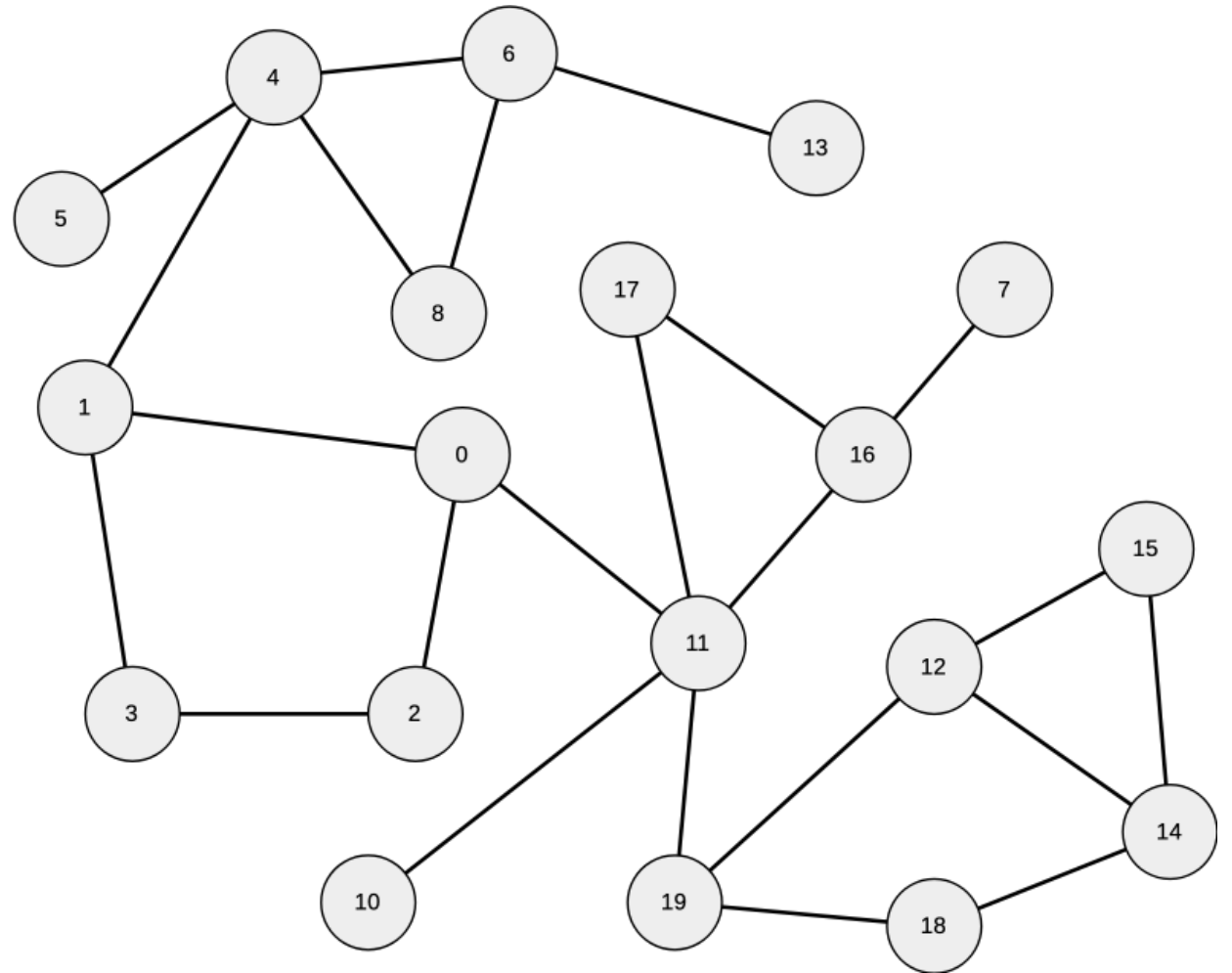


# Competitive Programming

Bridges and Articulation Points

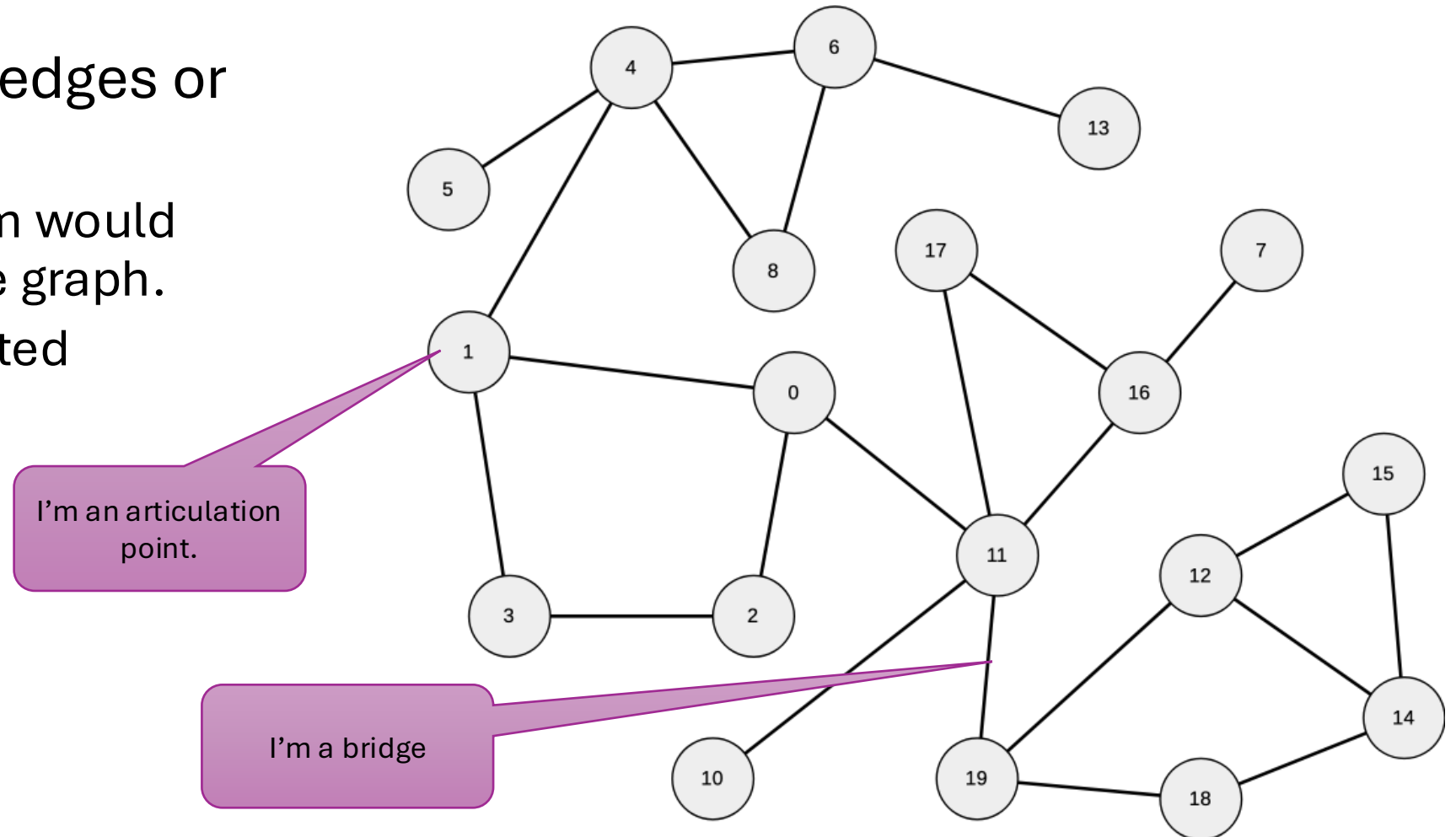
# Bridges and Articulation Points

- Identify critical edges or vertices.
  - Removing them would disconnect the graph.
  - For an undirected graph.



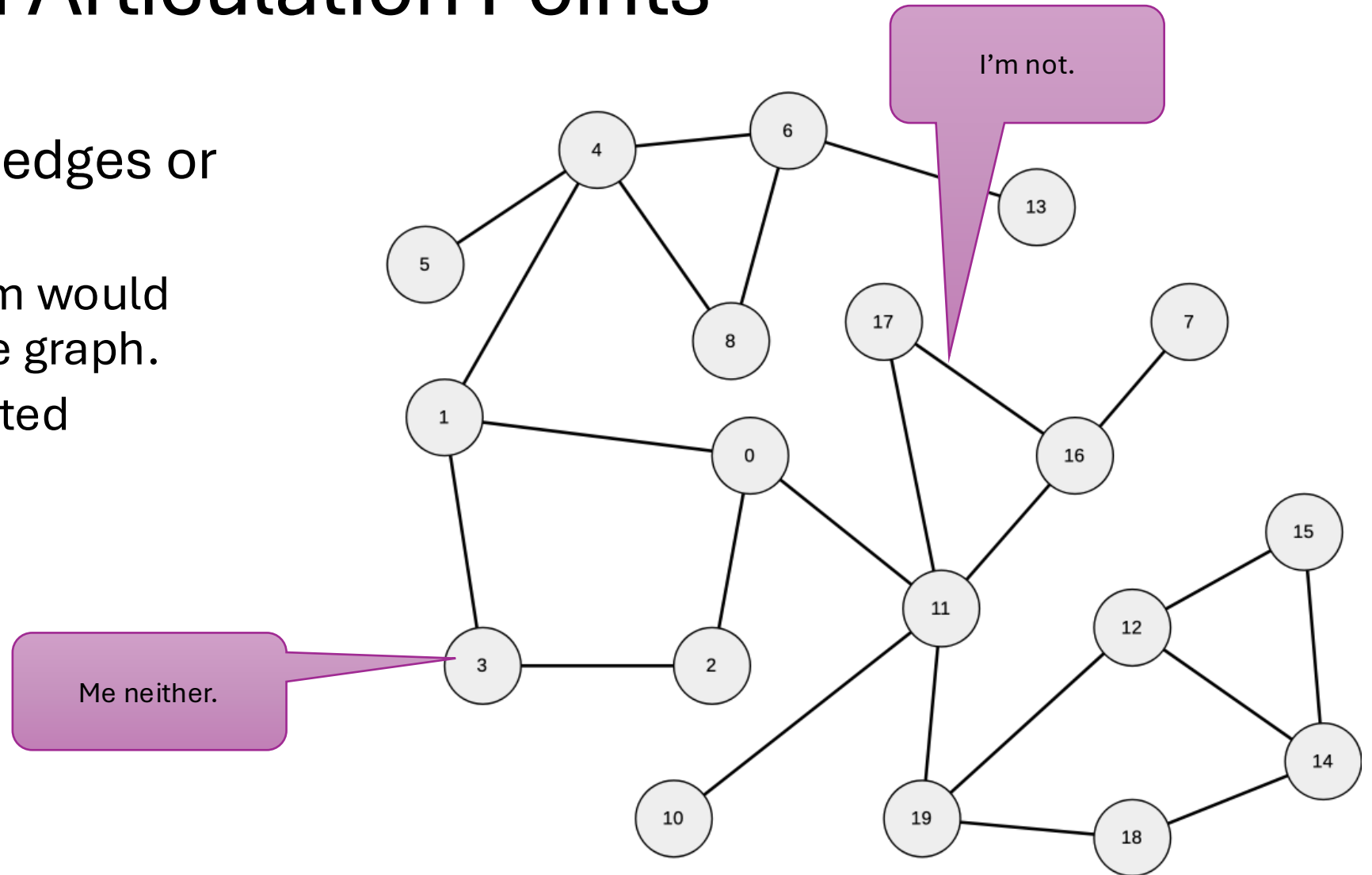
# Bridges and Articulation Points

- Identify critical edges or vertices.
  - Removing them would disconnect the graph.
  - For an undirected graph.



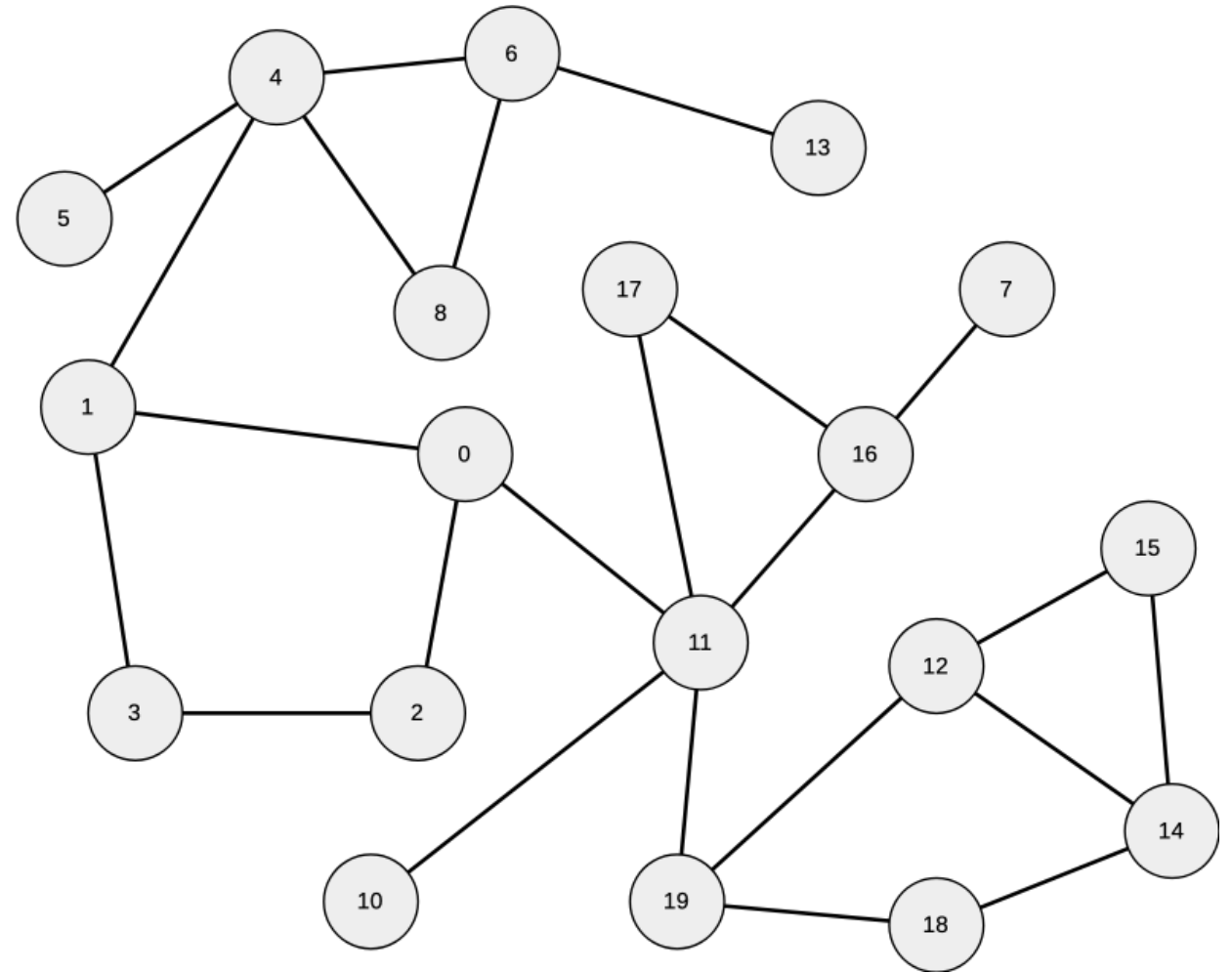
# Bridges and Articulation Points

- Identify critical edges or vertices.
  - Removing them would disconnect the graph.
  - For an undirected graph.



# Depth-First Traversal

- You can find these during a DFS.



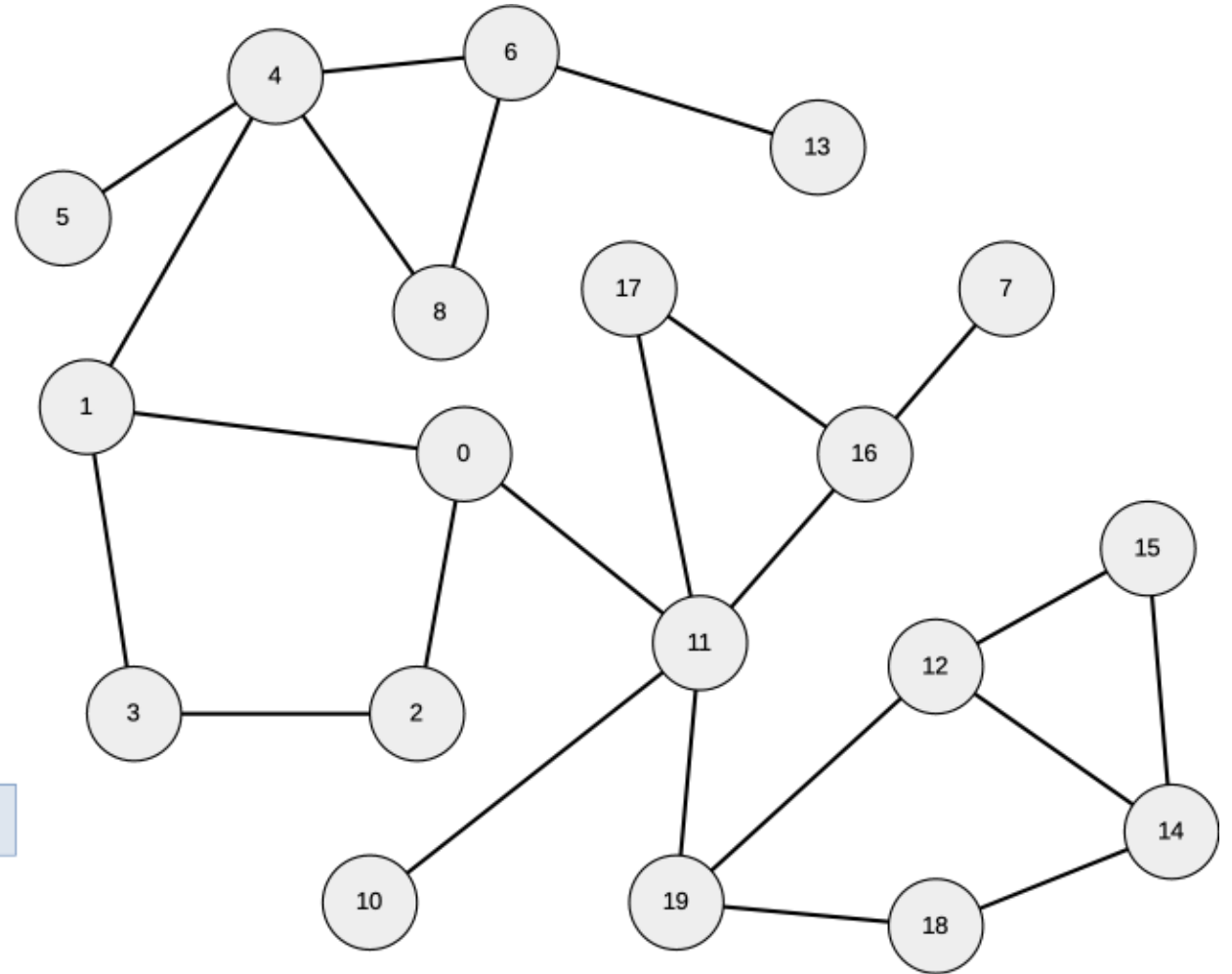
# Adjacency List Representation

- Adjacency List is your typical representation for a sparse graph.

A list of neighbors for every vertex.

Cost is linear in the total number of edges.

0	1	2	11		
1	0	3	4		
2	0	3			
3	1	2			
4	1	5	6	8	
5	4				
6	4	8	13		
7	16				
8	4	6			
9					
10	11				
11	0	10	16	17	19
12	14	15	19		
	•				
	•				
	•				



# Adjacency List Representation

- Easy to create and use in most common programming languages.

C++

```
int n;  
cin >> n;  
vector< vector< int > > edge( n );  
// add edges
```

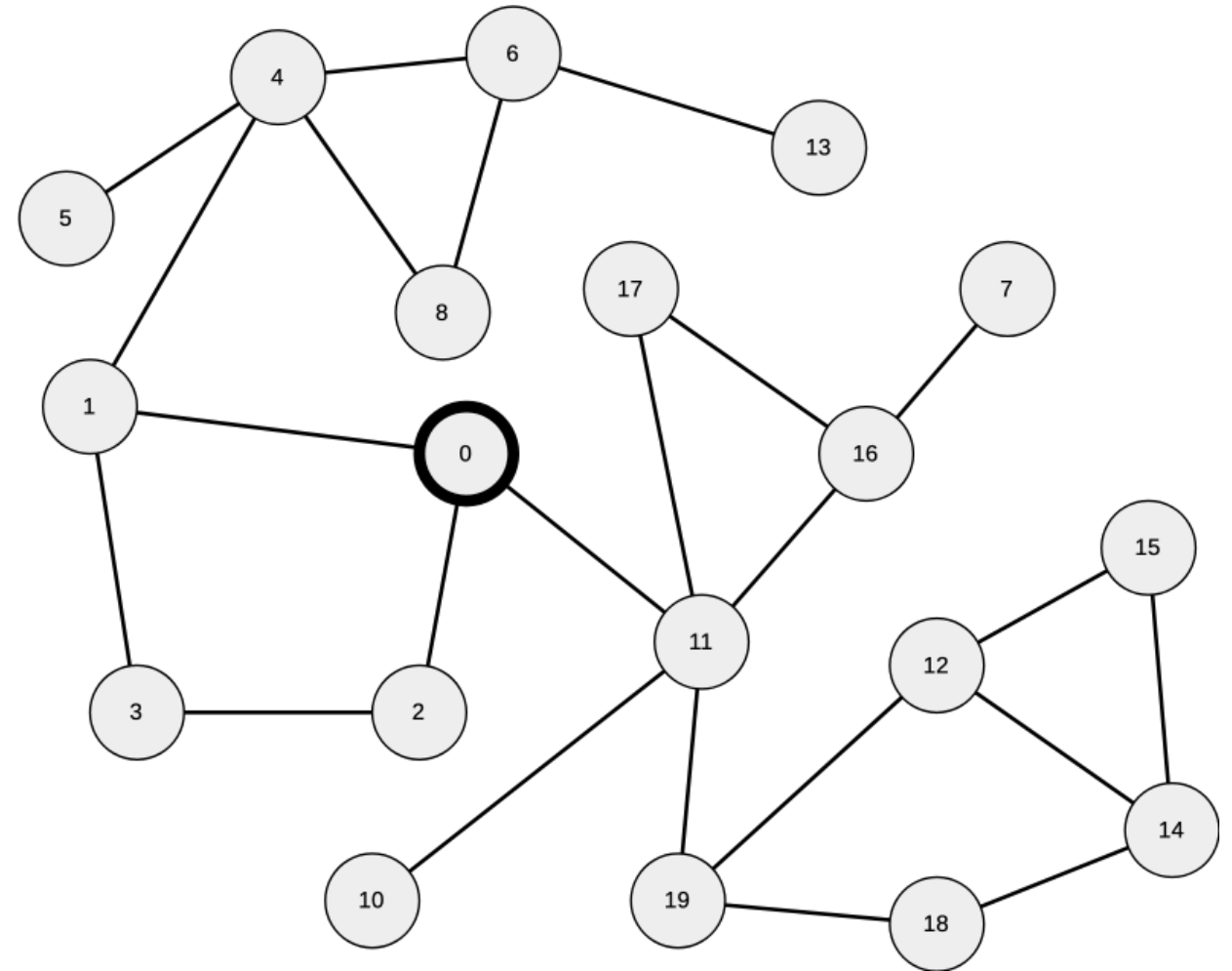
Python

```
n = int( input() )  
edge = [ [] for I in range( n ) ]  
// add edges
```

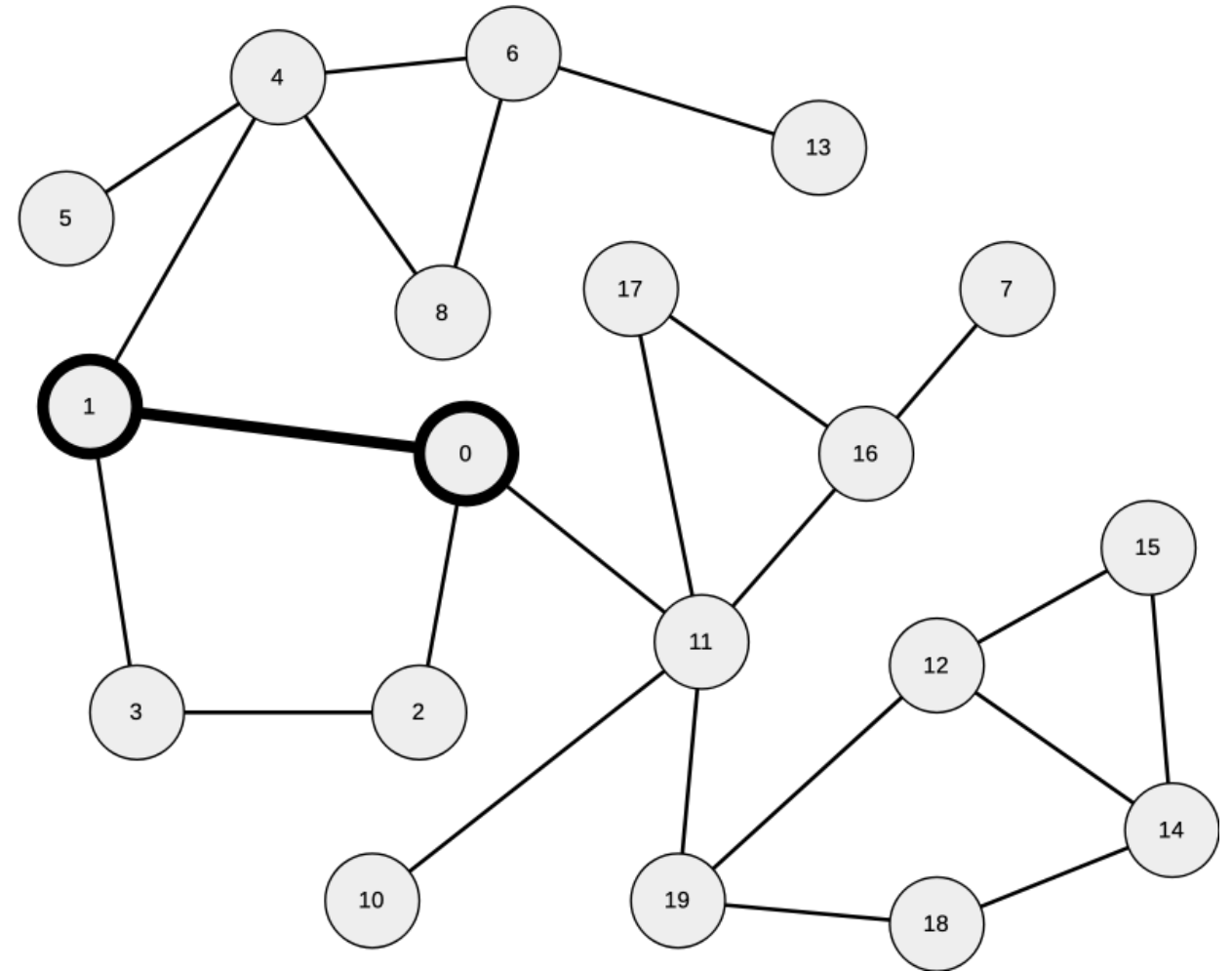
Java

```
int n = input.nextInt();  
ArrayList< ArrayList< Integer > > edge =  
    new ArrayList<>();  
for ( int i = 0; i < n; i++ ) {  
    edge.add( new ArrayList<>() );  
    // add edges for i.  
}
```

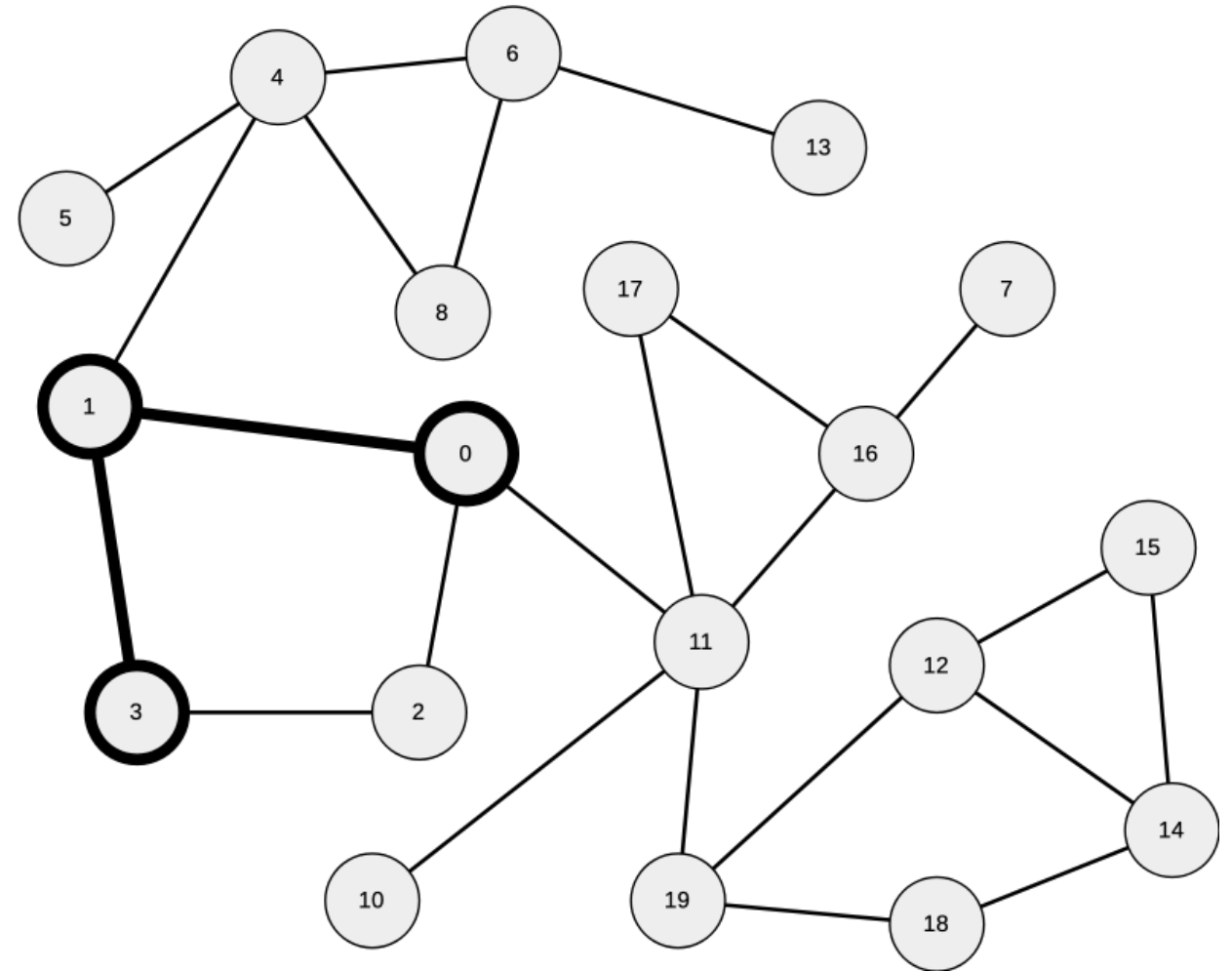
# Depth-First Traversal



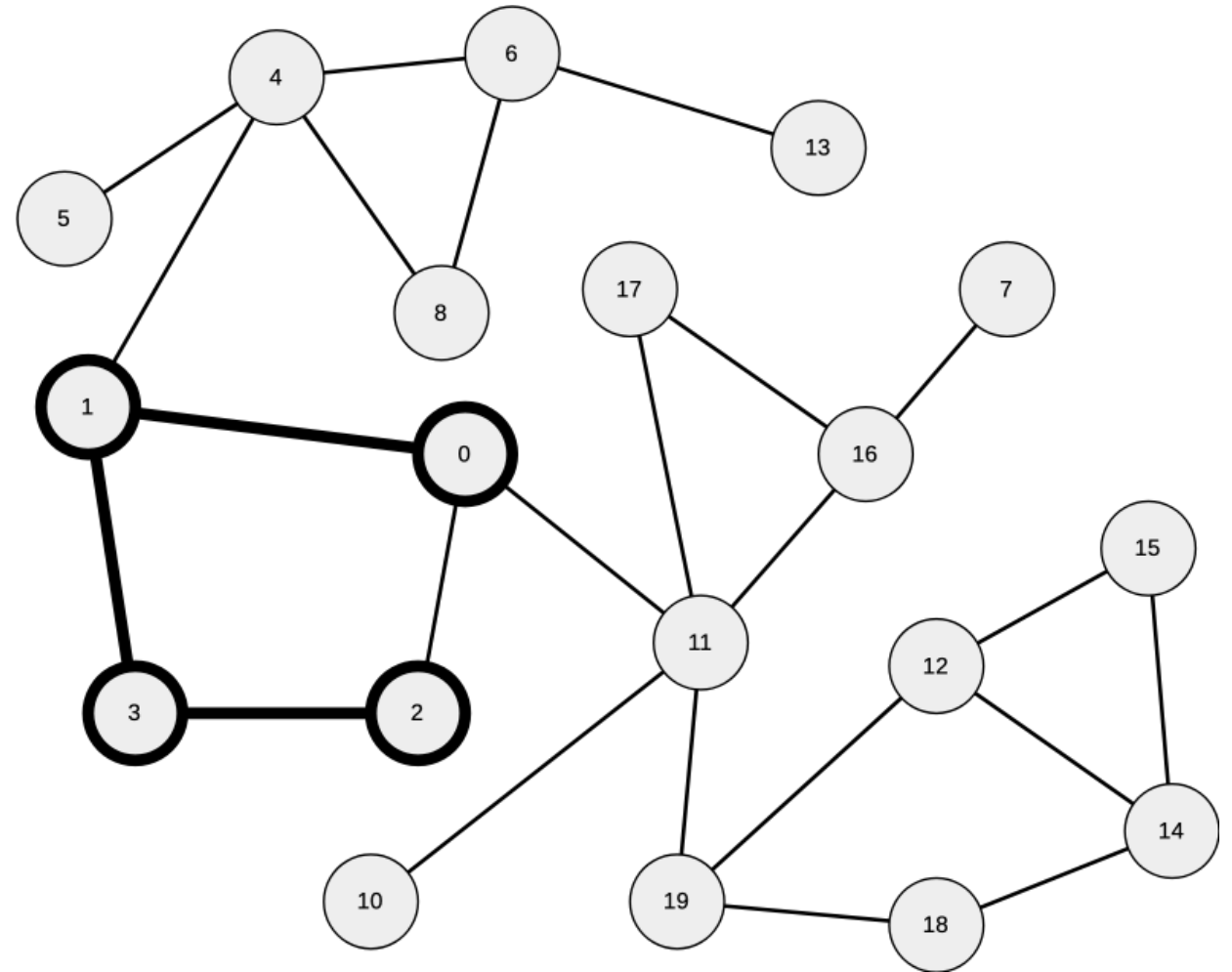
# Depth-First Traversal



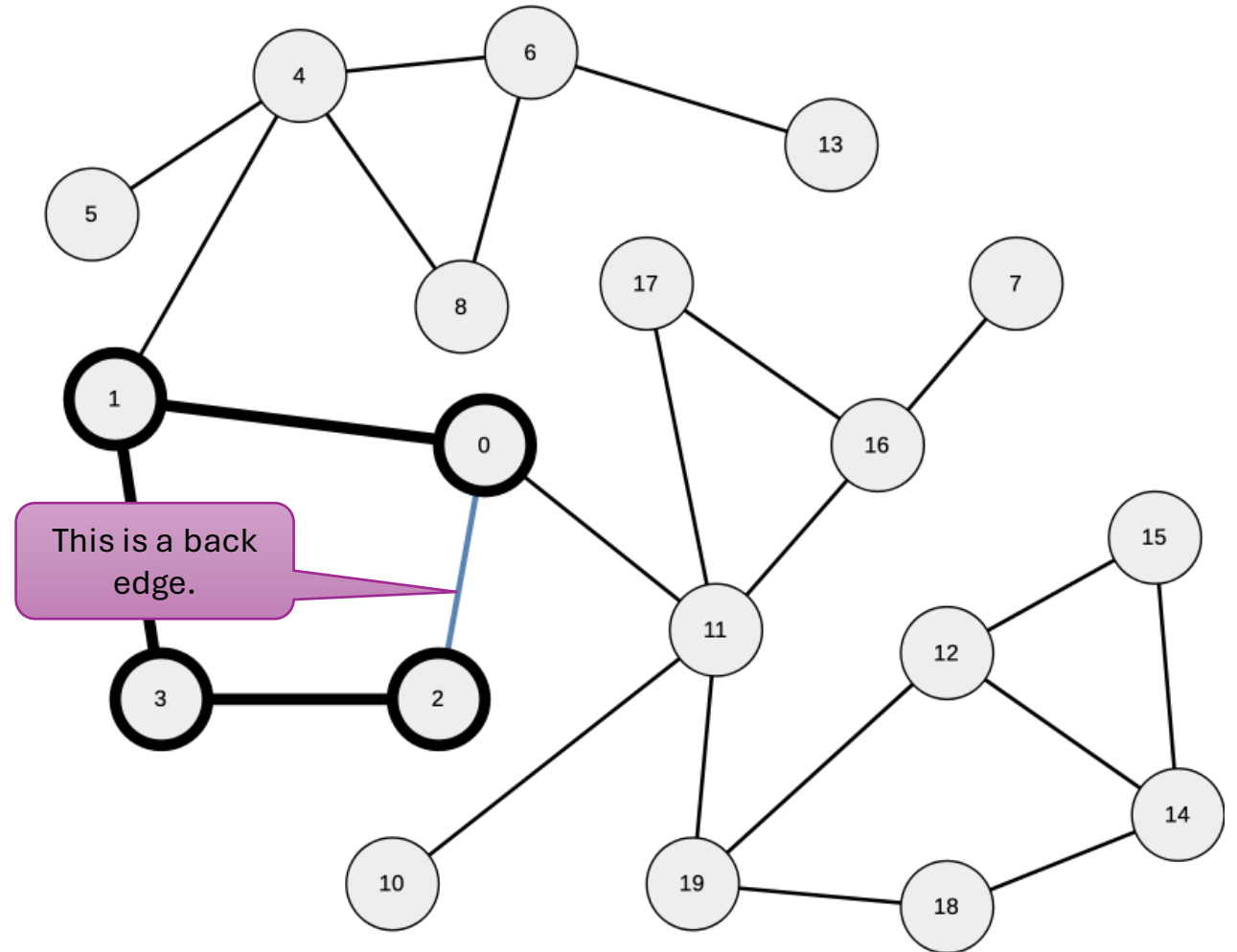
# Depth-First Traversal



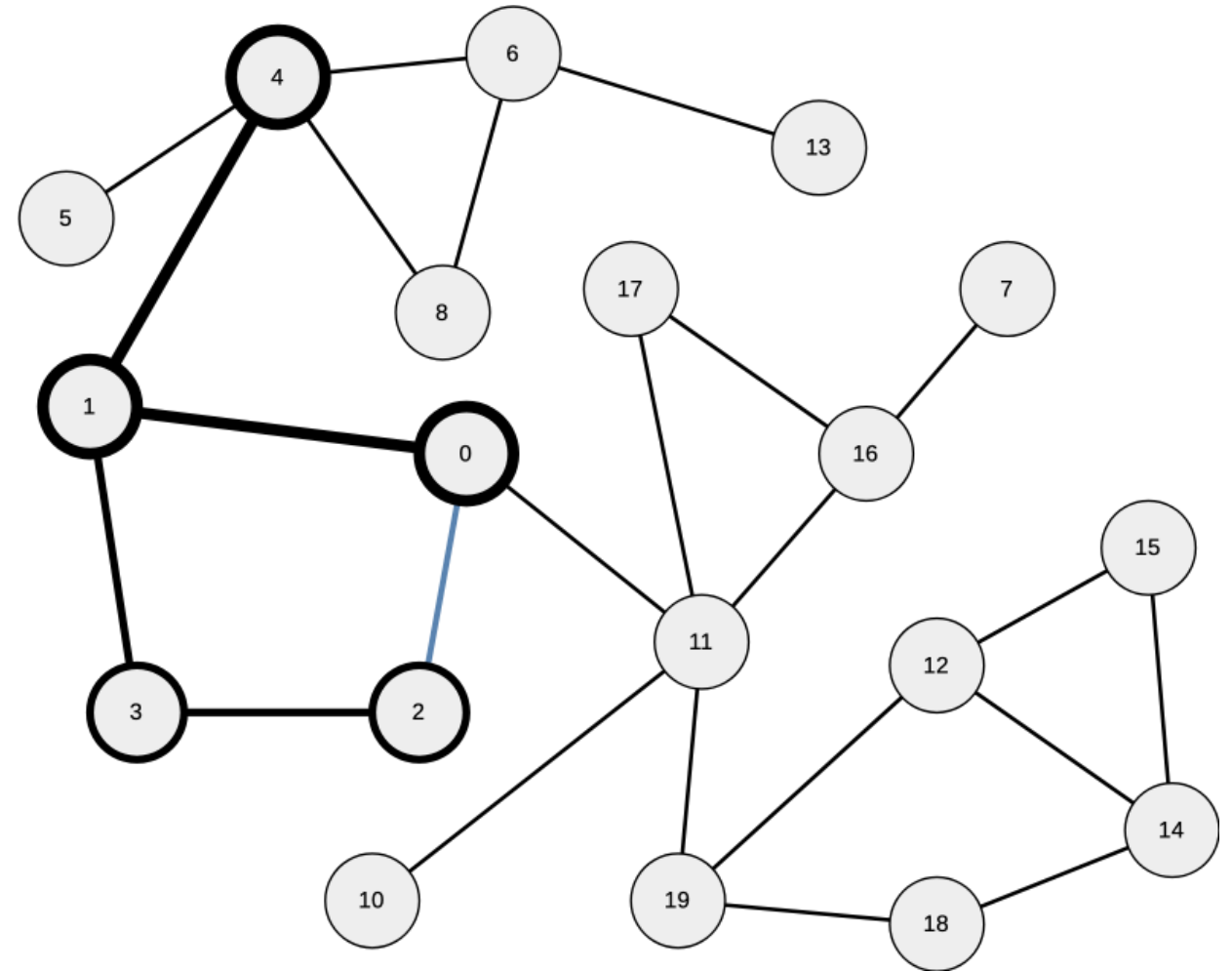
# Depth-First Traversal



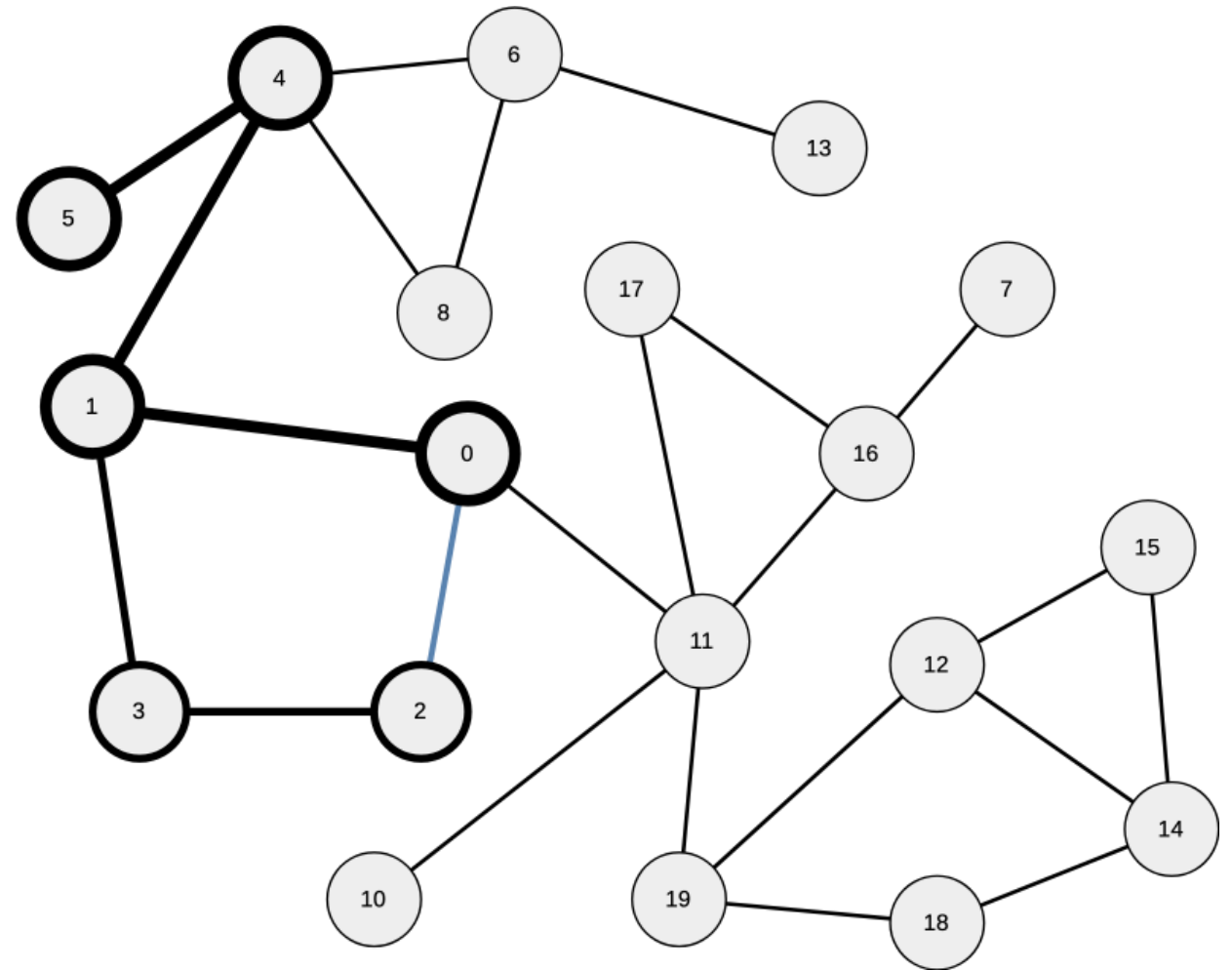
# Depth-First Traversal



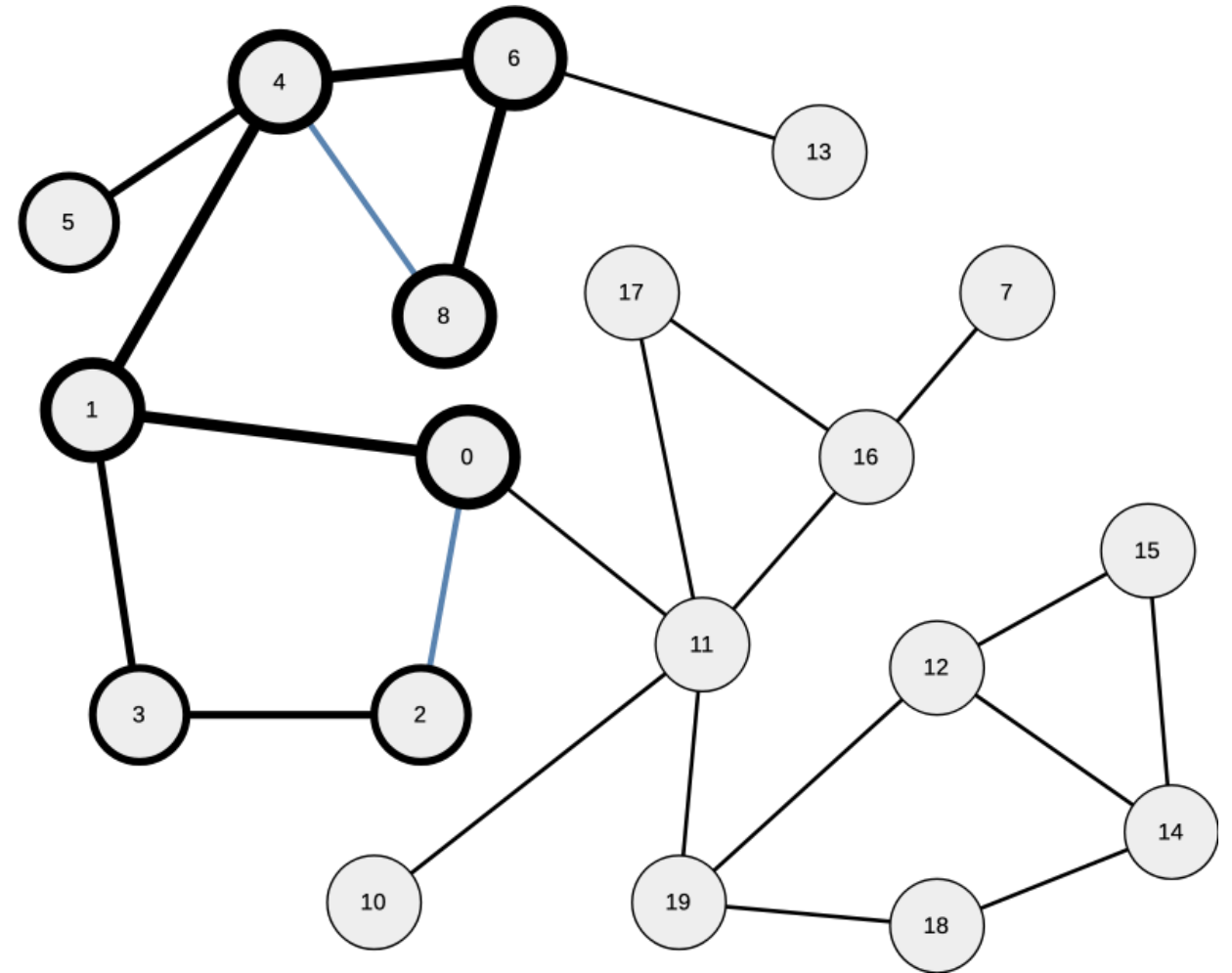
# Depth-First Traversal



# Depth-First Traversal

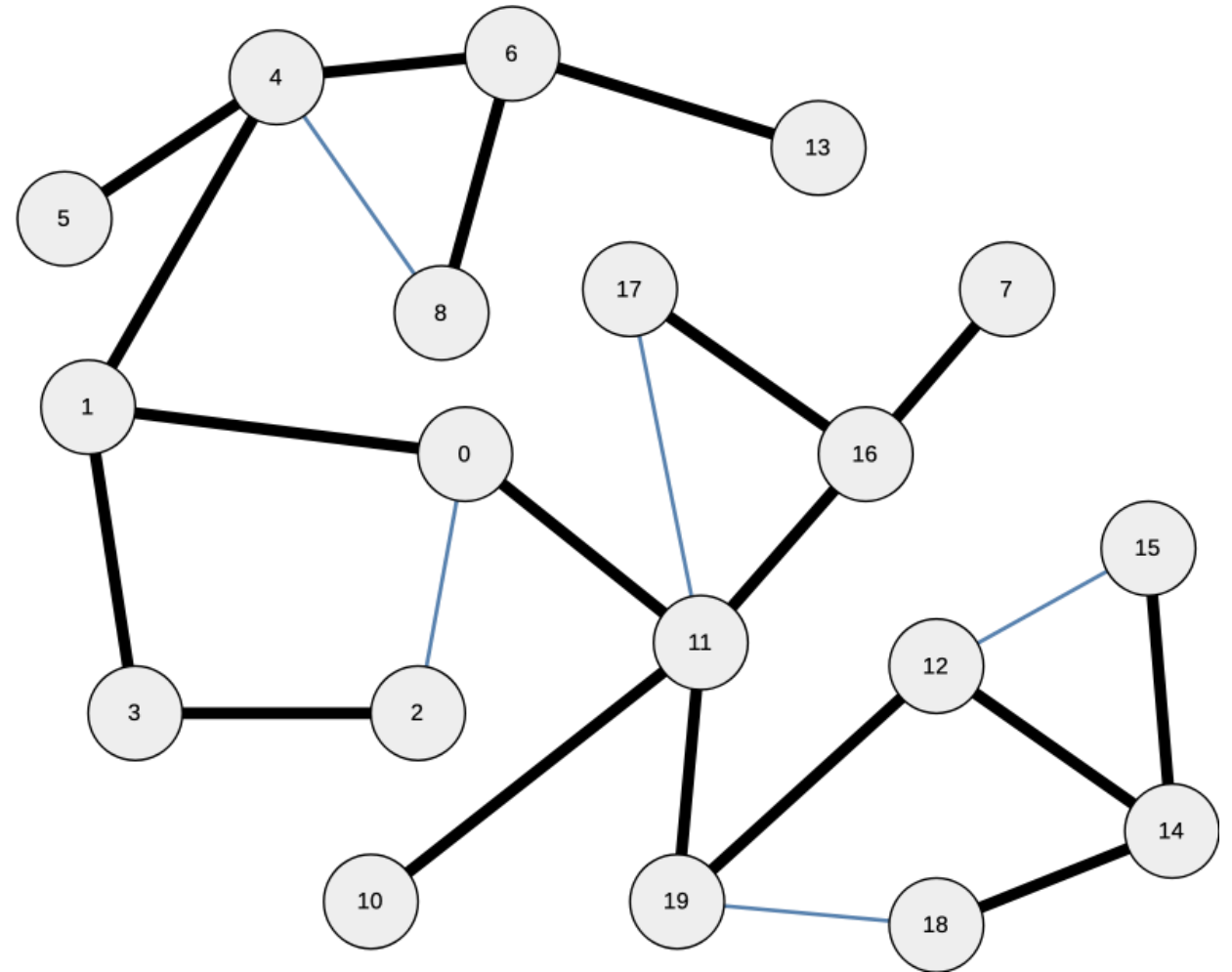


# Depth-First Traversal



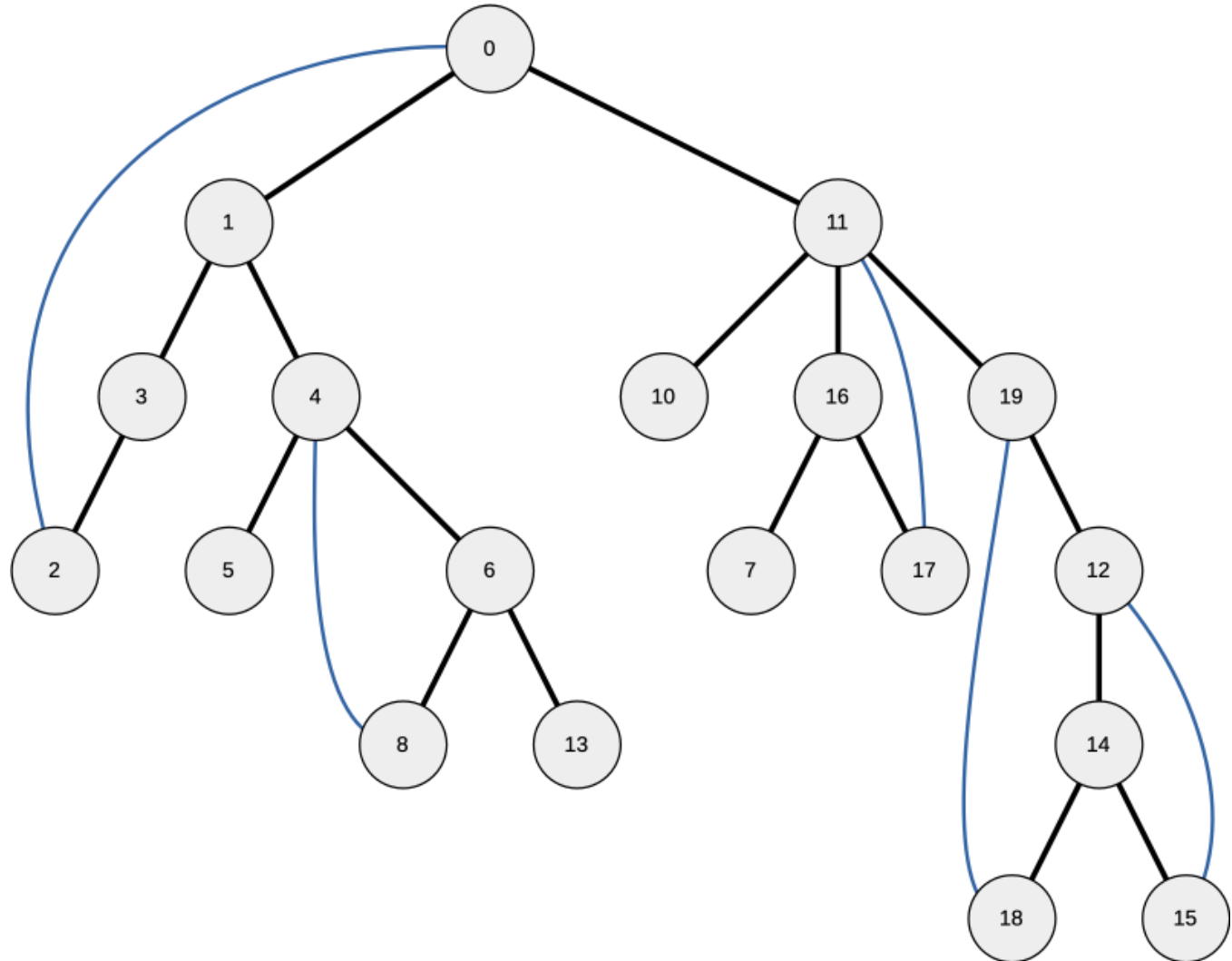
# Depth-First Traversal

- DFS alone is worth something
- Some programming problems require little more than graph traversal.



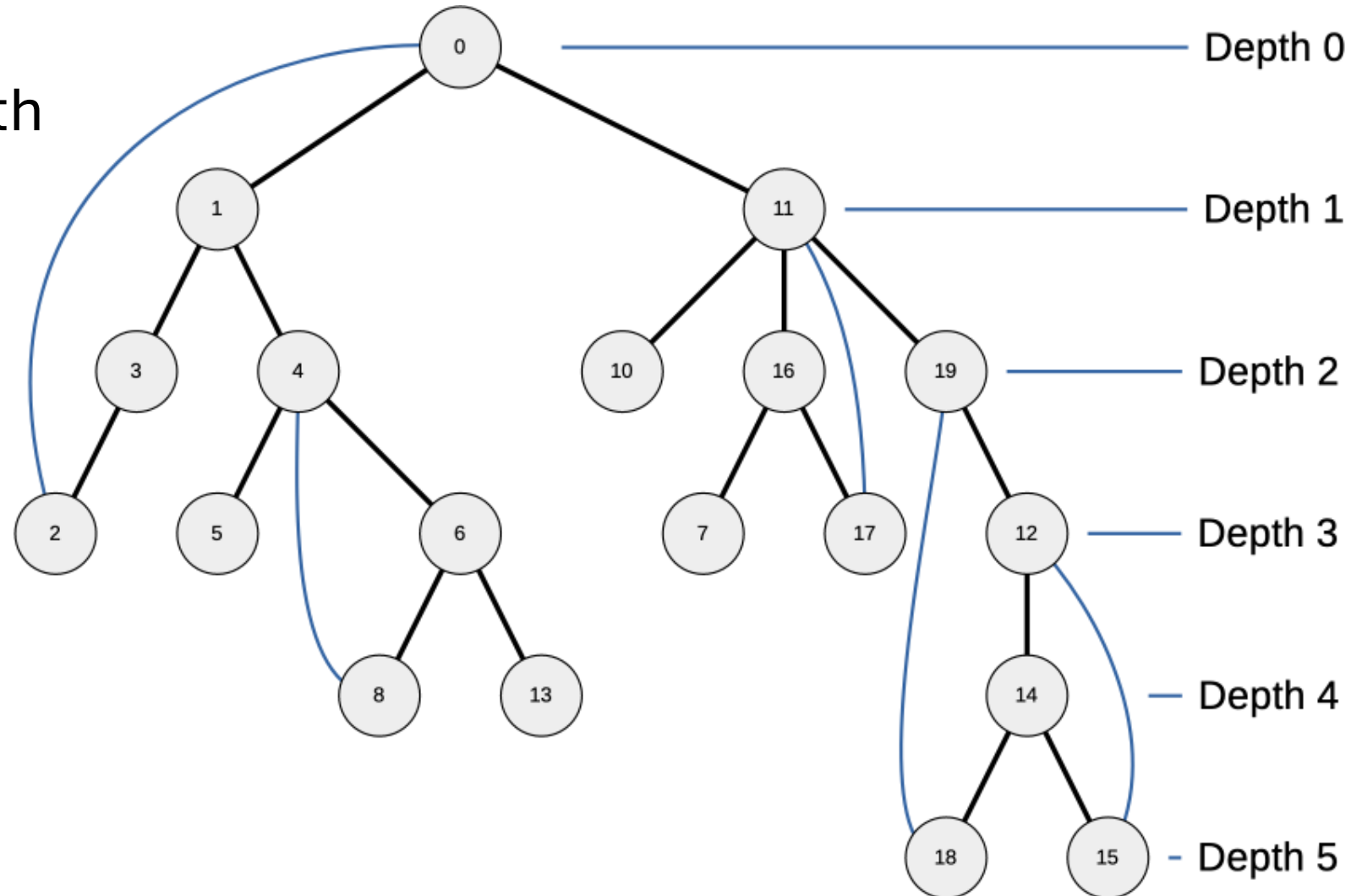
# DFS Depth

- DFS traversal gives us a tree (or forest) as a subset of the graph edges.
  - Forward edges go from parent to child.
  - Back edges go from a descendant to an ancestor.



# Depth-First Traversal

- We can assign a depth to each vertex during traversal.
  - Depth of a vertex is one greater than its parent's depth.



# DFS Traversal, with Depth

```
void dfs( int i, int d ) {  
    visited[ i ] = true;  
    depth[ i ] = d;  
  
    for ( int j : edge[ i ] ) {  
        if ( ! visited[ j ] ) {  
            parent[ j ] = i;  
            dfs( j, d + 1 );  
        } else  
            if ( j != parent[ i ] )  
                // i<->j is a back edge  
    }  
}
```

# DFS Traversal, with Depth

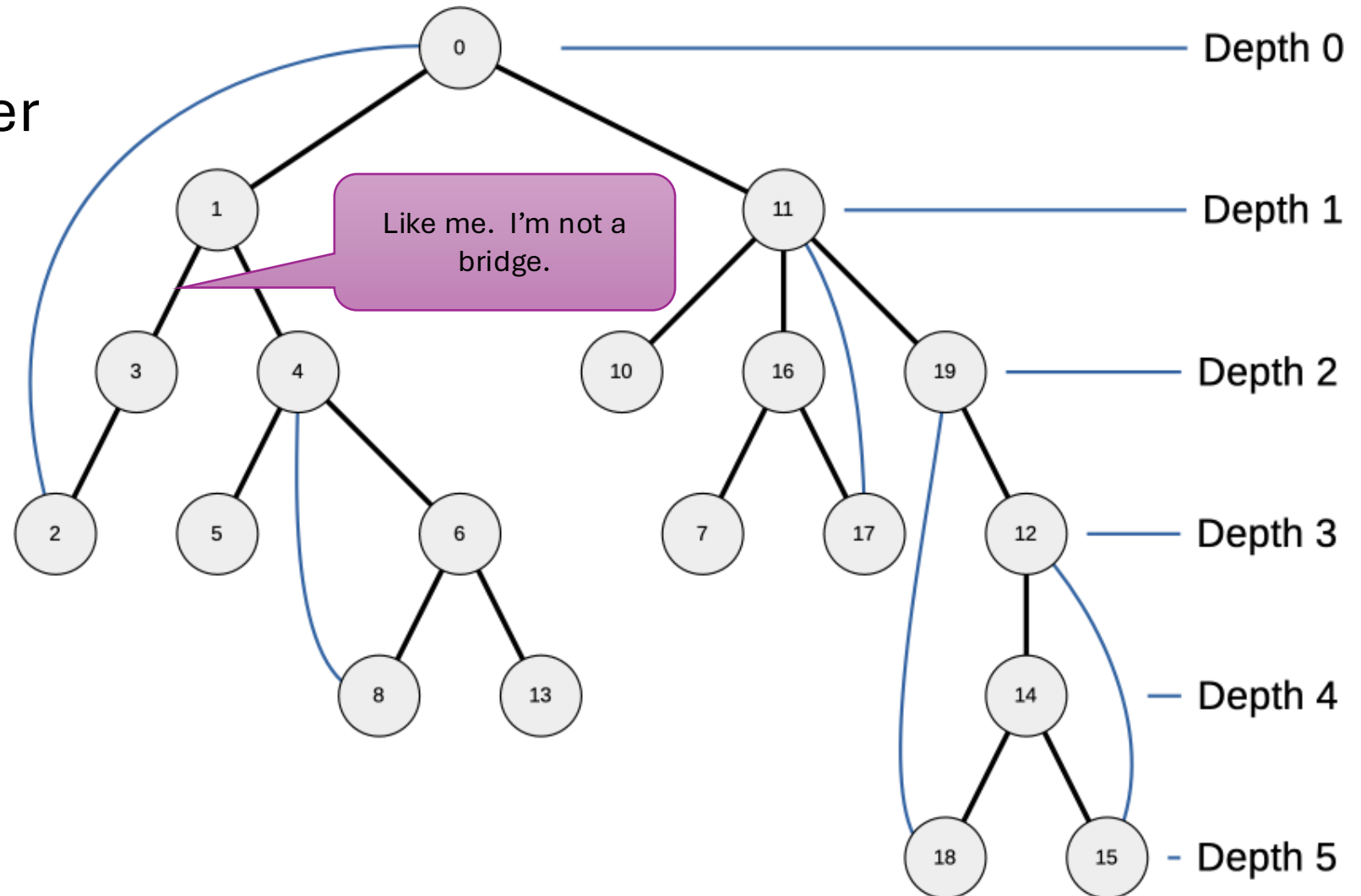
```
void dfs( int i, int d ) {  
    visited[ i ] = true;  
    depth[ i ] = d;  
  
    for ( int j : edge[ i ] ) {  
        if ( ! visited[ j ] ) {  
            parent[ j ] = i;  
            dfs( j, d + 1 );  
        } else  
            if ( j != parent[ i ] )  
                // i<->j is a back edge  
    }  
}
```

This prevents going  
right back where we  
came from.

There are other ways  
to solve this problem.

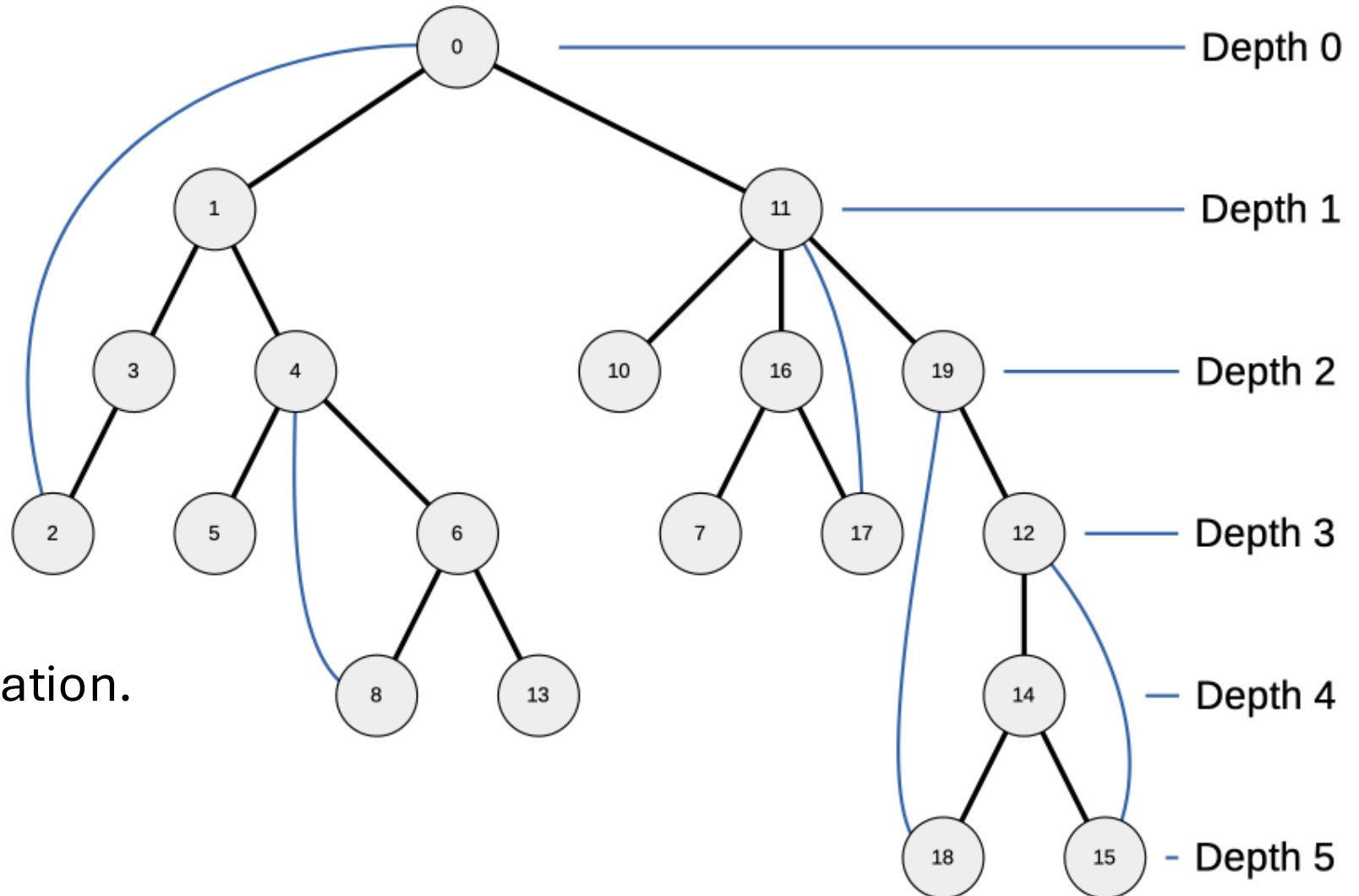
# Finding Bridges

- A back edge can never be a bridge.
- A forward edge is not a bridge if:
  - There's a back edge from the child or one of its descendants
  - ... to the parent or one of its ancestors.



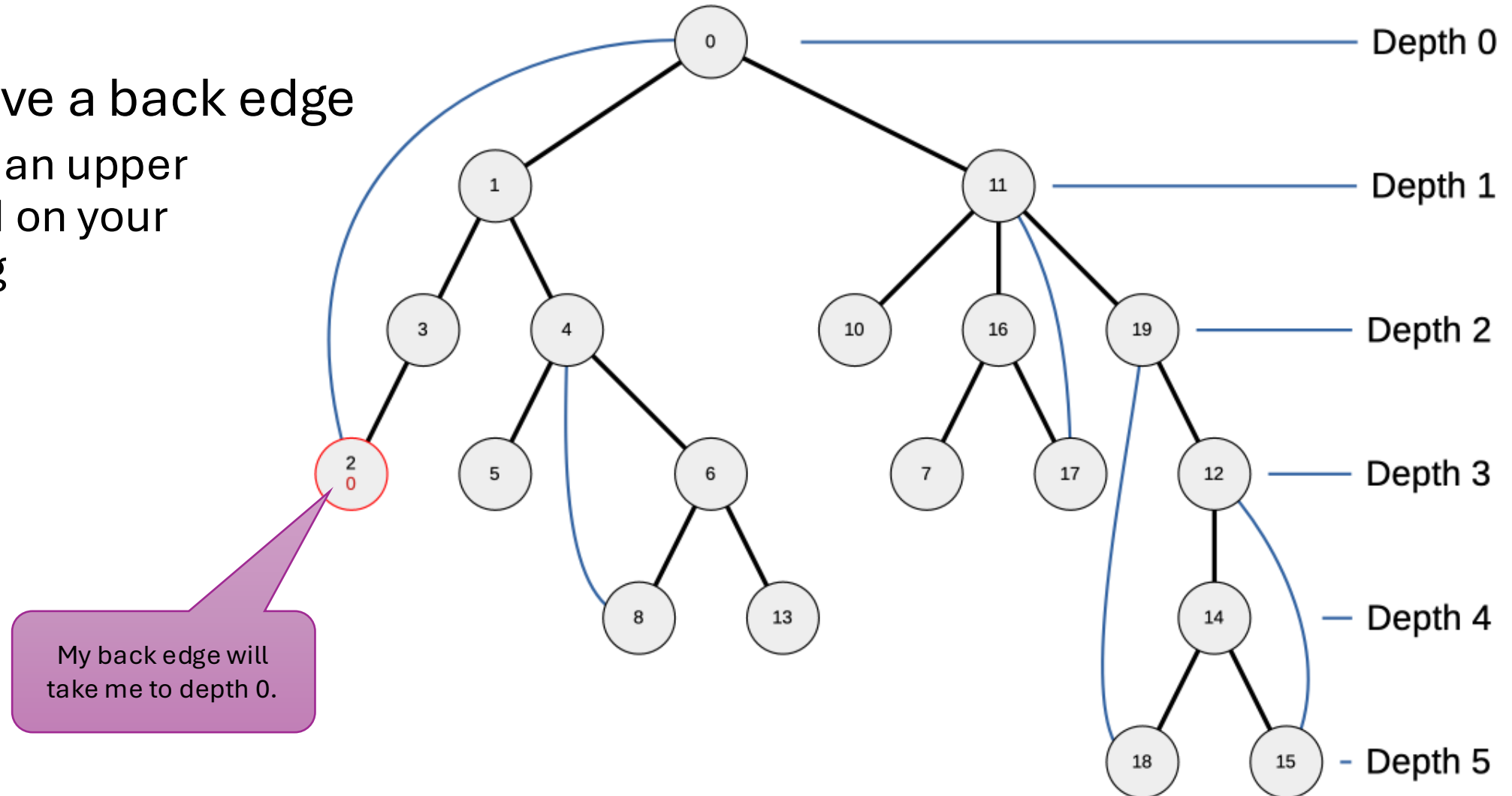
# Finding the Ceiling

- For every vertex, we will compute a ceiling value.
  - How far up the tree you can reach via a back edge in your subtree.
  - This is typically called “low” in a reference implementation.



# Finding the Ceiling

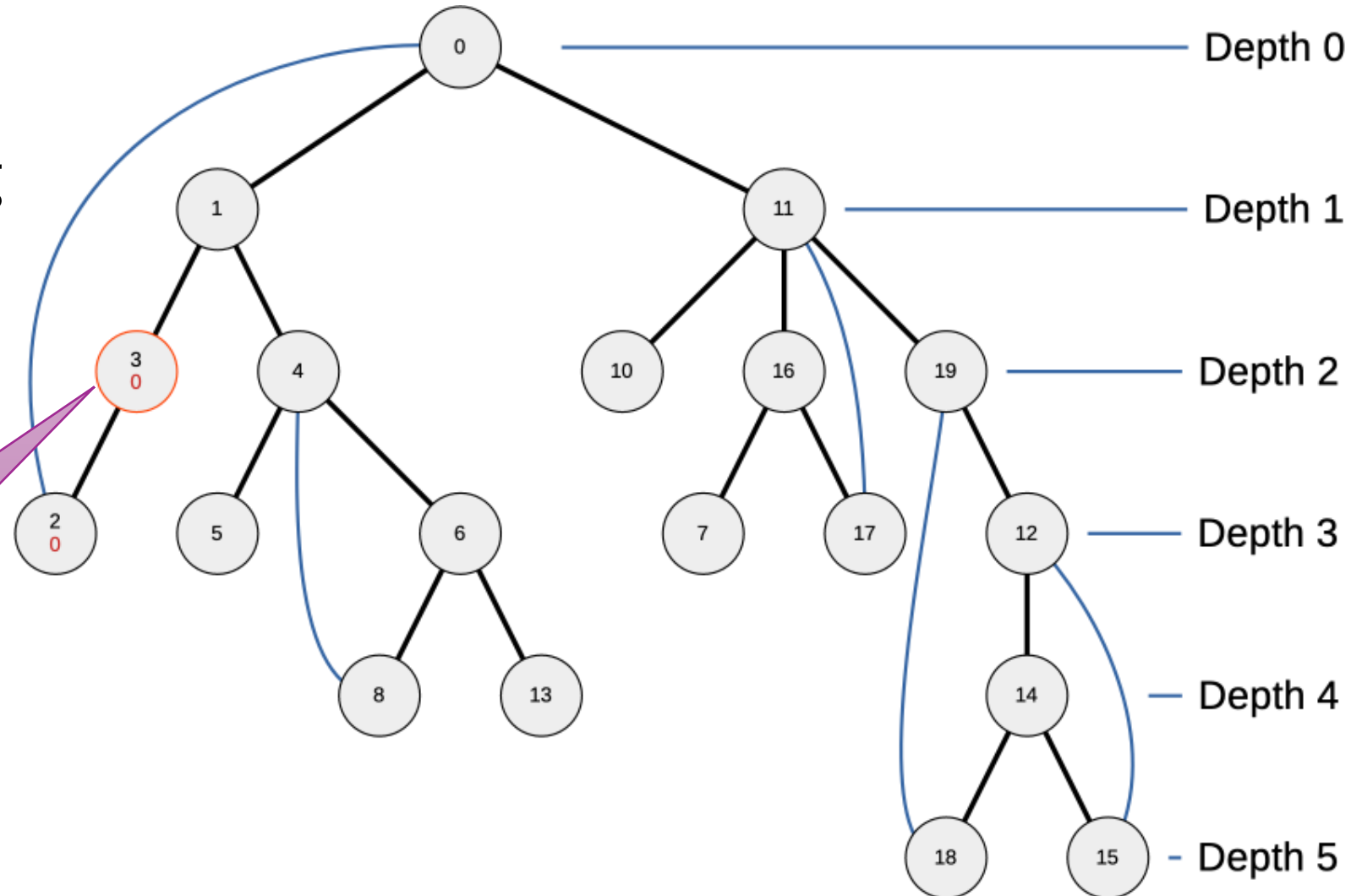
- If you have a back edge
  - That's an upper bound on your ceiling



# Finding the Ceiling

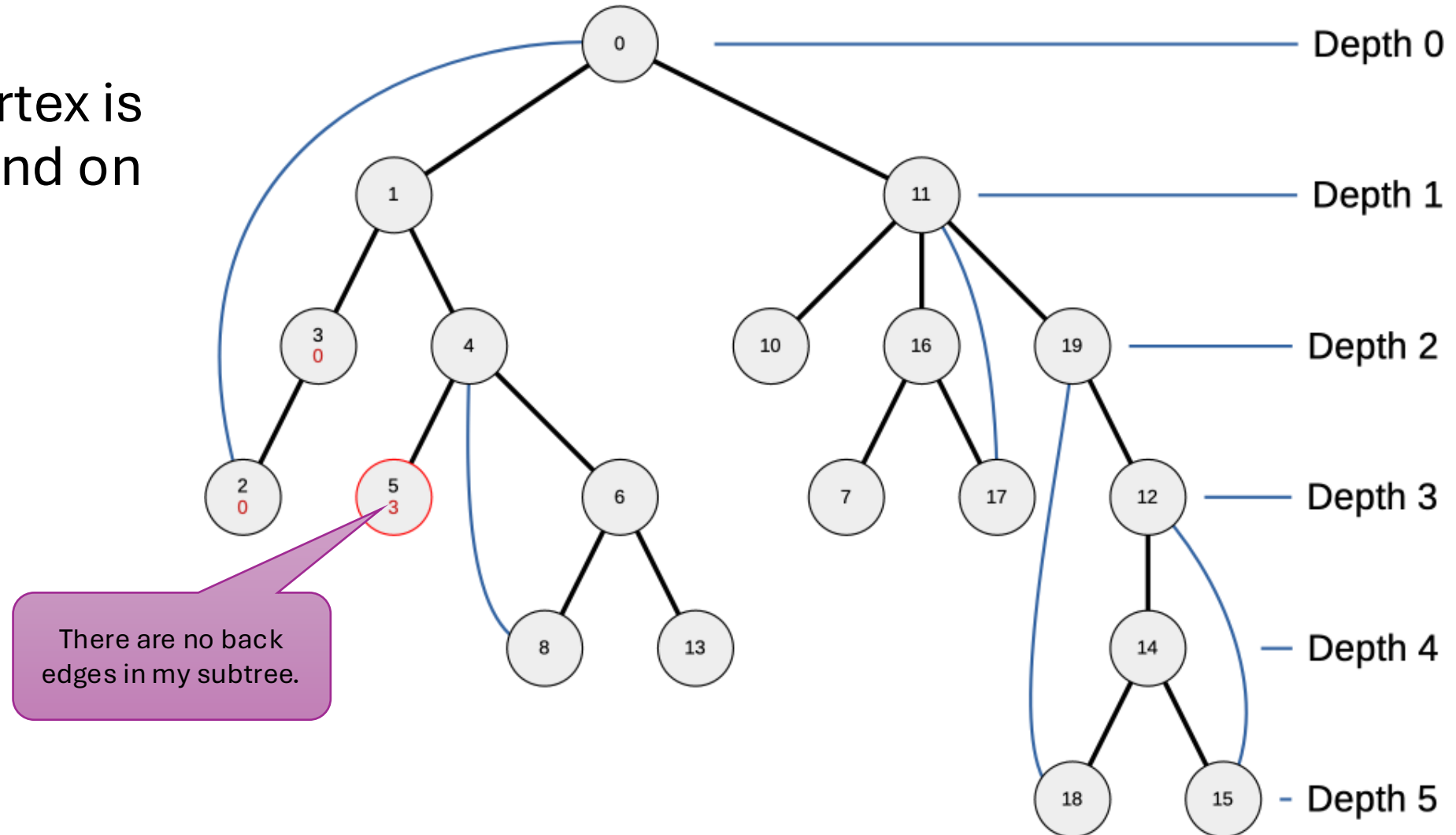
- Every vertex should get the lowest ceiling value in its subtree
  - Depth of the highest ancestor reachable via a back edge.

Vertex 2 has the lowest ceiling in my subtree.



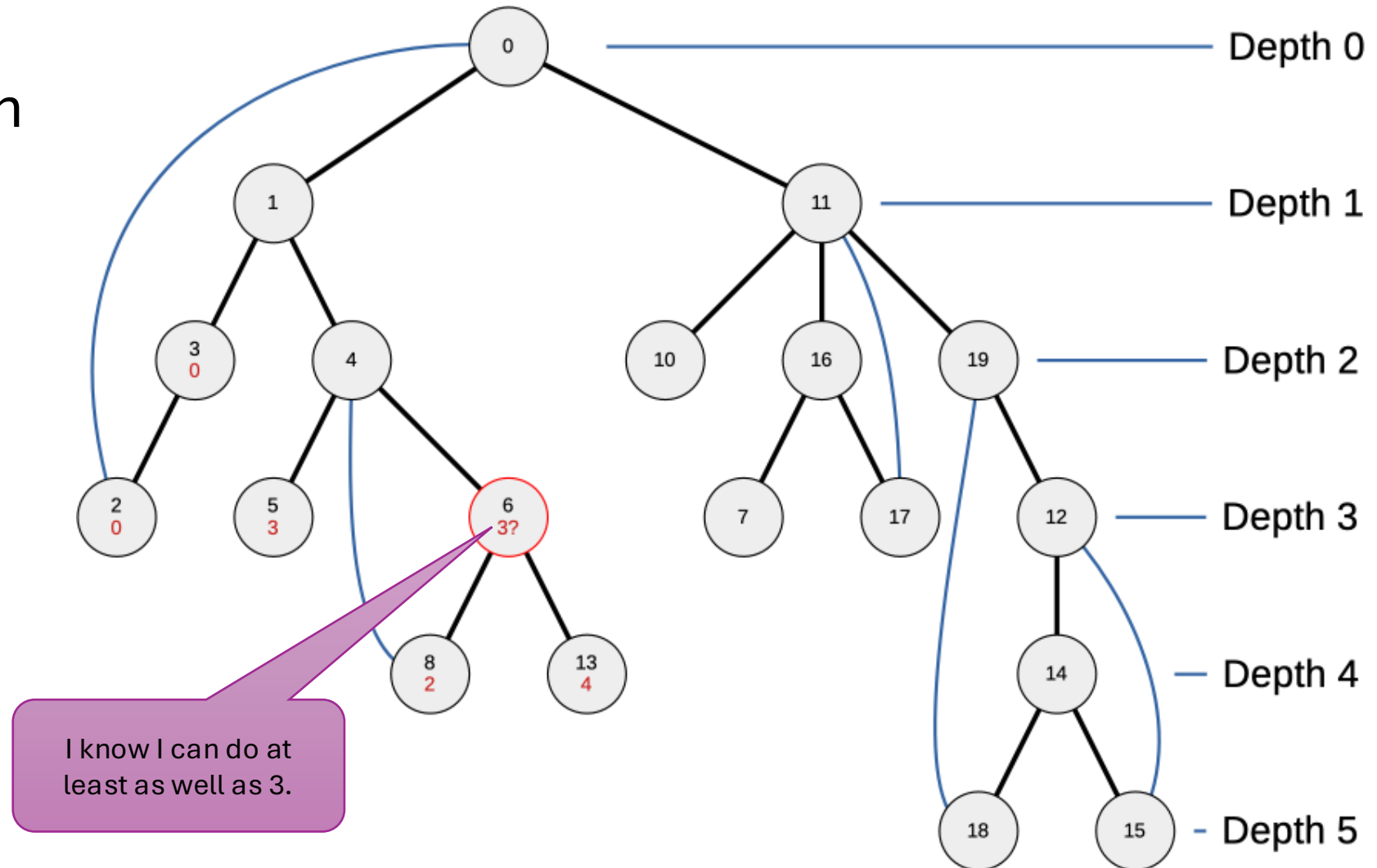
# Finding the Ceiling

- Depth of a vertex is an upper bound on its ceiling.



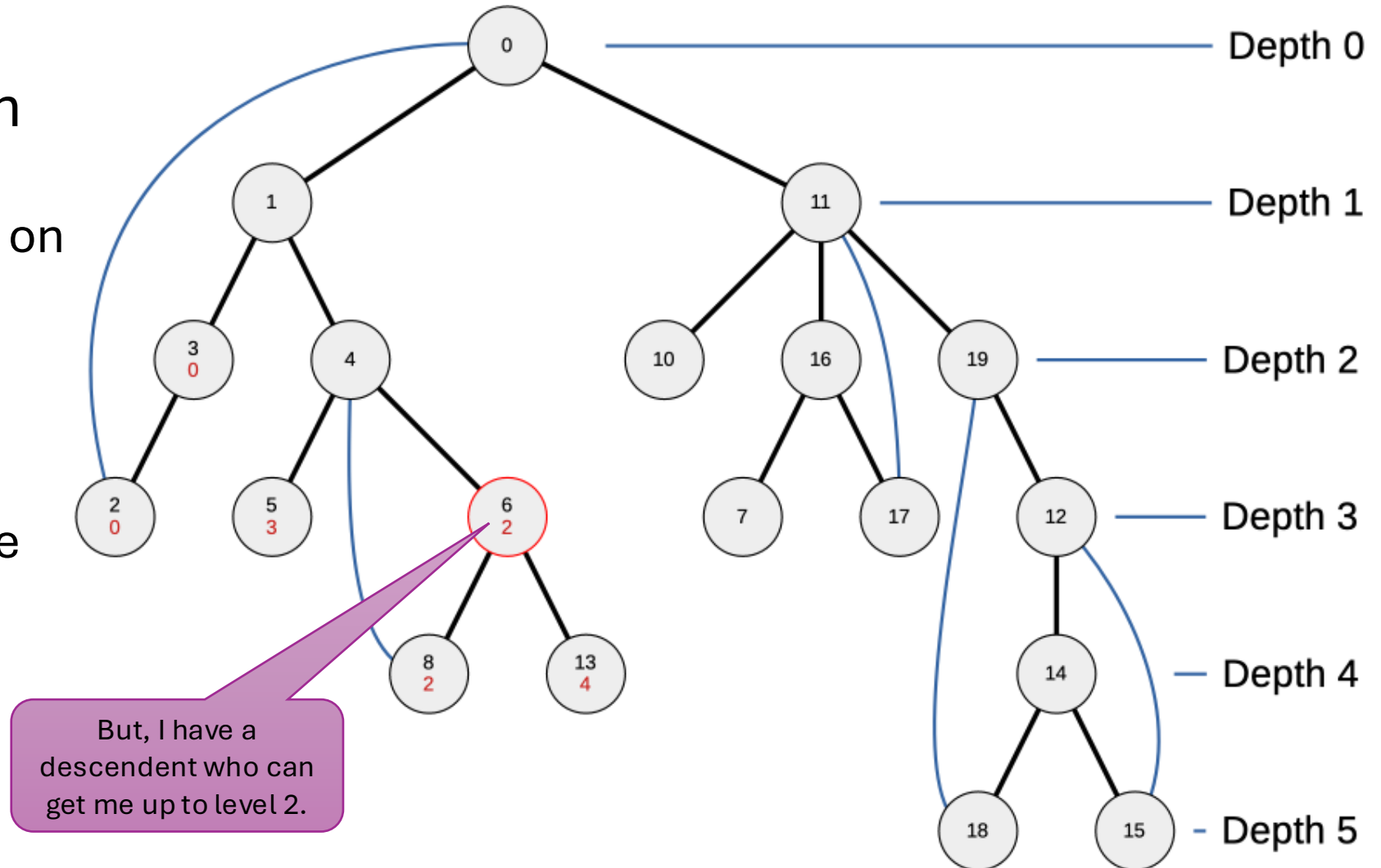
# Finding the Ceiling

- If you have children



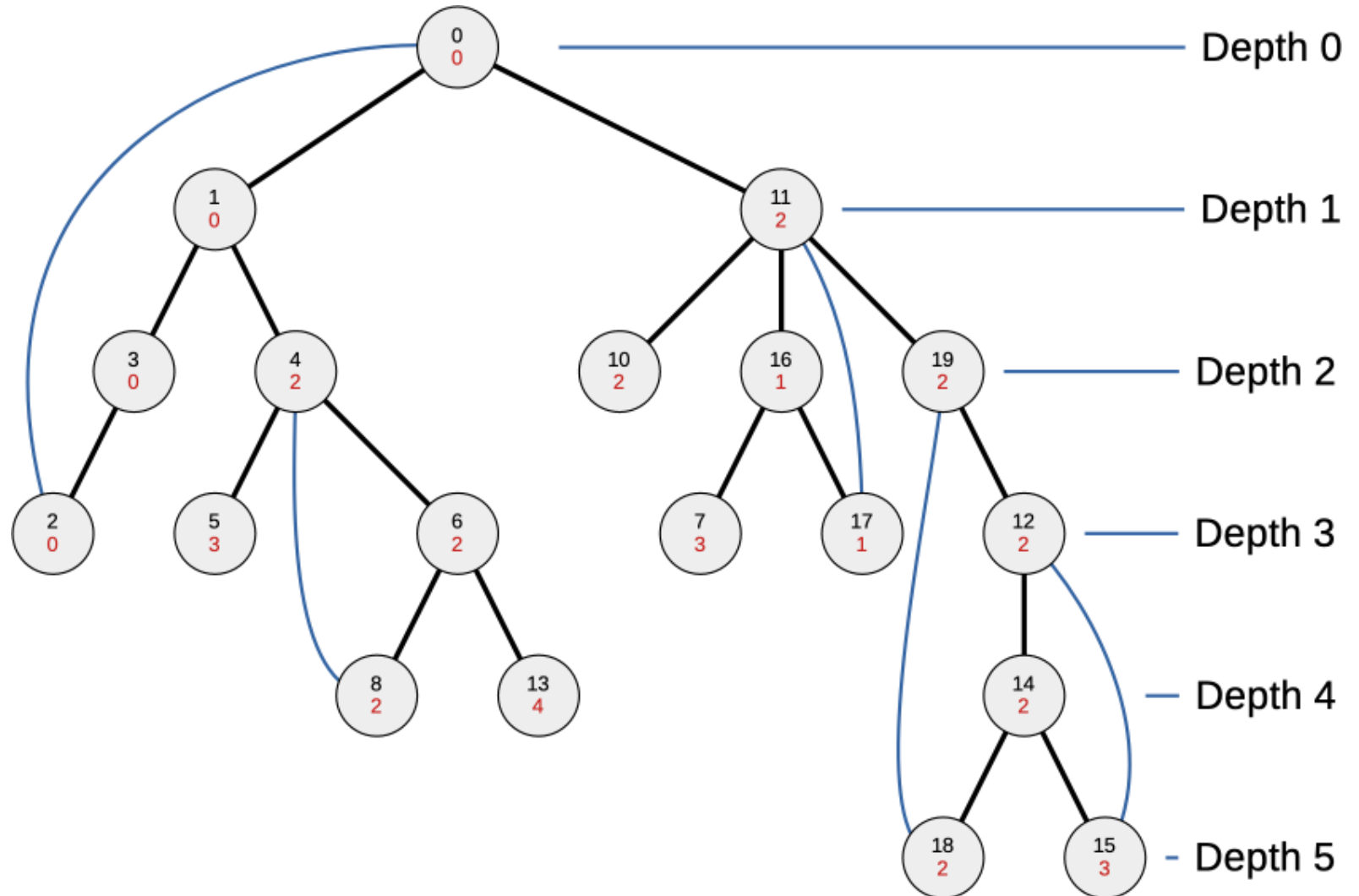
# Finding the Ceiling

- If you have children
  - Their ceiling value is an upper bound on your own.
  - We can compute this minimum during the DFS.
  - That's a linear-time algorithm.



# Finding the Bridges

- A forward edge is a bridge if
  - The ceiling value at the child
  - is greater than
  - The depth of the parent.



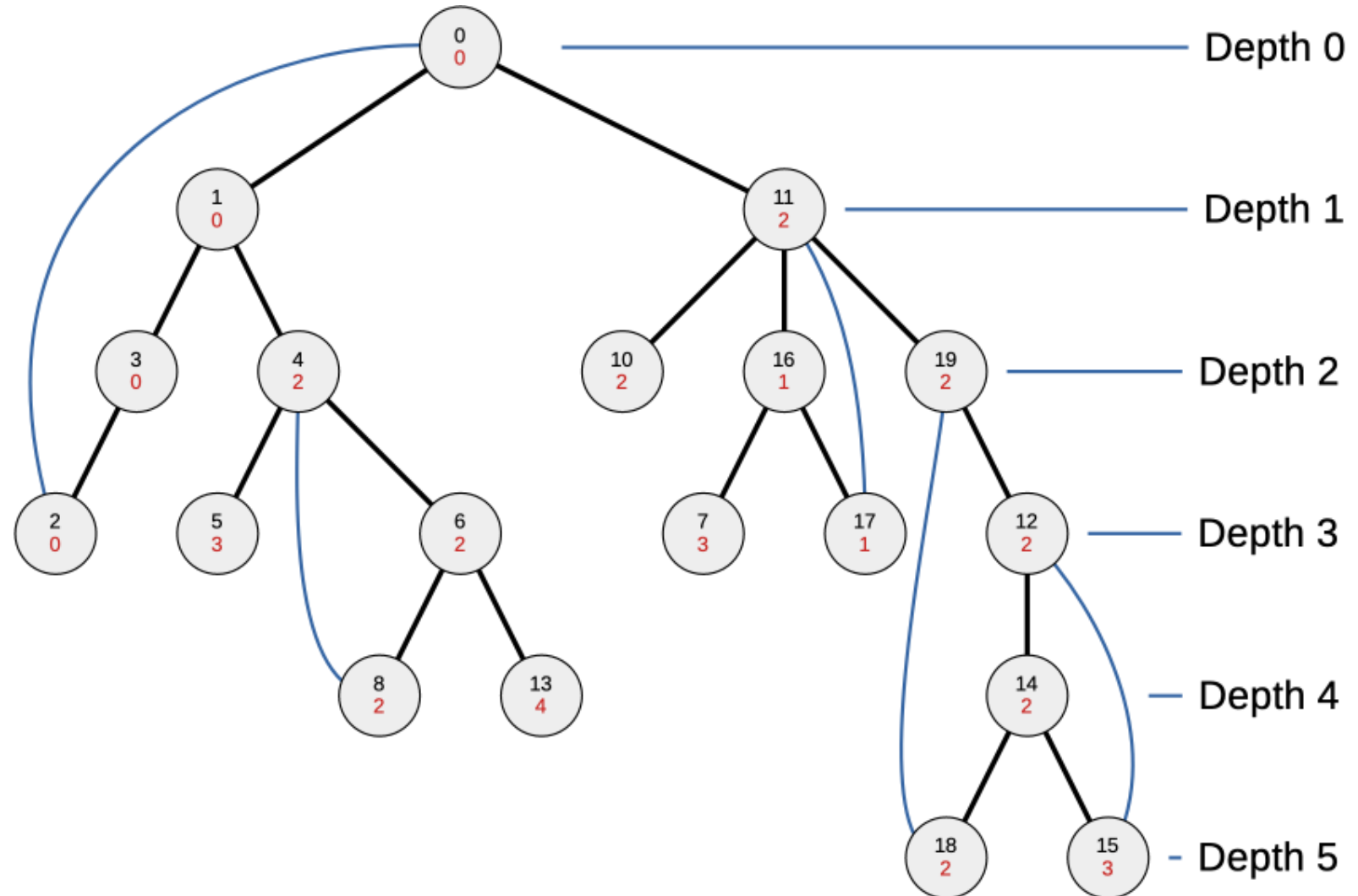
# DFS Traversal, with Depth

```
void dfs( int i, int d ) {
    visited[ i ] = true;
    depth[ i ] = d;
    c[ i ] = d;

    for ( int j : edge[ i ] ) {
        if ( ! visited[ j ] ) {
            parent[ j ] = i;
            dfs( j, d + 1 );
            if ( c[ j ] > depth[ i ] )
                // i<->j is a bridge.
                c[ i ] = min( c[ i ], c[ j ] );
        } else
            if ( j != parent[ i ] )
                c[ i ] = min( c[ i ], depth[ j ] );
    }
}
```

# Finding Articulation Points

- The root could be an articulation point
  - .. if it has more than one child.



# Finding Articulation Points

- An internal vertex,  $i$ , could be an articulation point
  - If  $i$  has a child with a ceiling greater than or equal to  $\text{depth}[i]$ .

