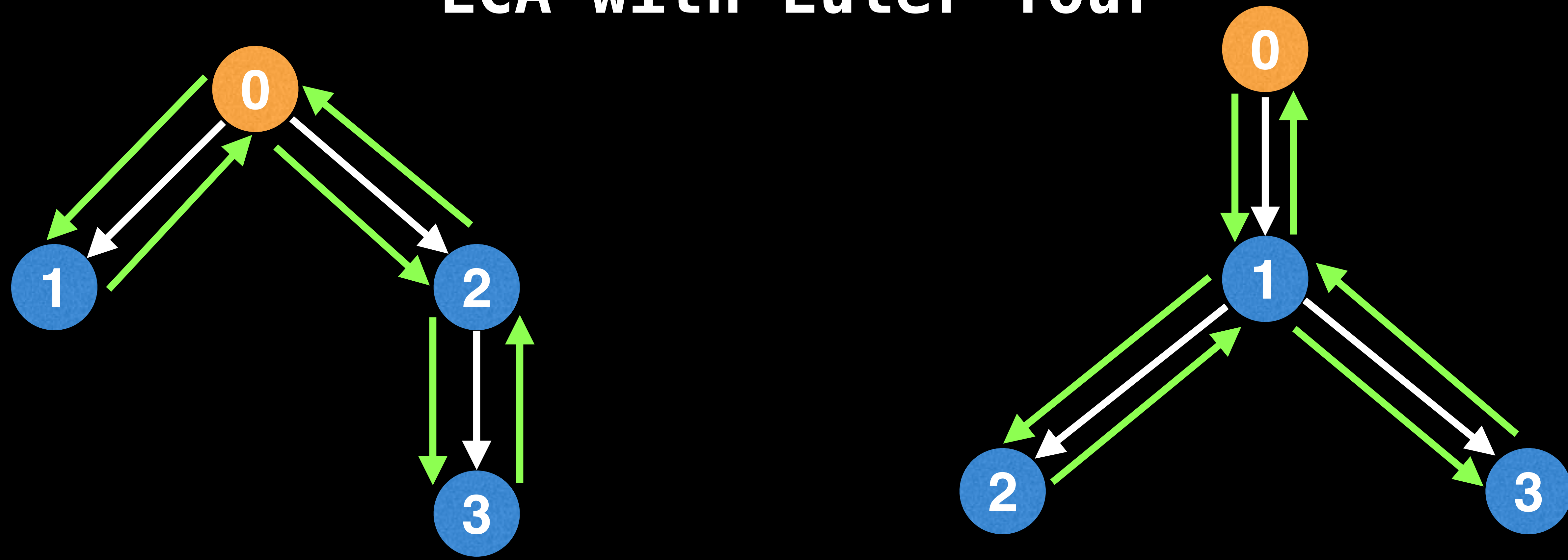

One easy way to index each node is by assigning each node a unique id between [0, n-1]

# LCA with Euler Tour




As you might have guessed, the Eulerian tour method begins by finding an Eulerian tour of the edges in a rooted tree.

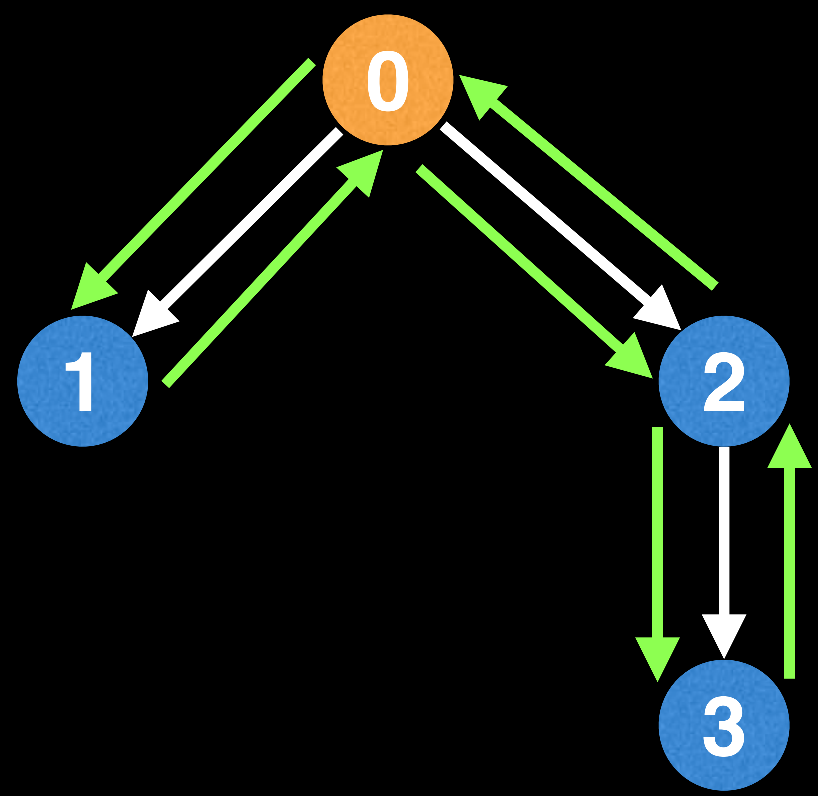
# LCA with Euler Tour



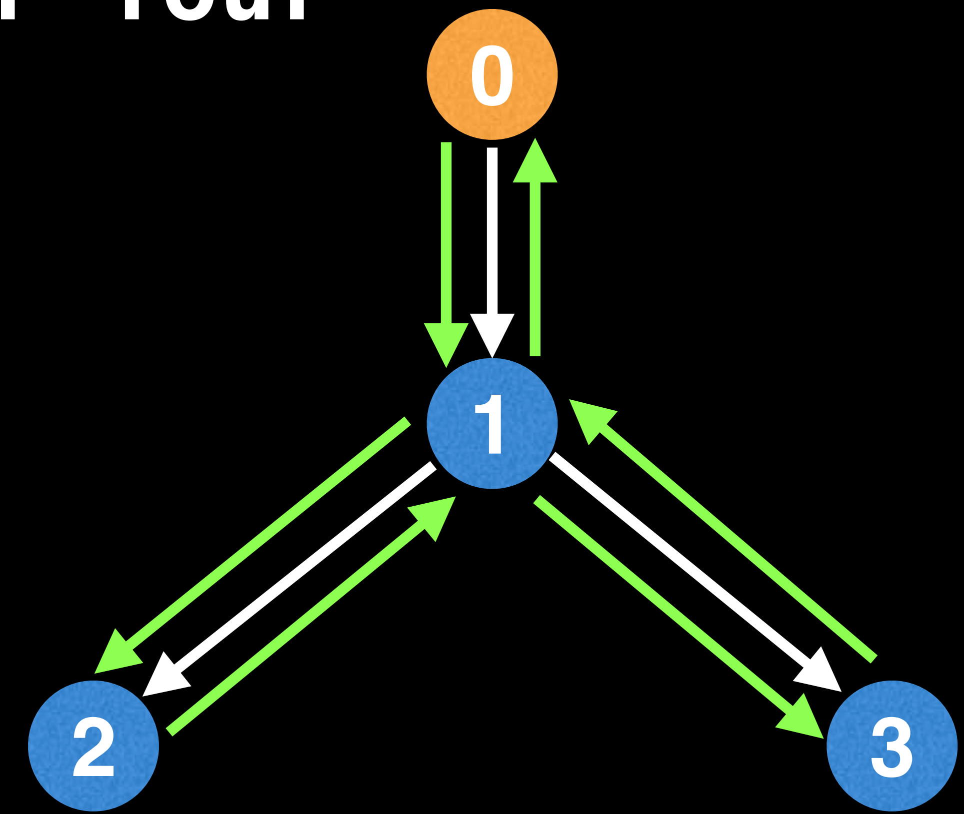

As you might have guessed, the Eulerian tour method begins by finding an Eulerian tour of the edges in a rooted tree.

Rather than doing the Euler tour on the white edges of our tree, we're 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.

# LCA with Euler Tour

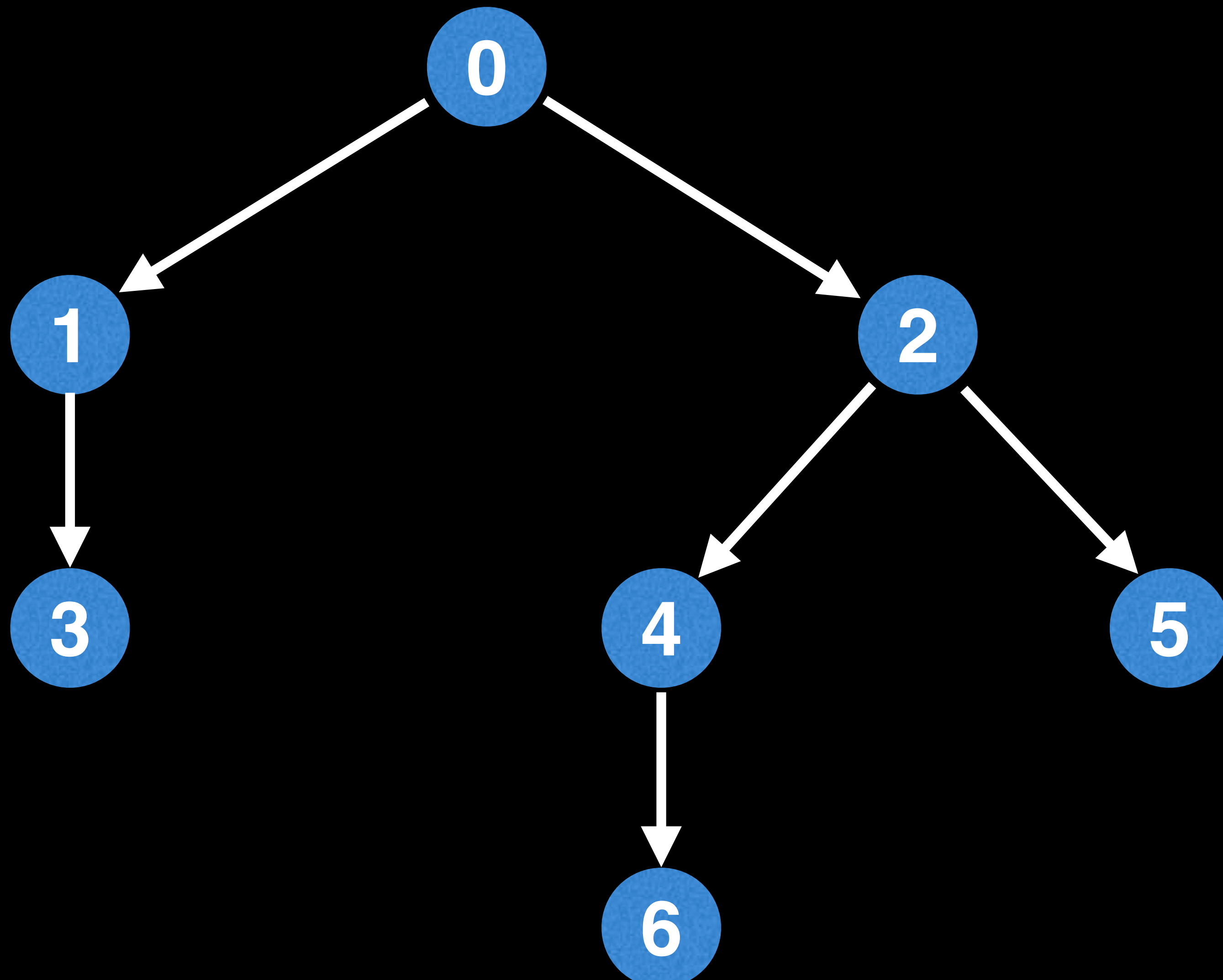



tour: [0,1,0,2,3,2,0]



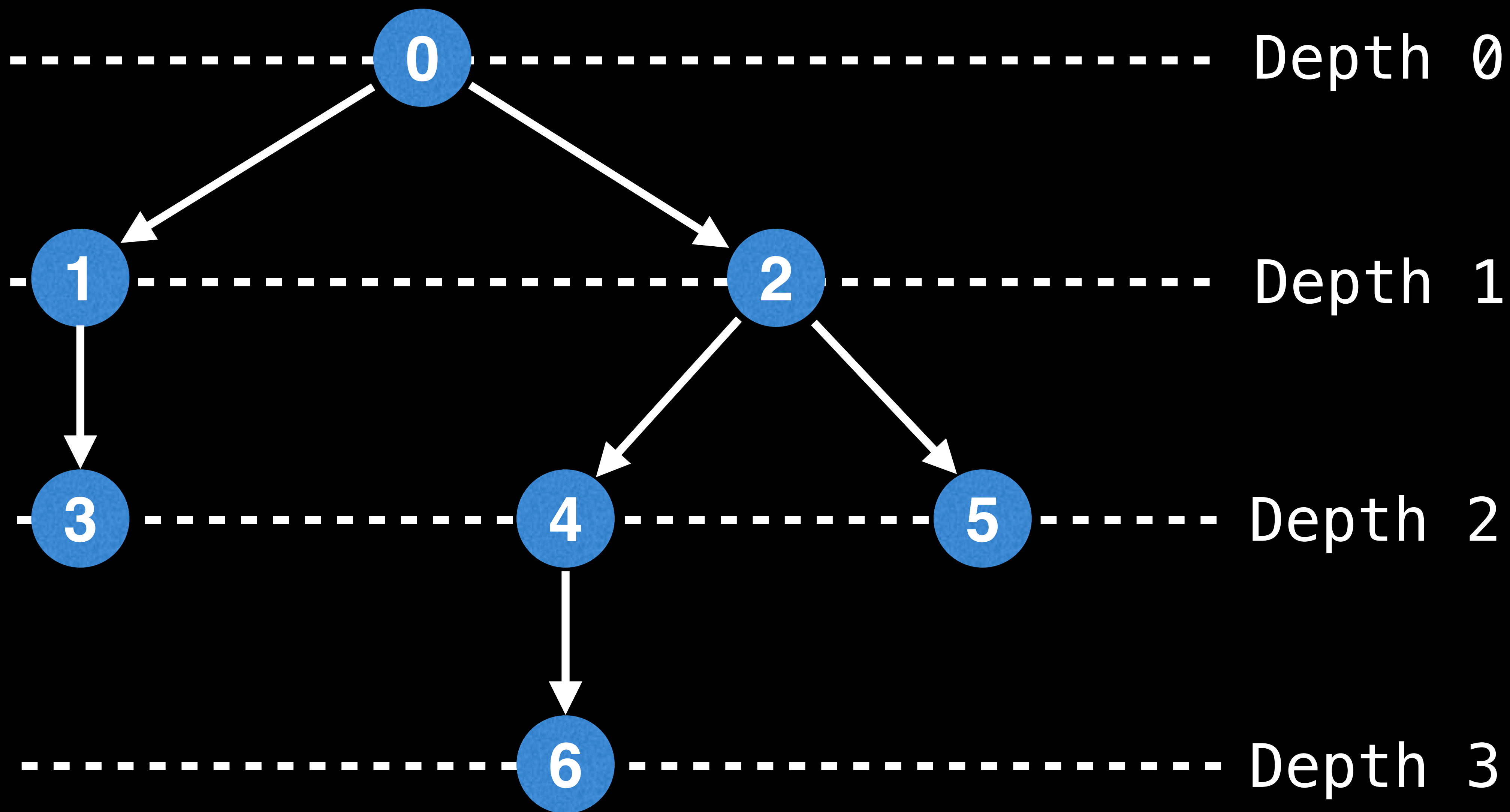

tour: [0,1,2,1,3,1,0]

Start an **Eulerian tour** (Eulerian circuit) 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.



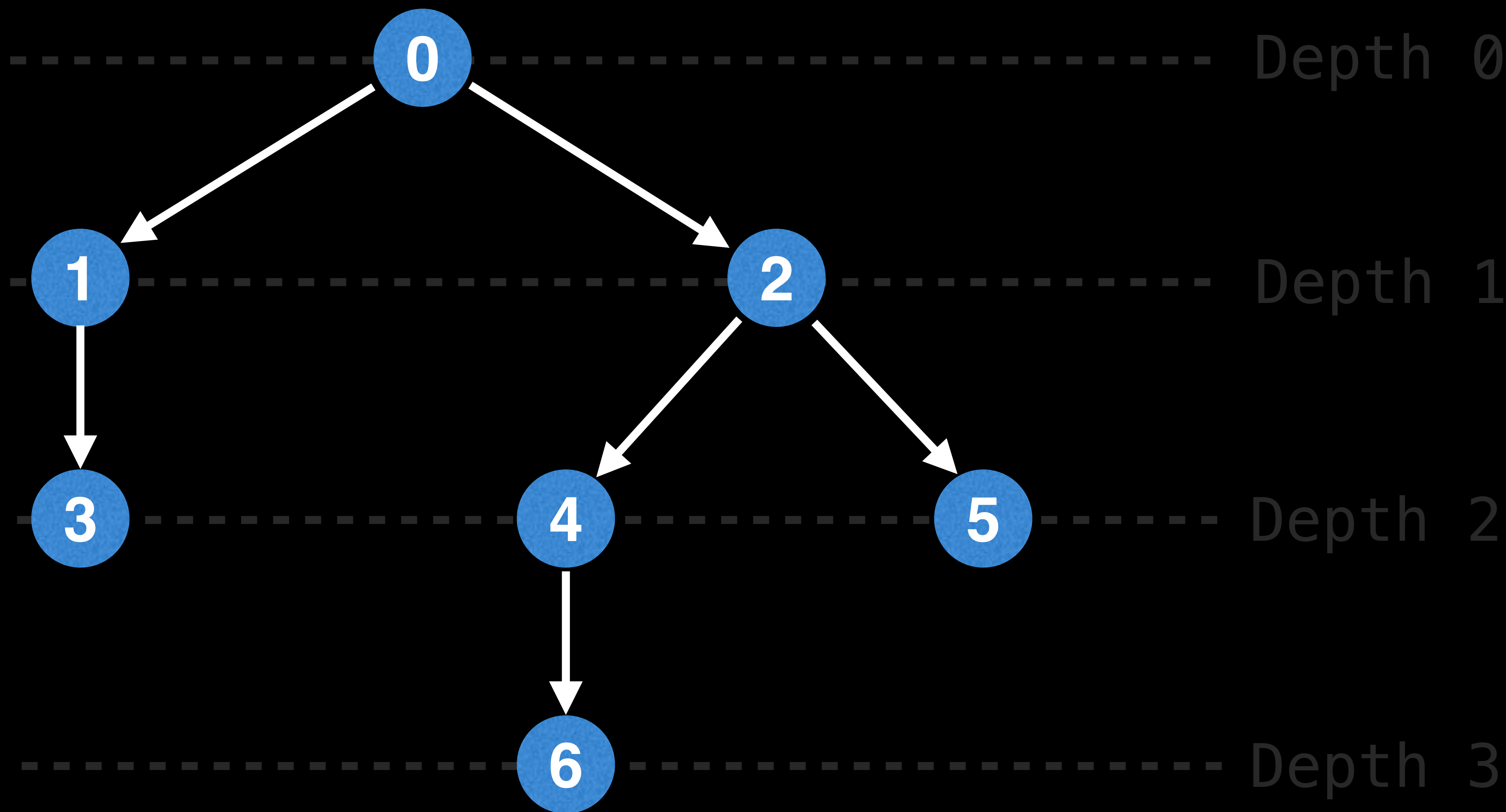

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| depth | | | | | | | | | | | | | |

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



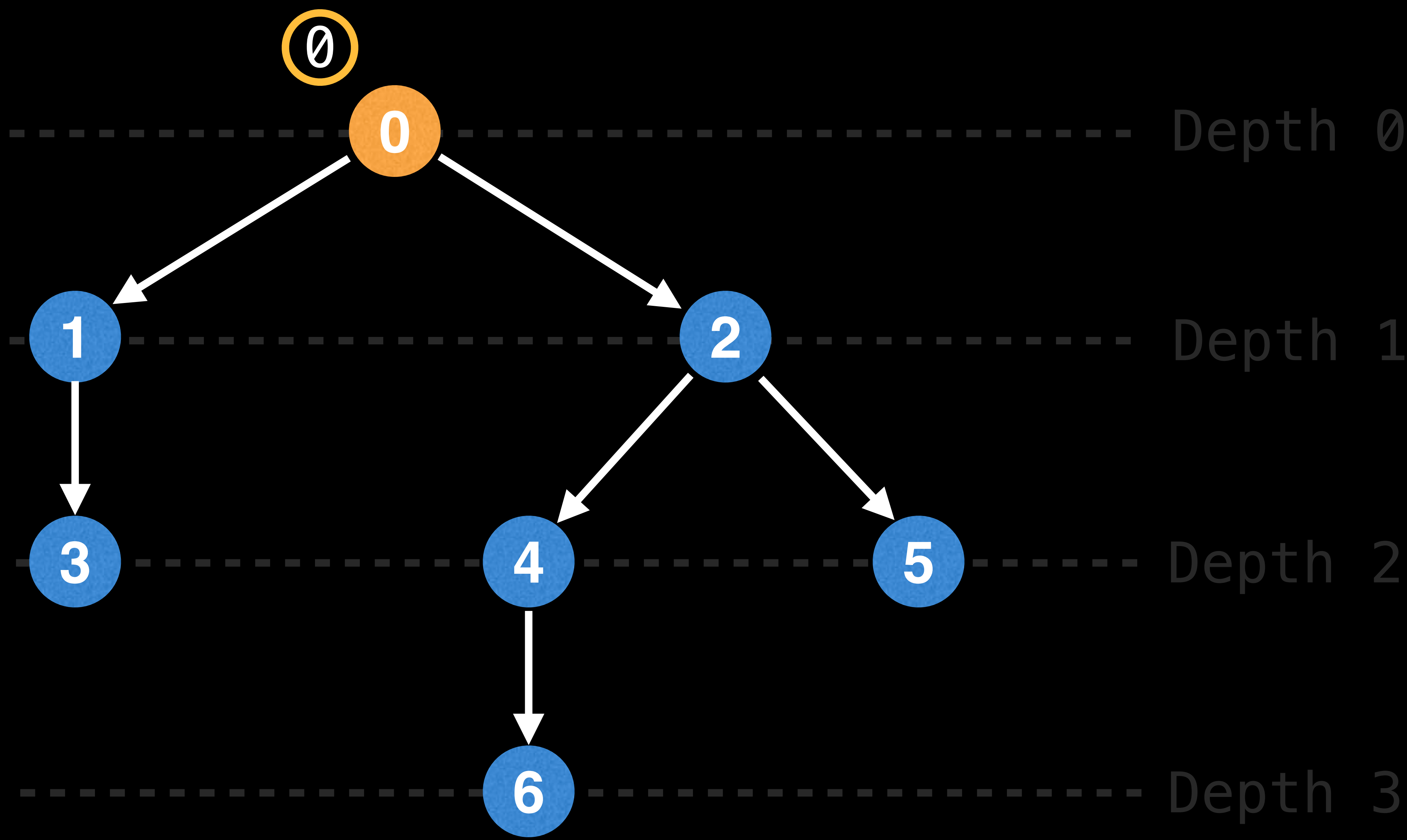

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| depth | | | | | | | | | | | | | |

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



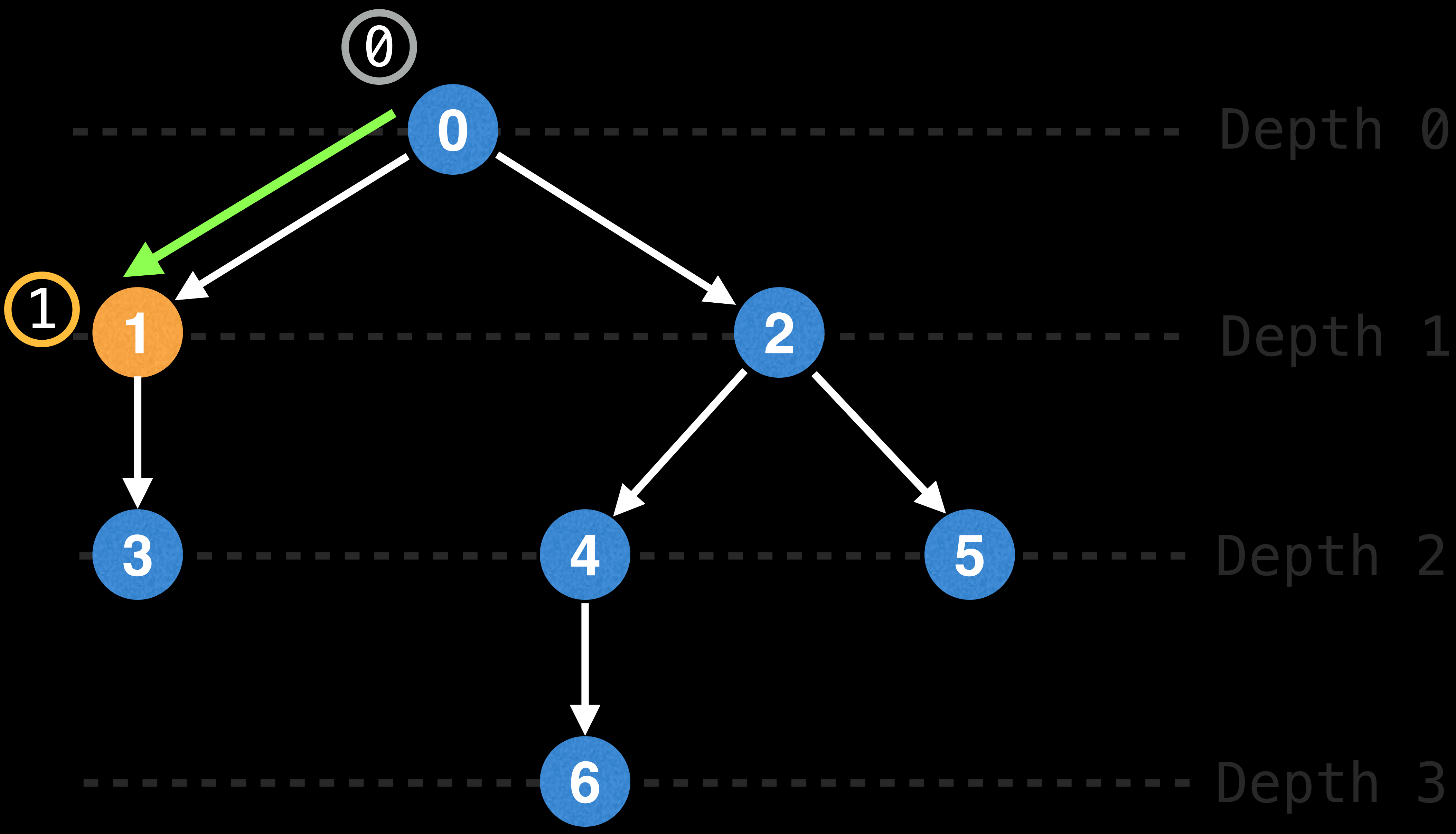

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| depth | | | | | | | | | | | | | |

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



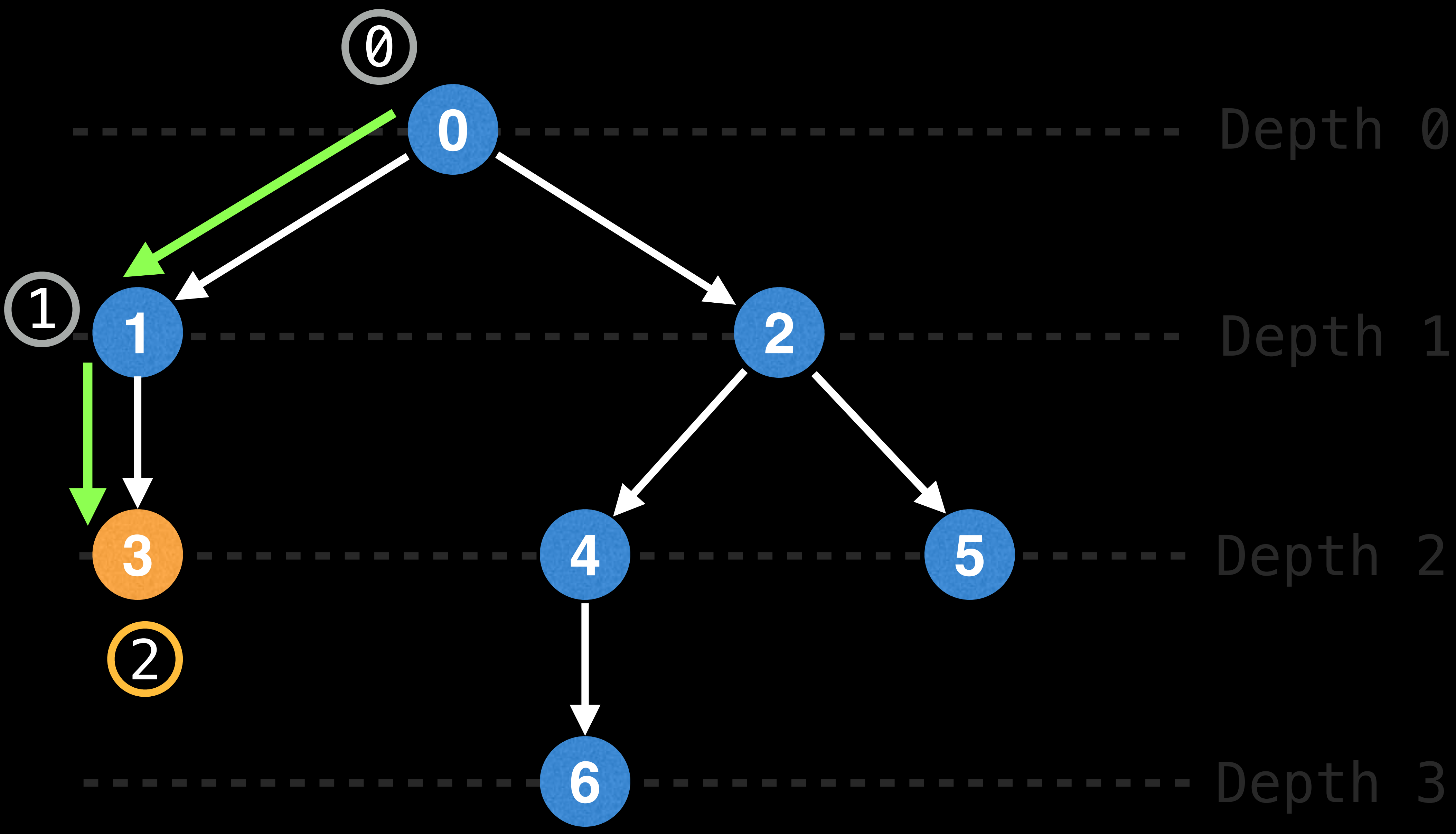

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| depth | 0 | | | | | | | | | | | | |

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



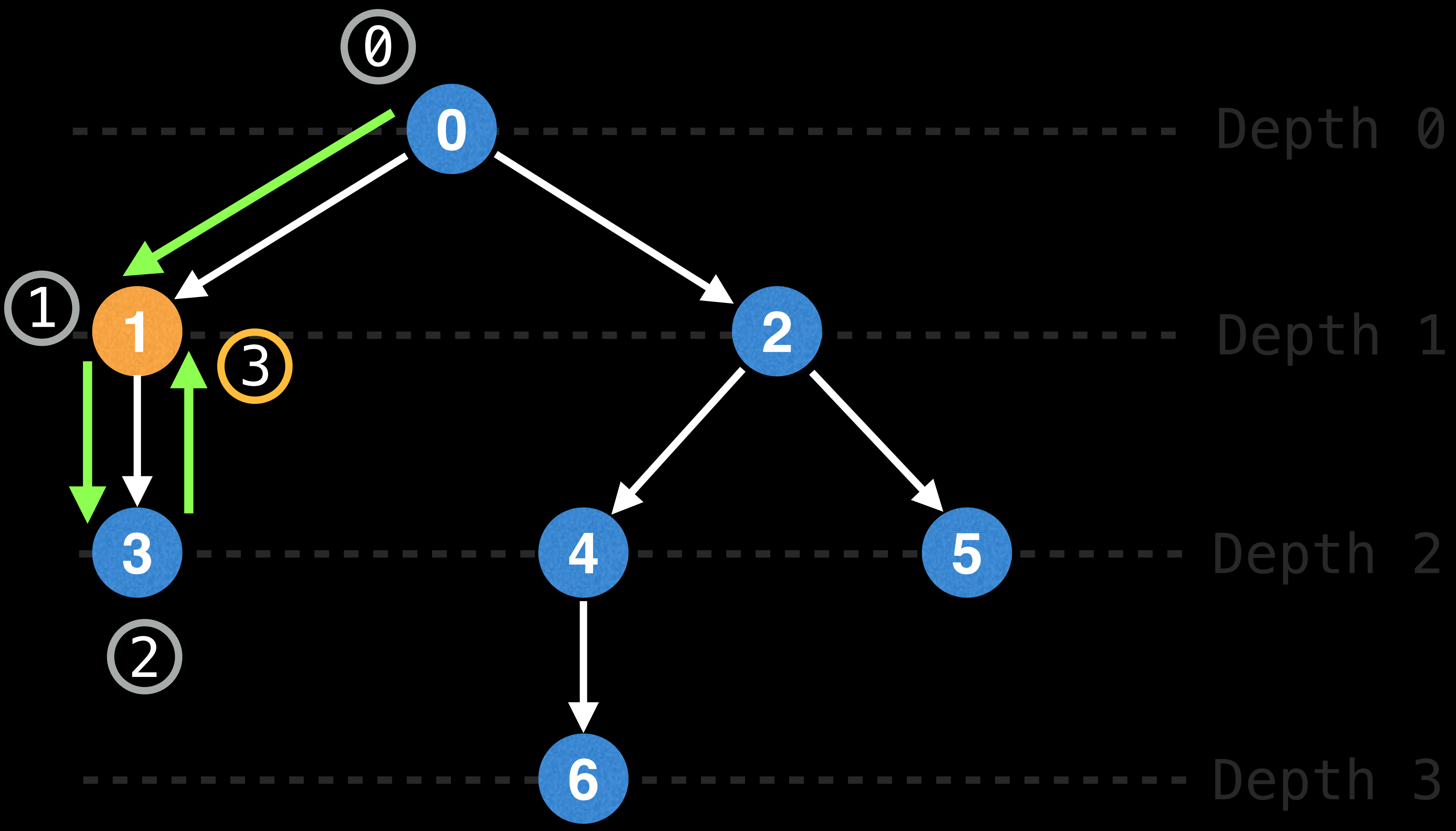

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| depth | 0 | 1 | | | | | | | | | | | |

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



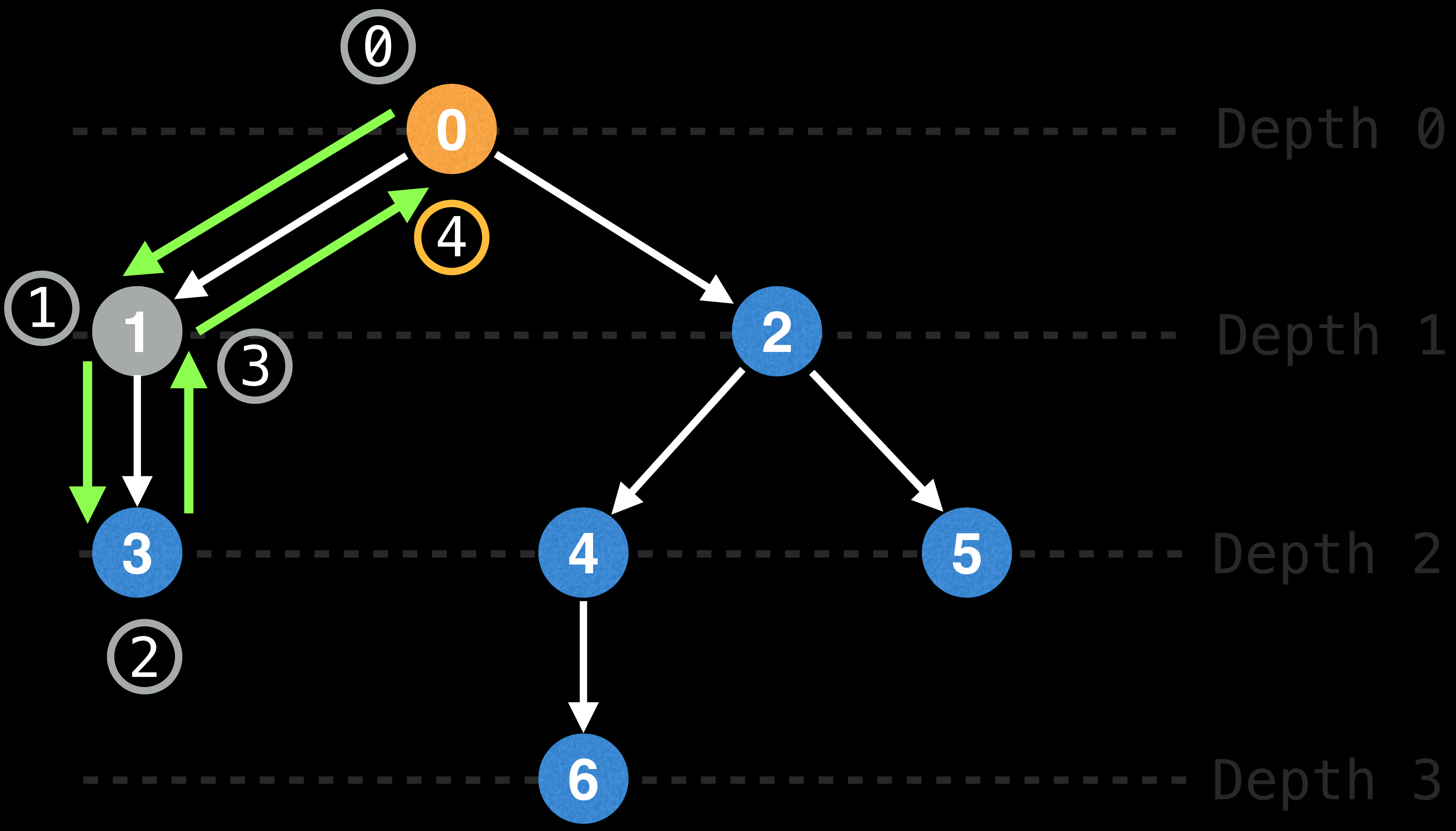

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| depth | 0 | 1 | 2 | | | | | | | | | | |

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



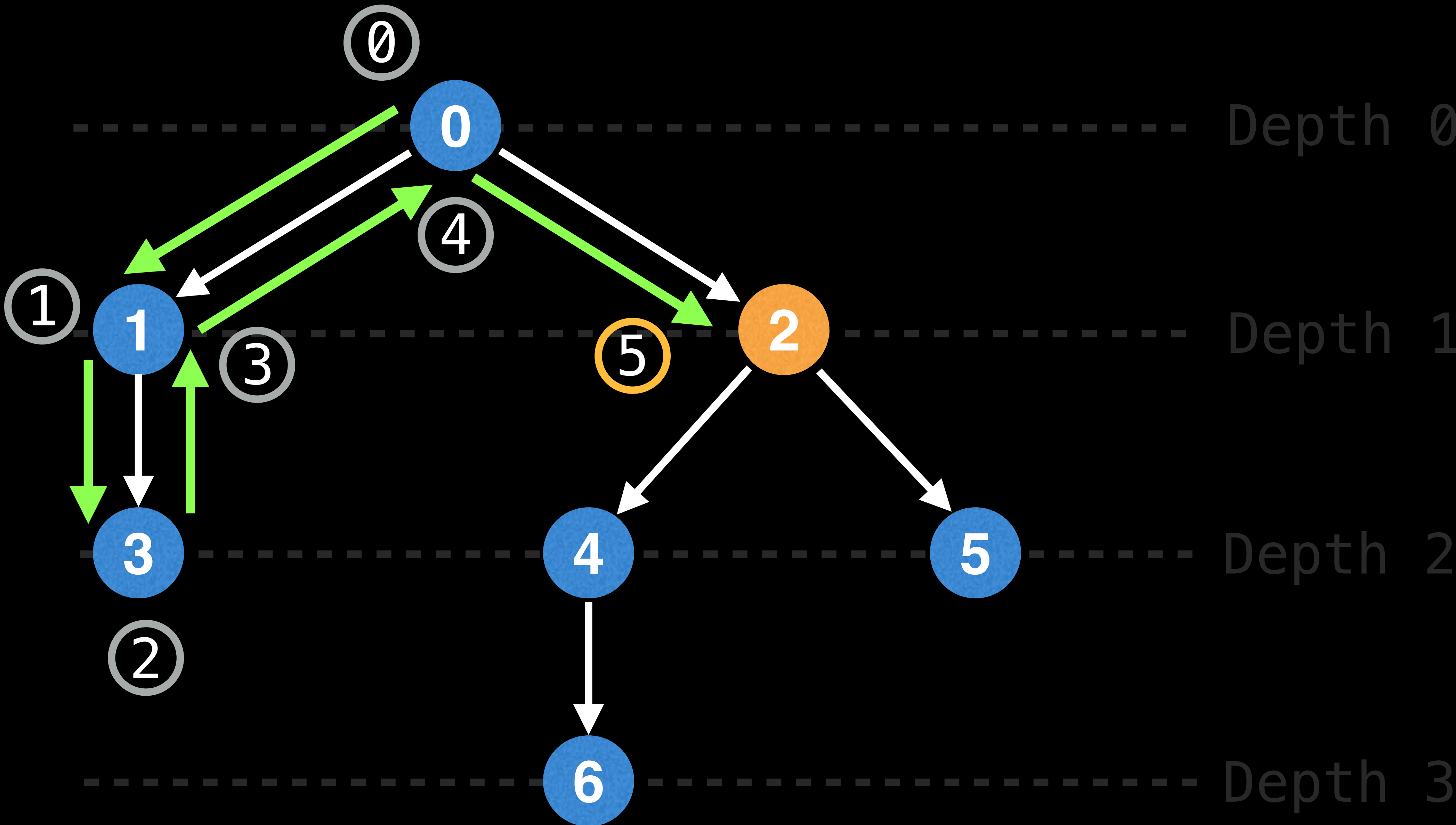

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| depth | 0 | 1 | 2 | 1 | | | | | | | | | |

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



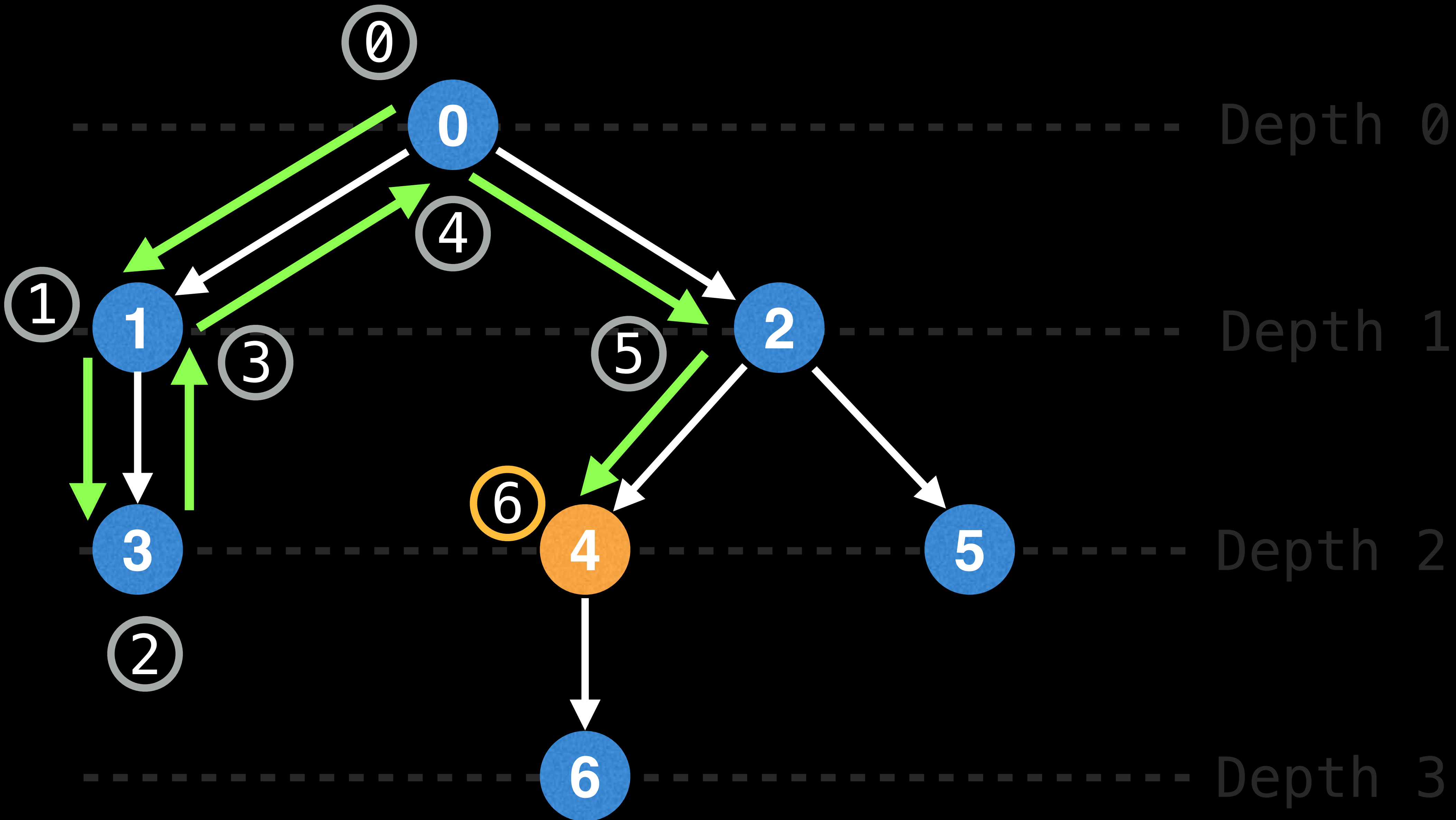

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| depth | 0 | 1 | 2 | 1 | 0 | | | | | | | | |

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



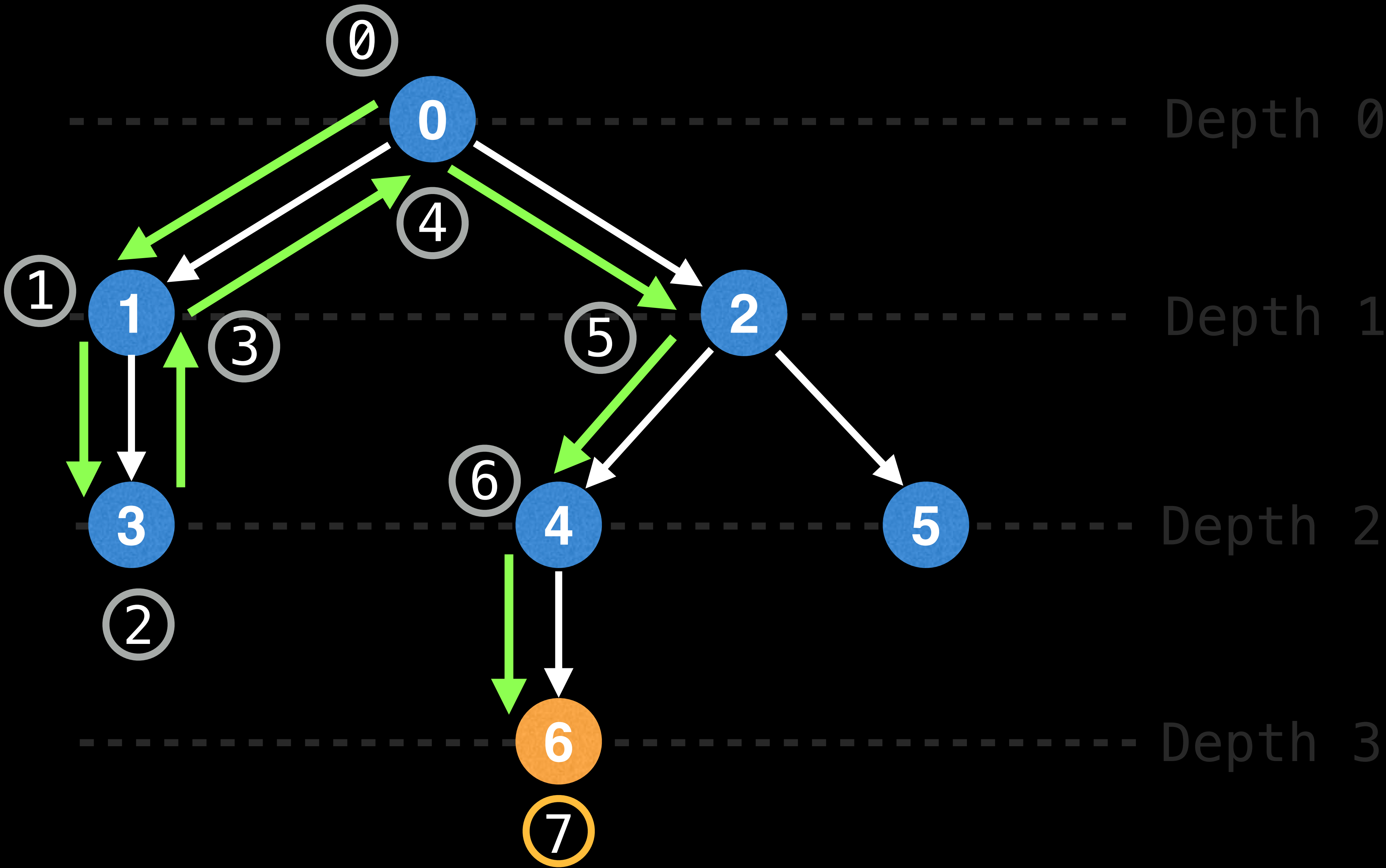

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| depth | 0 | 1 | 2 | 1 | 0 | 1 | | | | | | | |

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



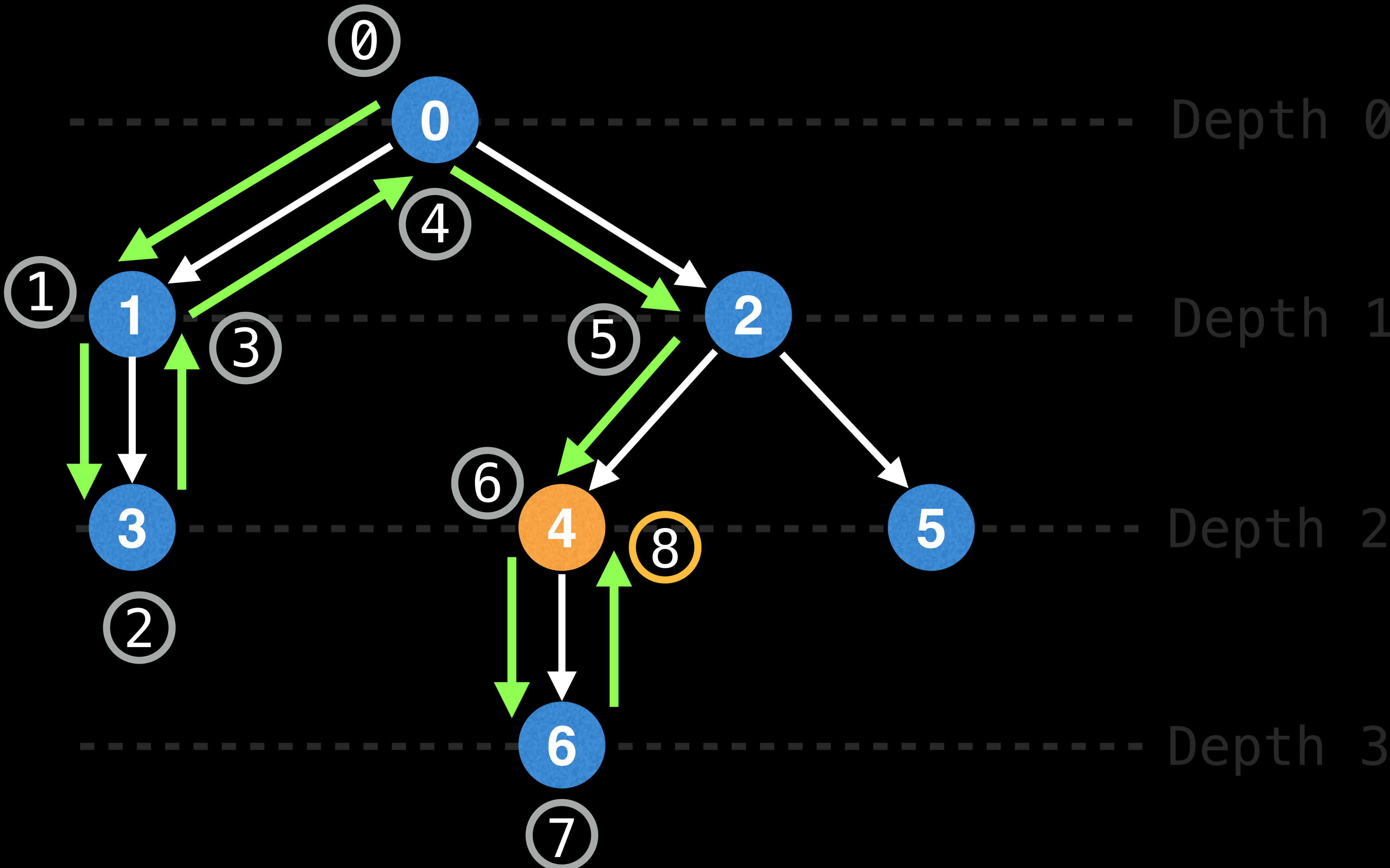

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| depth | 0 | 1 | 2 | 1 | 0 | 1 | 2 | | | | | | |

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



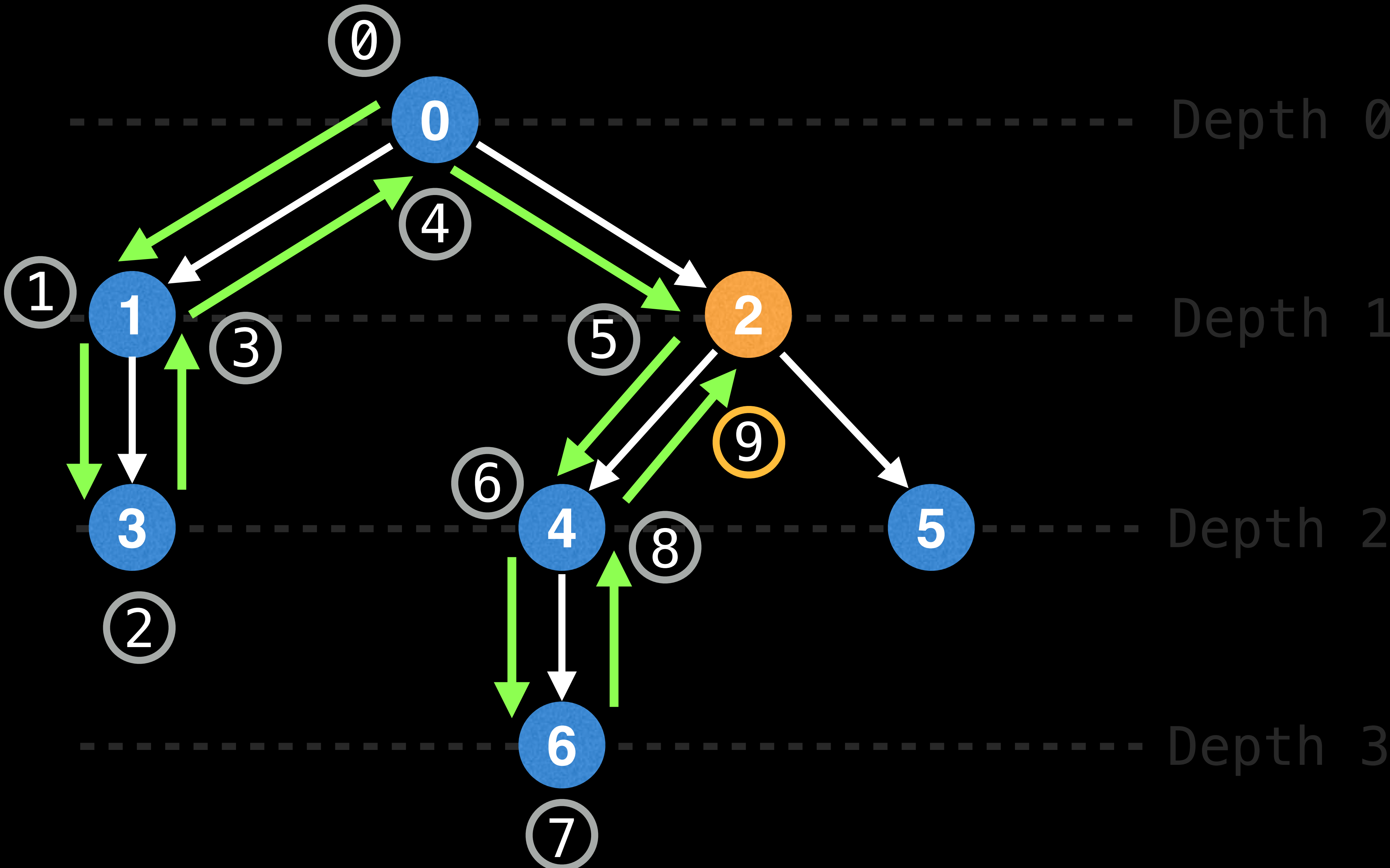

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| depth | 0 | 1 | 2 | 1 | 0 | 1 | 2 | 3 | | | | | |

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



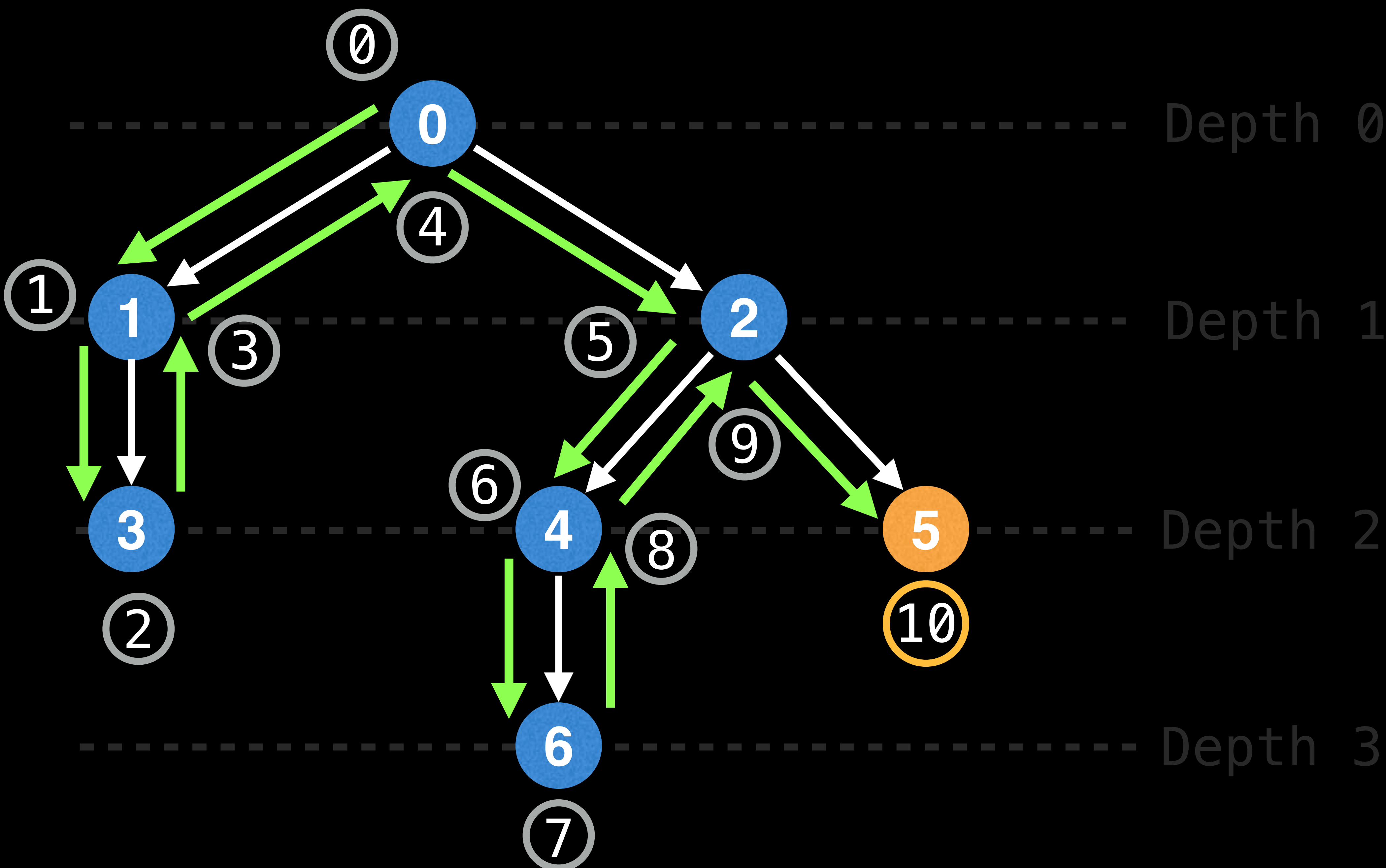

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| depth | 0 | 1 | 2 | 1 | 0 | 1 | 2 | 3 | 2 | | | | |

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



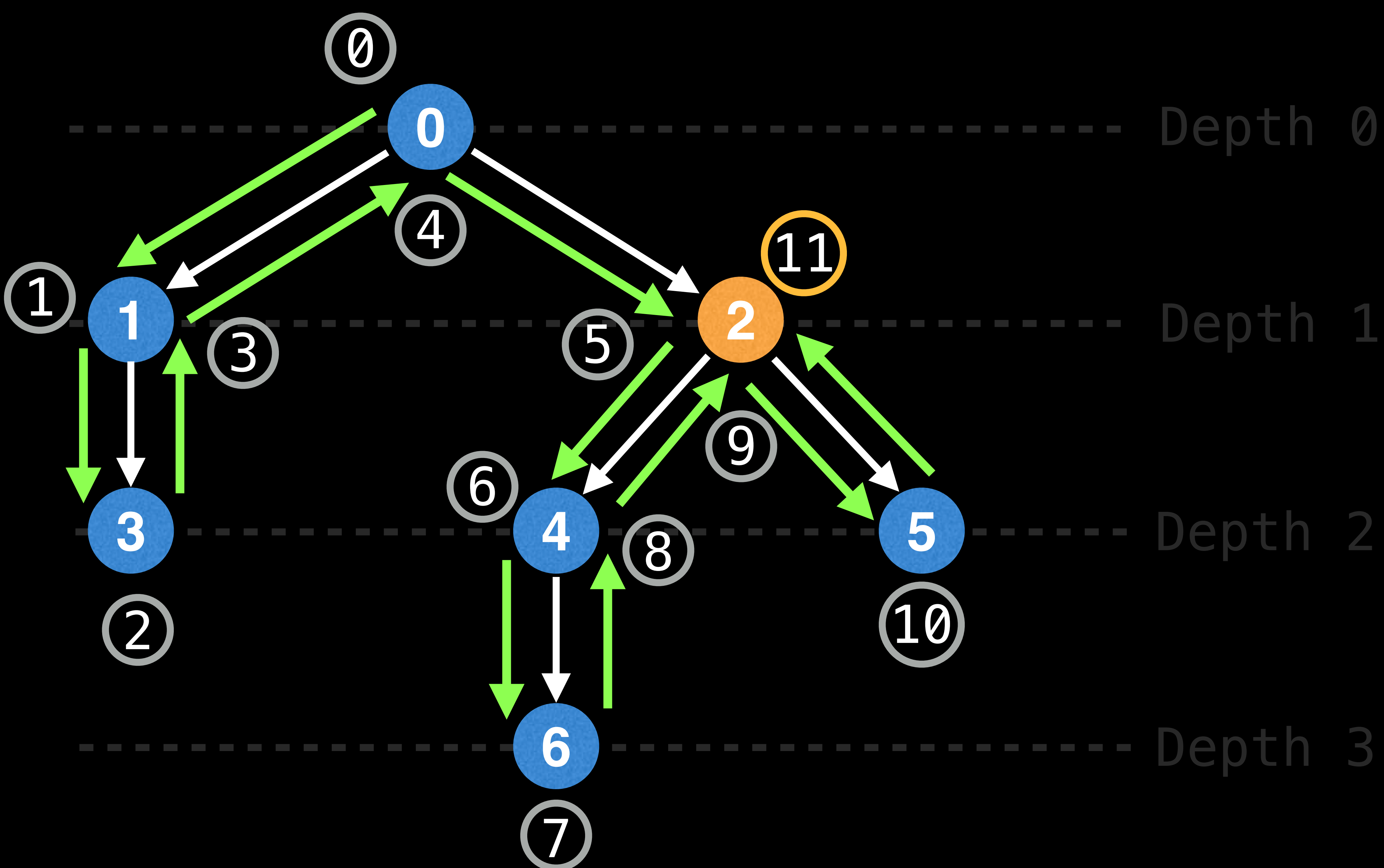

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| depth | 0 | 1 | 2 | 1 | 0 | 1 | 2 | 3 | 2 | 1 | | | |

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

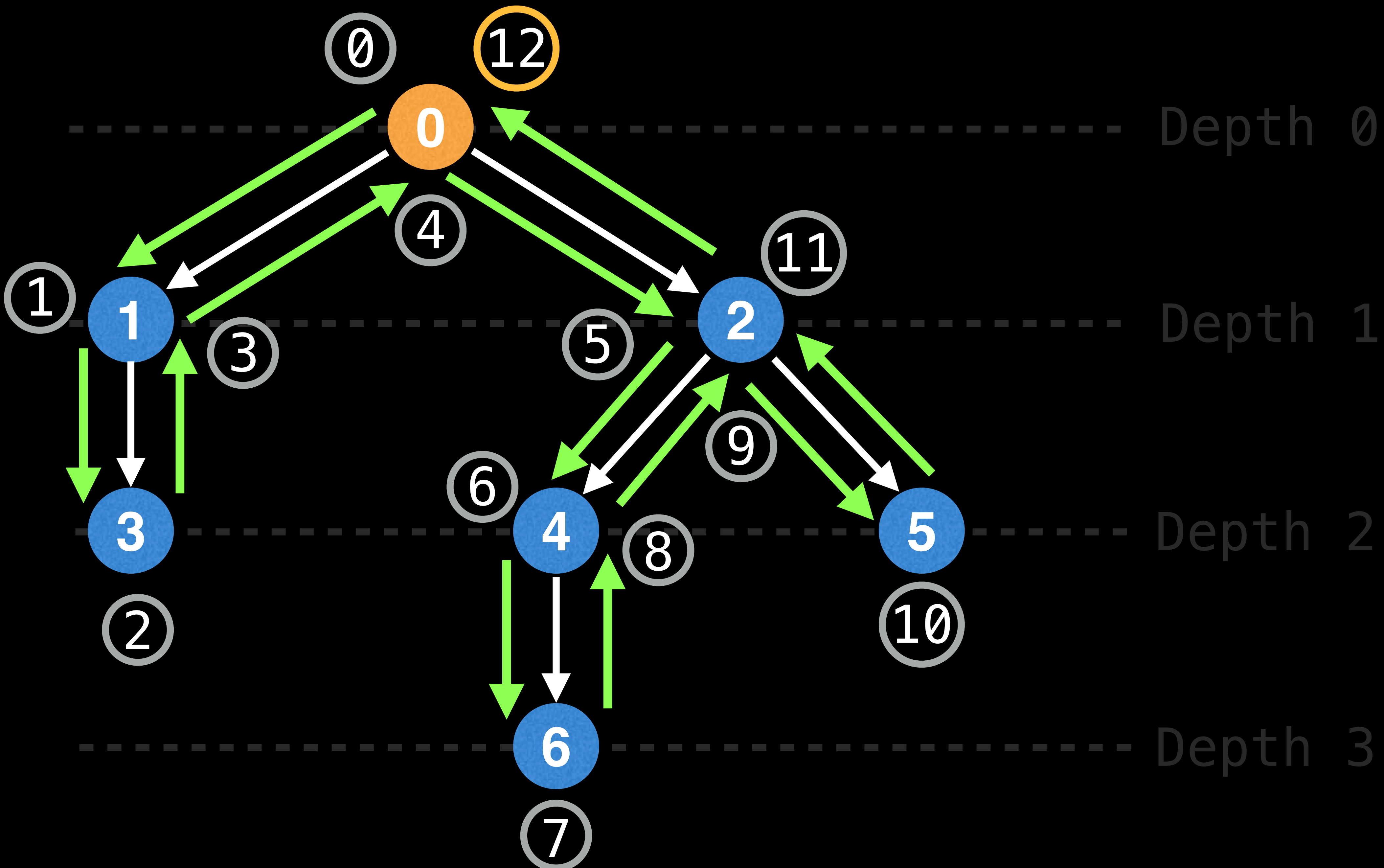



| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| depth | 0 | 1 | 2 | 1 | 0 | 1 | 2 | 3 | 2 | 1 | 2 | | |

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

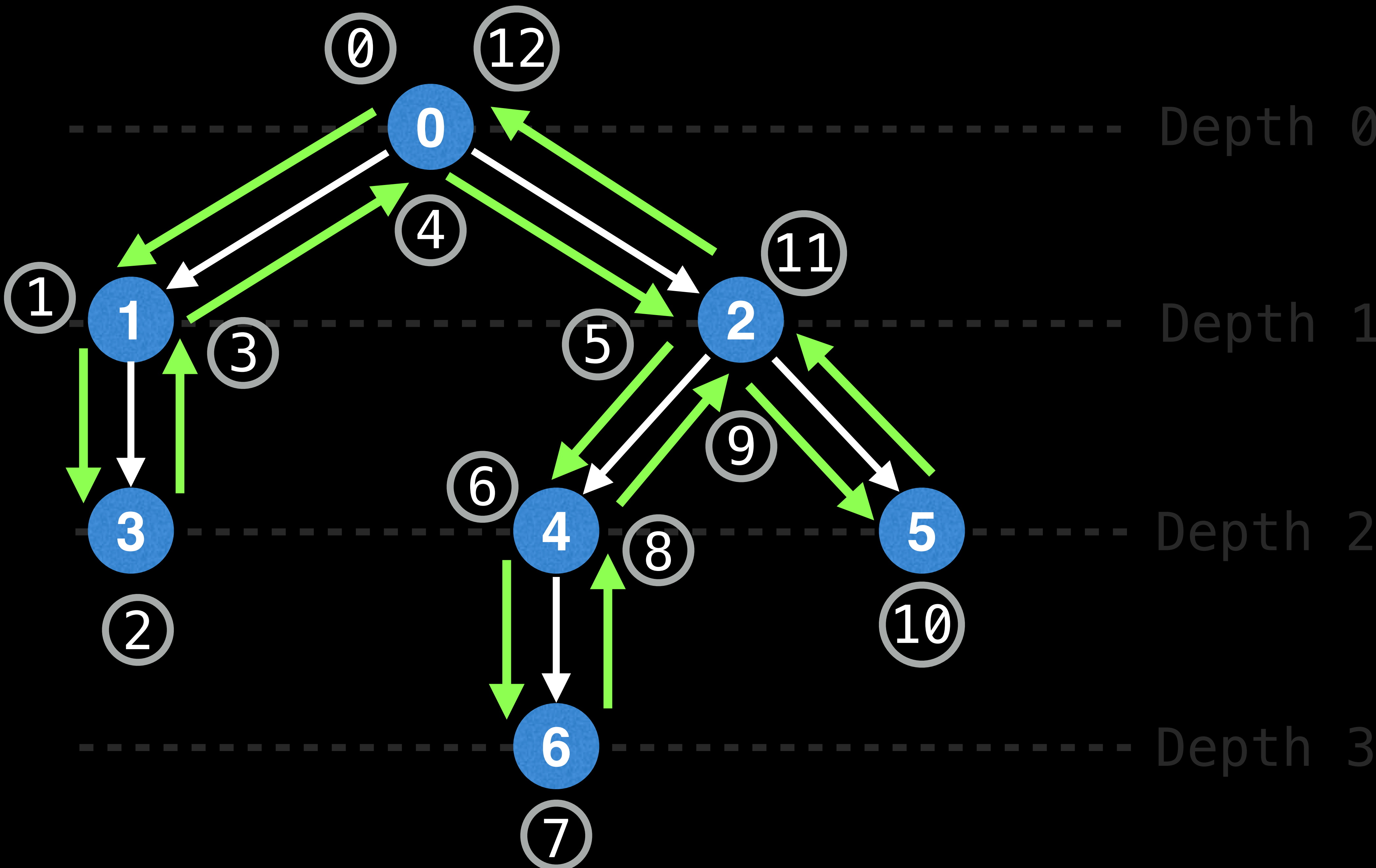



| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| depth | 0 | 1 | 2 | 1 | 0 | 1 | 2 | 3 | 2 | 1 | 2 | 1 | |
| nodes | 0 | 1 | 3 | 1 | 0 | 2 | 4 | 6 | 4 | 2 | 5 | 2 | |



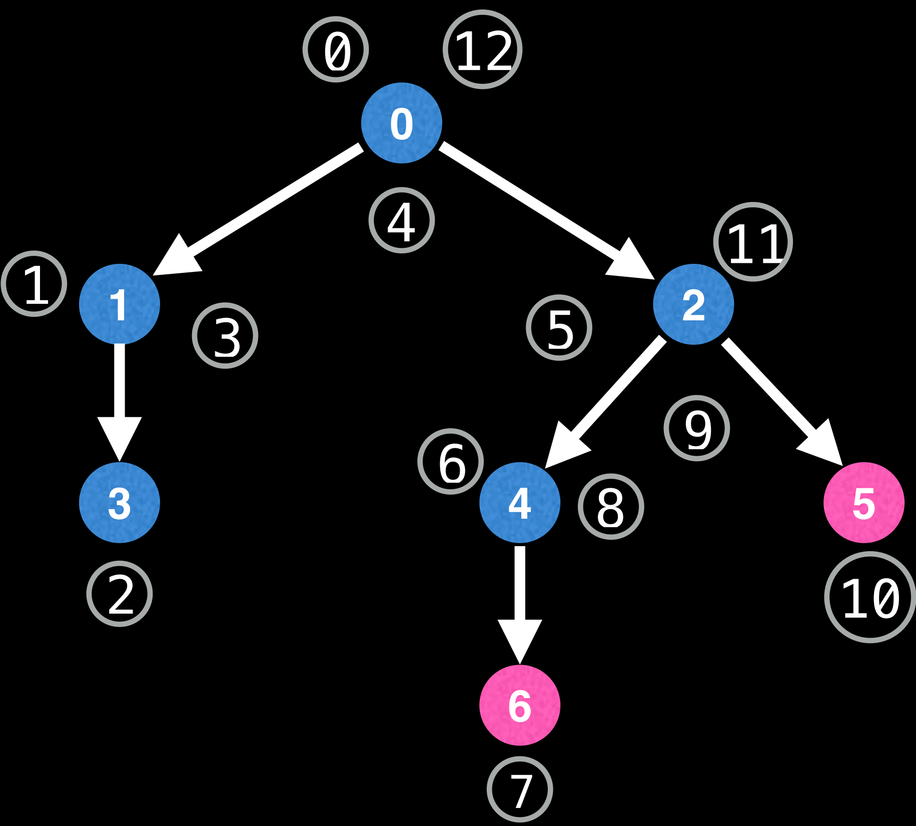

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| depth | 0 | 1 | 2 | 1 | 0 | 1 | 2 | 3 | 2 | 1 | 2 | 1 | 0 |

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




| | 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 |

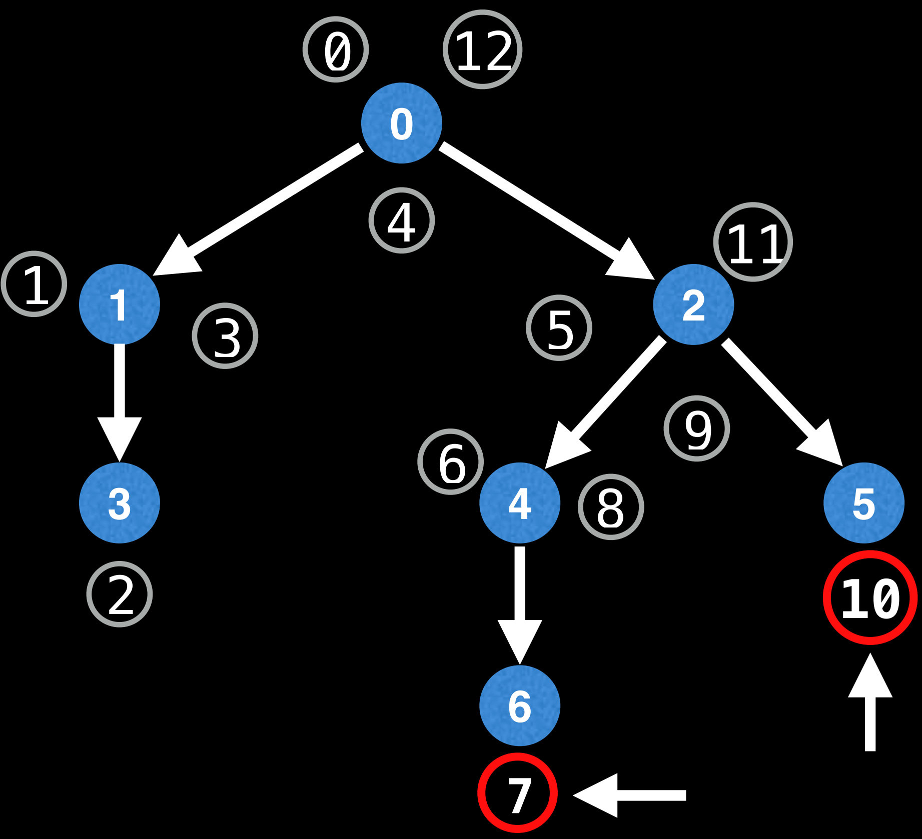
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| nodes | 0 | 1 | 3 | 1 | 0 | 2 | 4 | 6 | 4 | 2 | 5 | 2 | 0 |




Q: What is **LCA**(6, 5)?

| | 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 |

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




Q: What is **LCA**(6, 5)?

1. Find the index position value for the nodes \`a\` and \`b\` (5 and 6 respectably)

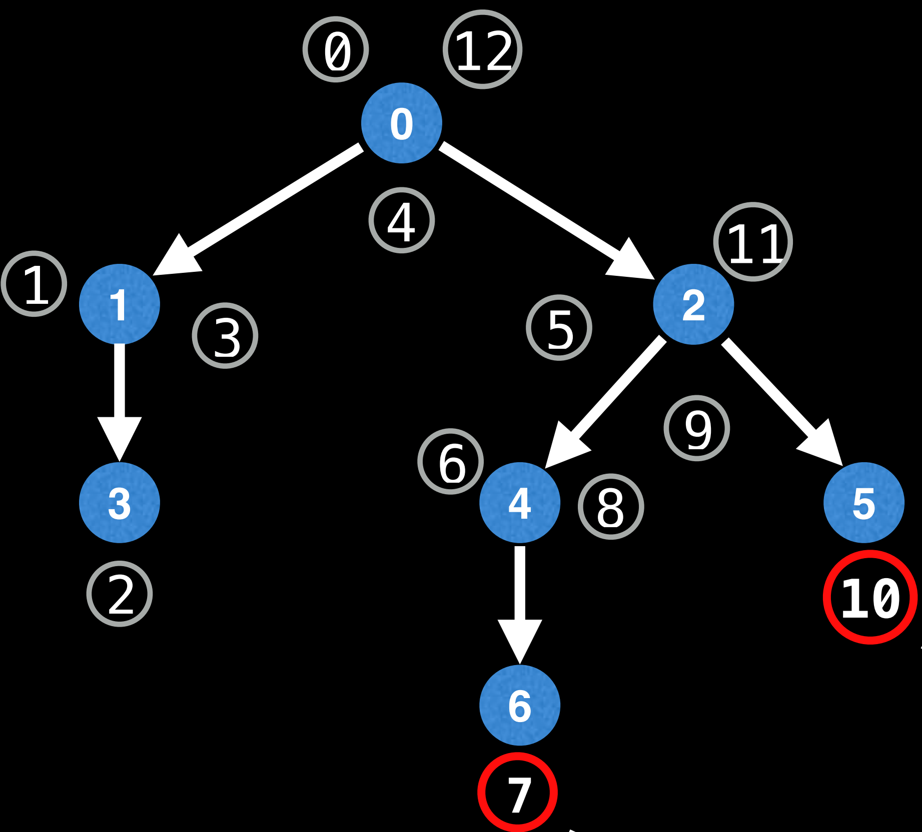
Nodes 5 and 6 map the the index positions 7 and 10 in the Euler Tour

| | 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 |

Q: What is **LCA**(6, 5)?

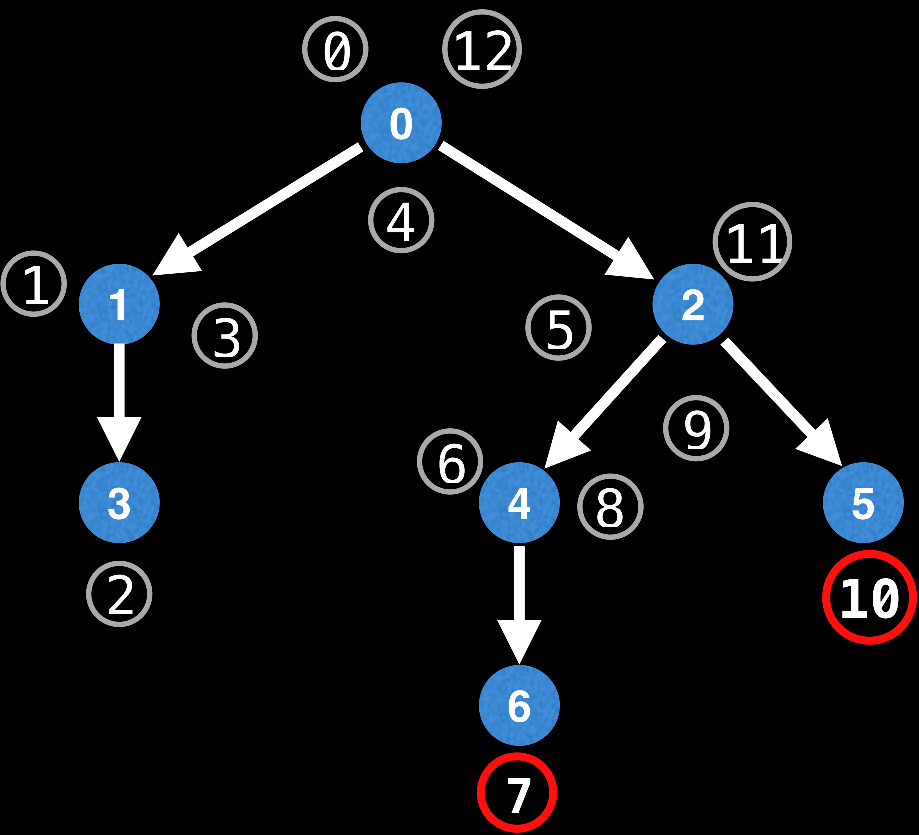
1. Find the index position value for the nodes \`a\` and \`b\` (5 and 6 respectably)

2. Using the depth array, find the index of the minimum value in the range of the indices obtained in step 1




| | 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 |

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




Q: What is **LCA**(6, 5)?

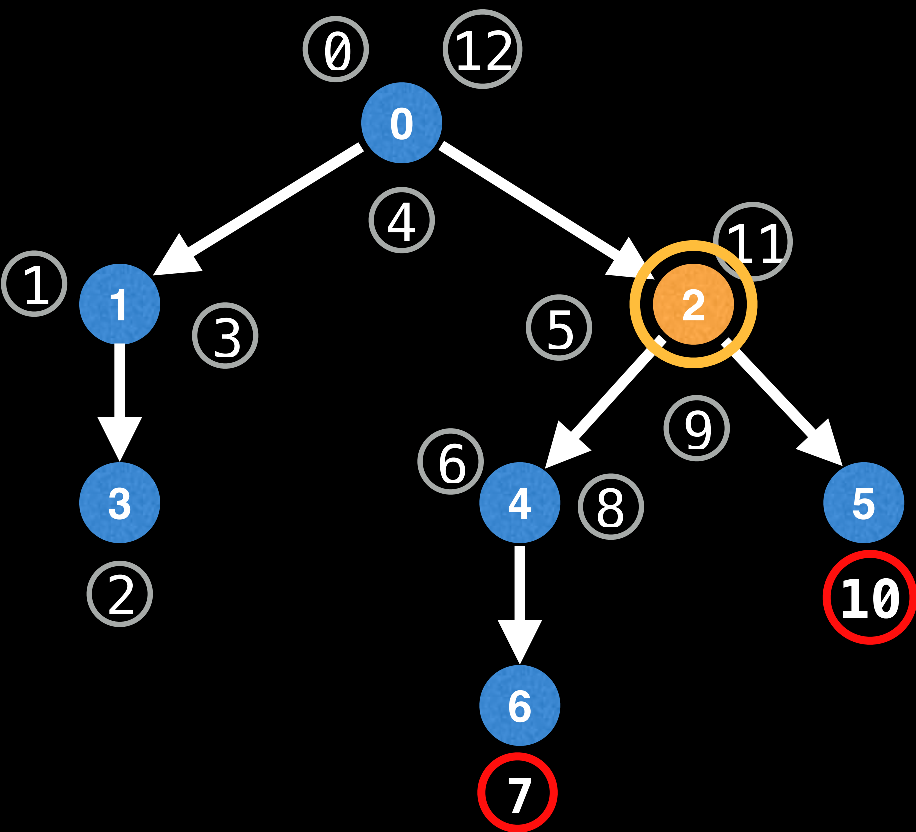
1. Find the index position value for the nodes `a` and `b` (5 and 6 respectably)

2. Using the depth array, find the index of the minimum value in the range of the indices obtained in step 1

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 a Sparse Table. For this example, the index is `9` with a value of `1`

| | 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 |

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




Q: What is **LCA**(6, 5)?

1. Find the index position value for the nodes \`a\` and \`b\` (5 and 6 respectably)

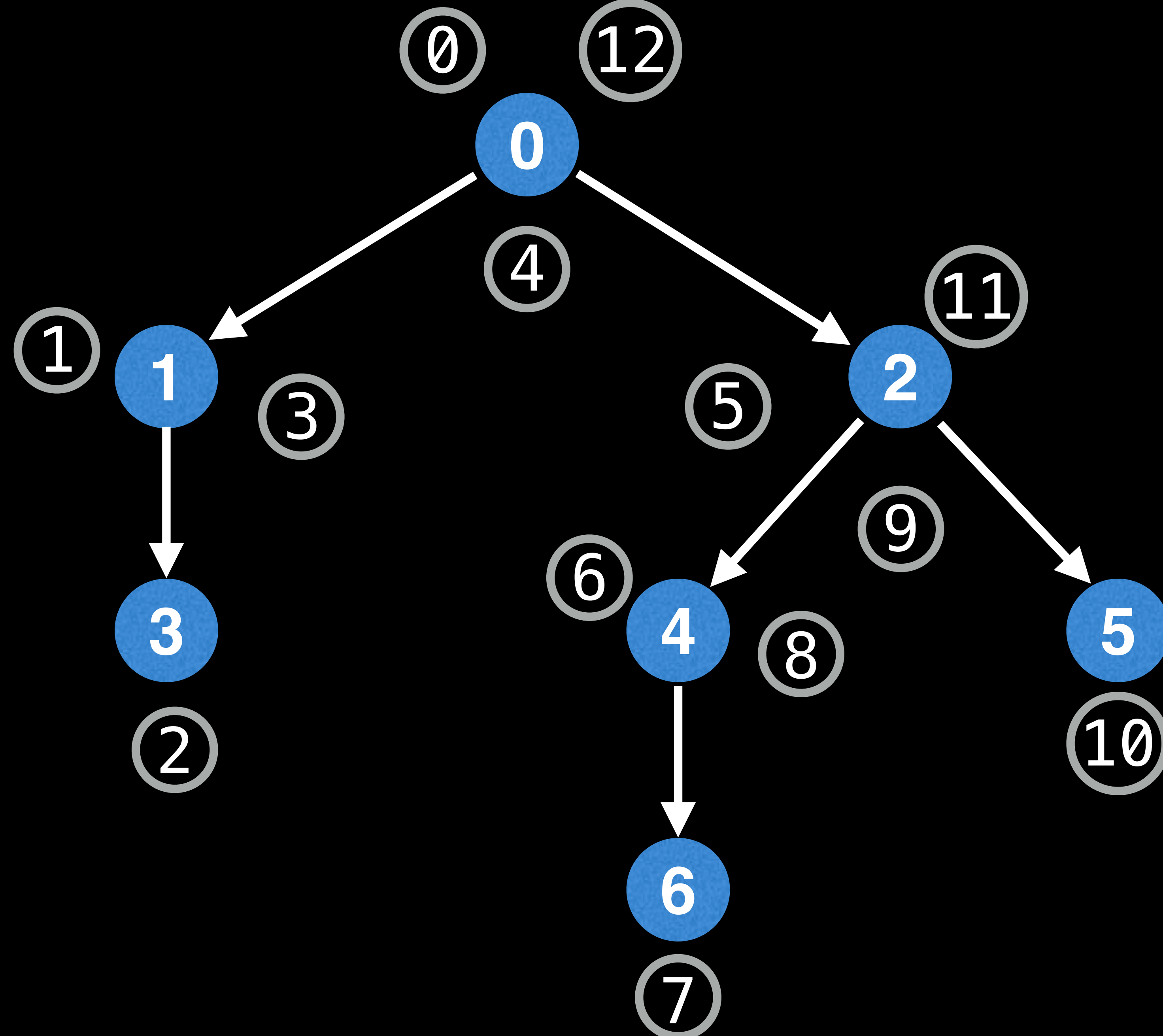
2. Using the depth array, find the index of the minimum value in the range of the indices obtained in step 1

3. Using the index obtained in step 2, find the LCA of \`a\` and \`b\` in the \`nodes\` array.

With index 9 found in the previous step, retrieve the LCA at nodes[9]

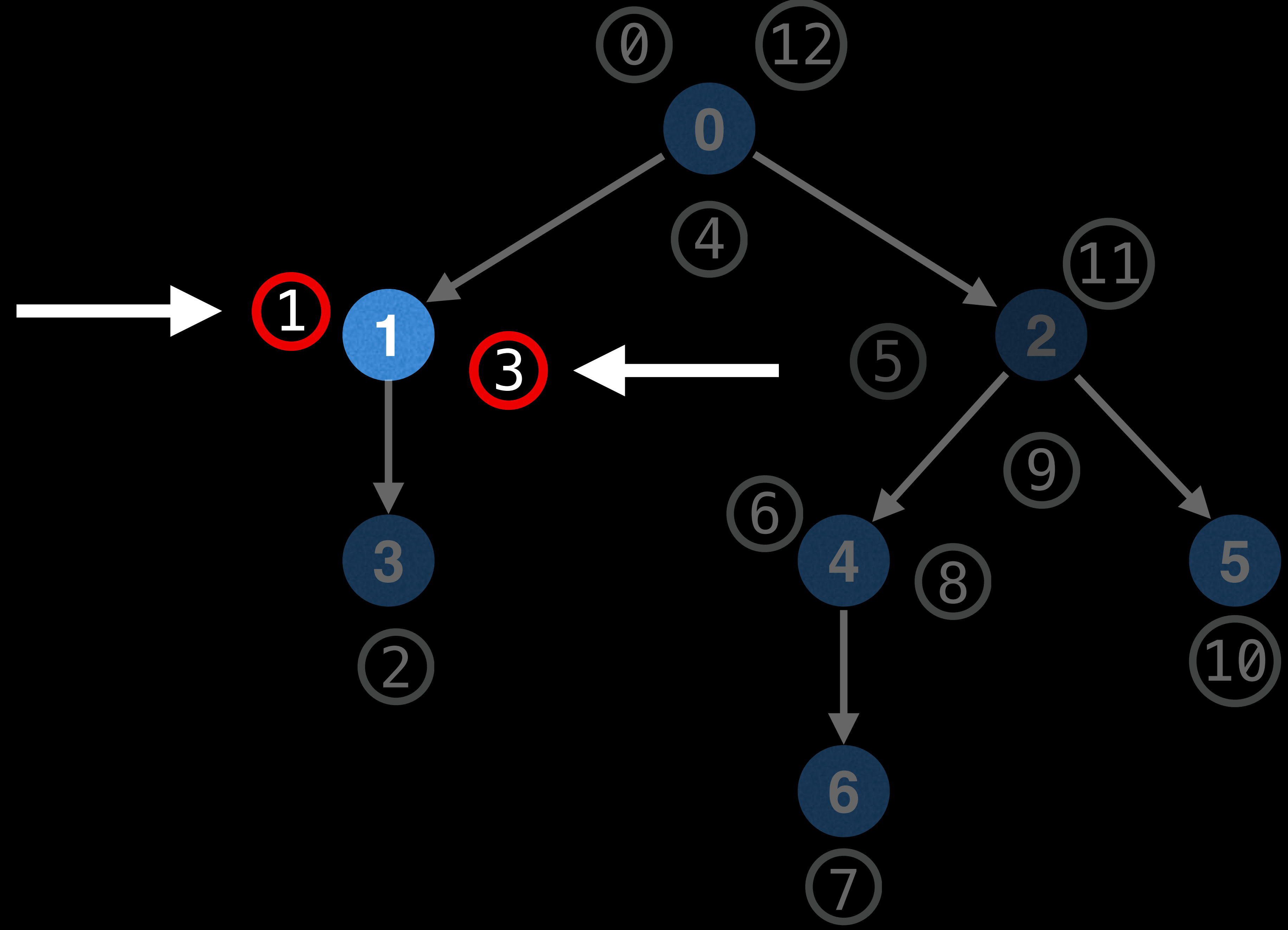
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| depth | 0 | 1 | 2 | 1 | 0 | 1 | 2 | 3 | 2 | 1 | 2 | 1 | 0 |

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

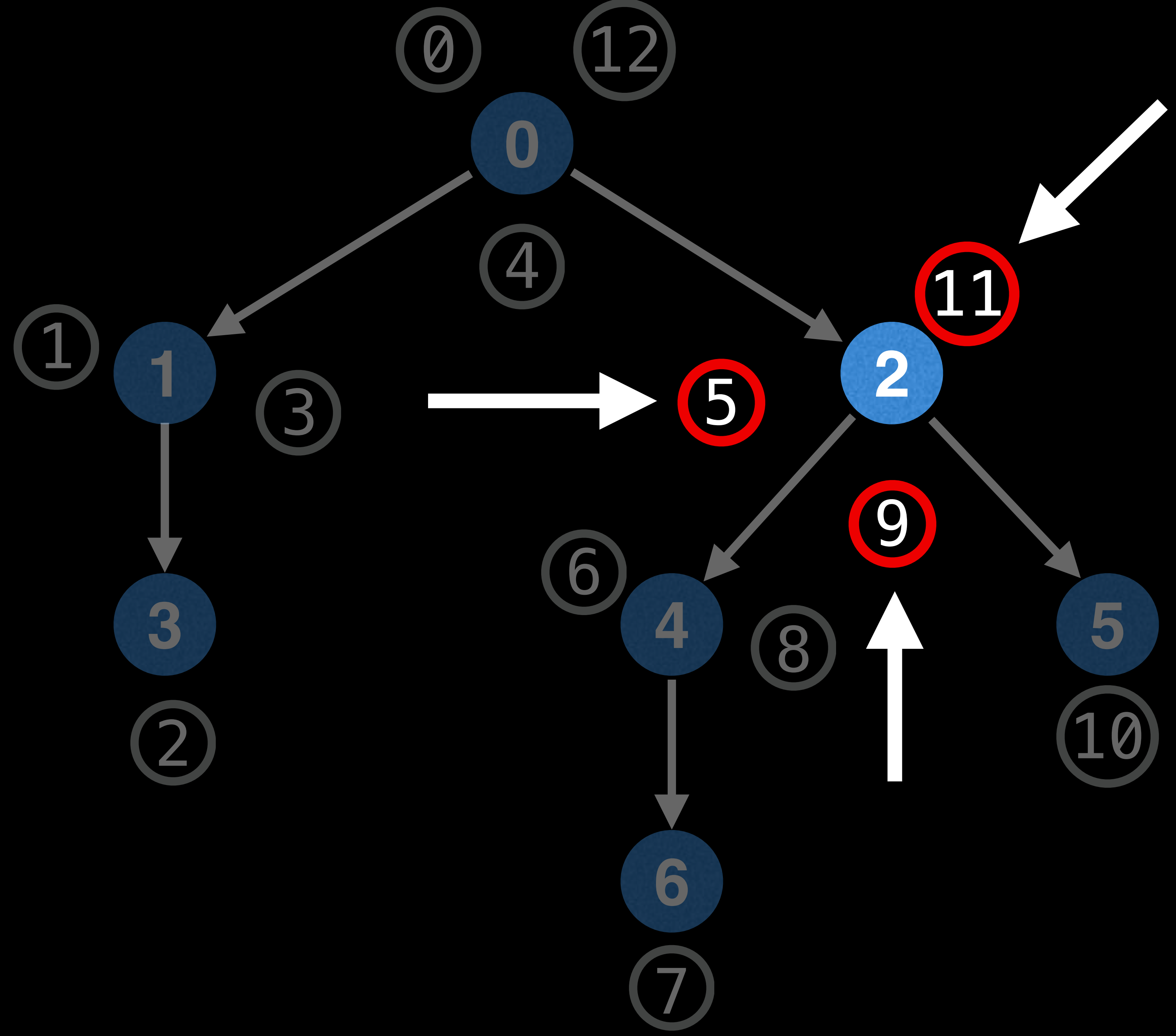



If you recall, step 1 required finding the index position for the two nodes with ids `a` and `b`.

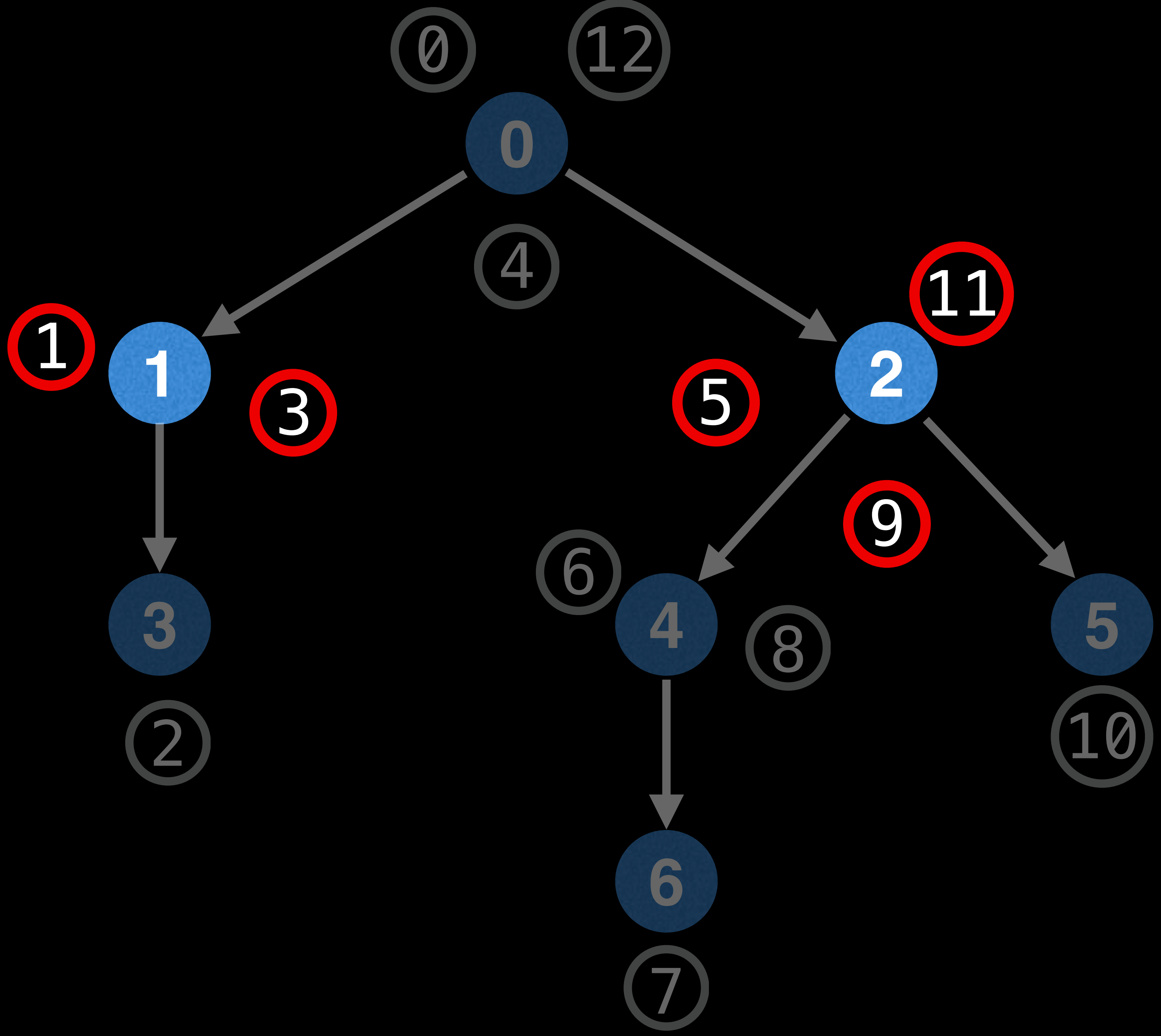
However, an issue we soon run into is that there are 2n - 1 nodes index positions in the Euler tour, and only n nodes in total, so a perfect 1 to 1 inverse mapping isn't possible.



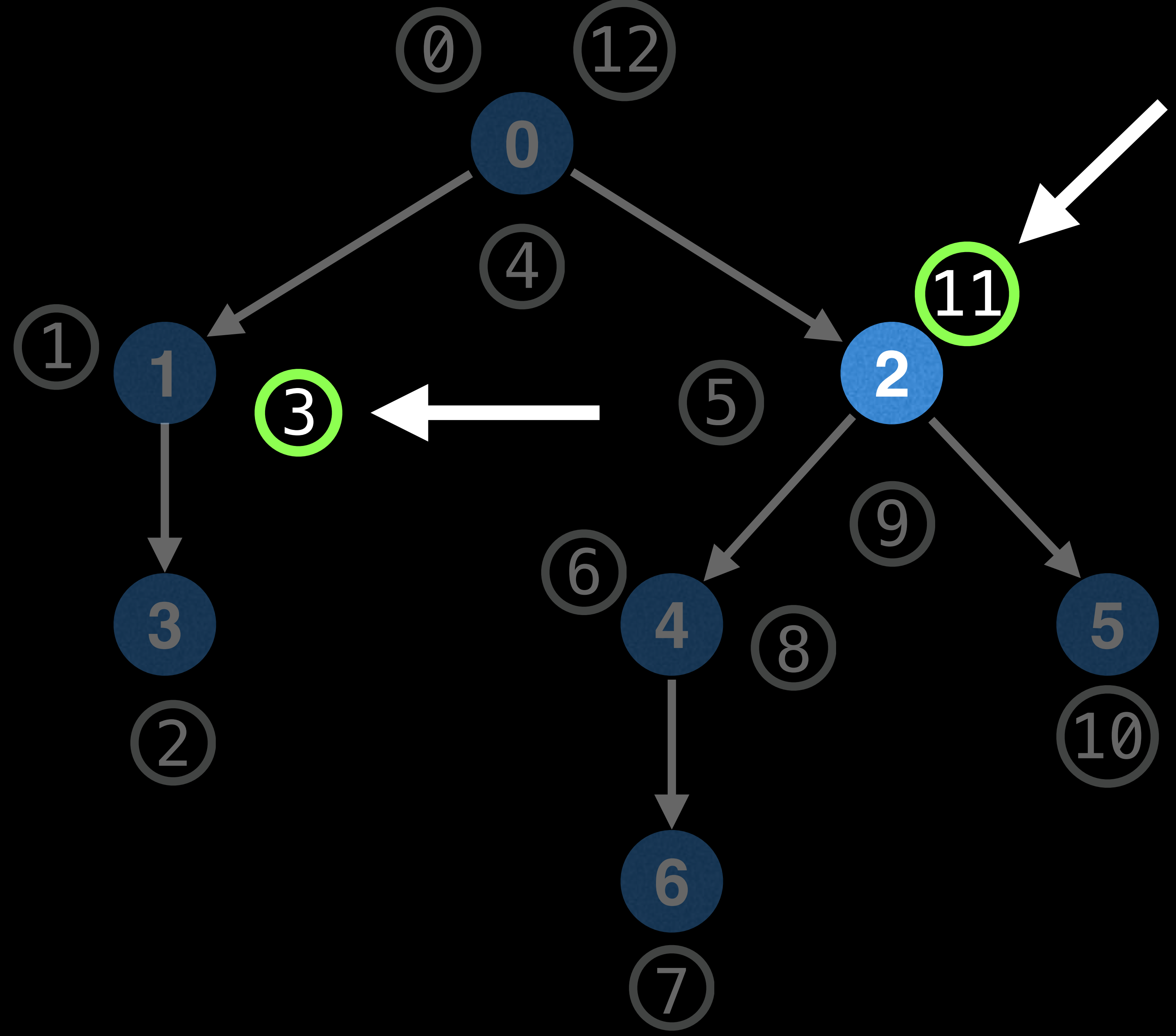

For example, the inverse mapping of node 1 could map to either index 1 or index 3 in the Euler tour.



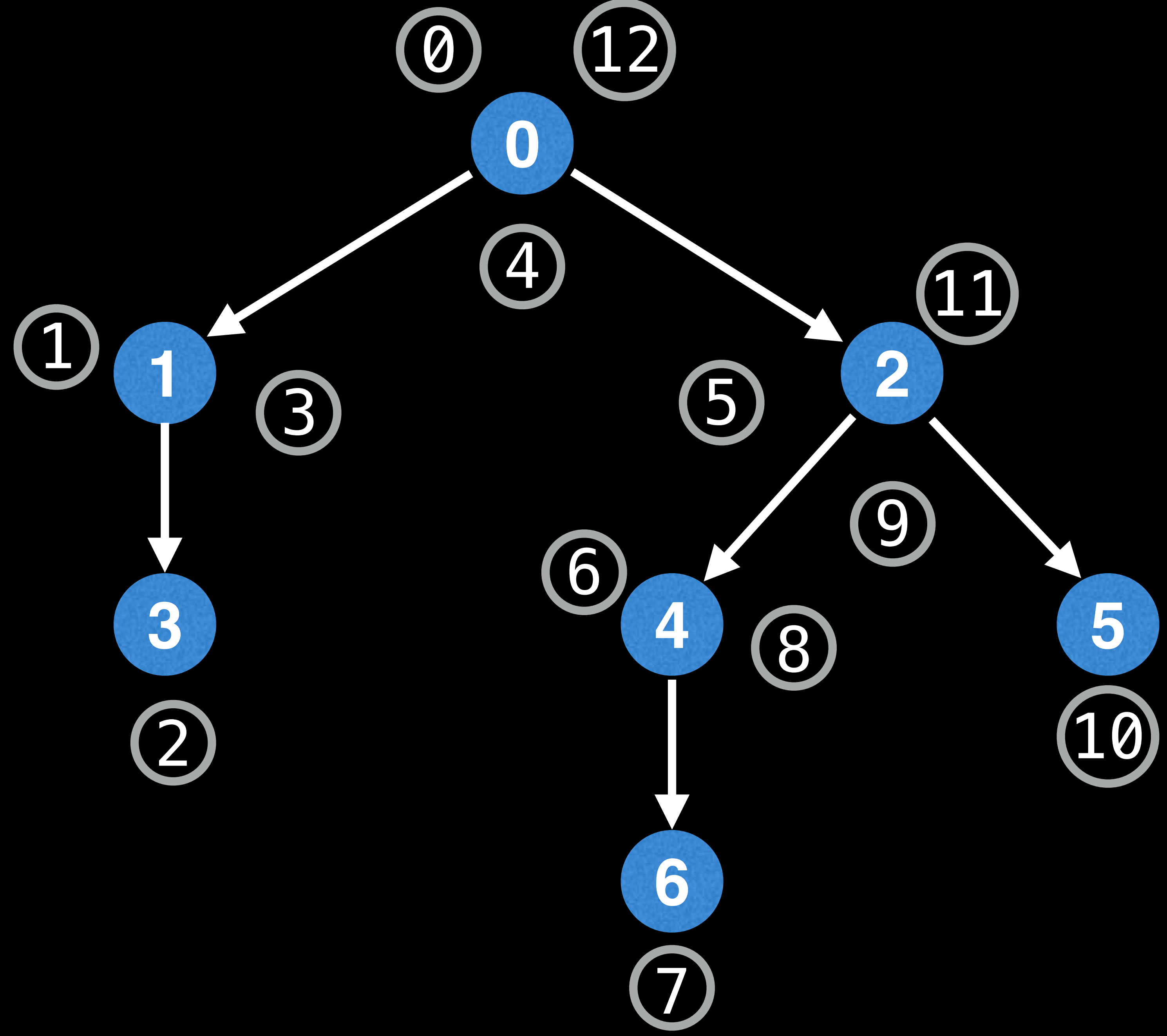

Similarly, the inverse mapping for node 2 could map to either index 5, 9 or 11 in the Euler tour.




So, which index values should we pick if we wanted to find the LCA of the nodes 1 and 2?

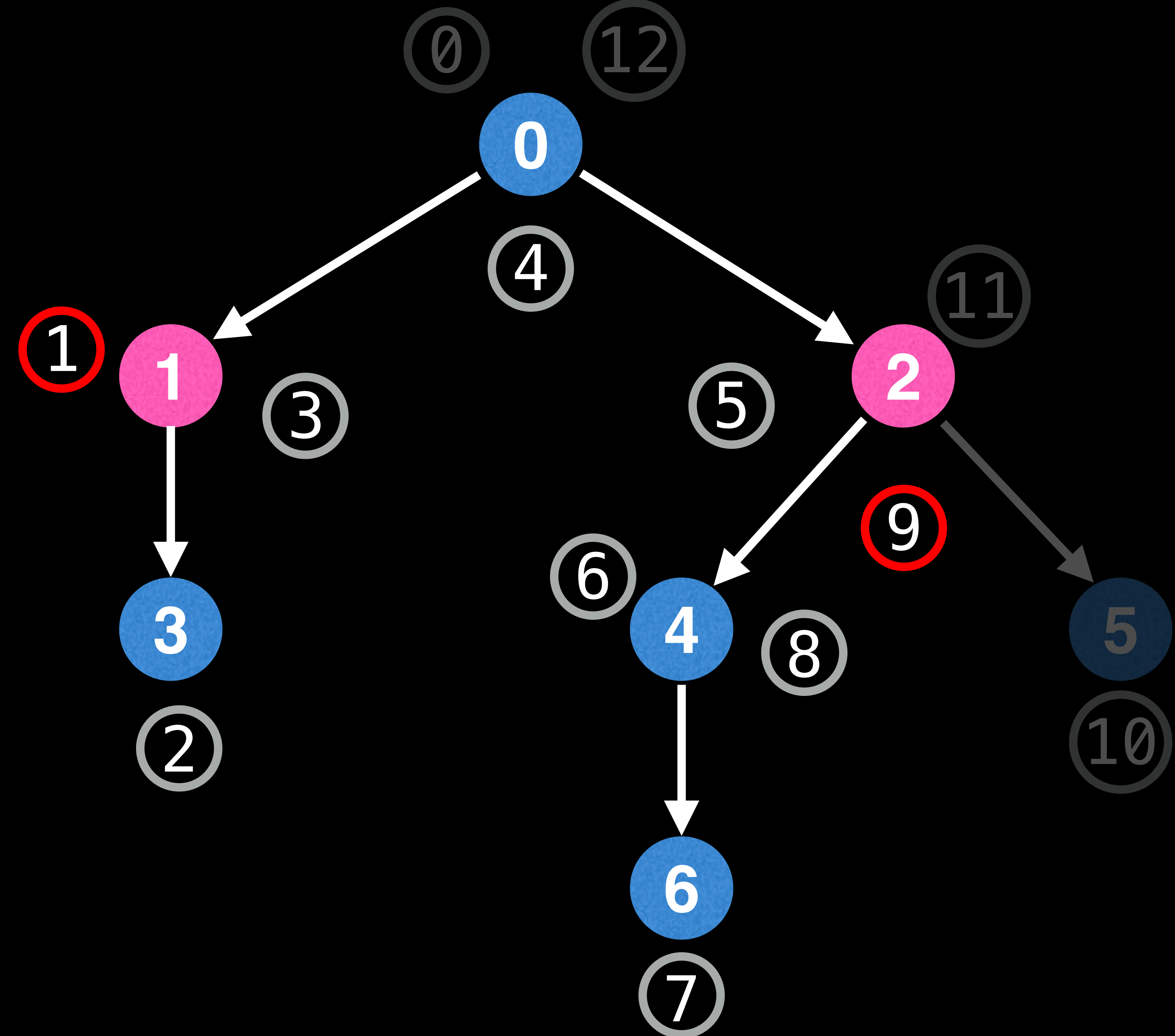



The answer is that it doesn't matter, any of the inverse index values will do. However, in practice, I find that it is easiest to select the **last encountered index** while doing the Euler tour.



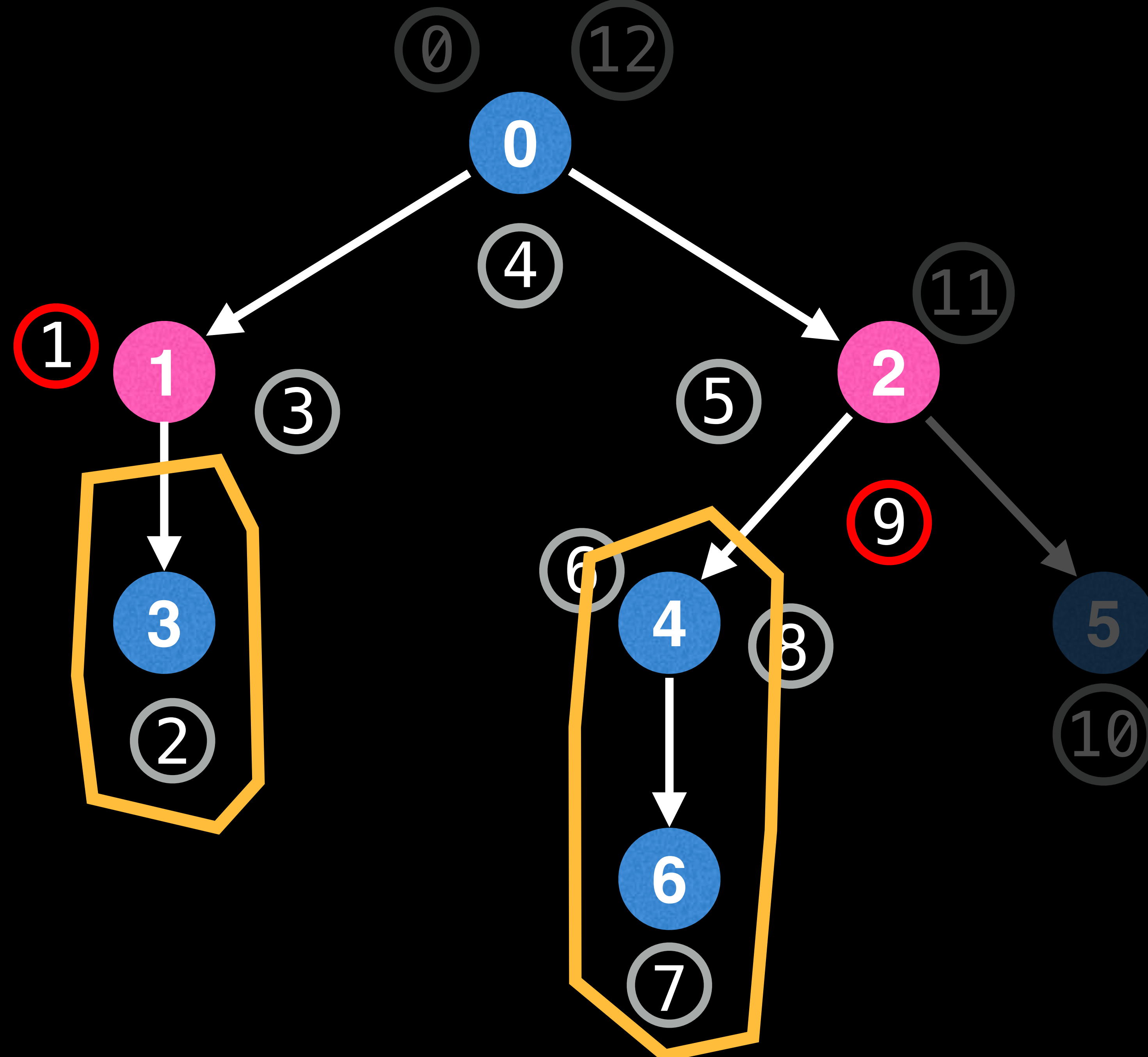

The reason the selection of the inverse index mapping doesn't matter is that it does not affect the value obtained from the **Range Minimum Query (RMQ)** in step 2

| | 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 |



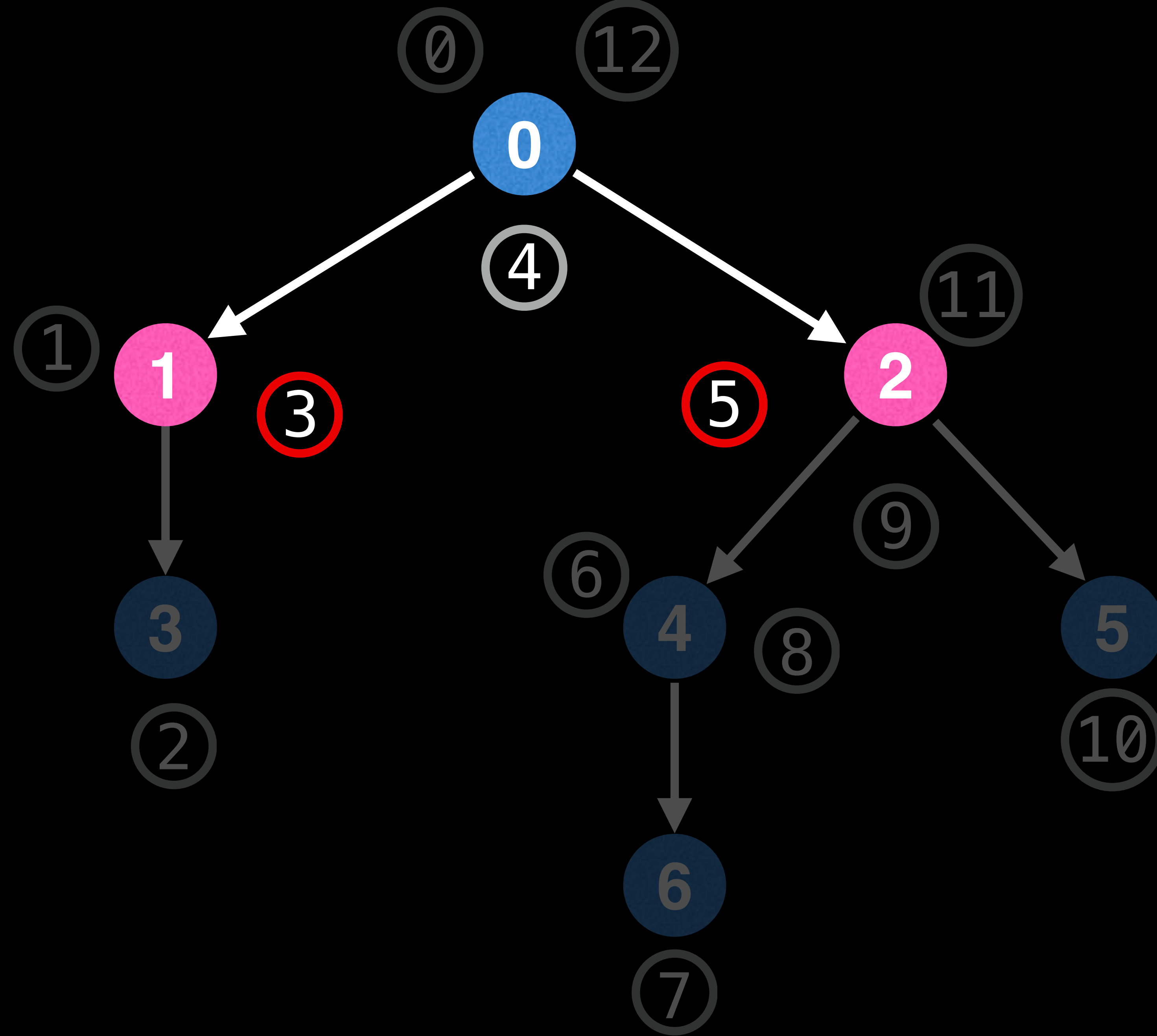

Suppose that for the LCA(1, 2) we selected index 1 for node 1 and index 9 for node 2, meaning the range [1, 9] in the depth array for the RMQ.

| | 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 |



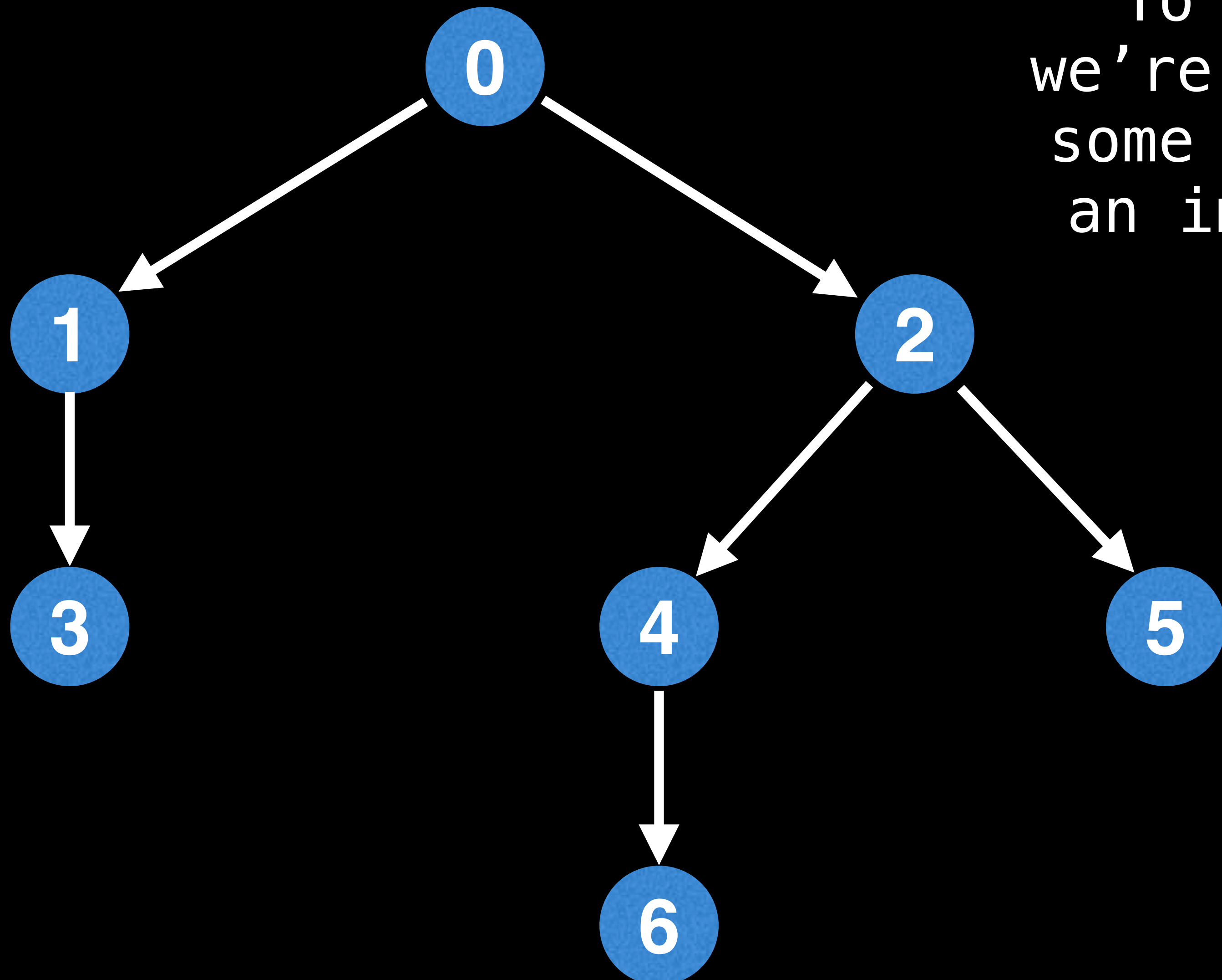

Even though the range [1, 9] includes some subtrees of the nodes 1 and 2, the depths of the subtree nodes are always more than the depths of nodes 1 and 2, so the value of the RMQ remains unchanged.

| | 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 |




You may think that choosing the index values 3 and 5 for nodes 1 and 2 would be better choice since the interval [3, 5] is smaller. However, this doesn't matter since RMQs take **O(1)** when using a sparse table.

| | 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 |

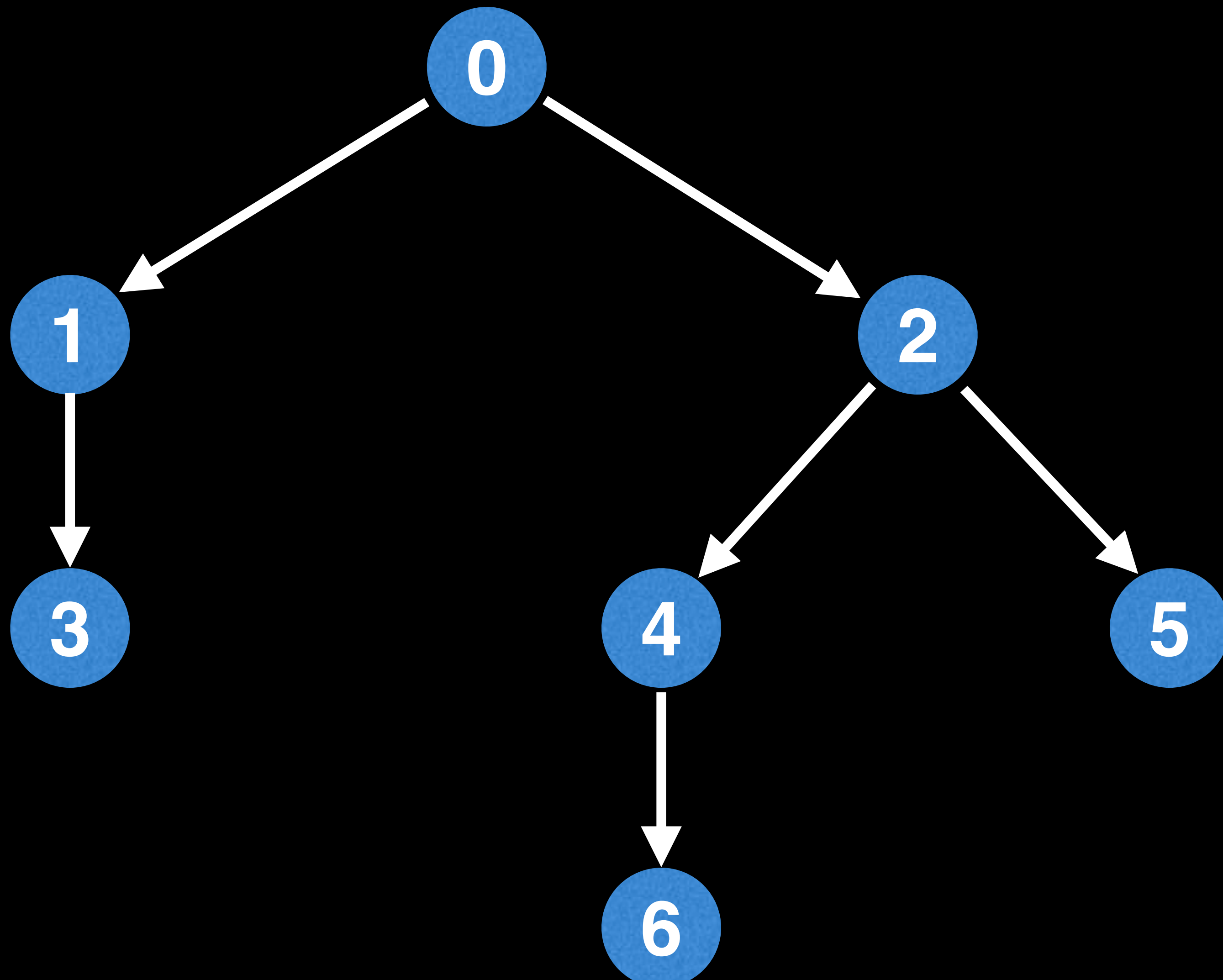



To maintain an inverse mapping, we're going to need to keep track of some additional information, namely an inverse map I will call `last`.

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| last | | | | | | | |

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

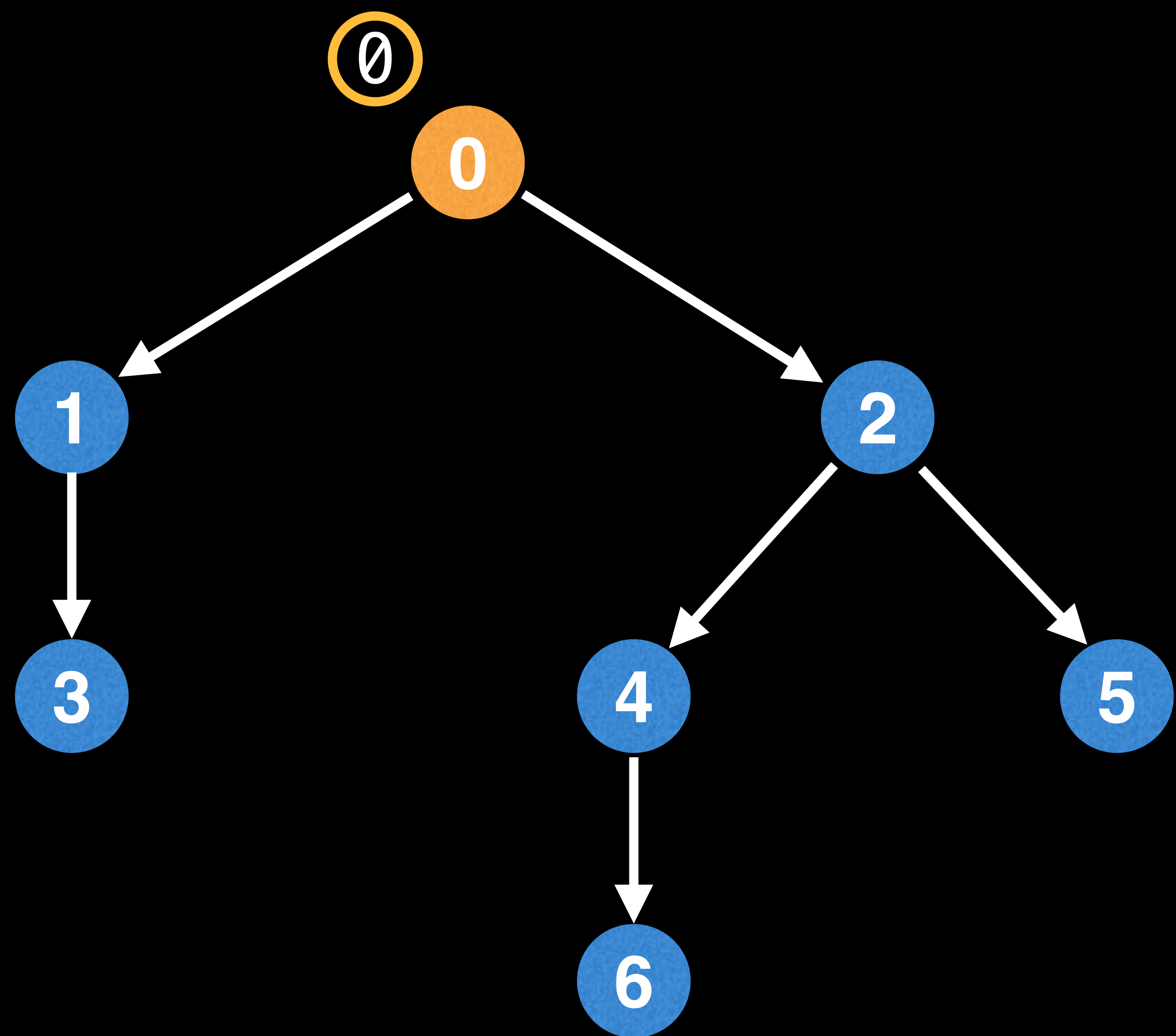
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| nodes | | | | | | | | | | | | | |




| | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| last | | | | | | | |

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

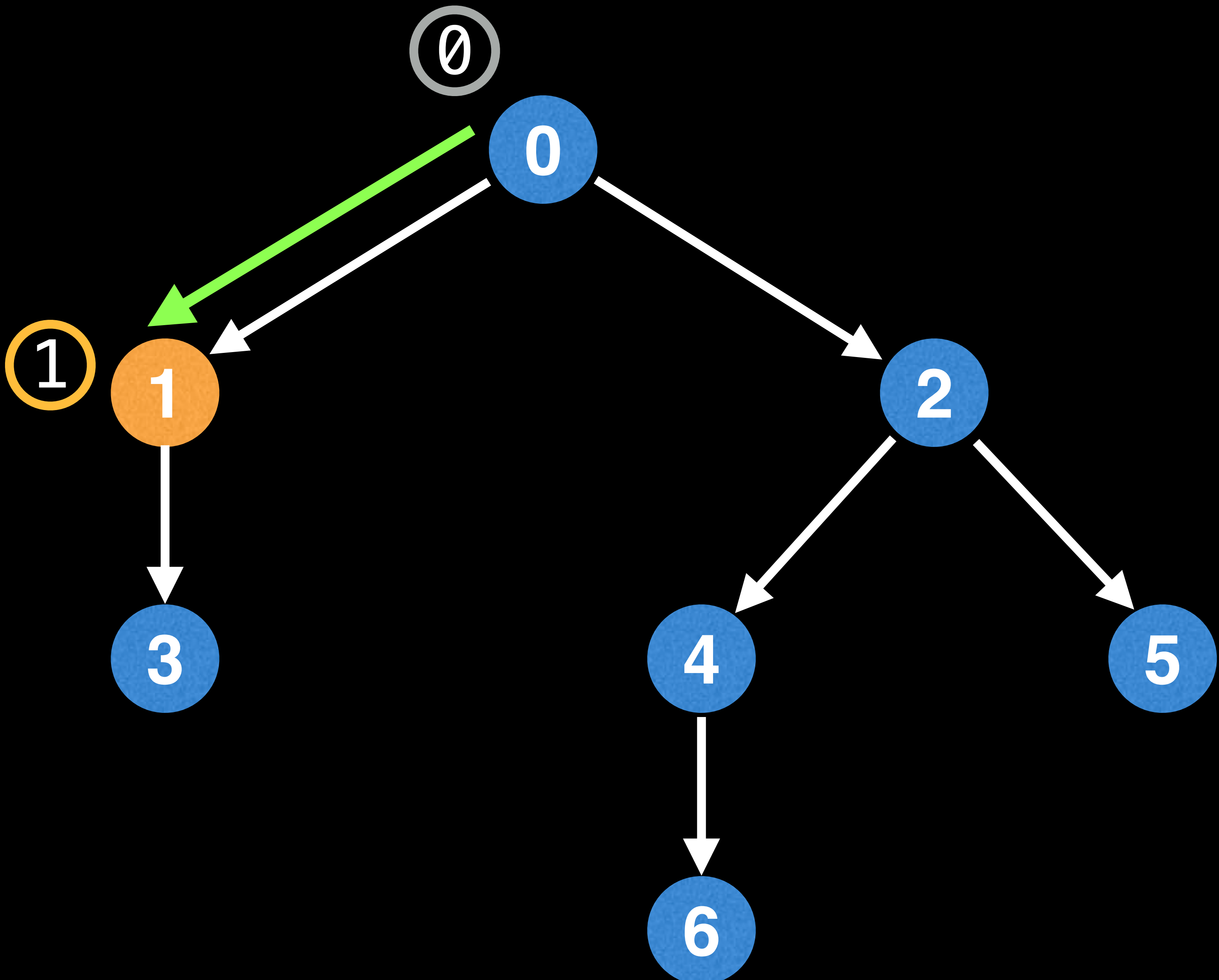
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| nodes | | | | | | | | | | | | | |




| | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| last | 0 | | | | | | |

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

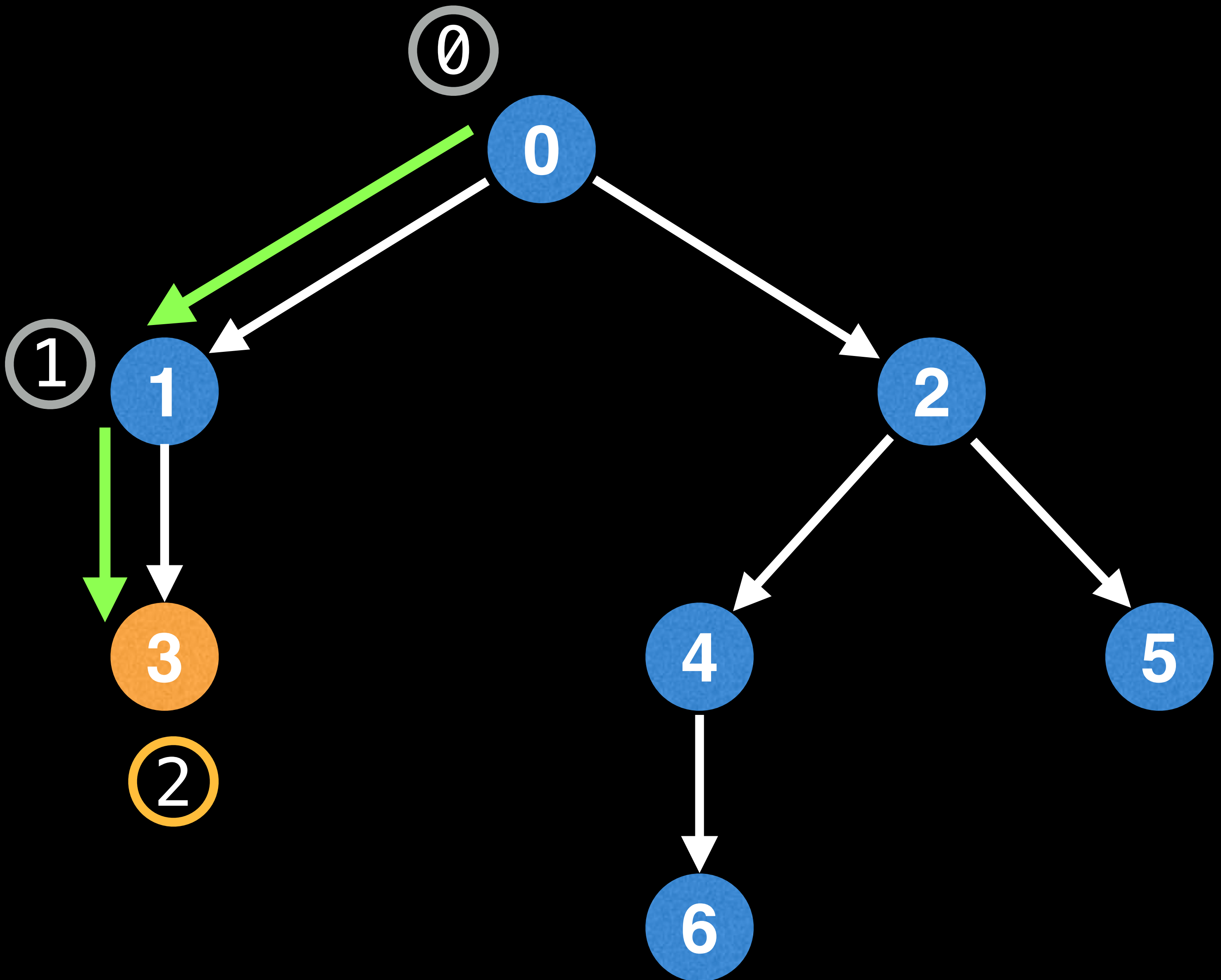
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| nodes | 0 | | | | | | | | | | | | |




| | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| last | 0 | 1 | | | | | |

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

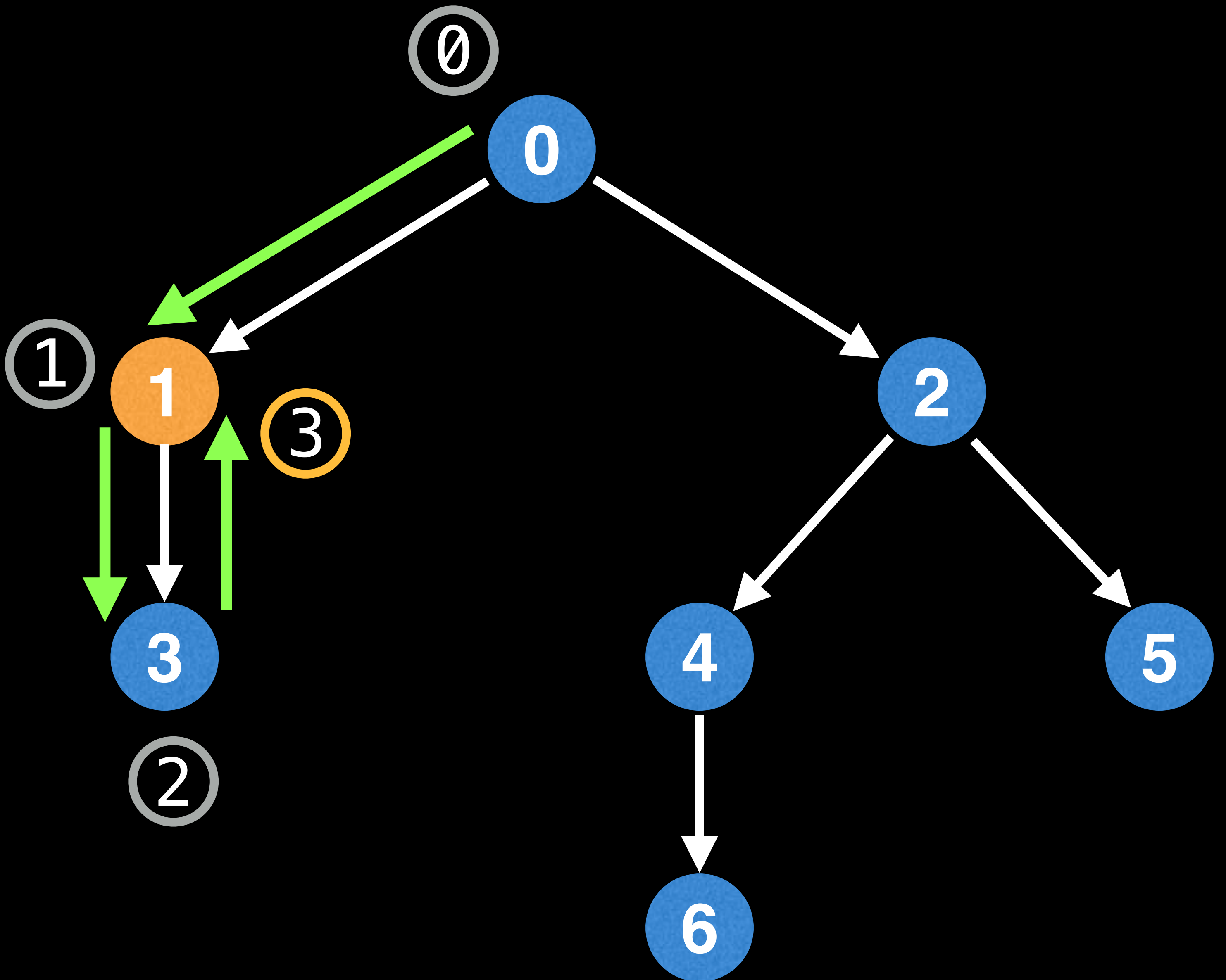
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| nodes | 0 | 1 | | | | | | | | | | | |




| | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| last | 0 | 1 | | 2 | | | |

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

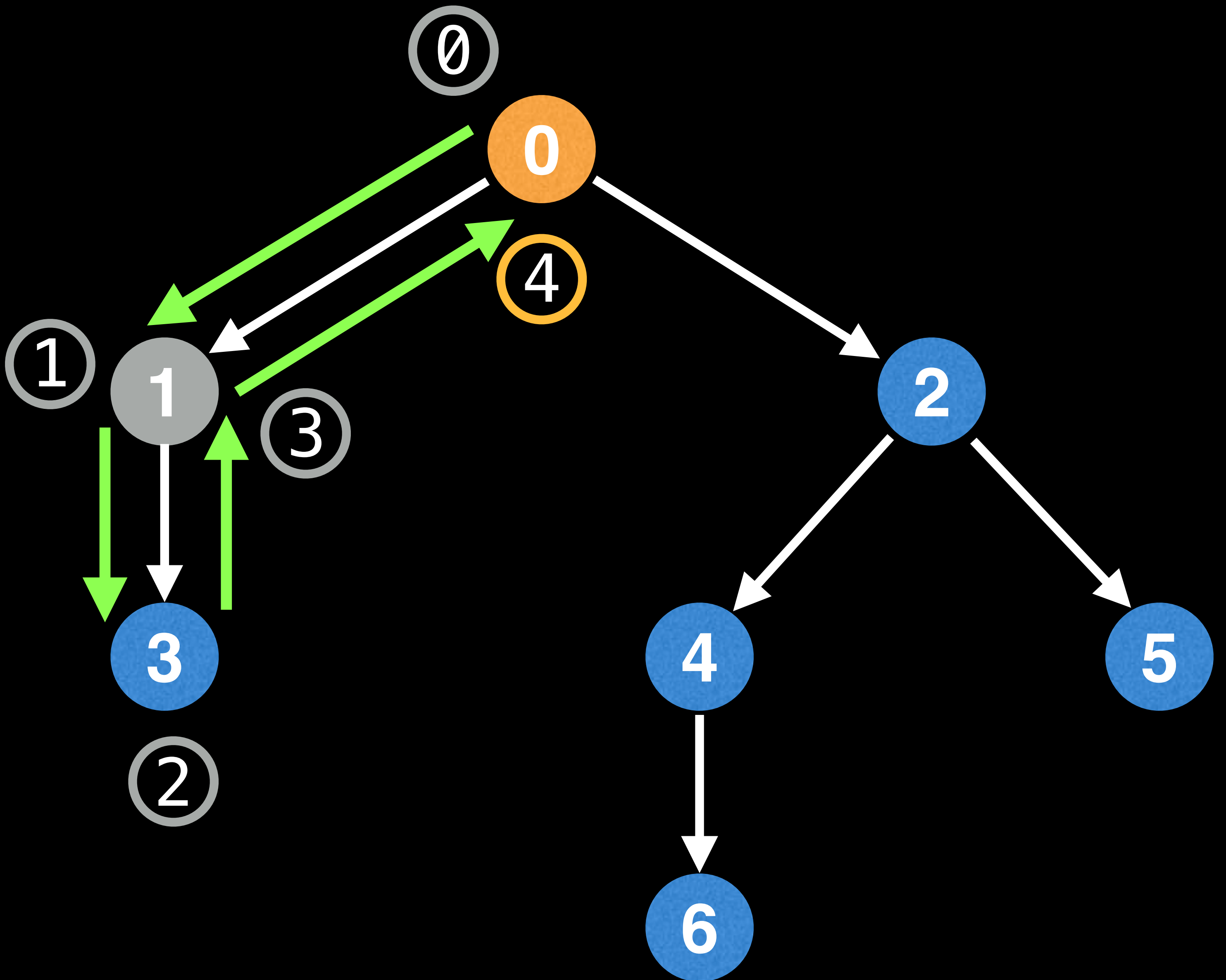
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| nodes | 0 | 1 | 3 | | | | | | | | | | |




| | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| last | 0 | 3 | | 2 | | | |

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

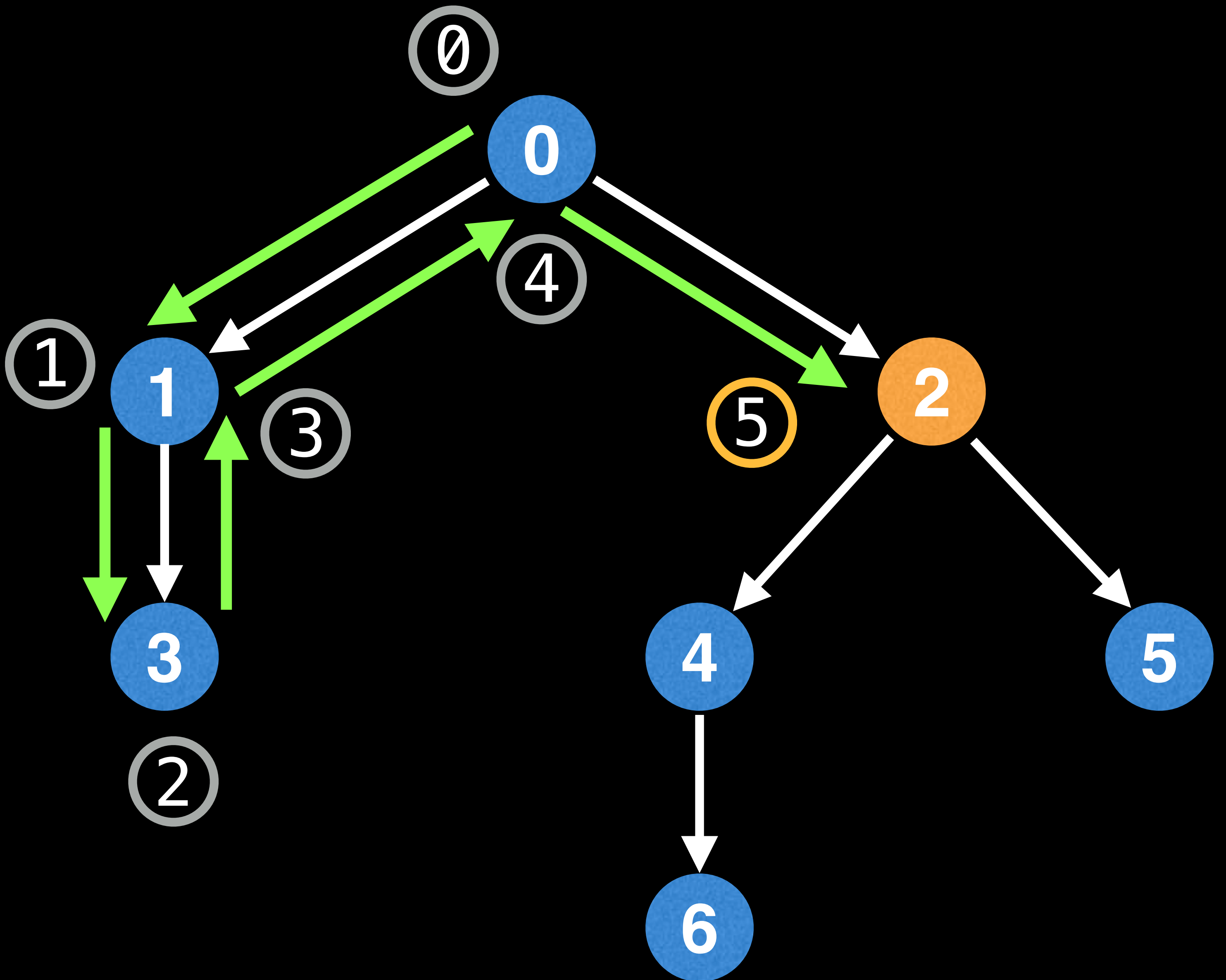
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| nodes | 0 | 1 | 3 | 1 | | | | | | | | | |




| | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| last | 4 | 3 | | 2 | | | |

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

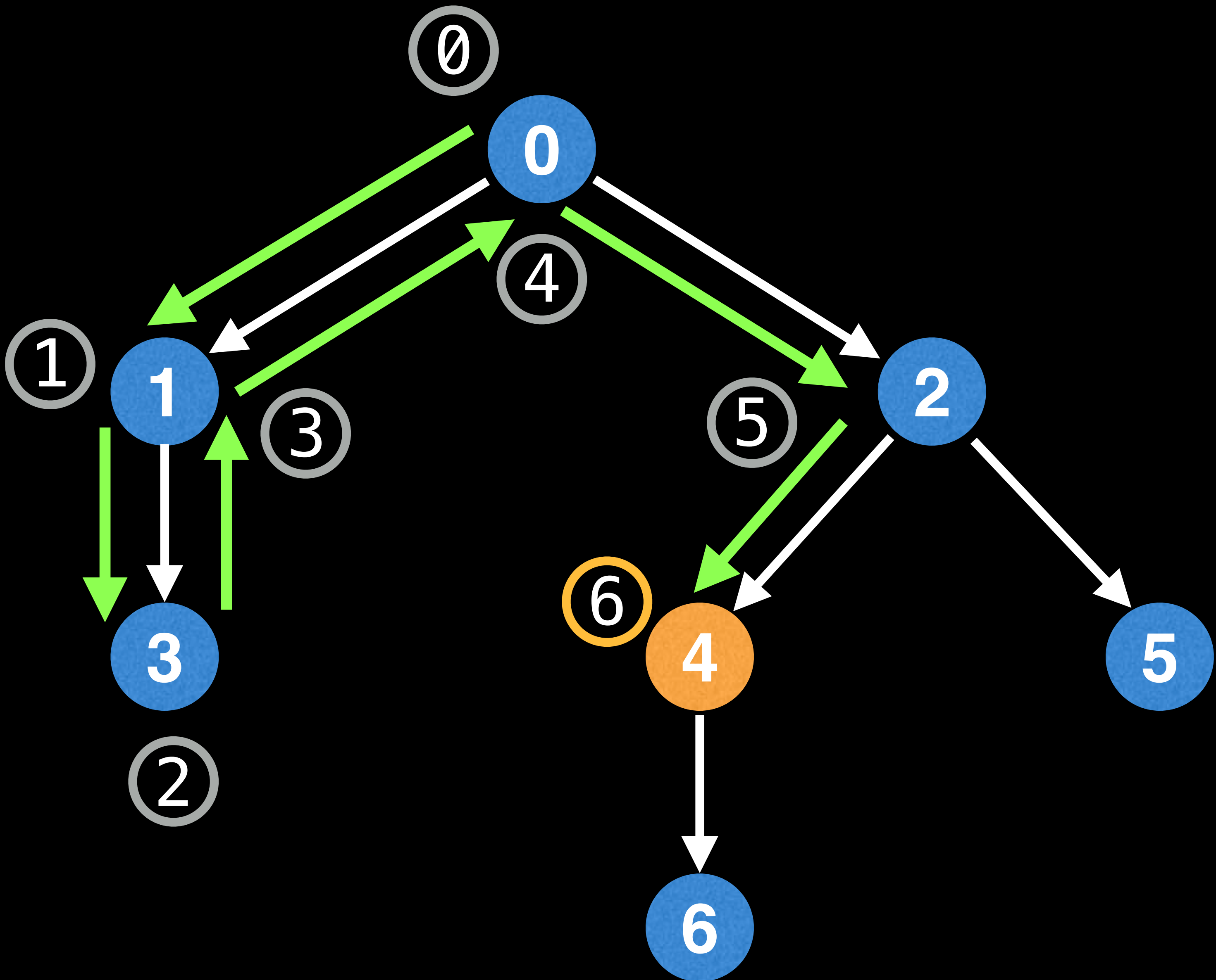
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| nodes | 0 | 1 | 3 | 1 | 0 | | | | | | | | |




| | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| last | 4 | 3 | 5 | 2 | | | |

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

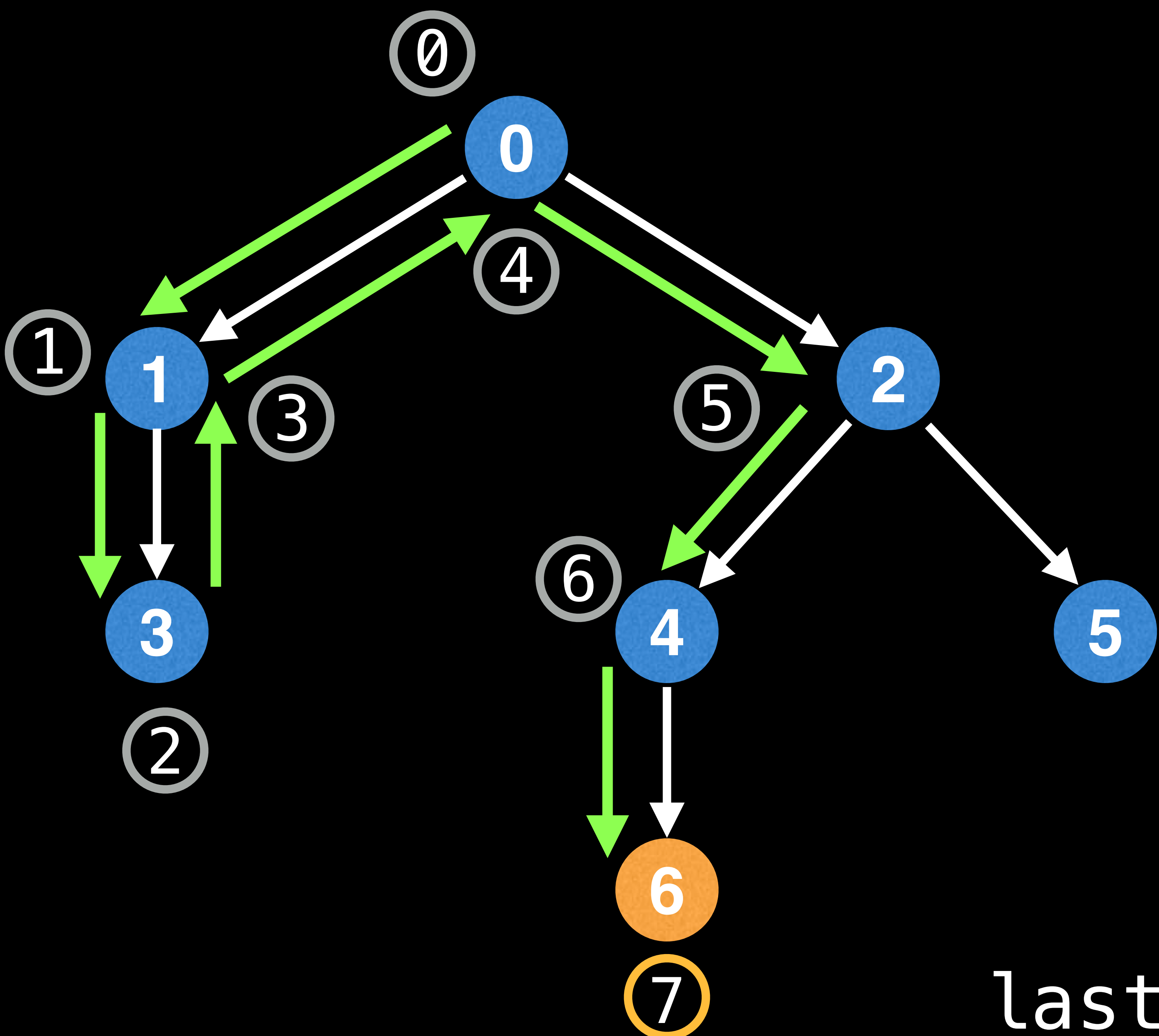
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| nodes | 0 | 1 | 3 | 1 | 0 | 2 | | | | | | | |




| | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| last | 4 | 3 | 5 | 2 | 6 | | |

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

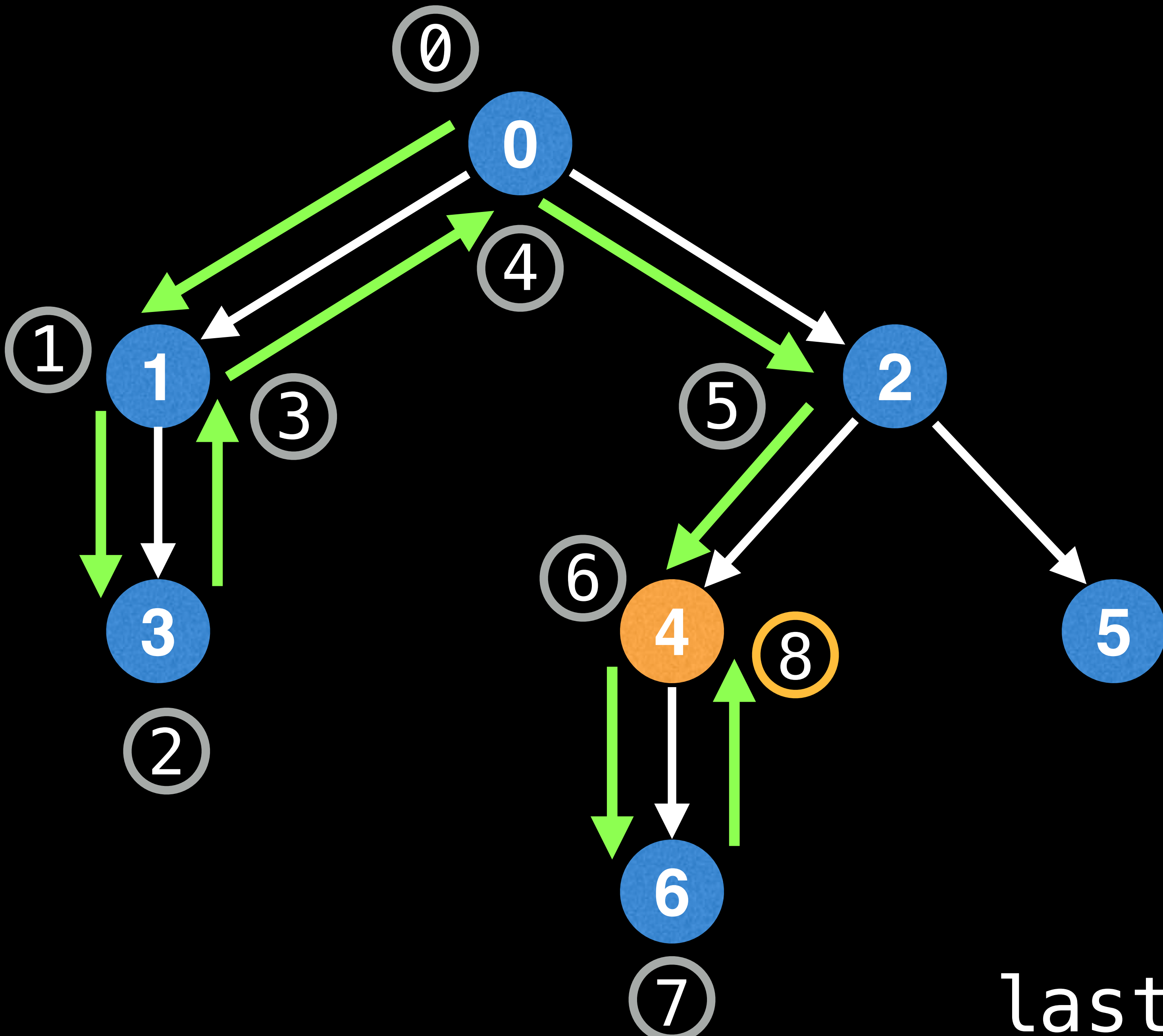
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| nodes | 0 | 1 | 3 | 1 | 0 | 2 | 4 | | | | | | |




| | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| last | 4 | 3 | 5 | 2 | 6 | | 7 |

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

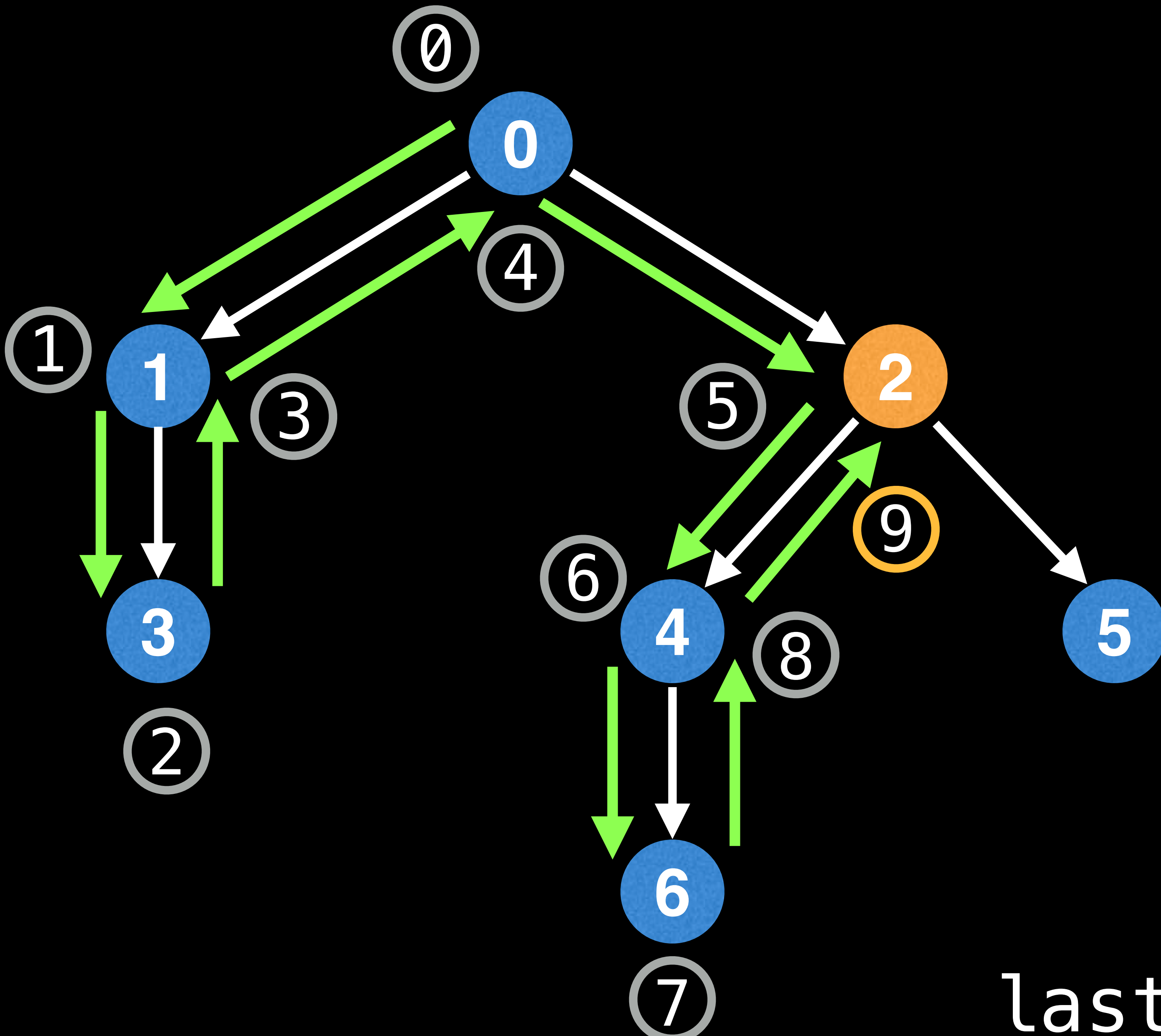
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| nodes | 0 | 1 | 3 | 1 | 0 | 2 | 4 | 6 | | | | | |




| | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| last | 4 | 3 | 5 | 2 | 8 | | 7 |

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

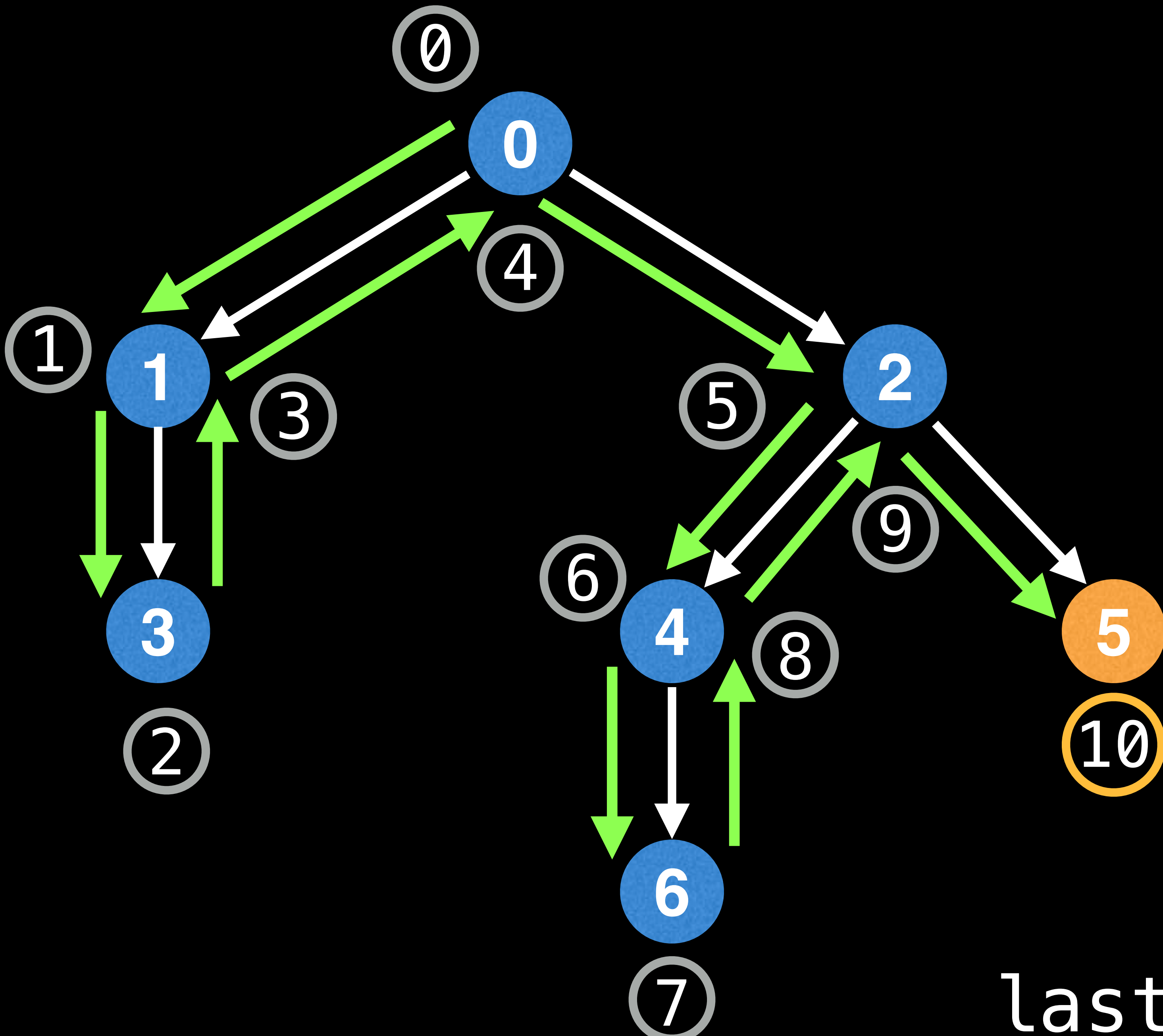
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| nodes | 0 | 1 | 3 | 1 | 0 | 2 | 4 | 6 | 4 | | | | |




| | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| last | 4 | 3 | 9 | 2 | 8 | | 7 |

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

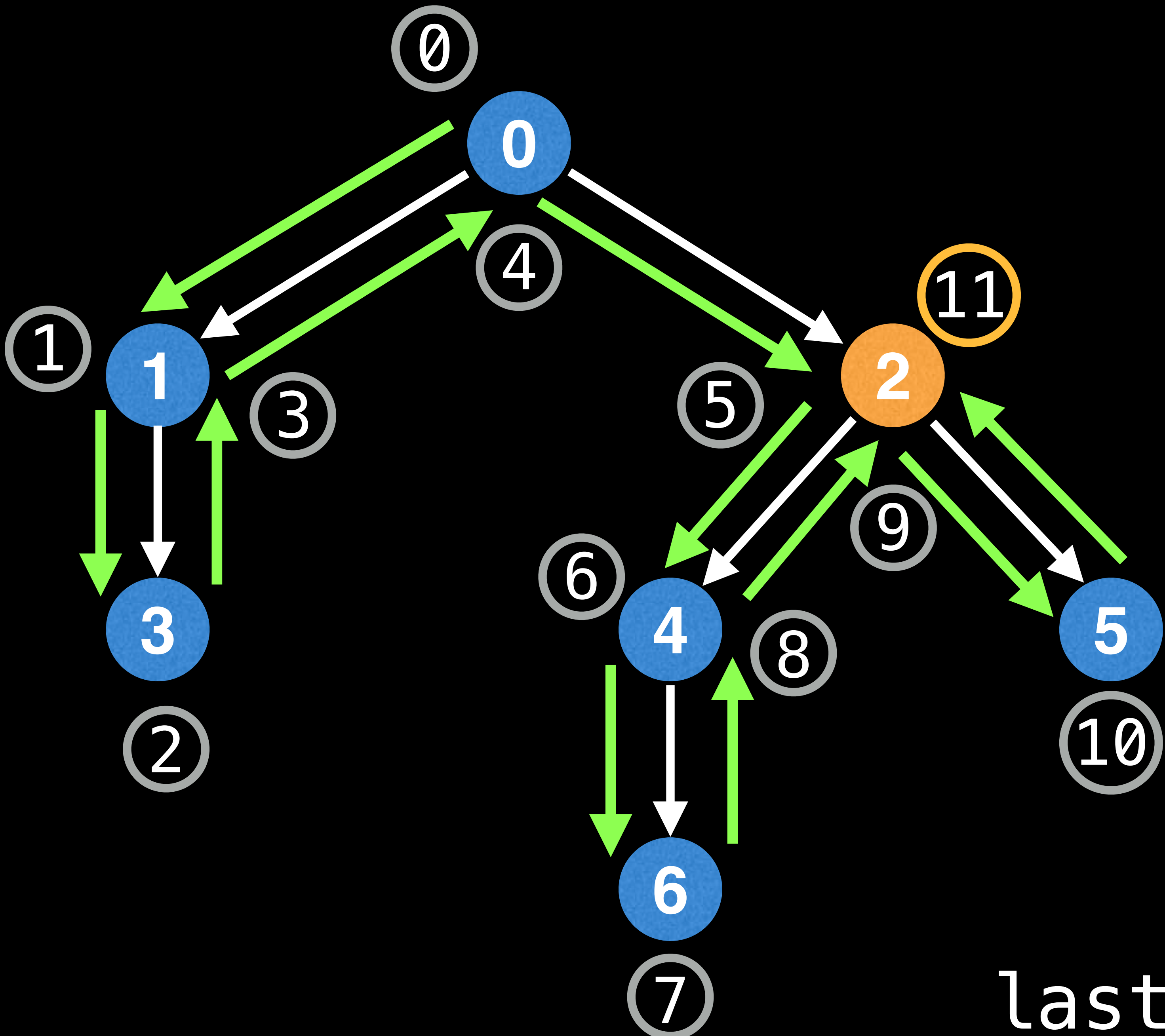
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| nodes | 0 | 1 | 3 | 1 | 0 | 2 | 4 | 6 | 4 | 2 | | | |




| | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| last | 4 | 3 | 9 | 2 | 8 | 10 | 7 |

| | 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 | | |

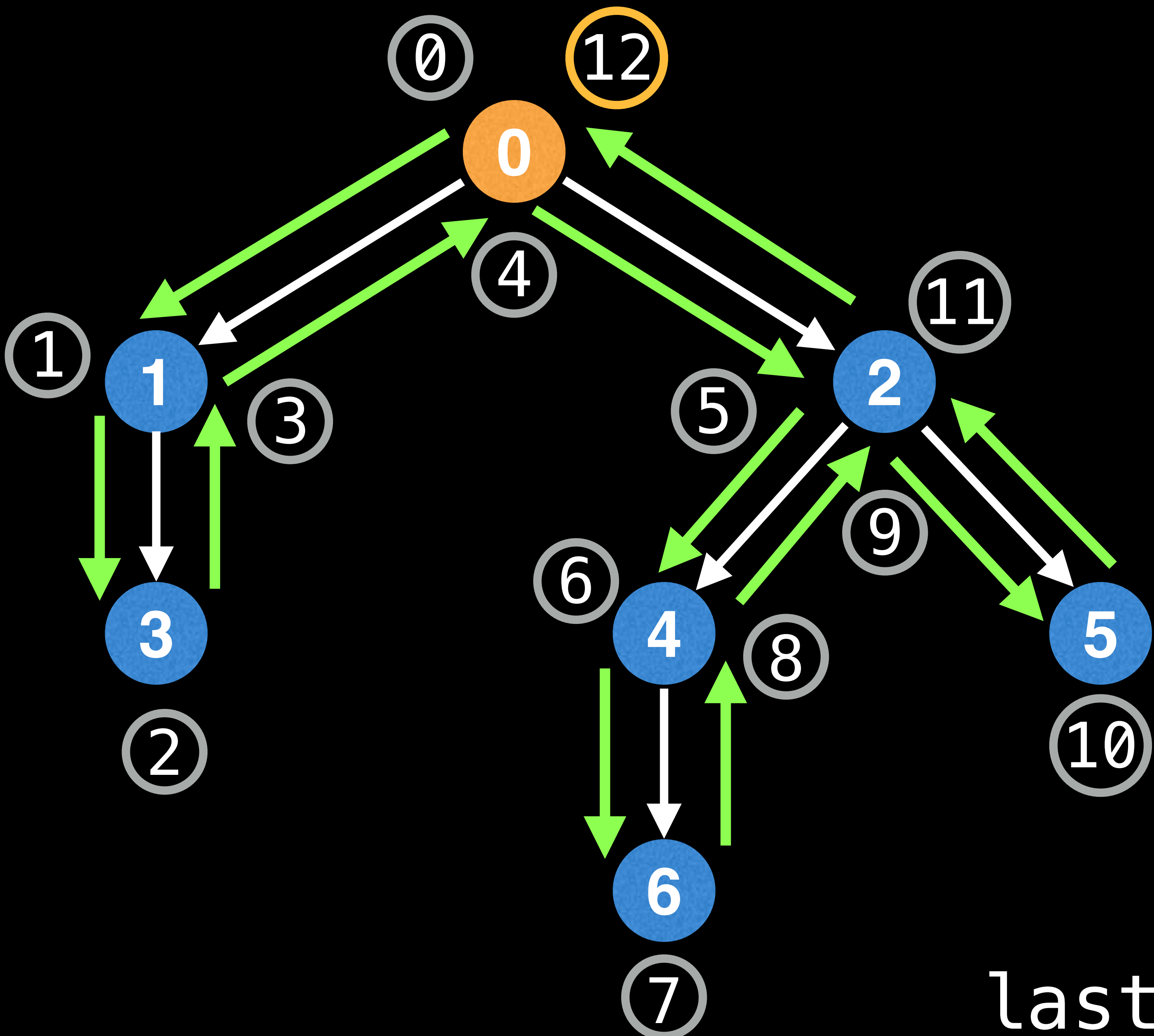
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| nodes | 0 | 1 | 3 | 1 | 0 | 2 | 4 | 6 | 4 | 2 | 5 | | |




| | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| last | 4 | 3 | 11 | 2 | 8 | 10 | 7 |

| | 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 | |

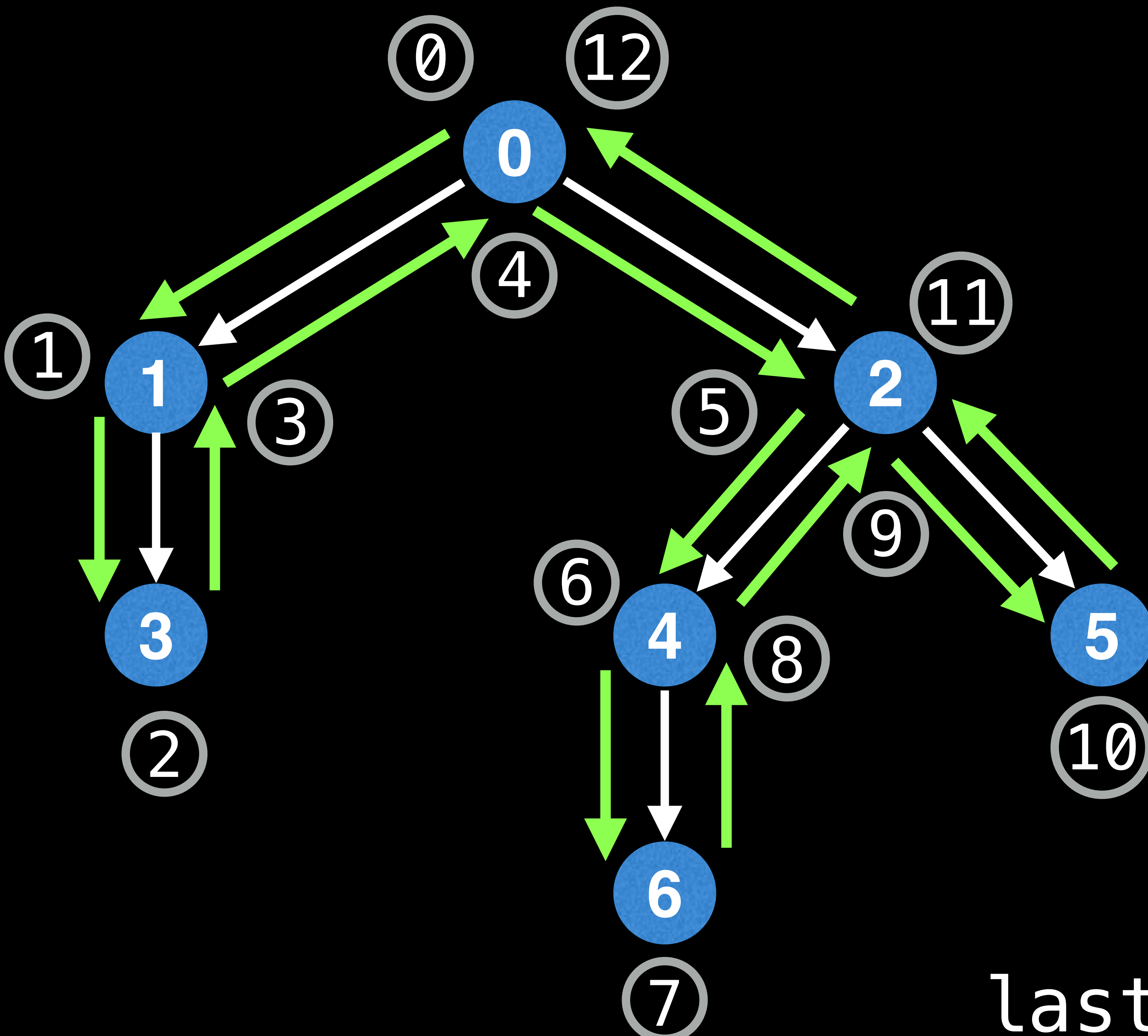
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| nodes | 0 | 1 | 3 | 1 | 0 | 2 | 4 | 6 | 4 | 2 | 5 | 2 | |




| | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| last | 12 | 3 | 11 | 2 | 8 | 10 | 7 |

| | 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 |

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




**last**

| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 12 | 3 | 11 | 2 | 8 | 10 | 7 |

**depth**

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 1 | 0 | 1 | 2 | 3 | 2 | 1 | 2 | 1 | 0 |

**nodes**

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 3 | 1 | 0 | 2 | 4 | 6 | 4 | 2 | 5 | 2 | 0 |

```
class TreeNode:

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

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

```
class TreeNode:

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

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

```
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)
```

```
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)
```

```
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)
```

```
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)
```

```
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)
```

```
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
```

```
# 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
```

```
# 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
```

```
# 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
```

```
# 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
```

```
# 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
```

```
# Query the Lowest Common Ancestor (LCA) of
# the two nodes with the indices `index1` and
# `index2`.
function lca(index1, index2):

  l = min(last[index1], last[index2])
  r = max(last[index1], last[index2])

  # Do RMQ to find the index of the minimum
  # element in the range [l, r]
  i = sparse_table.queryIndex(l, r)

  # Return the TreeNode object for the LCA
  return nodes[i]
```

```
# Query the Lowest Common Ancestor (LCA) of
# the two nodes with the indices `index1` and
# `index2`.
function lca(index1, index2):

  l = min(last[index1], last[index2])
  r = max(last[index1], last[index2])

  # Do RMQ to find the index of the minimum
  # element in the range [l, r]
  i = sparse_table.queryIndex(l, r)

  # Return the TreeNode object for the LCA
  return nodes[i]
```

```
# Query the Lowest Common Ancestor (LCA) of
# the two nodes with the indices `index1` and
# `index2`.
function lca(index1, index2):

  l = min(last[index1], last[index2])
  r = max(last[index1], last[index2])

  # Do RMQ to find the index of the minimum
  # element in the range [l, r]
  i = sparse_table.queryIndex(l, r)

  # Return the TreeNode object for the LCA
  return nodes[i]
```

```
# Query the Lowest Common Ancestor (LCA) of
# the two nodes with the indices `index1` and
# `index2`.
function lca(index1, index2):

  l = min(last[index1], last[index2])
  r = max(last[index1], last[index2])

  # Do RMQ to find the index of the minimum
  # element in the range [l, r]
  i = sparse_table.queryIndex(l, r)

  # Return the TreeNode object for the LCA
  return nodes[i]
```

```
# Query the Lowest Common Ancestor (LCA) of
# the two nodes with the indices `index1` and
# `index2`.
function lca(index1, index2):

  l = min(last[index1], last[index2])
  r = max(last[index1], last[index2])

  # Do RMQ to find the index of the minimum
  # element in the range [l, r]
  i = sparse_table.queryIndex(l, r)

  # Return the TreeNode object for the LCA
  return nodes[i]
```

Unused slides follow

# Popular LCA methods on static trees

1. Find the **Eulerian Tour** of a rooted tree, and subsequently do Range Minimum Queries to find the LCA. Requires **O(nlogn)** preprocessing with a Sparse Table, and gives **O(1)** LCA queries

# Popular LCA methods on static trees

1. Find the **Eulerian Tour** of a rooted tree, and subsequently do Range Minimum Queries to find the LCA. Requires **O(nlogn)** preprocessing with a Sparse Table, and gives **O(1)** LCA queries

2. **Tarjan's offline LCA algorithm**. This algorithm uses a union find to find the LCA between two nodes, but requires all LCA queries to be specified in advance.

# Popular LCA methods on static trees

1. Find the **Eulerian Tour** of a rooted tree, and subsequently do Range Minimum Queries to find the LCA. Requires **O(nlogn)** preprocessing with a Sparse Table, and gives **O(1)** LCA queries

2. **Tarjan's offline LCA algorithm.** This algorithm uses a union find to find the LCA between two nodes, but requires all LCA queries to be specified in advance.

3. Use the **Heavy-Light Decomposition** technique to break a tree into disjoint chains, and use this structure to do LCA queries.

# Popular LCA methods on static trees

1. Find the **Eulerian Tour** of a rooted tree, and subsequently do Range Minimum Queries to find the LCA. Requires **O(nlogn)** preprocessing with a Sparse Table, and gives **O(1)** LCA queries

2. **Tarjan’s offline LCA algorithm.** This algorithm uses a union find to find the LCA between two nodes, but requires all LCA queries to be specified in advance.

3. Use the **Heavy-Light Decomposition** technique to break a tree into disjoint chains, and use this structure to do LCA queries.

4. **Farach-Colton and Bender technique** improves on the Euler Tour + RMQ solution by reducing the pre-processing time to **O(n)**.

# Popular LCA methods on static trees

1. Find the **Eulerian Tour** of a rooted tree, and subsequently do Range Minimum Queries to find the LCA. Requires **O(nlogn)** preprocessing with a Sparse Table, and gives **O(1)** LCA queries

2. **Tarjan’s offline LCA algorithm.** This algorithm uses a union find to find the LCA between two nodes, but requires all LCA queries to be specified in advance.

3. Use the **Heavy-Light Decomposition** technique to break a tree into disjoint chains, and use this structure to do LCA queries.

4. **Farach-Colton and Bender technique** improves on the Euler Tour + RMQ solution by reducing the pre-processing time to **O(n)**.

5. … and several more algorithms like **Binary Lifting** and the naive approach of walking up the tree.

# Popular LCA methods on static trees

1. Find the **Eulerian Tour** of a rooted tree, and subsequently do Range Minimum Queries to find the LCA. Requires **O(nlogn)** preprocessing with a Sparse Table, and gives **O(1)** LCA queries

2. **Tarjan's offline LCA algorithm.** This algorithm uses a union find to find the LCA between two nodes, but requires all LCA queries to be specified in advance.

3. Use the **Heavy-Light Decomposition** technique to break a tree into disjoint chains, and use this structure to do LCA queries.

4. **Farach-Colton and Bender technique** improves on the Euler Tour + RMQ solution by reducing the pre-processing time to **O(n)**.

5. … and several more algorithms like **Binary Lifting** and the naive approach of walking up the tree.

# Lowest Common Ancestor