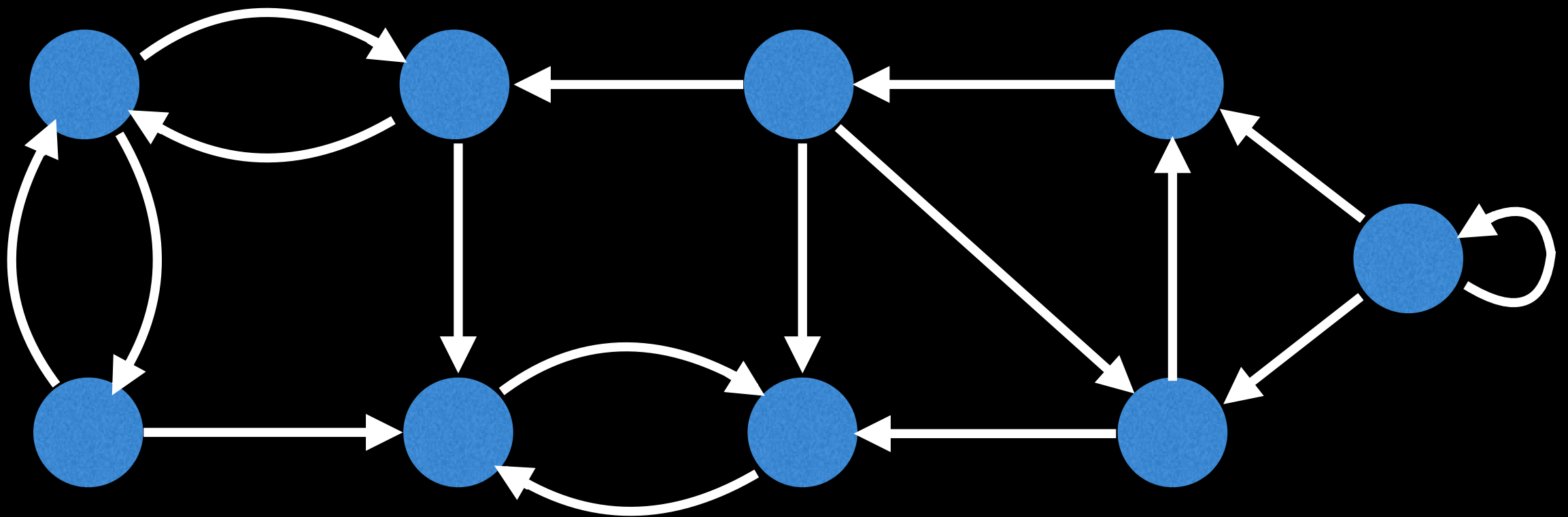


# Tarjan's Algorithm for Finding Strongly Connected Components

William Fiset

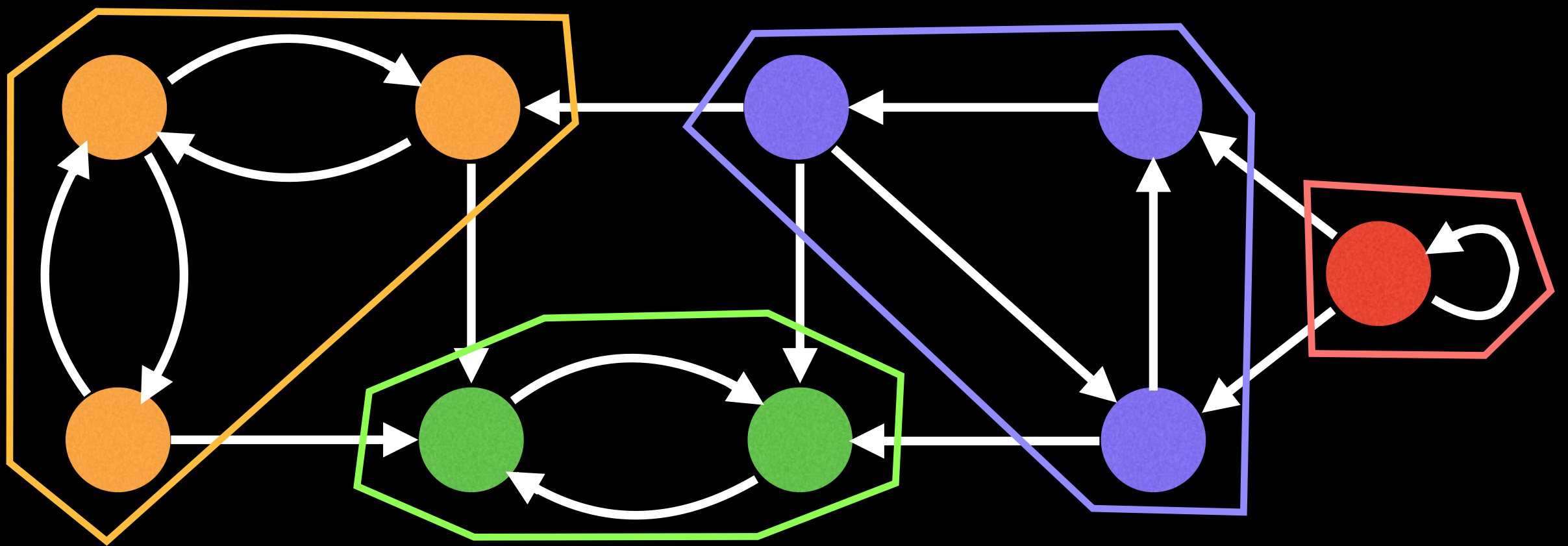
# What are SCCs?

Strongly Connected Components (SCCs) can be thought of as **self-contained cycles** within a **directed graph** where every vertex in a given cycle can reach every other vertex in the same cycle.



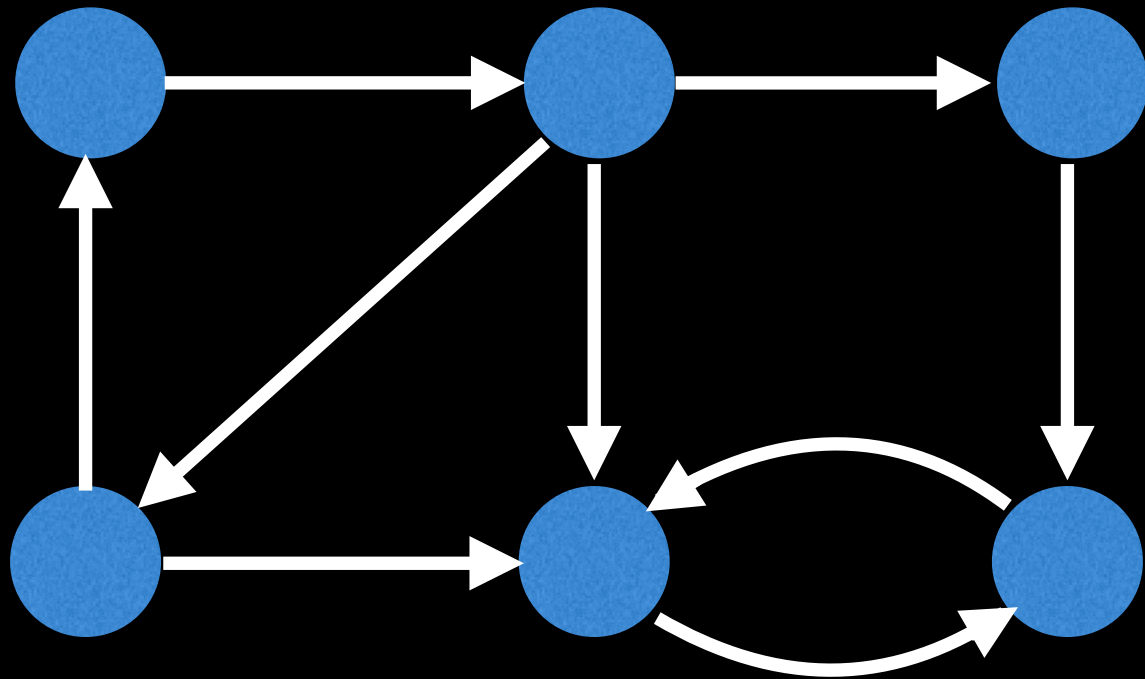
# What are SCCs?

Strongly Connected Components (SCCs) can be thought of as **self-contained cycles** within a **directed graph** where every vertex in a given cycle can reach every other vertex in the same cycle.



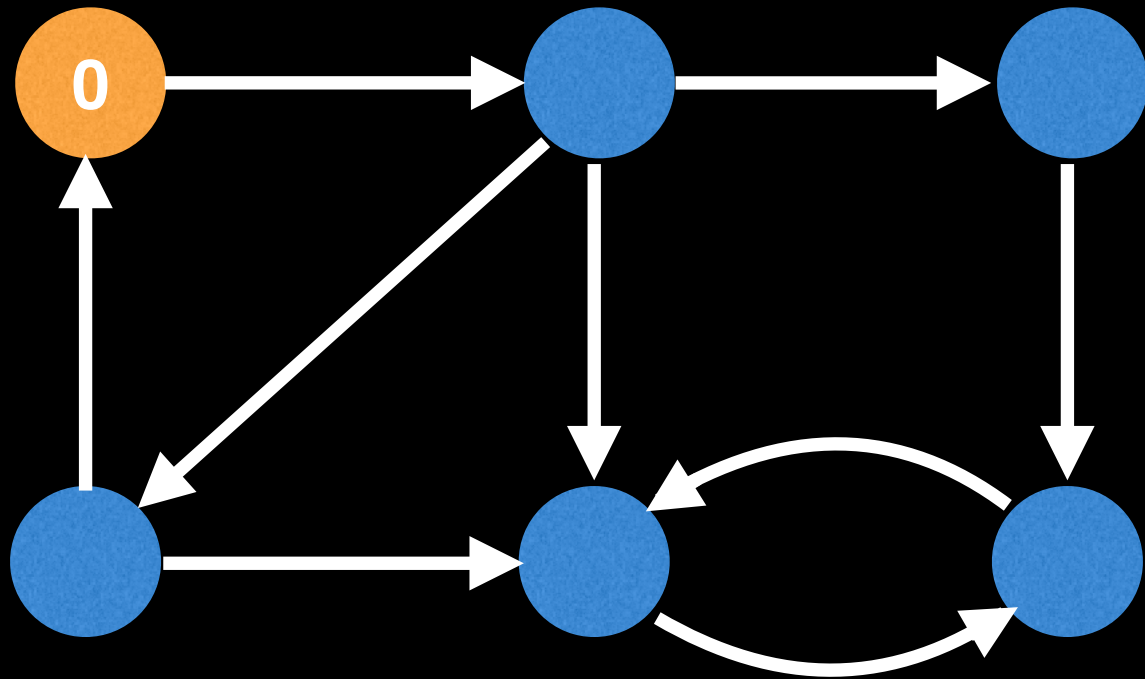
# Low-Link Values

The **low-link** value of a node is the smallest [lowest] node id reachable from that node when doing a DFS (including itself).



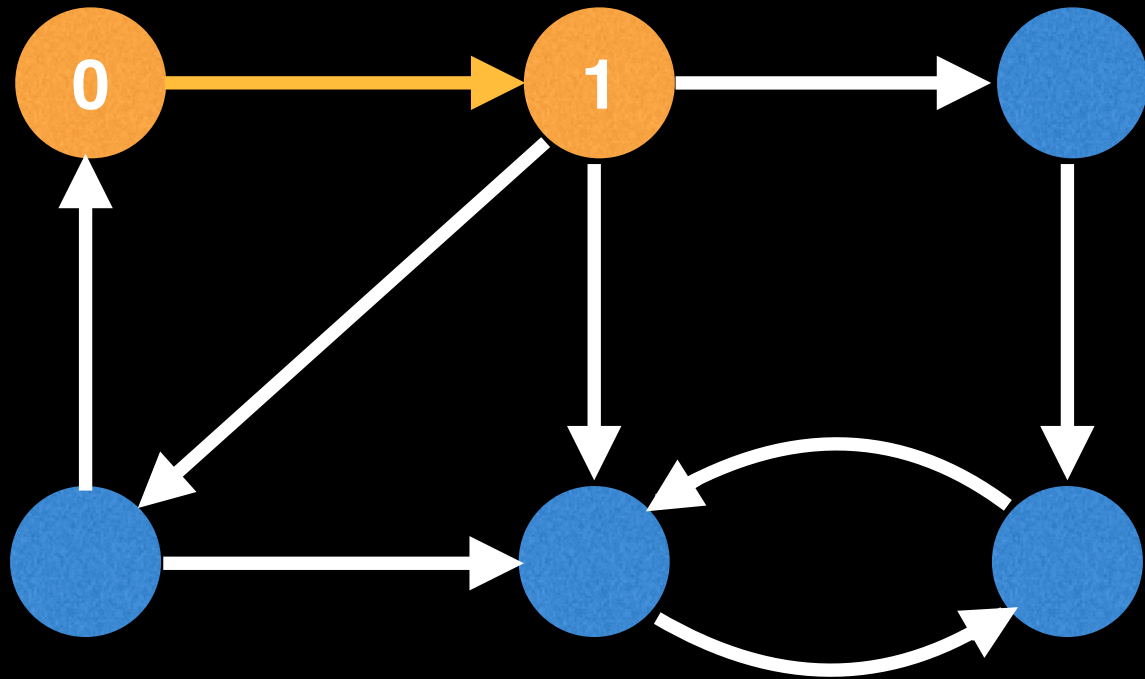
# Low-Link Values

The **low-link** value of a node is the smallest [lowest] node id reachable from that node when doing a DFS (including itself).



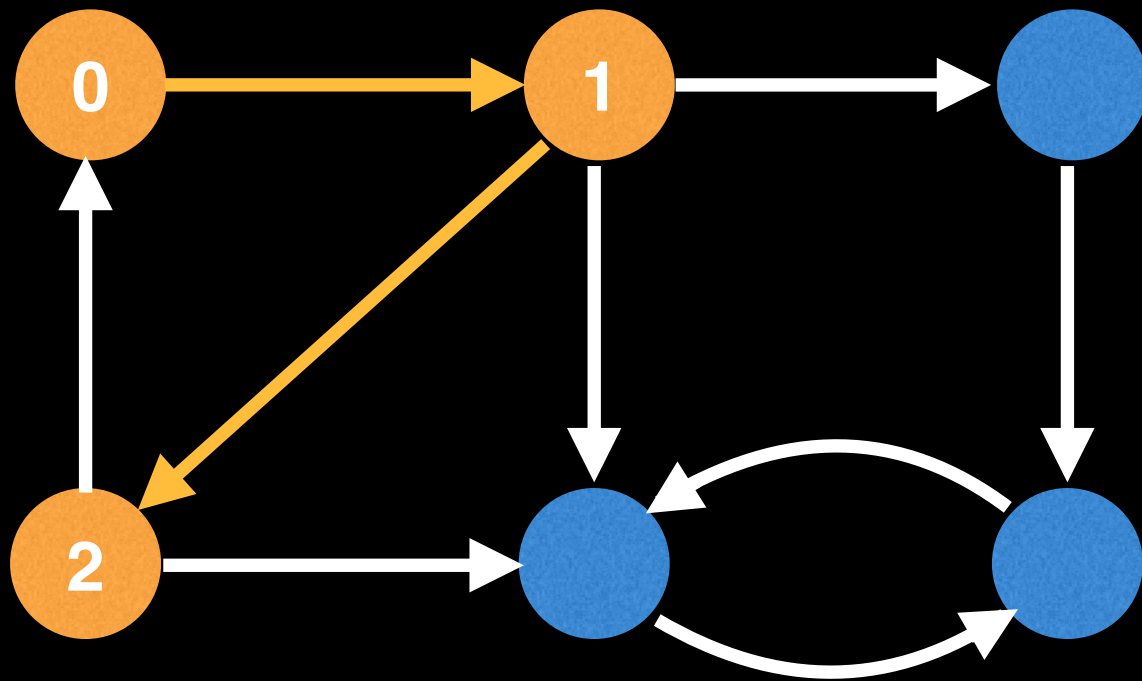
# Low-Link Values

The **low-link** value of a node is the smallest [lowest] node id reachable from that node when doing a DFS (including itself).



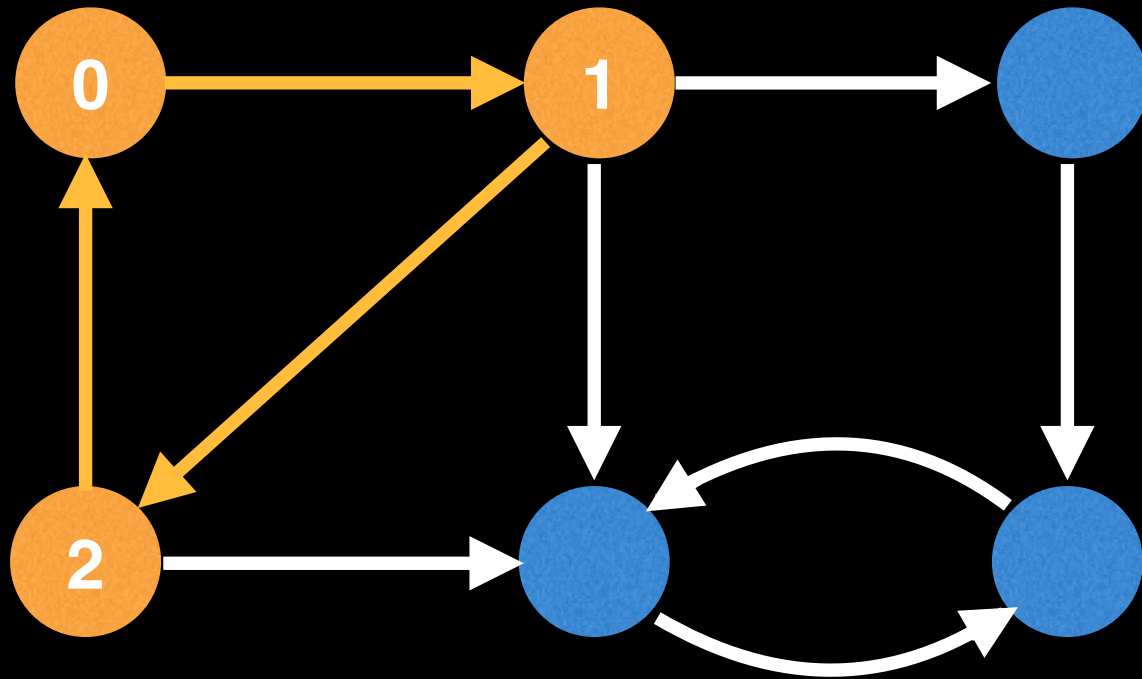
# Low-Link Values

The **low-link** value of a node is the smallest [lowest] node id reachable from that node when doing a DFS (including itself).



# Low-Link Values

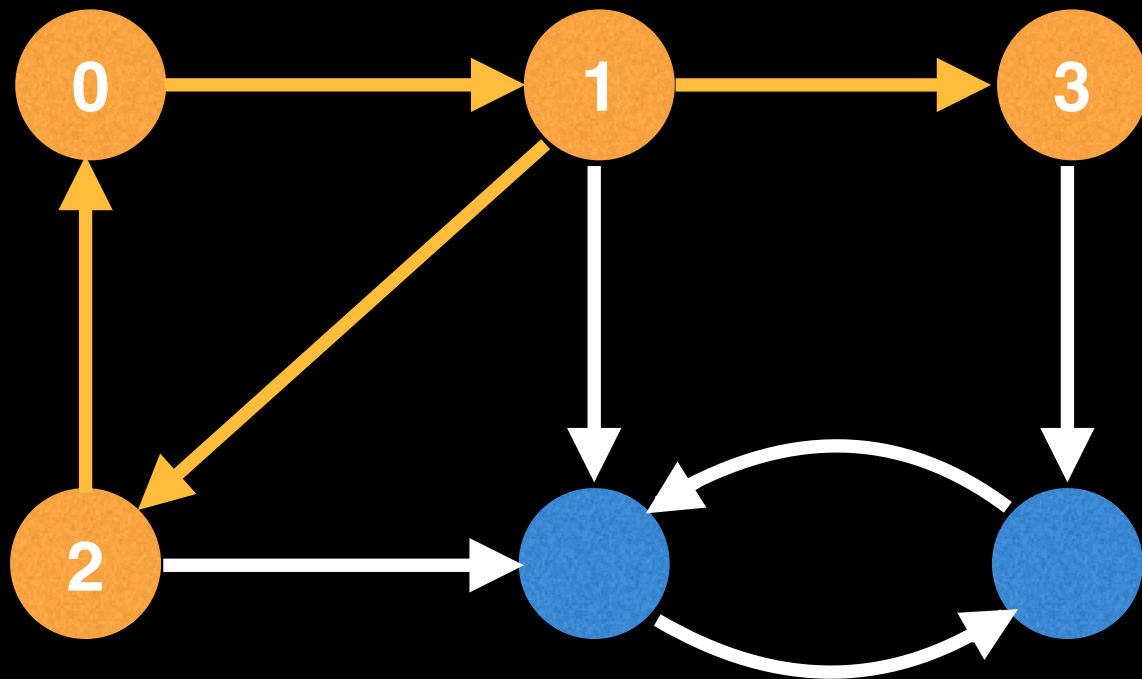
The **low-link** value of a node is the smallest [lowest] node id reachable from that node when doing a DFS (including itself).





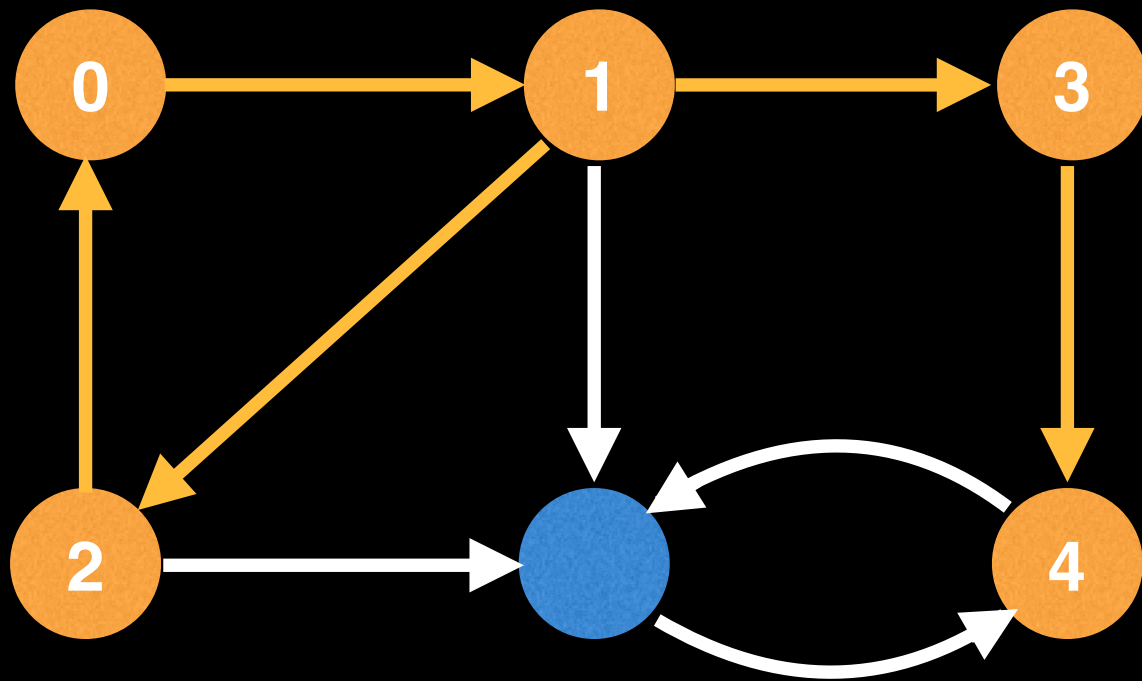
# Low-Link Values

The **low-link** value of a node is the smallest [lowest] node id reachable from that node when doing a DFS (including itself).



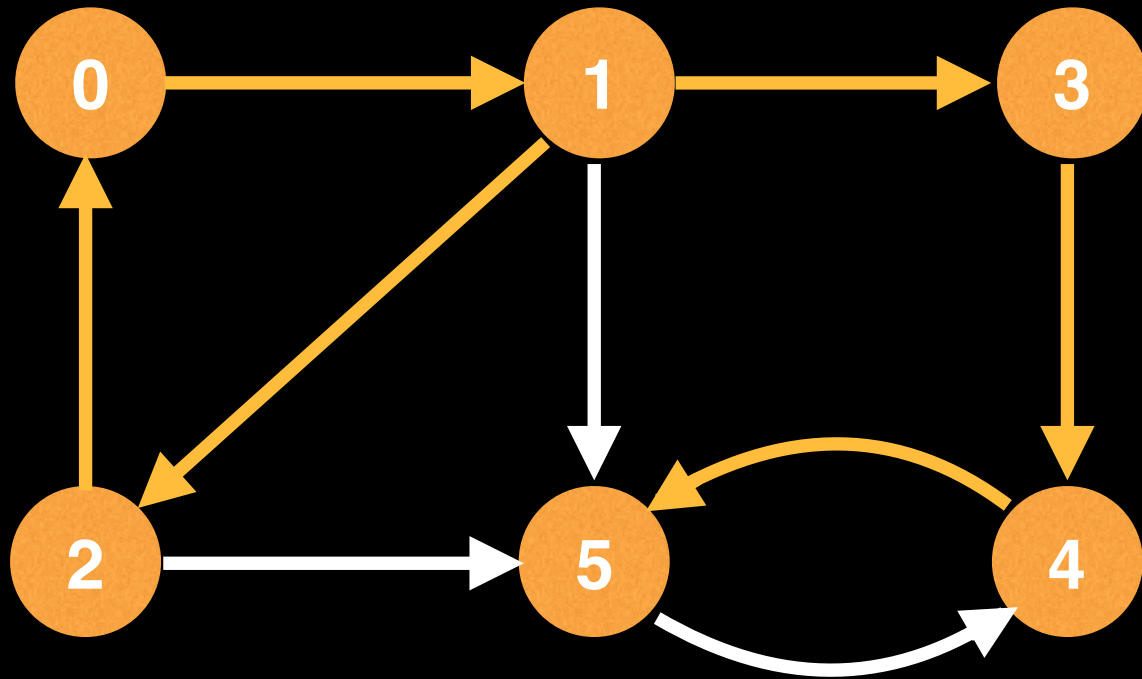
# Low-Link Values

The **low-link** value of a node is the smallest [lowest] node id reachable from that node when doing a DFS (including itself).



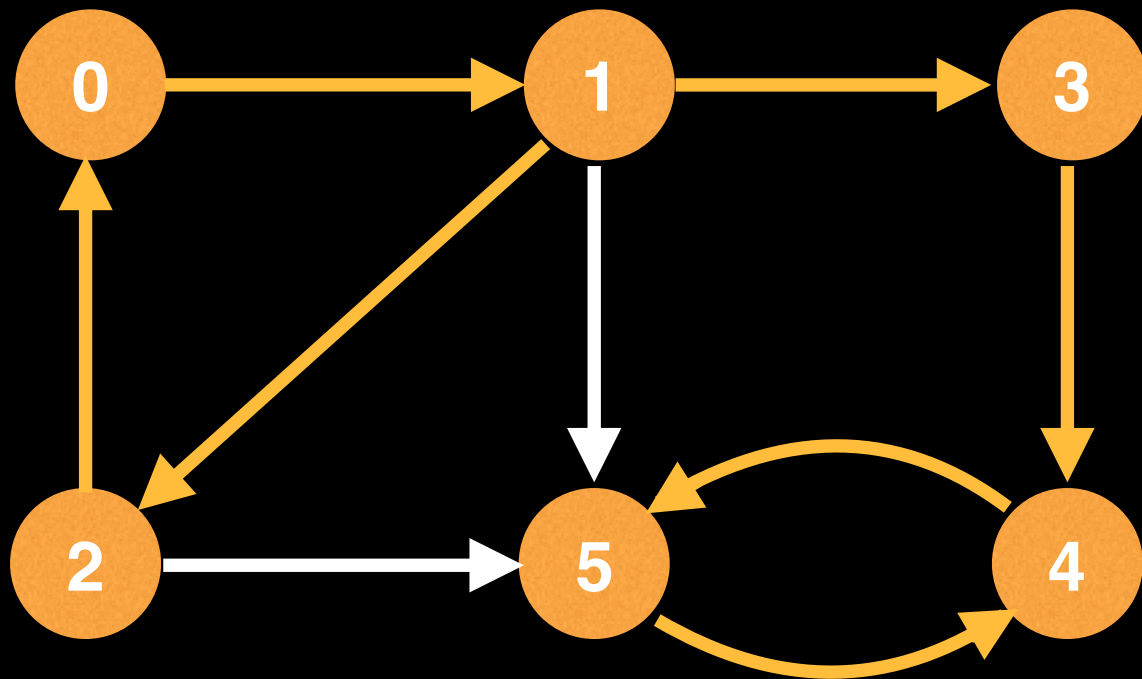
# Low-Link Values

The **low-link** value of a node is the smallest [lowest] node id reachable from that node when doing a DFS (including itself).



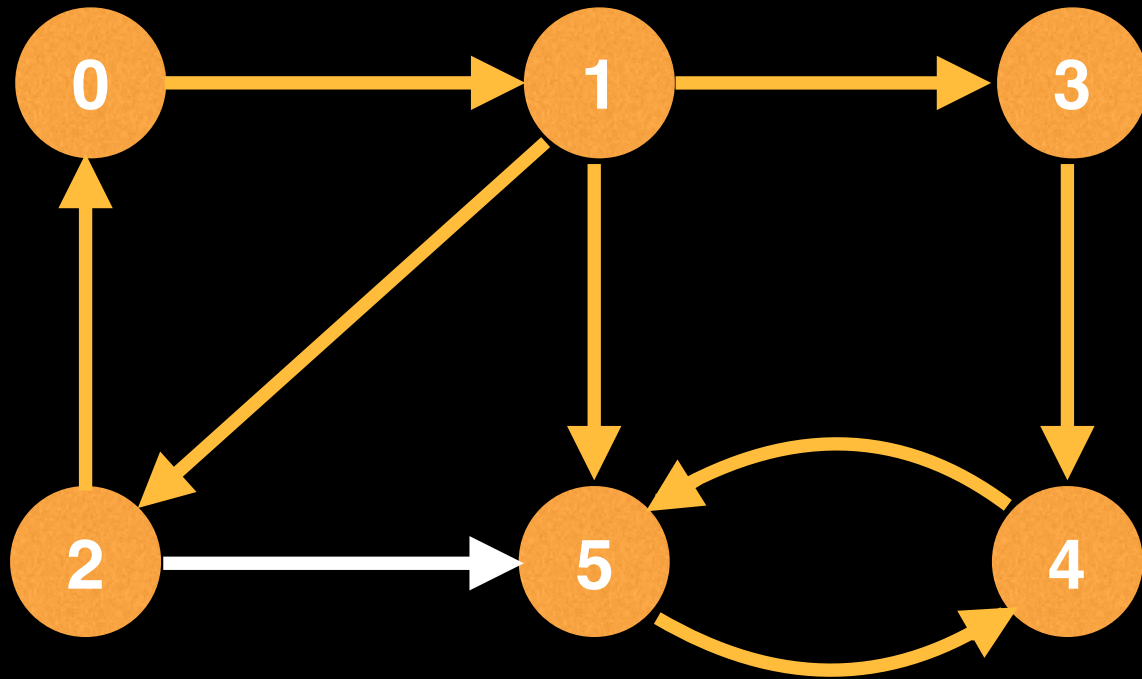
# Low-Link Values

The **low-link** value of a node is the smallest [lowest] node id reachable from that node when doing a DFS (including itself).



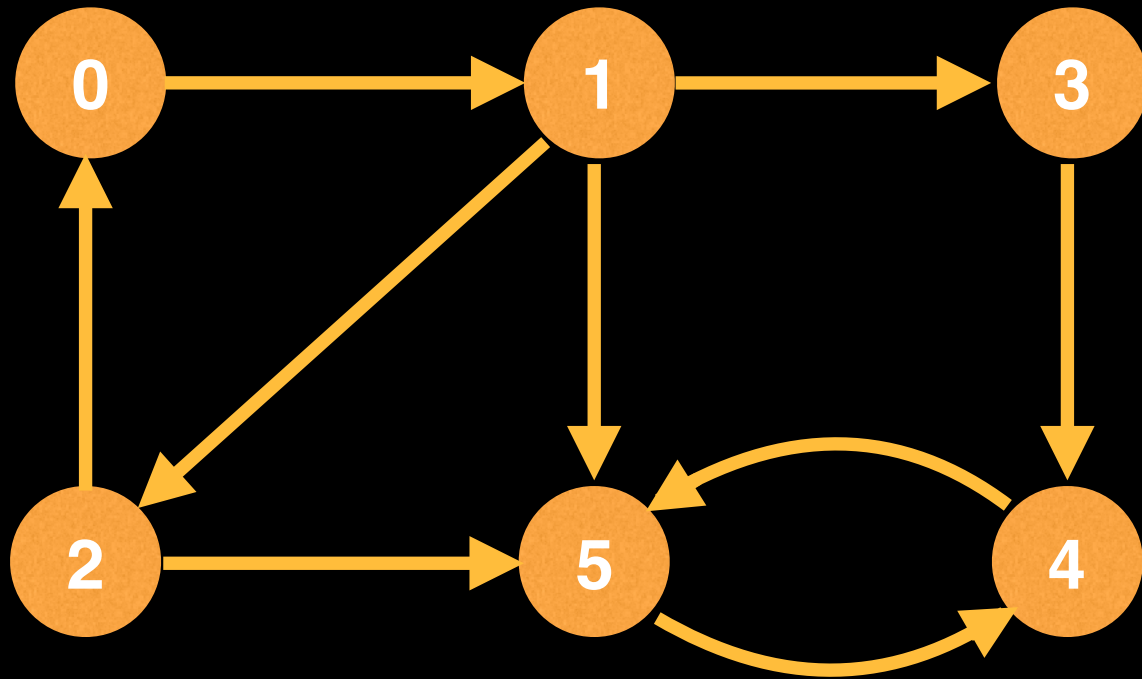
# Low-Link Values

The **low-link** value of a node is the smallest [lowest] node id reachable from that node when doing a DFS (including itself).



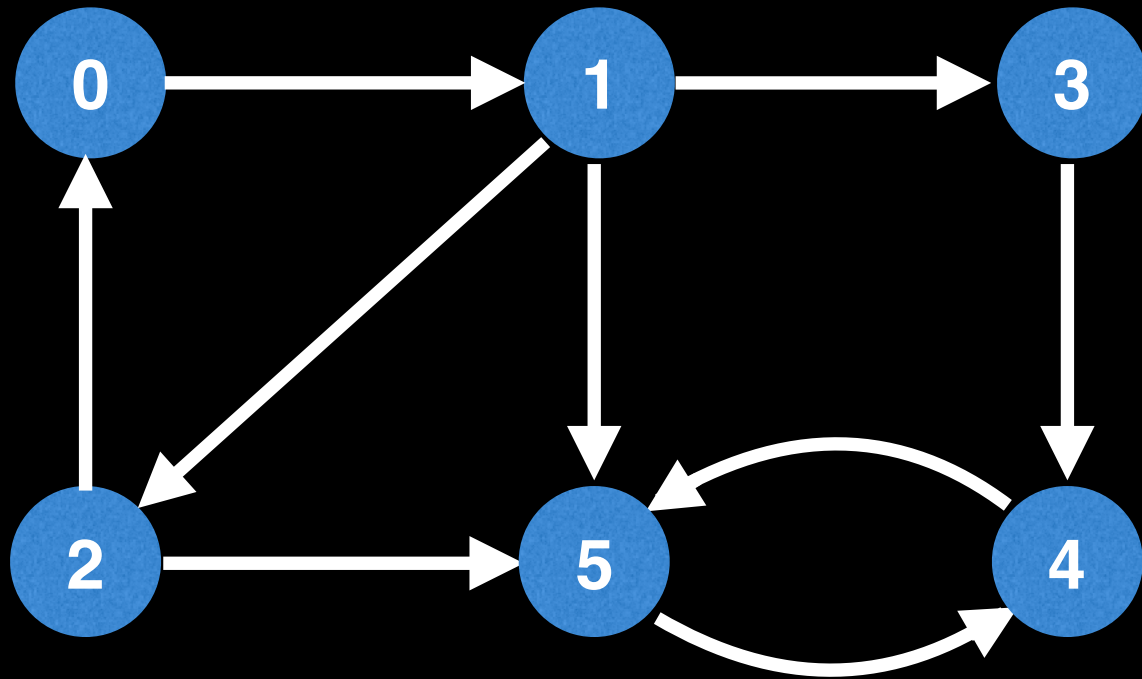
# Low-Link Values

The **low-link** value of a node is the smallest [lowest] node id reachable from that node when doing a DFS (including itself).



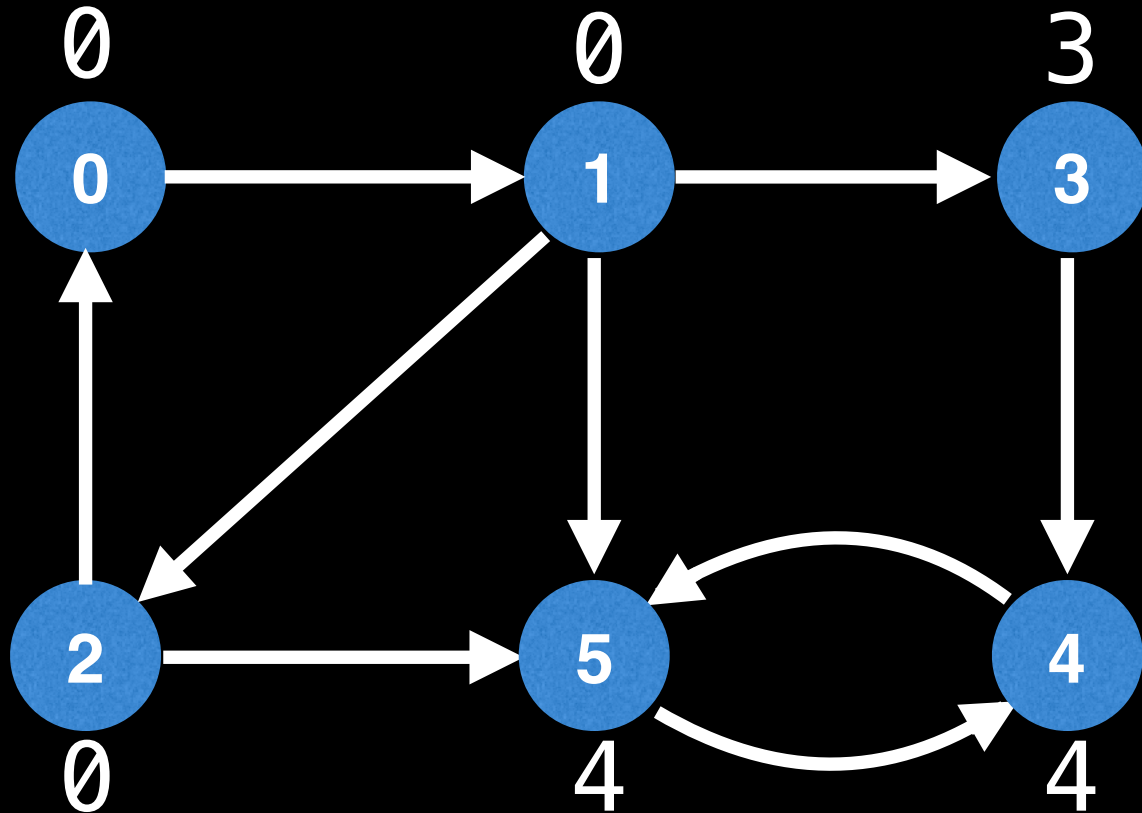
# Low-Link Values

The **low-link** value of a node is the smallest [lowest] node id reachable from that node when doing a DFS (including itself).



# Low-Link Values

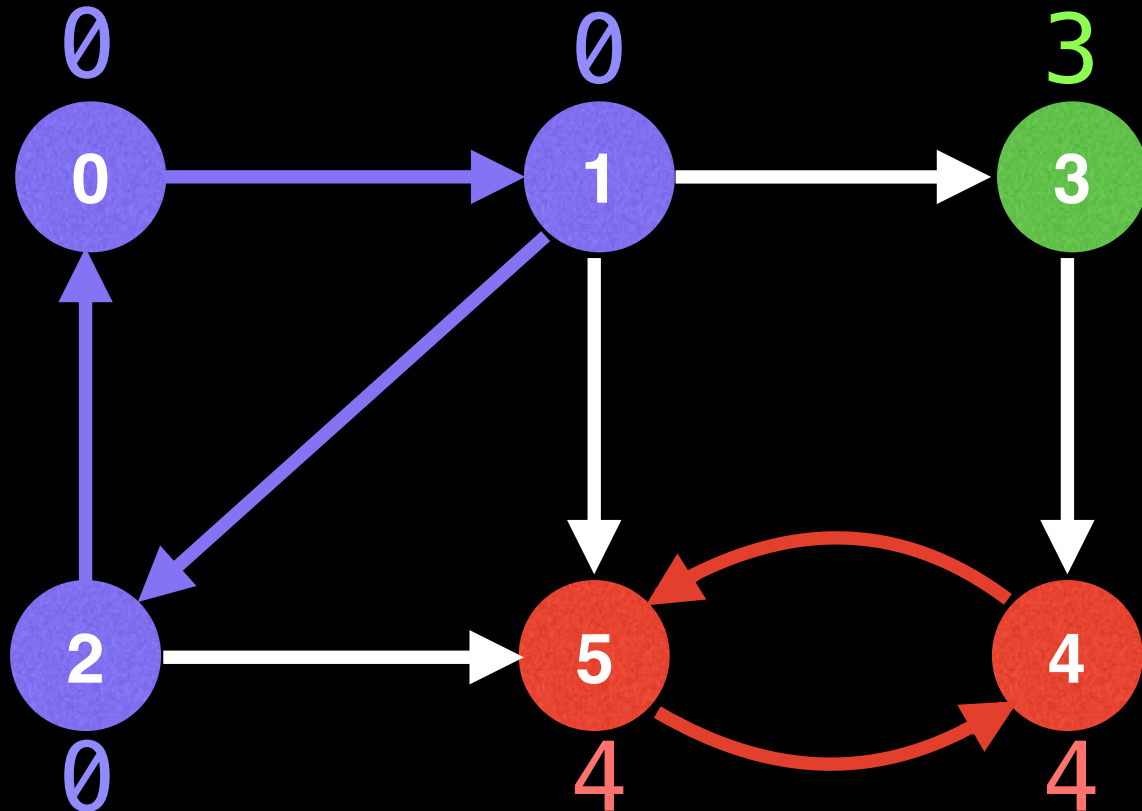
The **low-link** value of a node is the smallest [lowest] node id reachable from that node when doing a DFS (including itself).





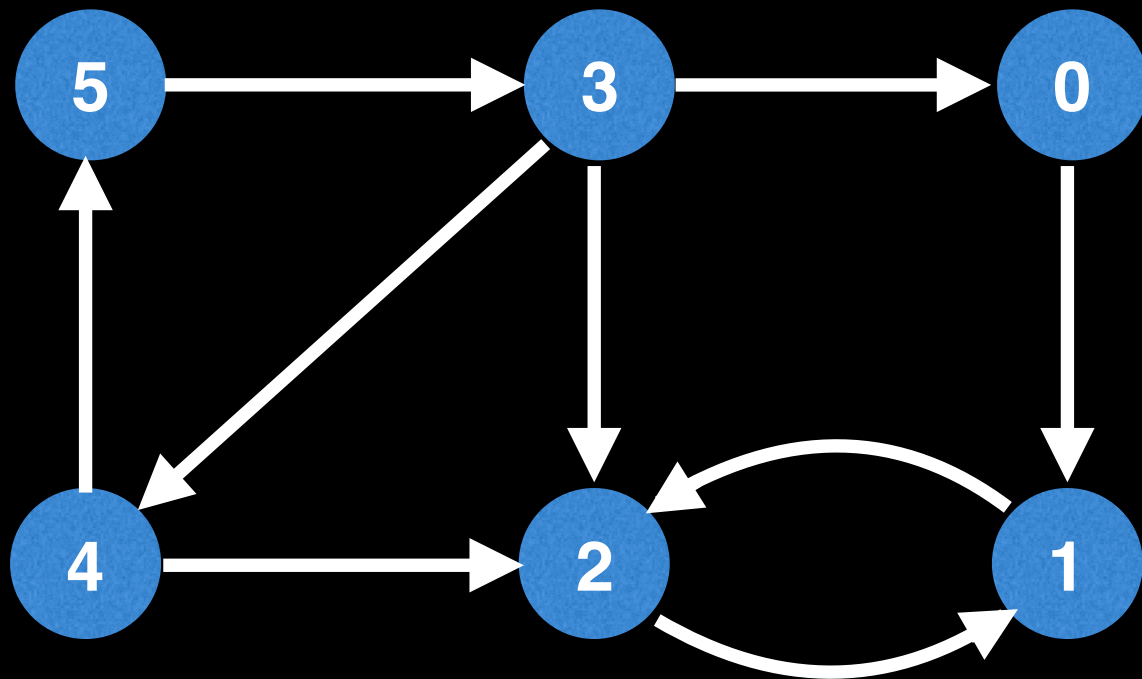
# Low-Link Values

The **low-link** value of a node is the smallest [lowest] node id reachable from that node when doing a DFS (including itself).



# Low-Link Values

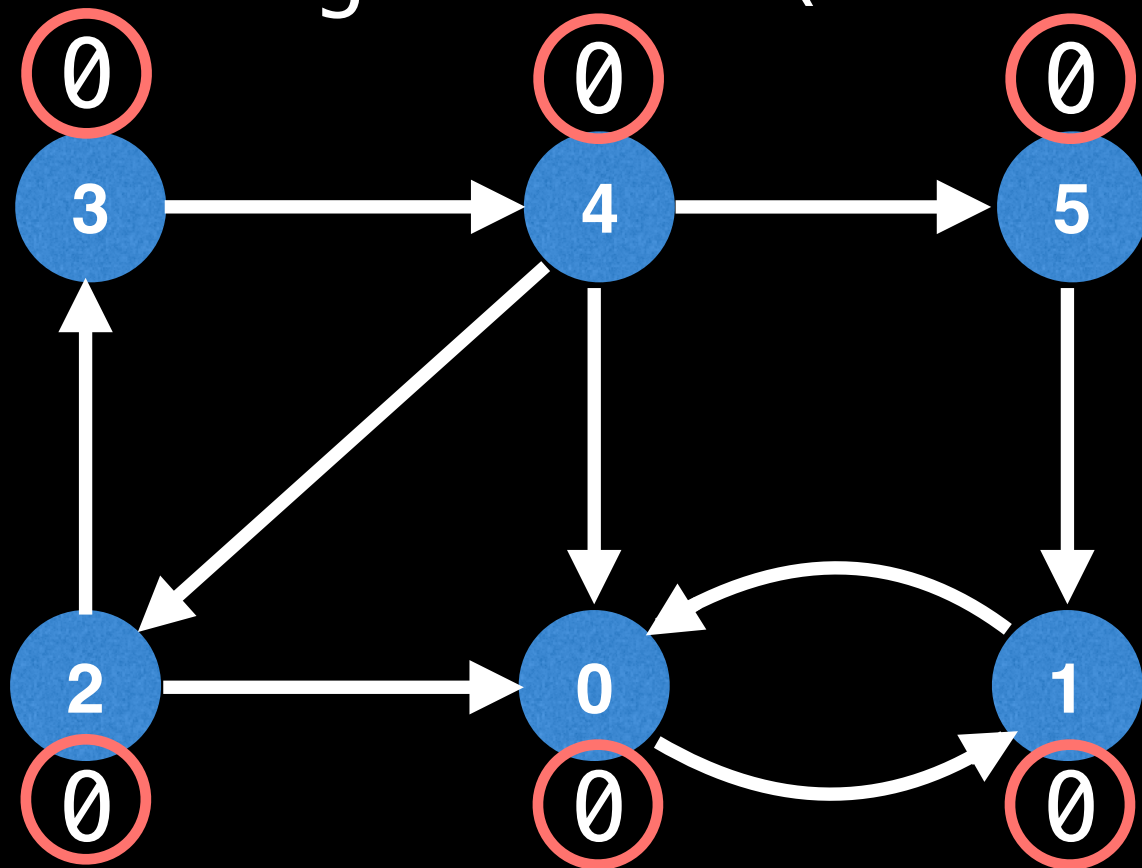
The **low-link** value of a node is the smallest [lowest] node id reachable from that node when doing a DFS (including itself).



**IMPORTANT:** Depending on where the DFS starts and which edges are visited the low-link values could be wrong. In the context of Tarjan's SCC algorithm we maintain an invariant that prevents SCCs to interfere with each others' low-link values.

# Low-Link Values

The **low-link** value of a node is the smallest [lowest] node id reachable from that node when doing a DFS (including itself).



All low link values are the same but there are multiple SCCs!

**IMPORTANT:** Depending on where the DFS starts and which edges are visited the low-link values could be wrong. In the context of Tarjan's SCC algorithm we maintain an invariant that prevents SCCs to interfere with each others' low-link values.

# The Stack Invariant

To cope with the random traversal order of the DFS, Tarjan's algorithm maintains a set (often as a stack) of valid nodes from which to update low-link values from.

Nodes are added to the stack [set] of valid nodes as they're explored for the first time.

Nodes are removed from the stack [set] each time a complete SCC is found.

# New low-link update condition

If  $u$  and  $v$  are nodes in a graph and we're currently exploring  $u$  then our new low-link update condition is that:

To update node  $u$ 's low-link value to node  $v$ 's low-link value there has to be a path of edges from  $u$  to  $v$  and **node  $v$  must be on the stack.**

# Time Complexity

Another difference we're going to make to finding all low-link values is that instead of finding low-link values after the fact we're going to update them "on the fly" during the DFS so we can get a linear  $O(V+E)$  time complexity :)

# Tarjan's Algorithm Overview

Mark the id of each node as unvisited.

Start DFS. Upon visiting a node assign it an id and a low-link value. Also mark current nodes as visited and add them to a seen stack.

On DFS callback, if the previous node is on the stack then min the current node's low-link value with the last node's low-link value\*.

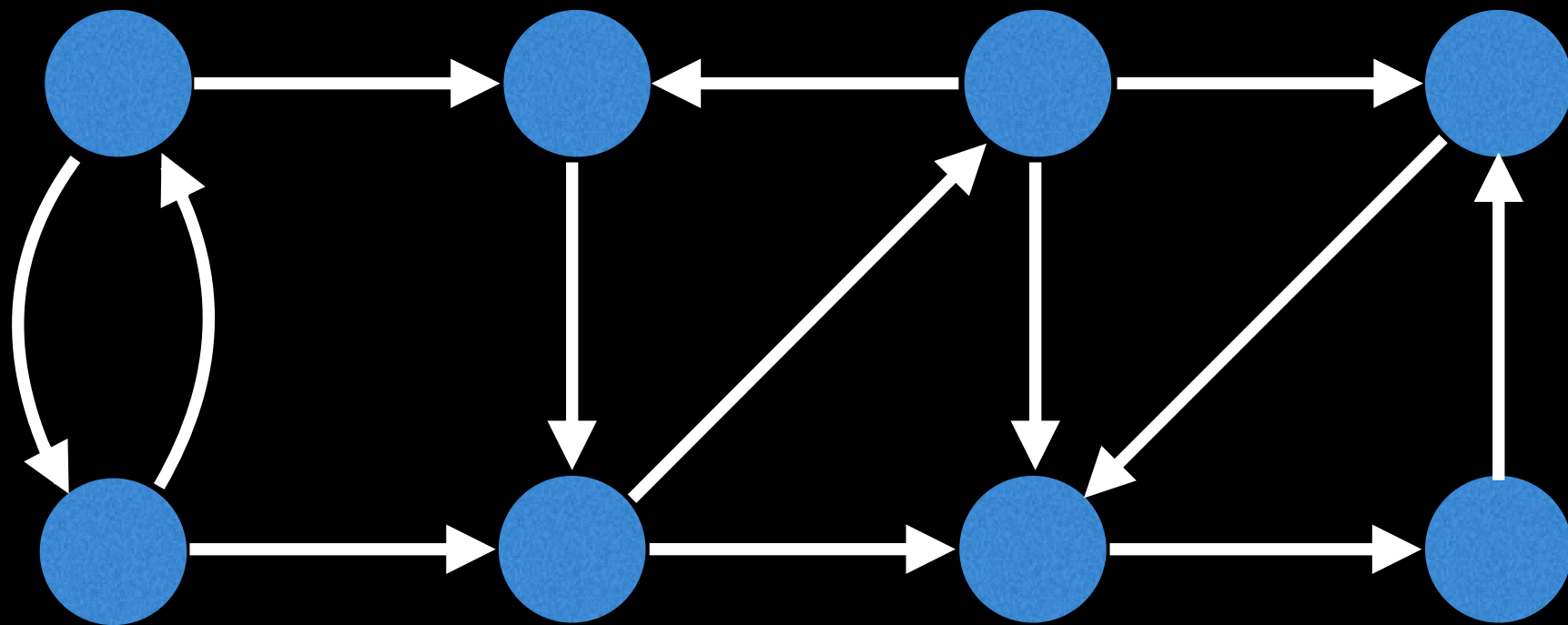
After visiting all neighbors, if the current node started a connected component\*\* then pop nodes off stack until current node is reached.

\*This allows low-link values to propagate throughout cycles.

\*\*As we will see, a node started a connected component if its **id equals its low link value**

● Unvisited      ● Visiting neighbours      ● Visited all neighbours

Stack

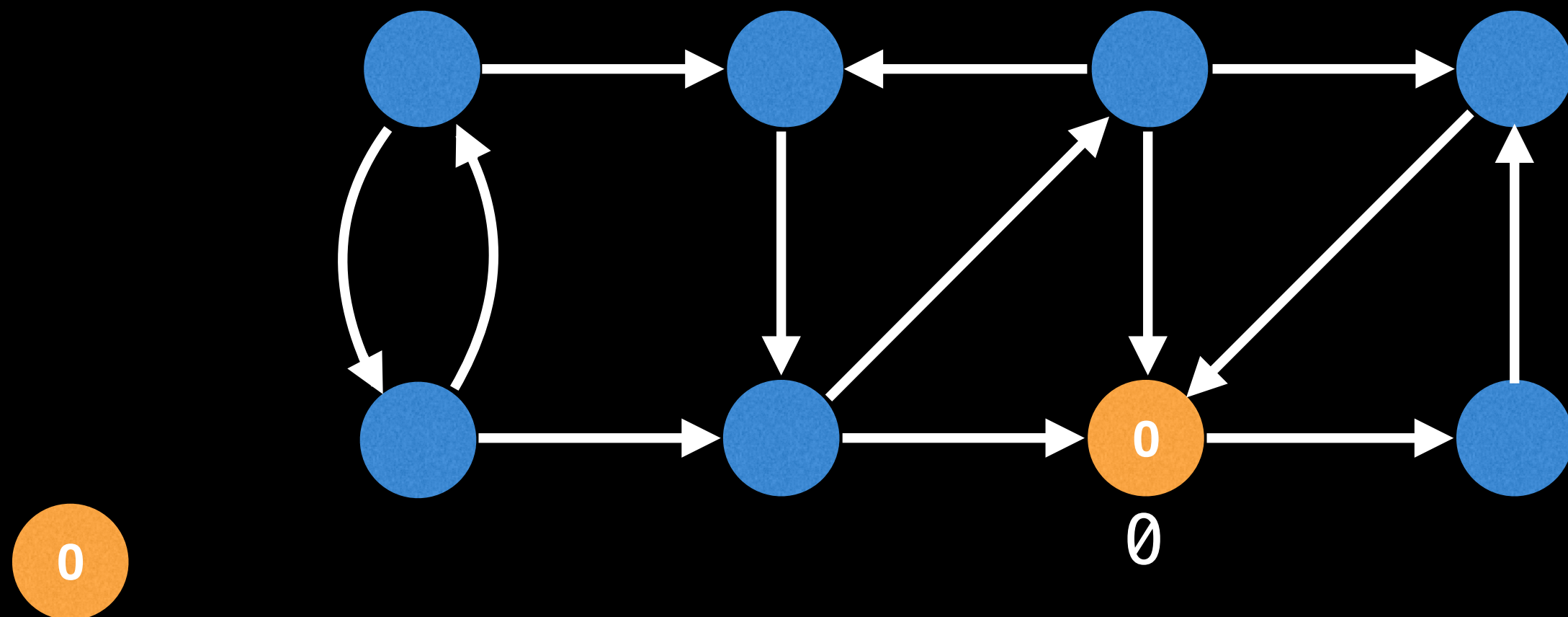


If a node's colour is **grey** or **orange** then it is on the stack and we can update its low-link value.



● Unvisited      ● Visiting neighbours      ● Visited all neighbours

Stack



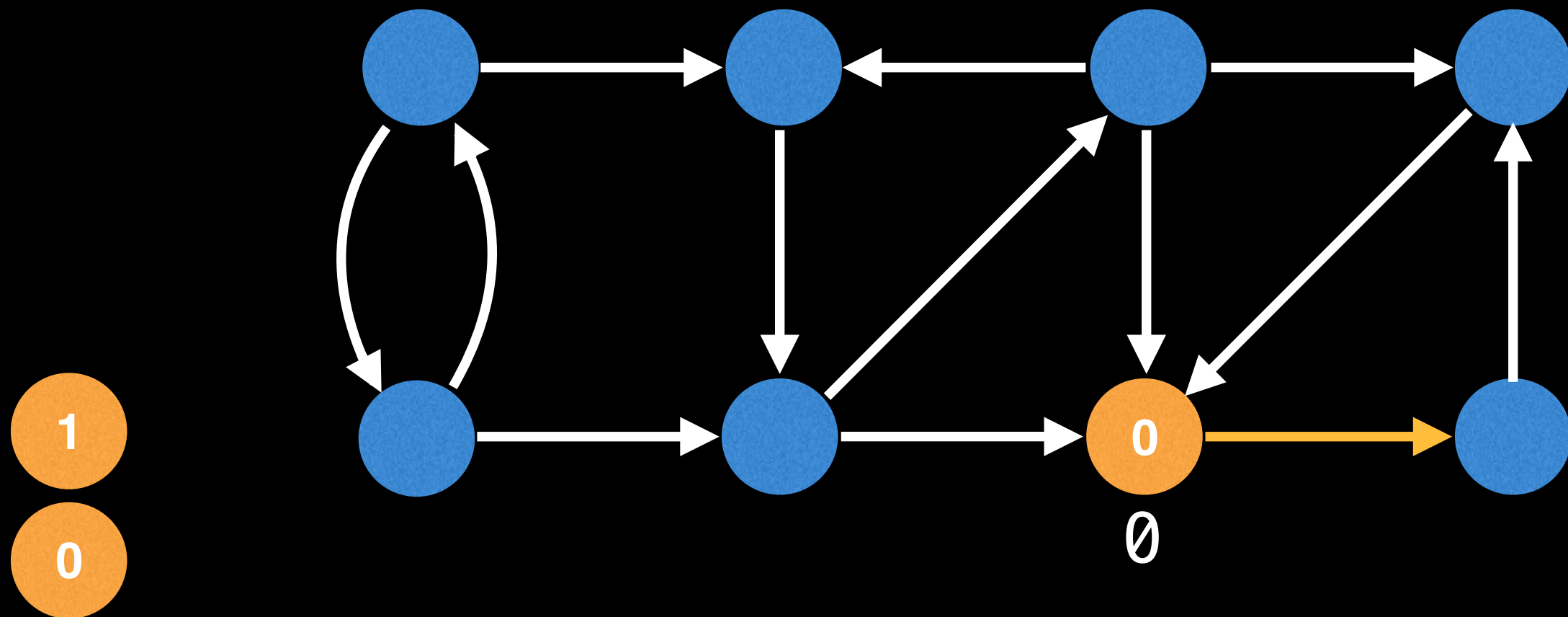
Start DFS anywhere.



## visiting neighbours

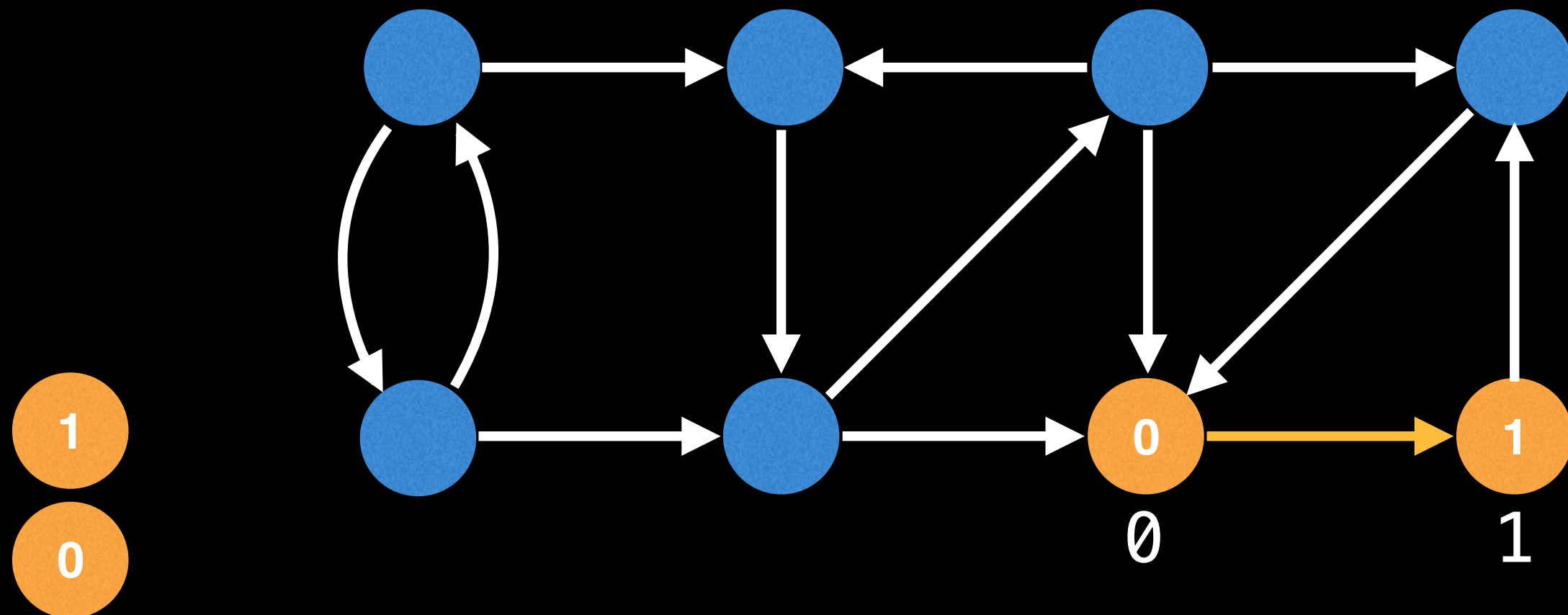
- Visited all neighbours

# Stack



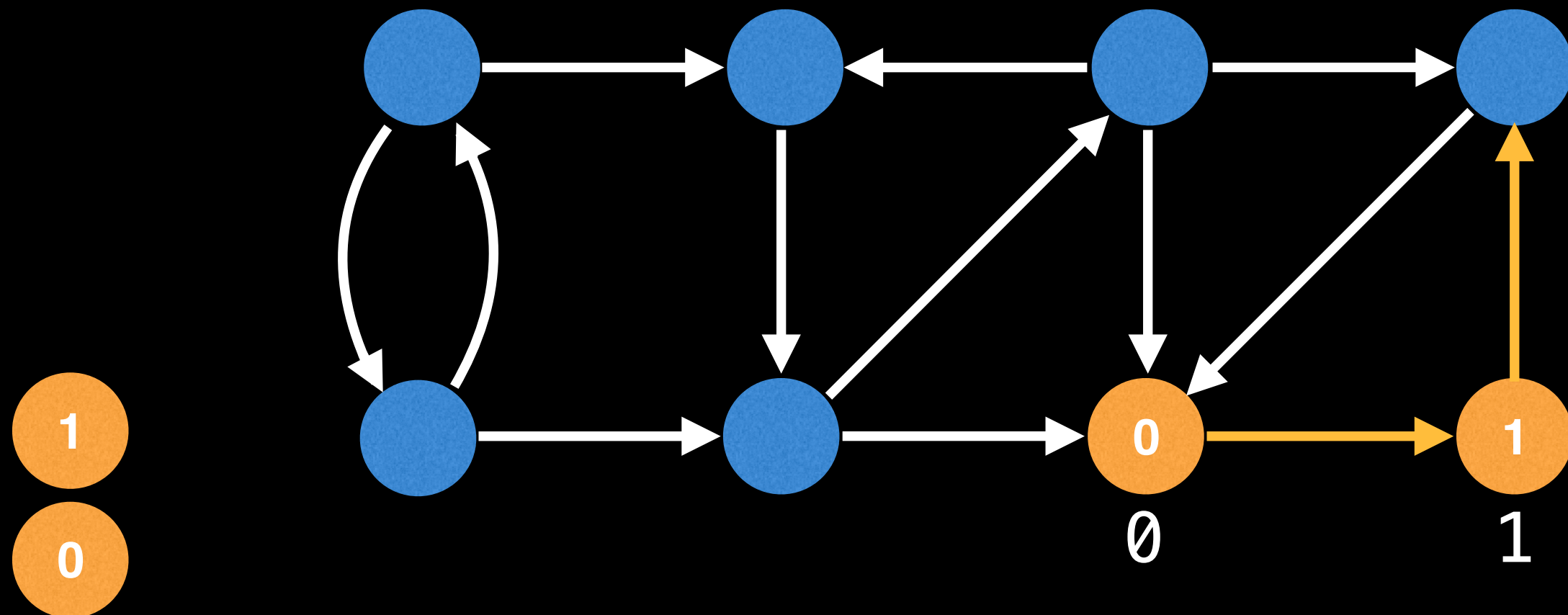
● Unvisited      ● Visiting neighbours      ● Visited all neighbours

Stack



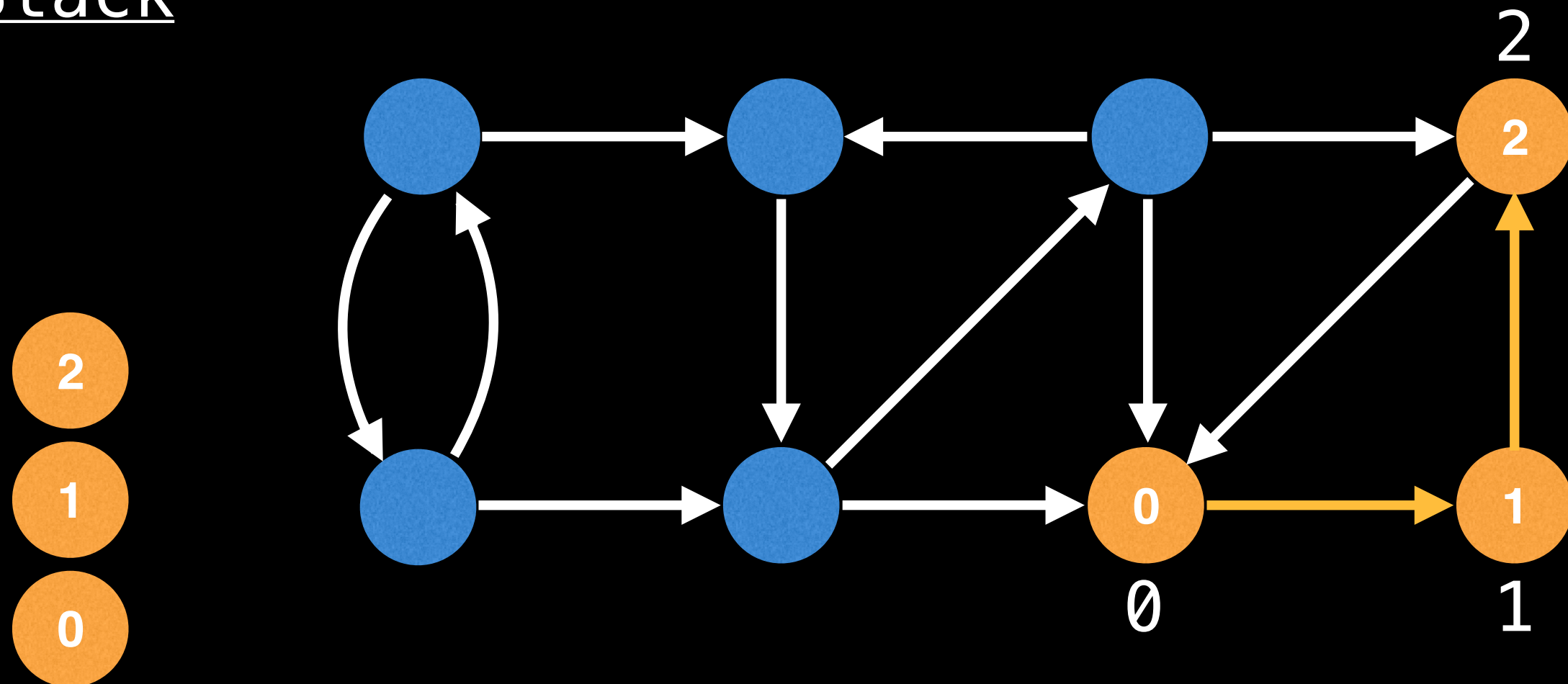
● Unvisited      ● Visiting neighbours      ● Visited all neighbours

Stack



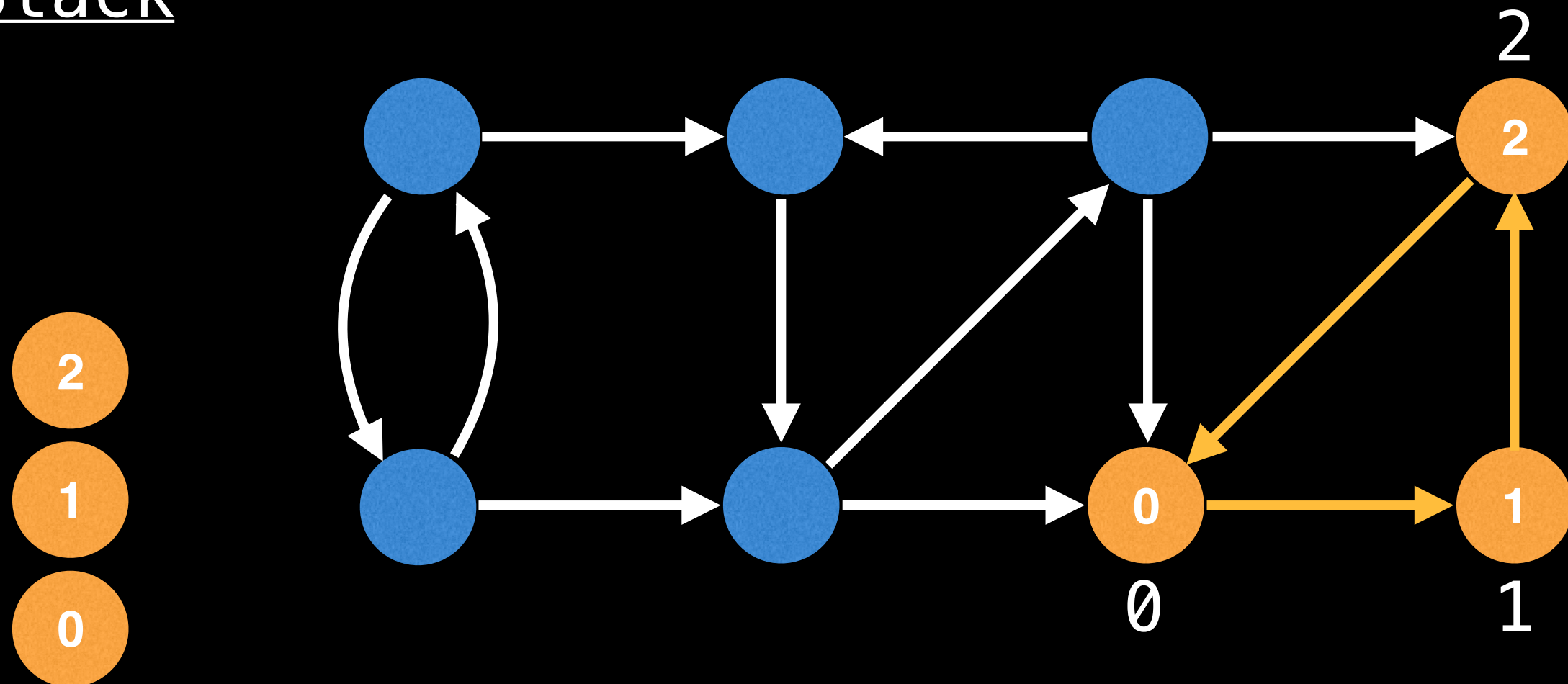
● Unvisited      ● Visiting neighbours      ● Visited all neighbours

Stack



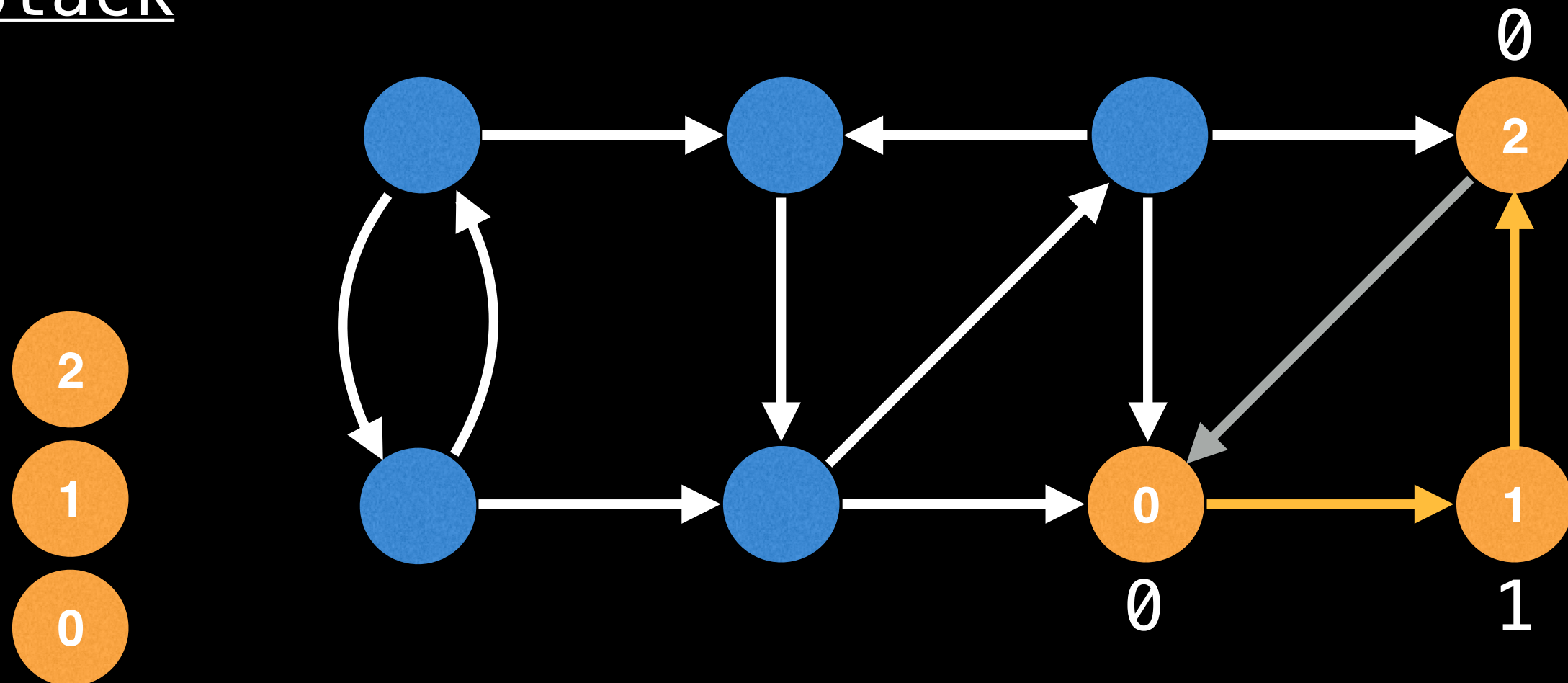
● Unvisited      ● Visiting neighbours      ● Visited all neighbours

Stack



● Unvisited     
 ● Visiting neighbours     
 ● Visited all neighbours

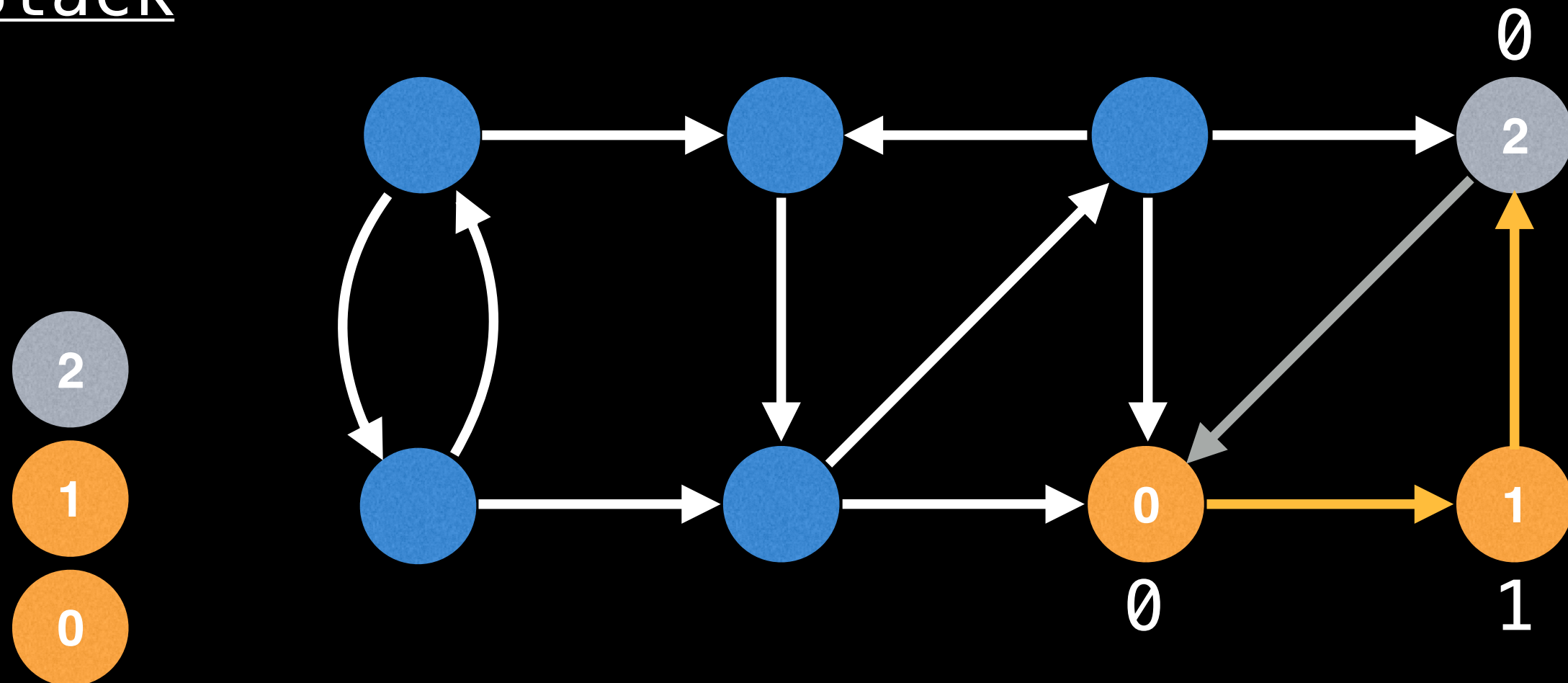
Stack



$\text{lowlink}[2] = \min(\text{lowlink}[2], \text{lowlink}[0])$   
 $= 0$

● Unvisited      ● Visiting neighbours      ● Visited all neighbours

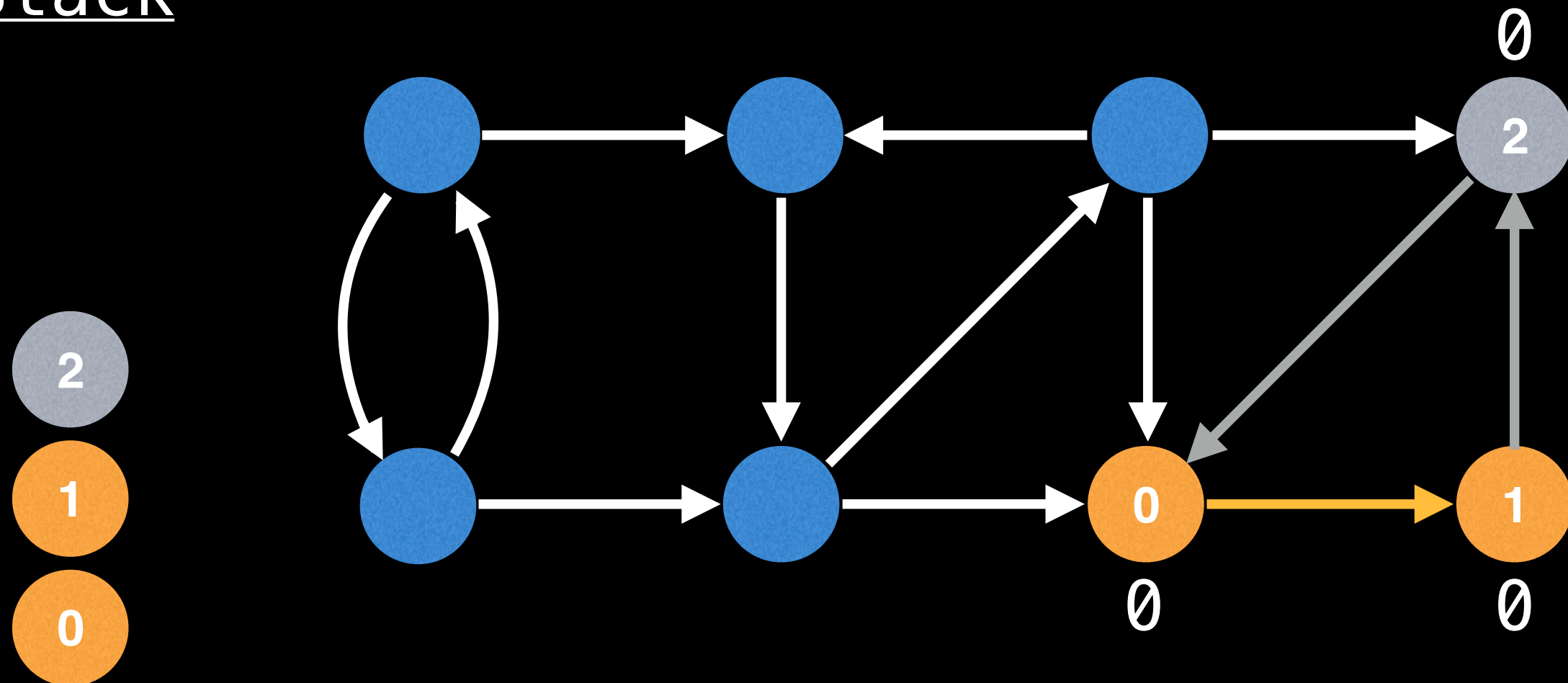
Stack





● Unvisited     
 ● Visiting neighbours     
 ● Visited all neighbours

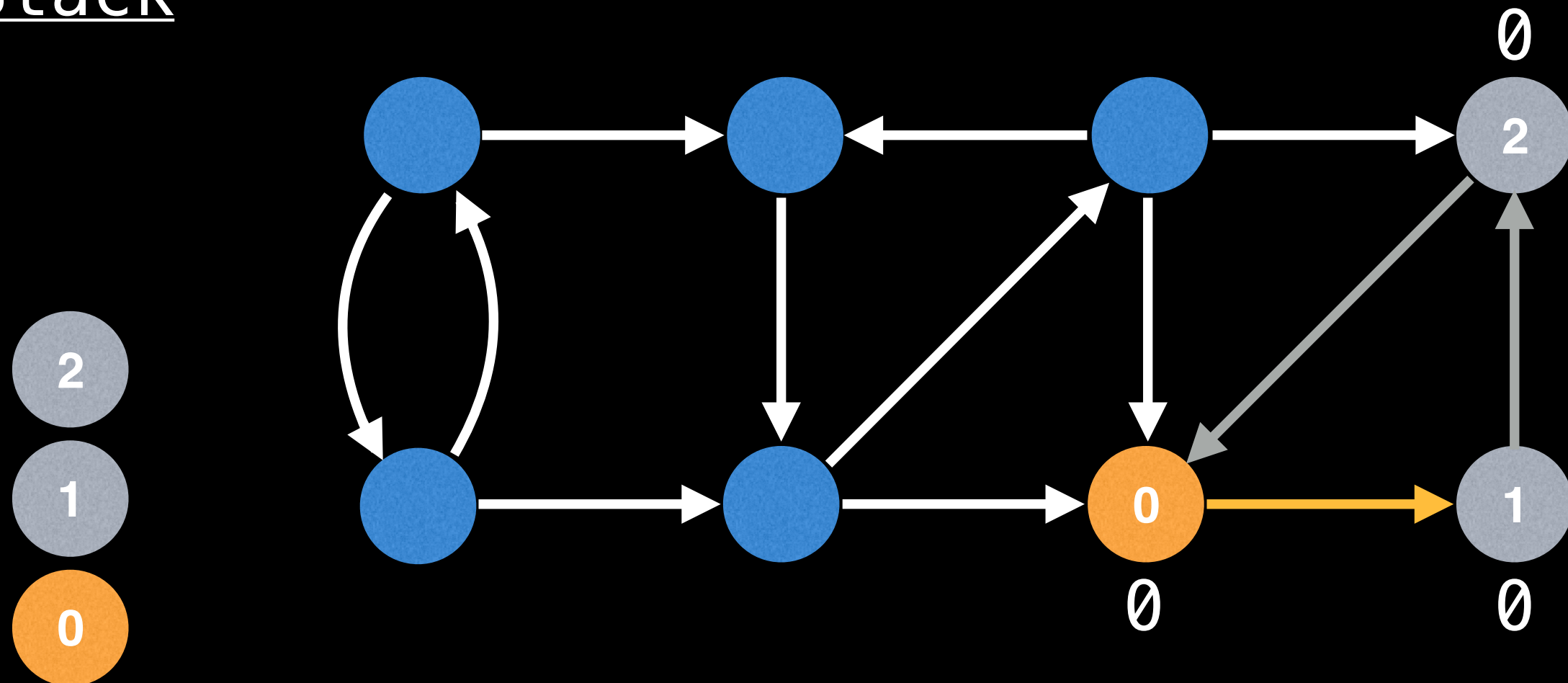
Stack



$\text{lowlink}[1] = \min(\text{lowlink}[1], \text{lowlink}[2])$   
 $= 0$

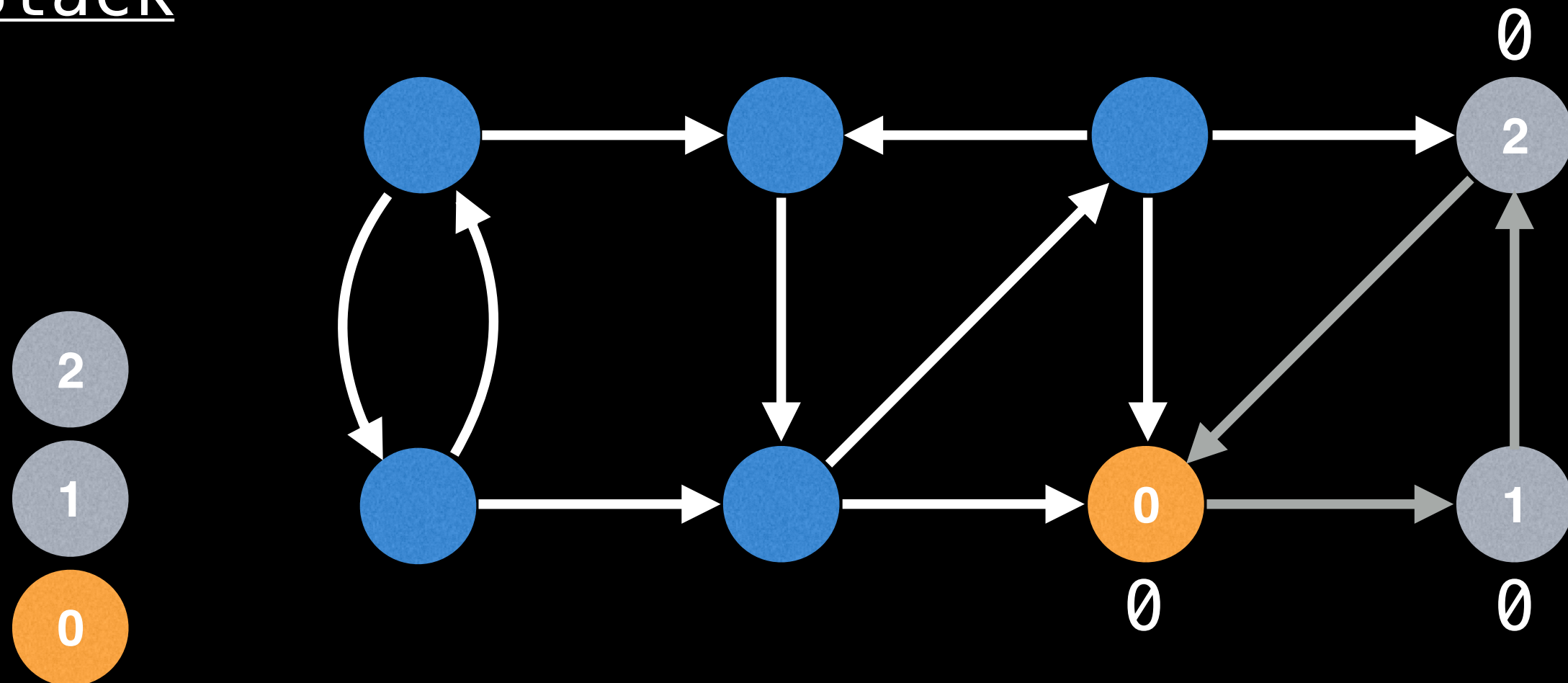
● Unvisited      ● Visiting neighbours      ● Visited all neighbours

Stack



● Unvisited     
 ● Visiting neighbours     
 ● Visited all neighbours

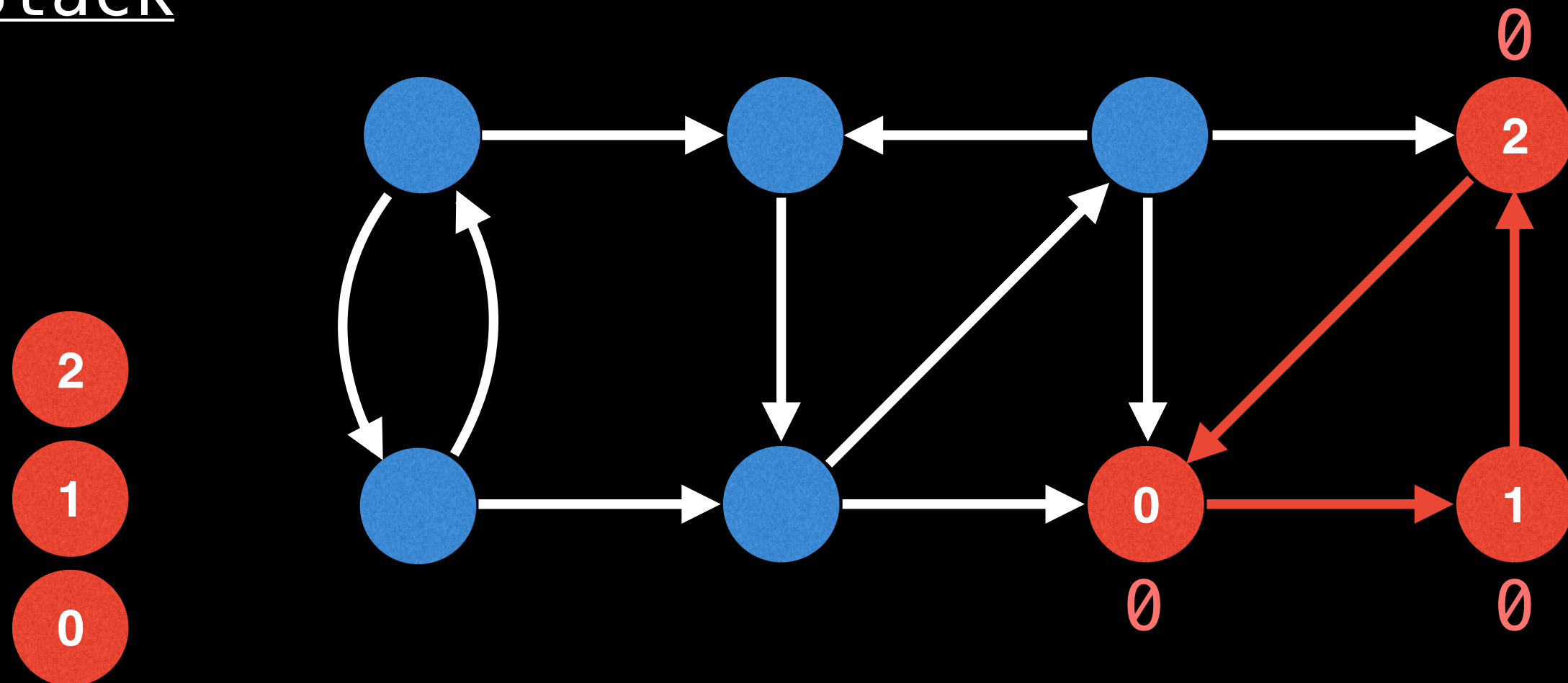
Stack



$\text{lowlink}[0] = \min(\text{lowlink}[0], \text{lowlink}[1])$   
 $= 0$

● Unvisited      ● Visiting neighbours      ● Visited all neighbours

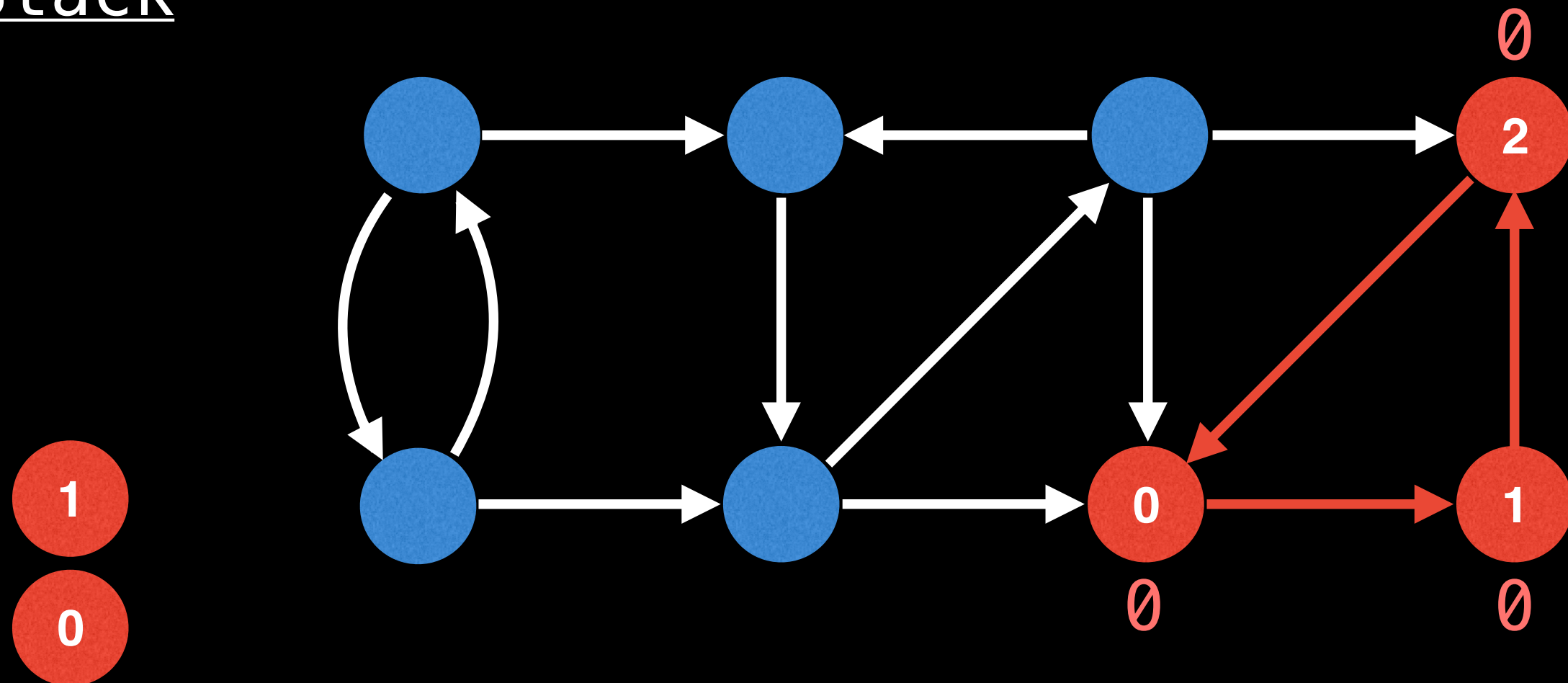
Stack



When a completed SCC is found (current node has visited all its neighbours and its lowlink value equals its id) pop off all associated nodes off the stack.

● Unvisited      ● Visiting neighbours      ● Visited all neighbours

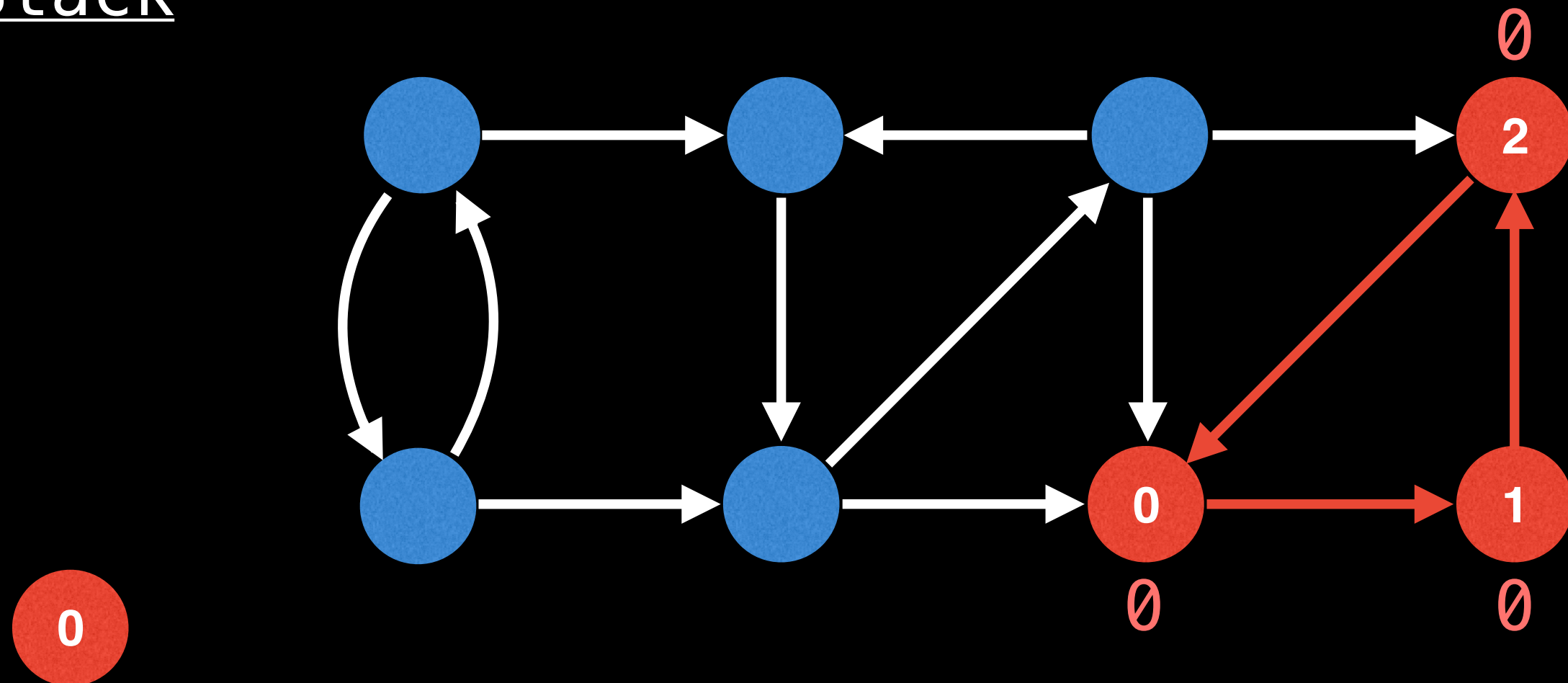
Stack



When a completed SCC is found (current node has visited all its neighbours and its lowlink value equals its id) pop off all associated nodes off the stack.

● Unvisited      ● Visiting neighbours      ● Visited all neighbours

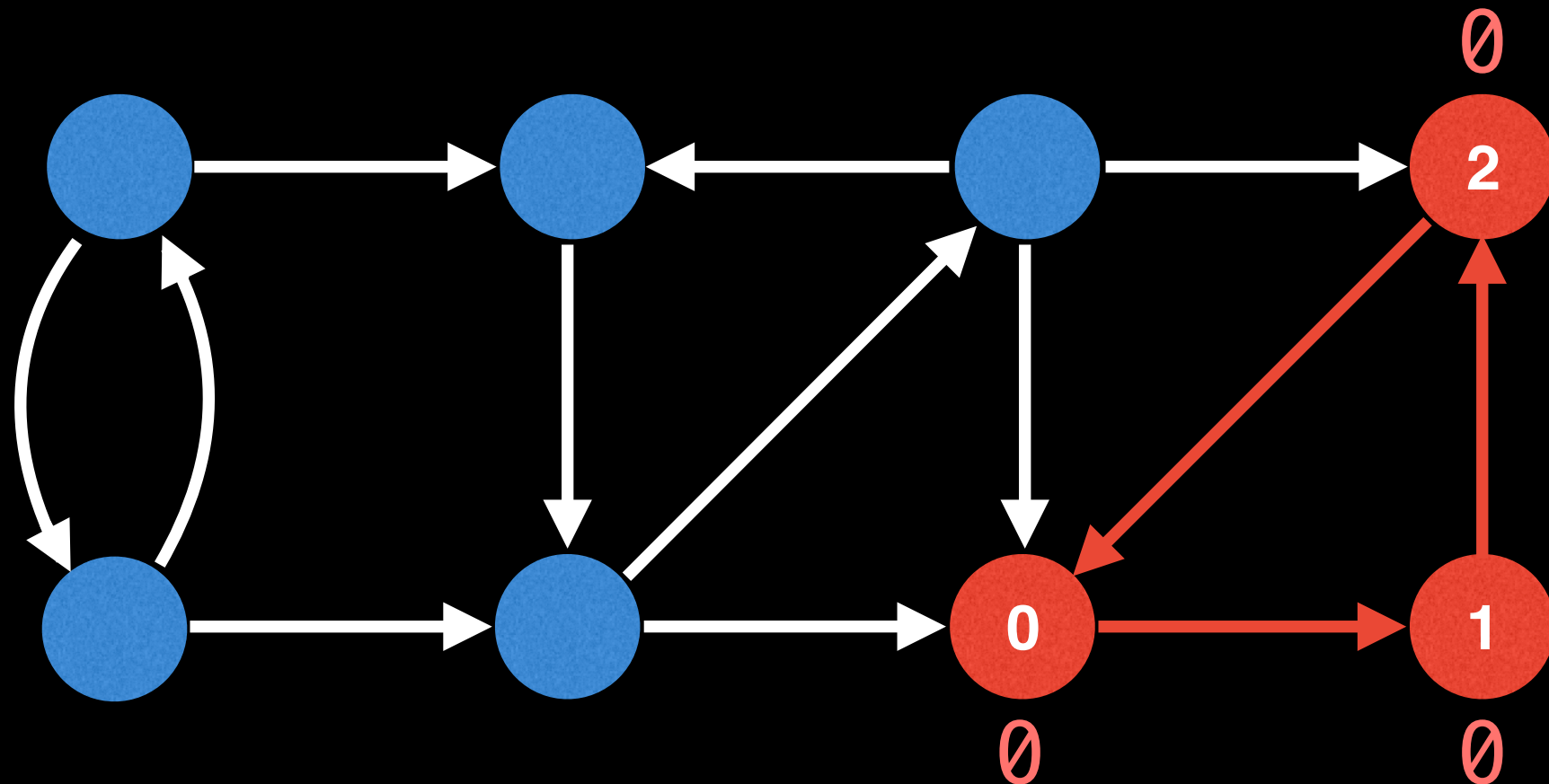
Stack



When a completed SCC is found (current node has visited all its neighbours and its lowlink value equals its id) pop off all associated nodes off the stack.

● Unvisited      ● Visiting neighbours      ● Visited all neighbours

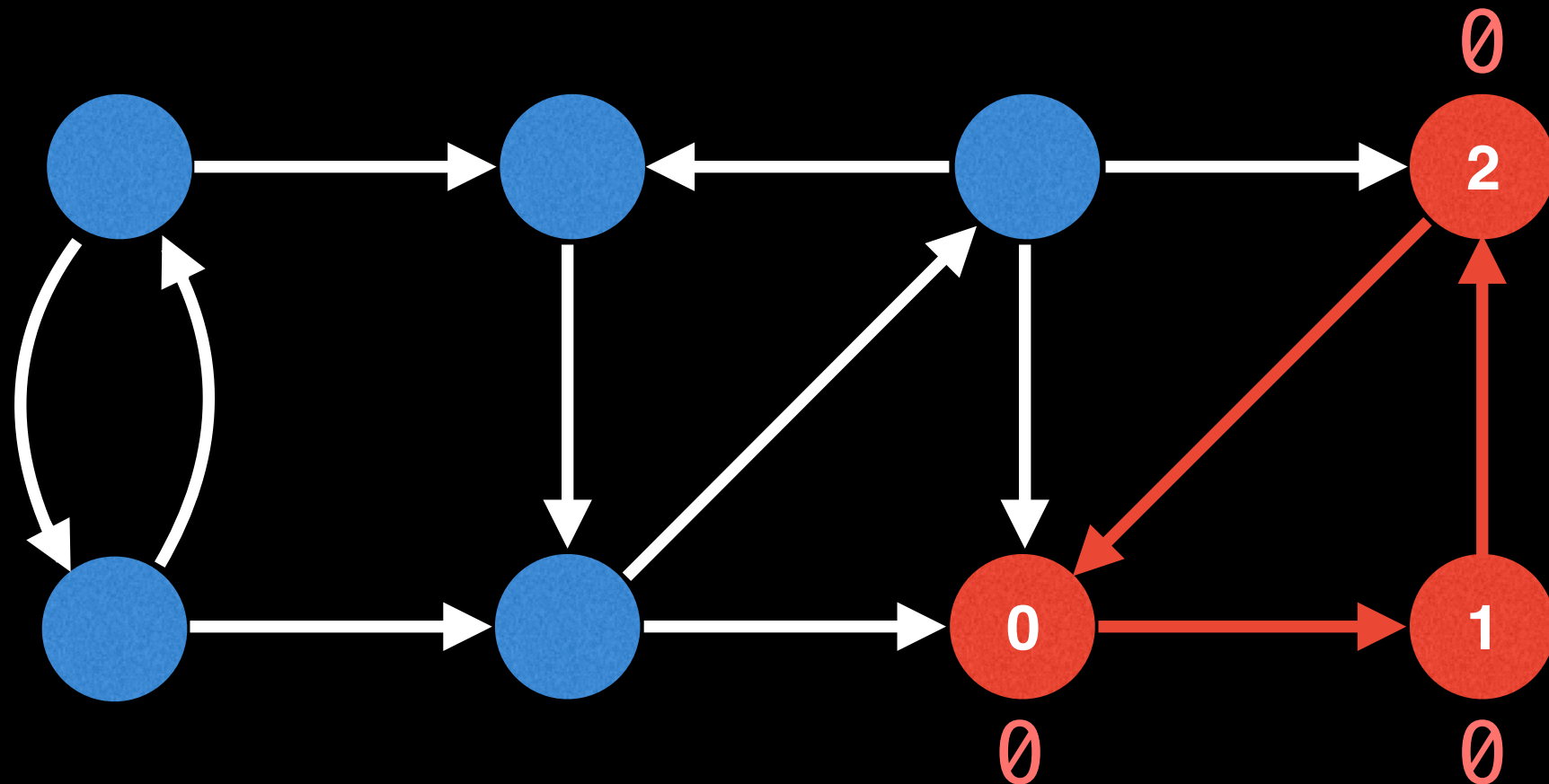
Stack



When a completed SCC is found (current node has visited all its neighbours and its lowlink value equals its id) pop off all associated nodes off the stack.

● Unvisited      ● Visiting neighbours      ● Visited all neighbours

Stack

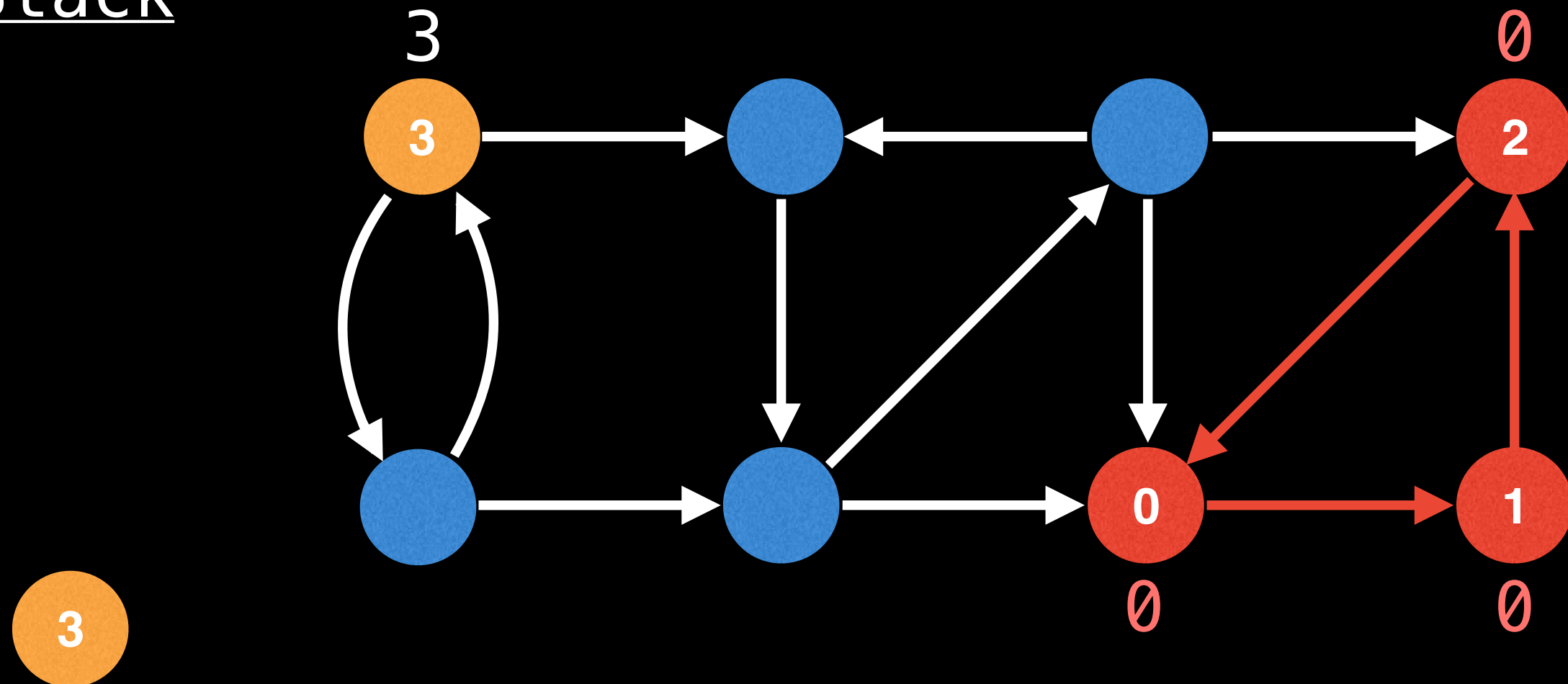


We're not done exploring the graph so pick another starting node at random.



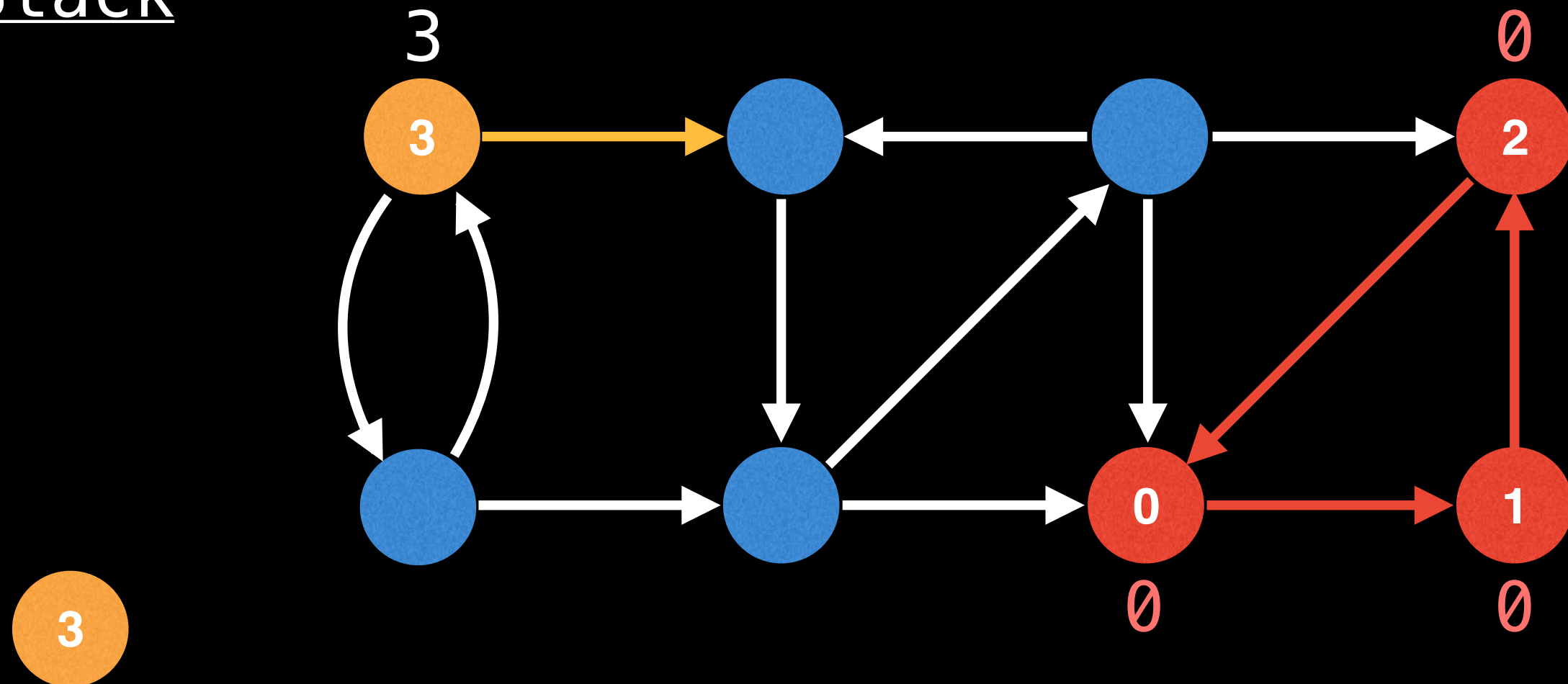
● Unvisited      ● Visiting neighbours      ● Visited all neighbours

Stack



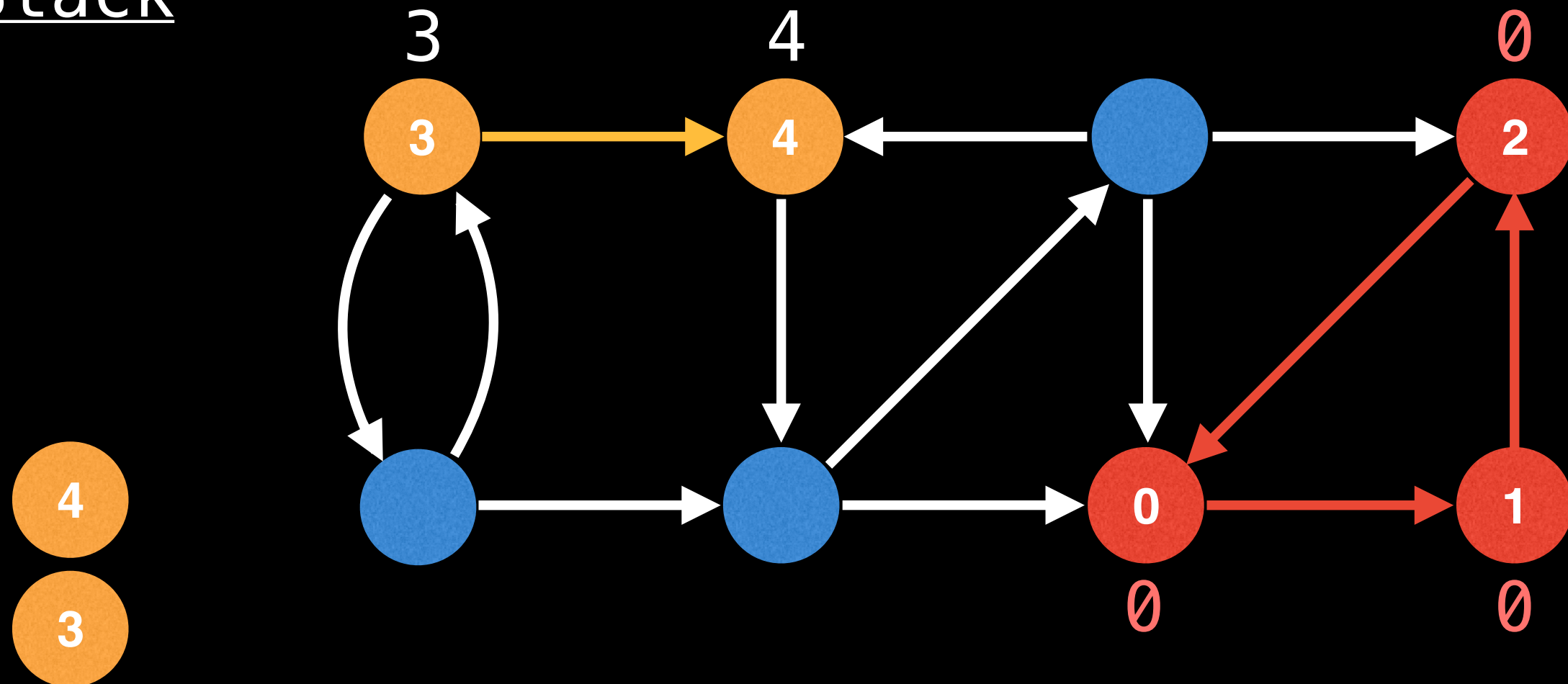
● Unvisited      ● Visiting neighbours      ● Visited all neighbours

Stack



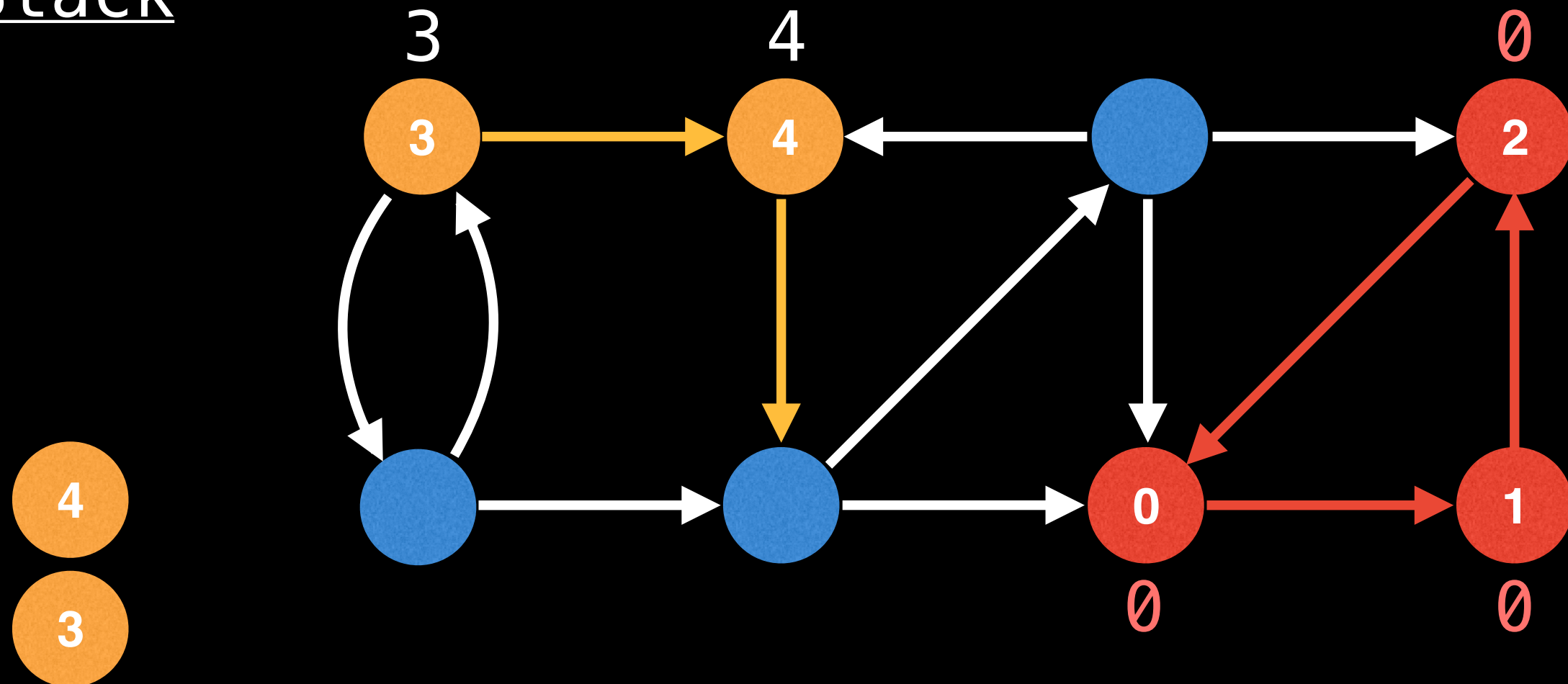
● Unvisited      ● Visiting neighbours      ● Visited all neighbours

Stack



● Unvisited      ● Visiting neighbours      ● Visited all neighbours

Stack

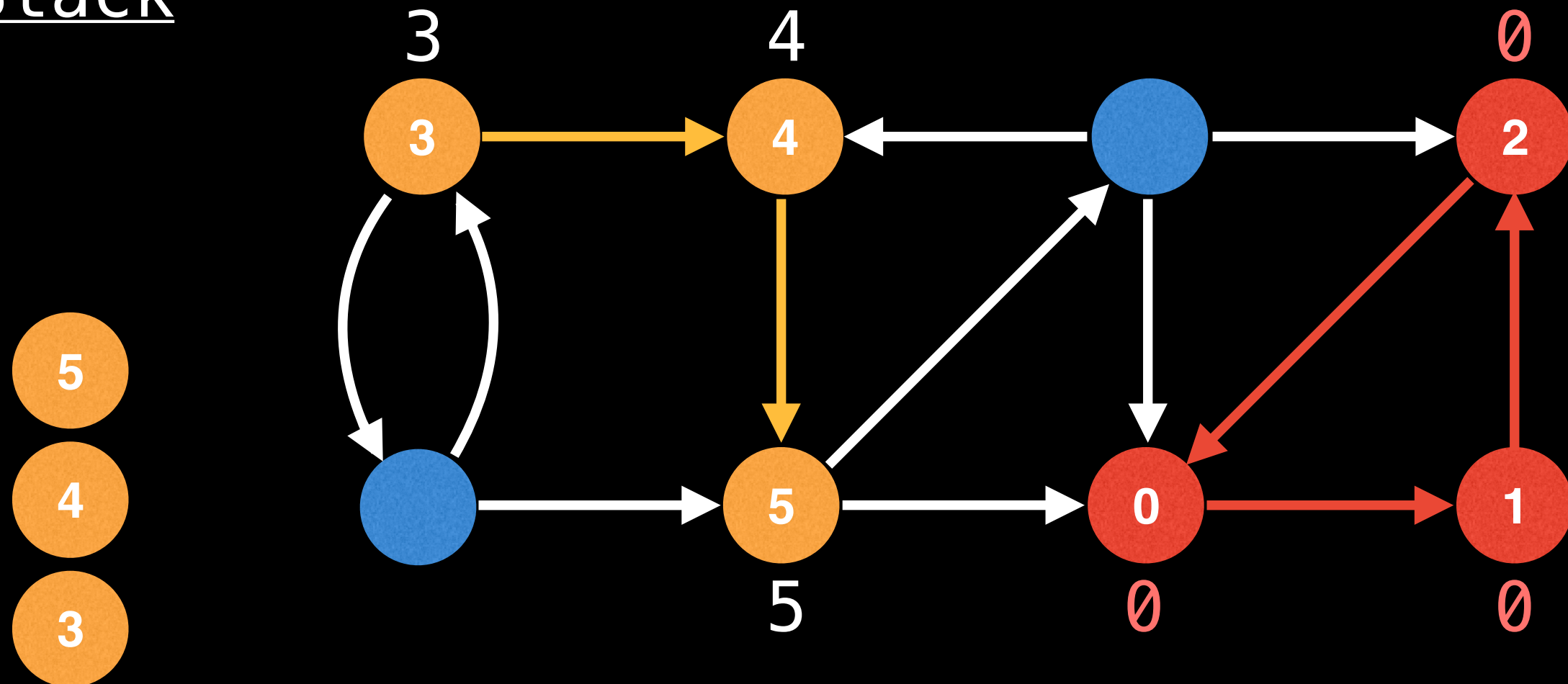


 Unvisited

# visiting neighbours

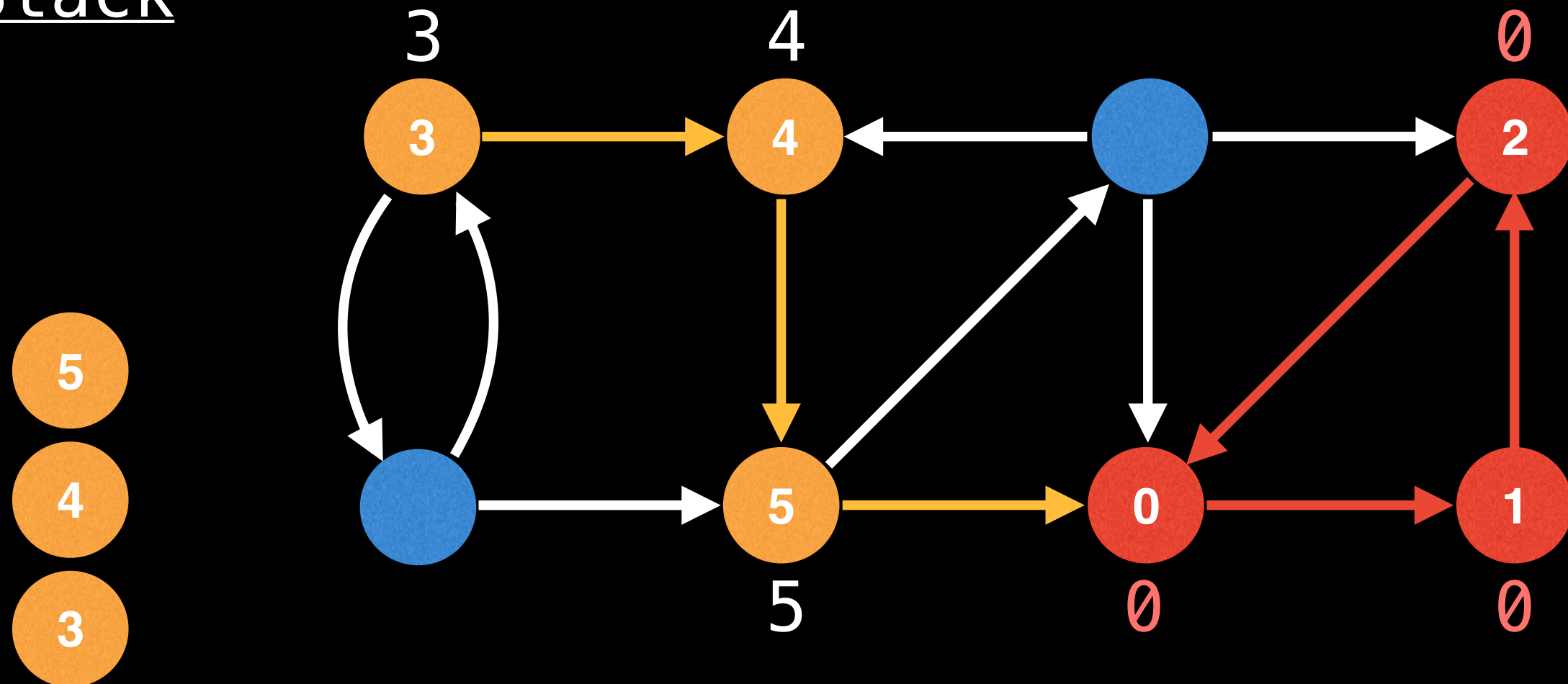
● visited all neighbours

# Stack



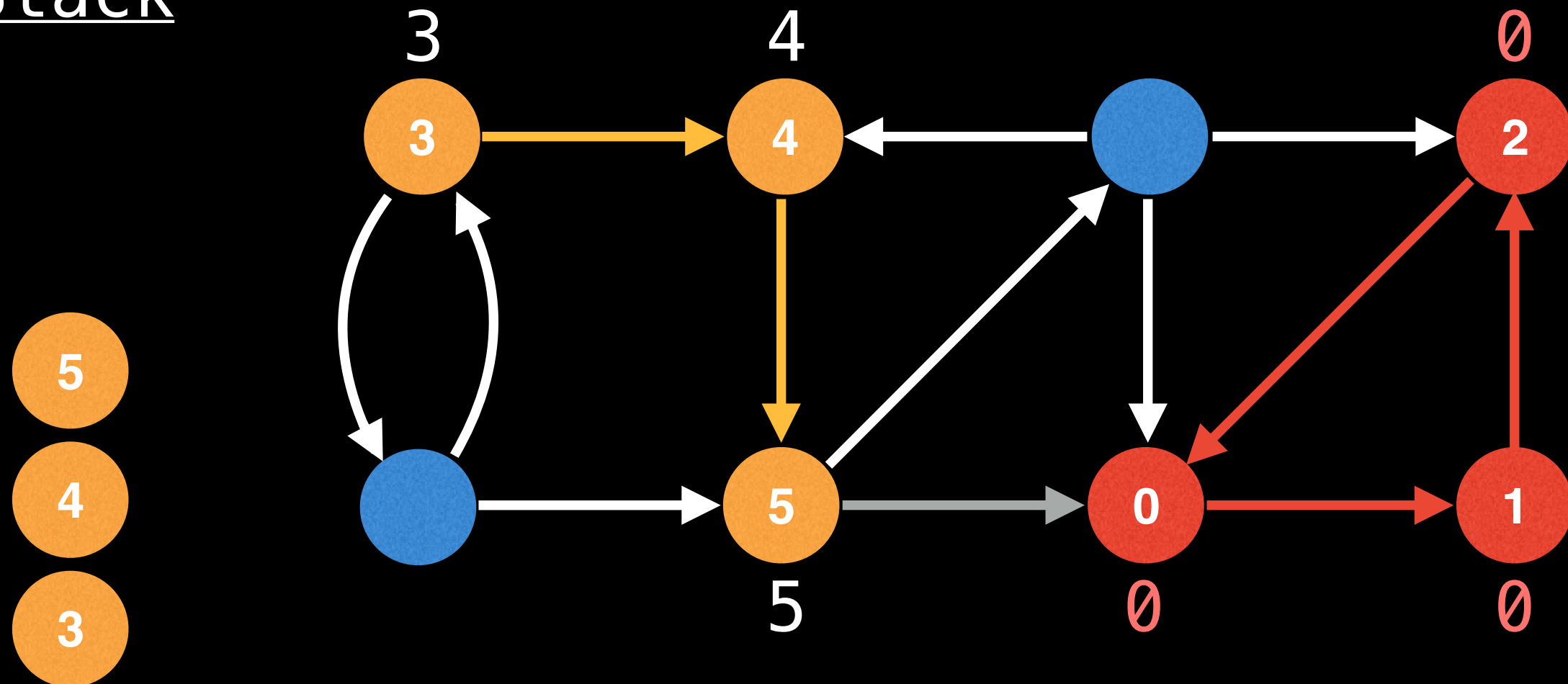
● Unvisited      ● Visiting neighbours      ● Visited all neighbours

Stack



● Unvisited      ● Visiting neighbours      ● Visited all neighbours

Stack



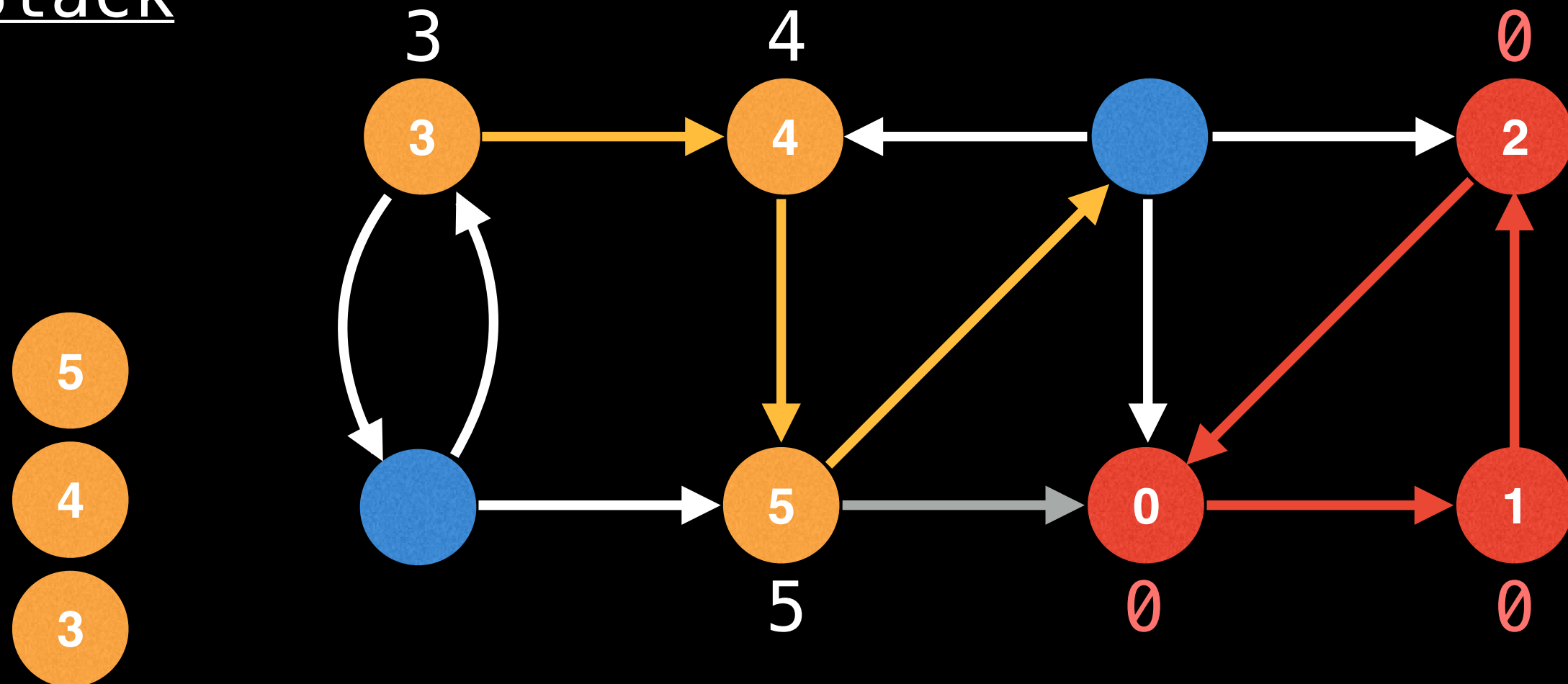
Node 0 is not on stack so don't min  
with its low-link value.

 Unvisited

# visiting neighbours

● visited all neighbours

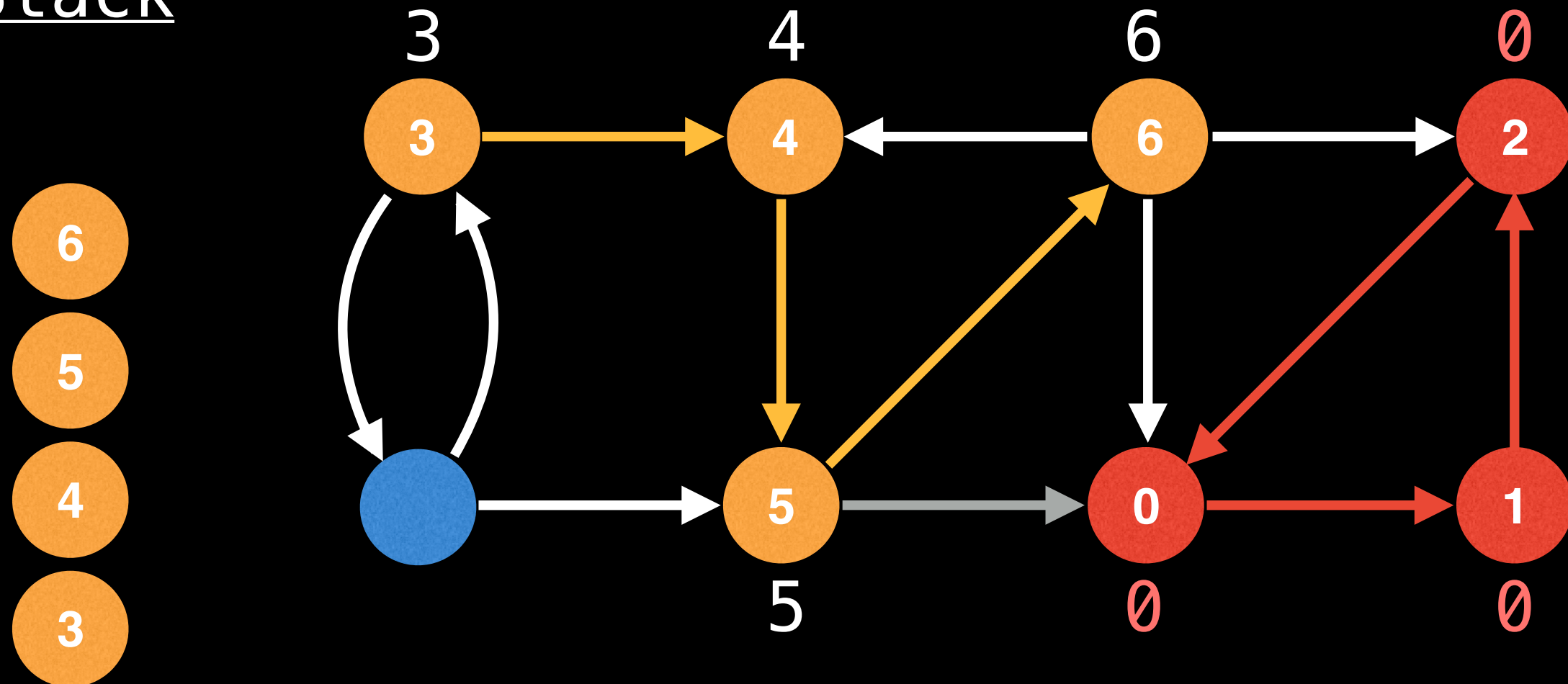
# Stack





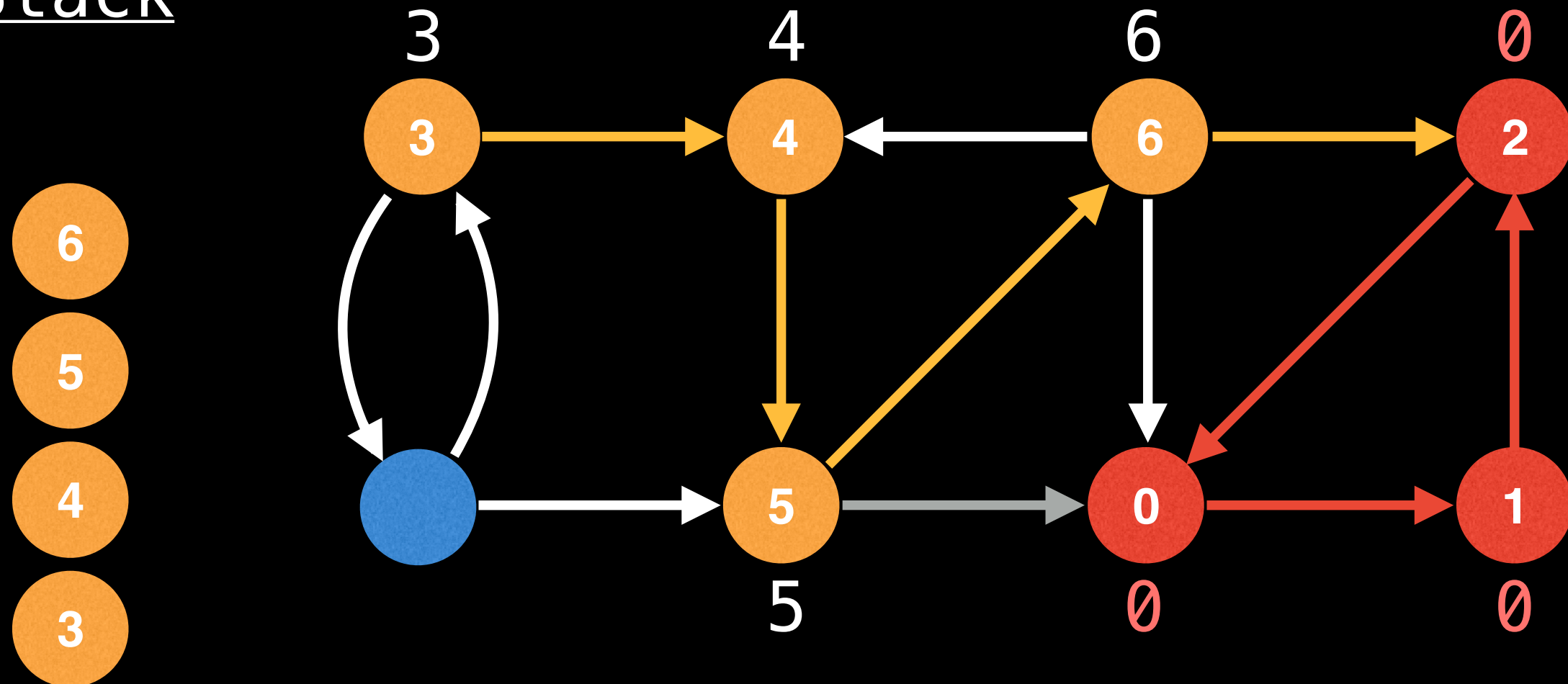
● Unvisited      ● Visiting neighbours      ● Visited all neighbours

Stack



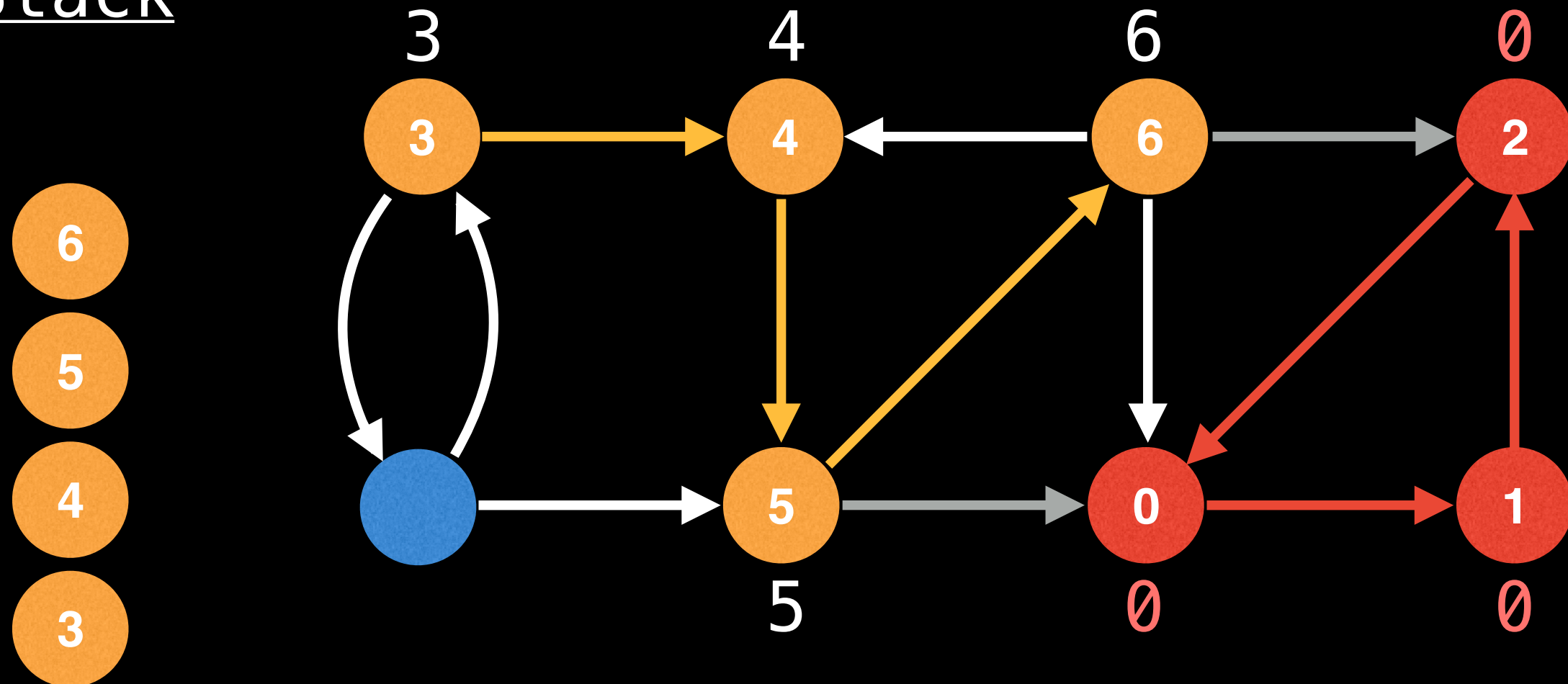
● Unvisited      ● Visiting neighbours      ● Visited all neighbours

Stack



● Unvisited      ● Visiting neighbours      ● Visited all neighbours

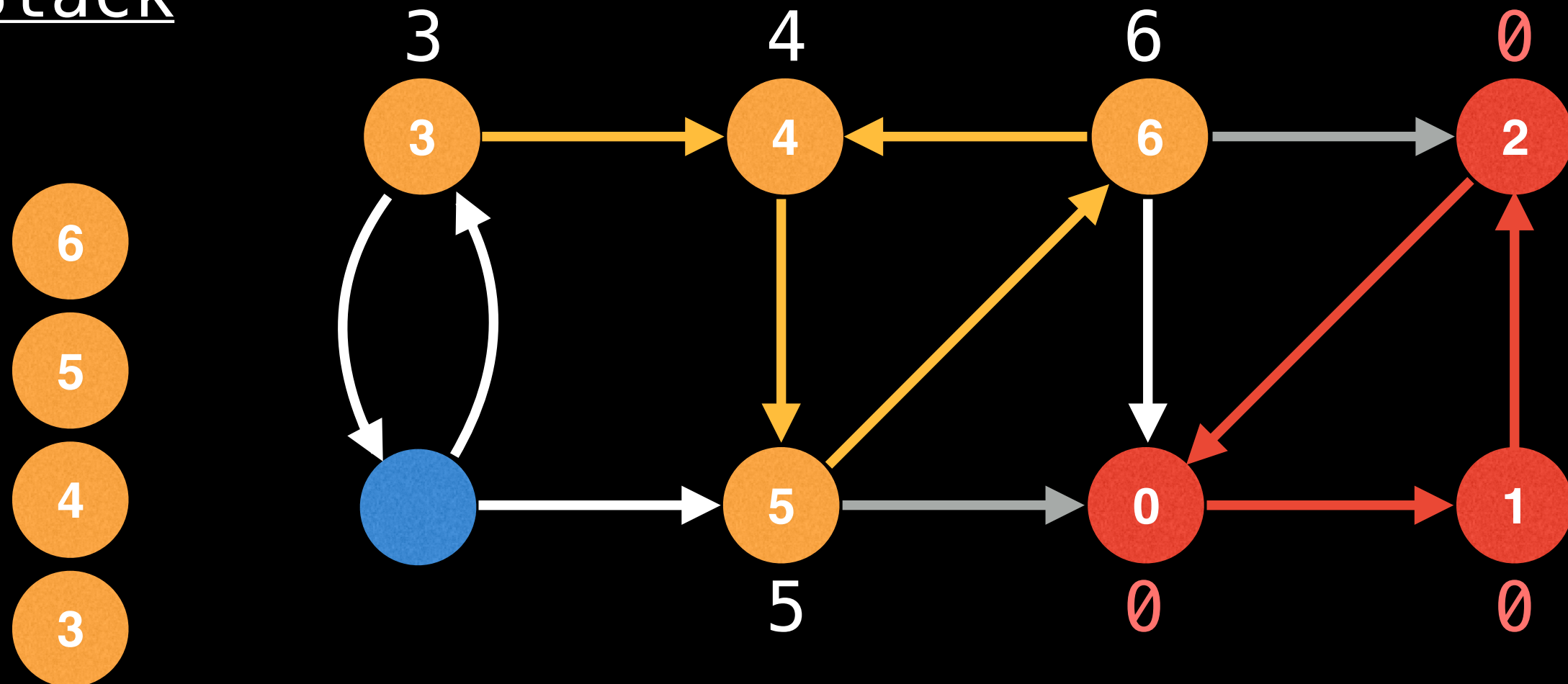
Stack



Node 2 is not on stack so don't min  
with its low-link value.

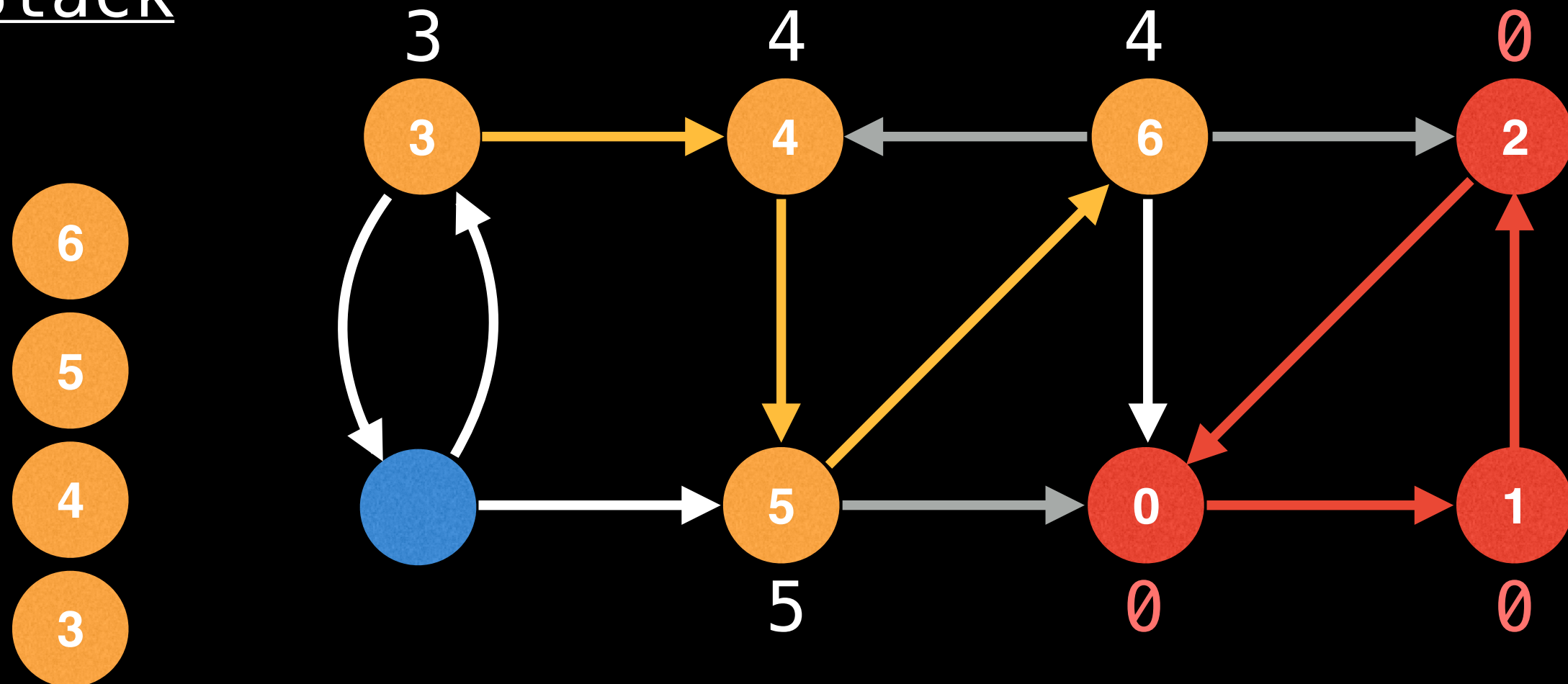
● Unvisited      ● Visiting neighbours      ● Visited all neighbours

Stack



● Unvisited     
 ● Visiting neighbours     
 ● Visited all neighbours

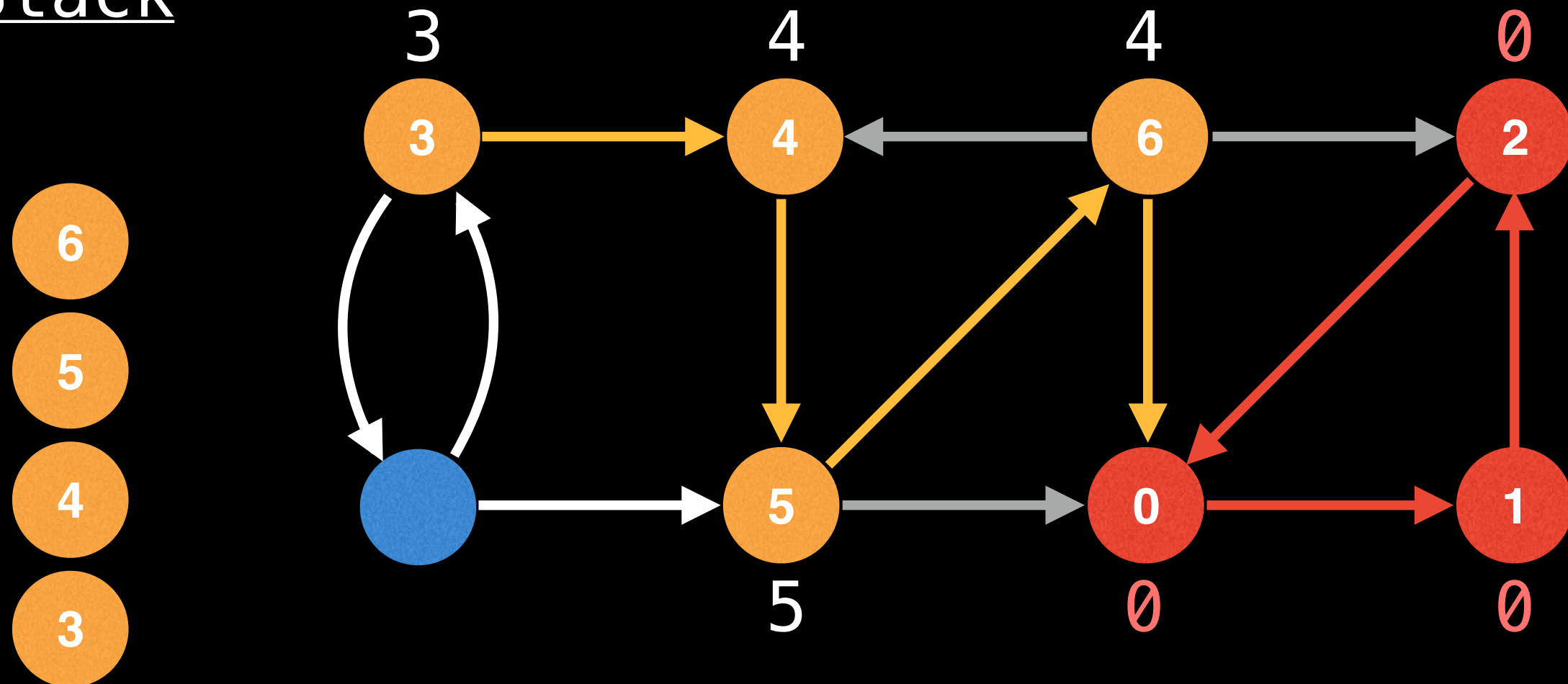
Stack



$\text{lowlink}[6] = \min(\text{lowlink}[6], \text{lowlink}[4])$   
 $= 4$

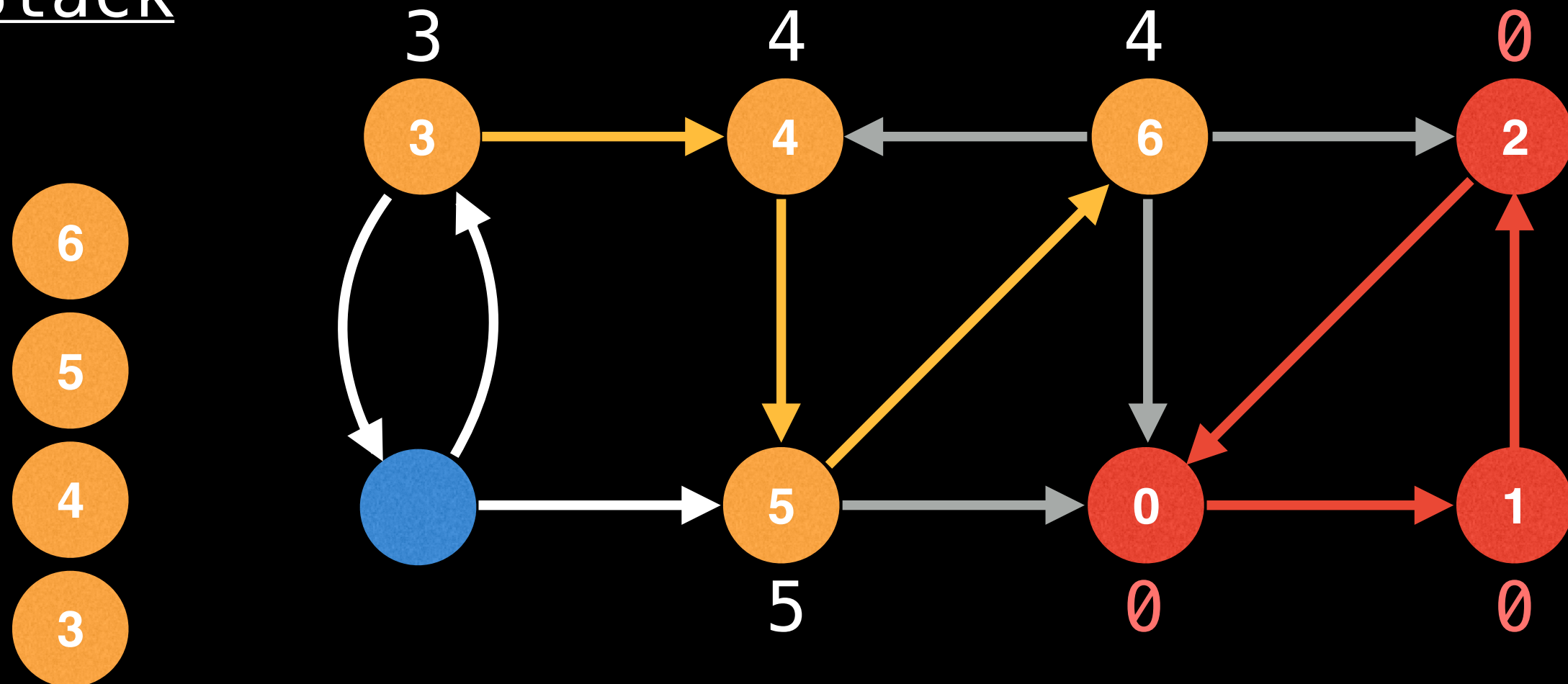
● Unvisited      ● Visiting neighbours      ● Visited all neighbours

Stack



● Unvisited      ● Visiting neighbours      ● Visited all neighbours

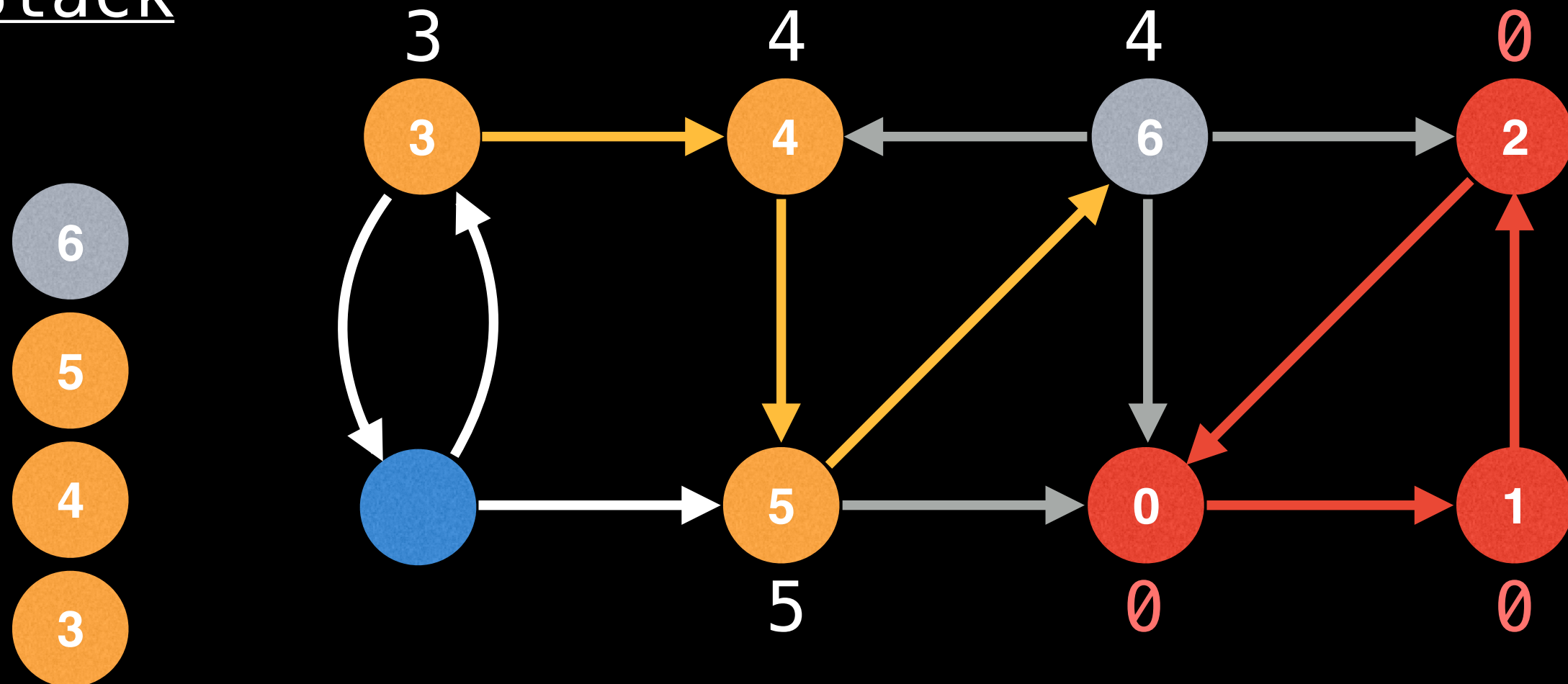
Stack



Node 0 is not on stack so don't min  
with its low-link value.

● Unvisited      ● Visiting neighbours      ● Visited all neighbours

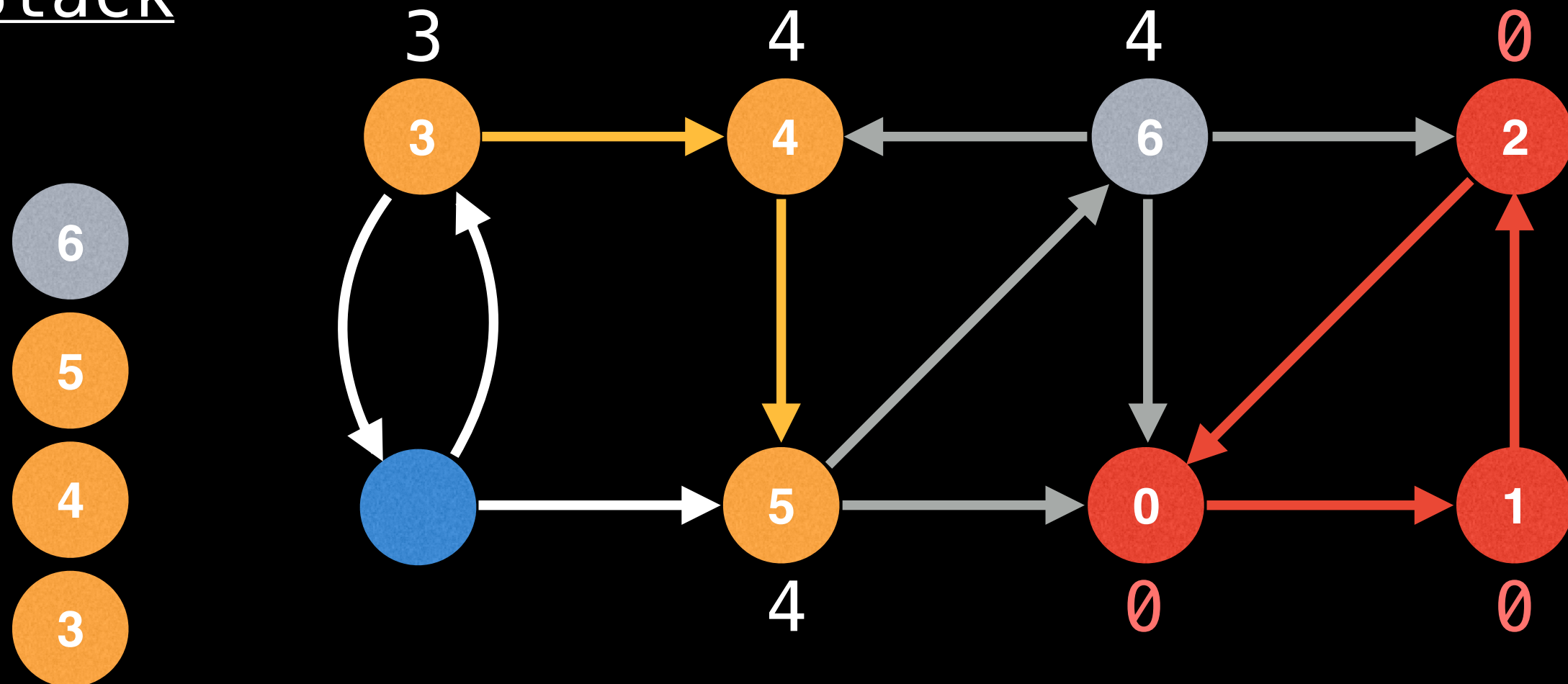
Stack





● Unvisited     
 ● Visiting neighbours     
 ● Visited all neighbours

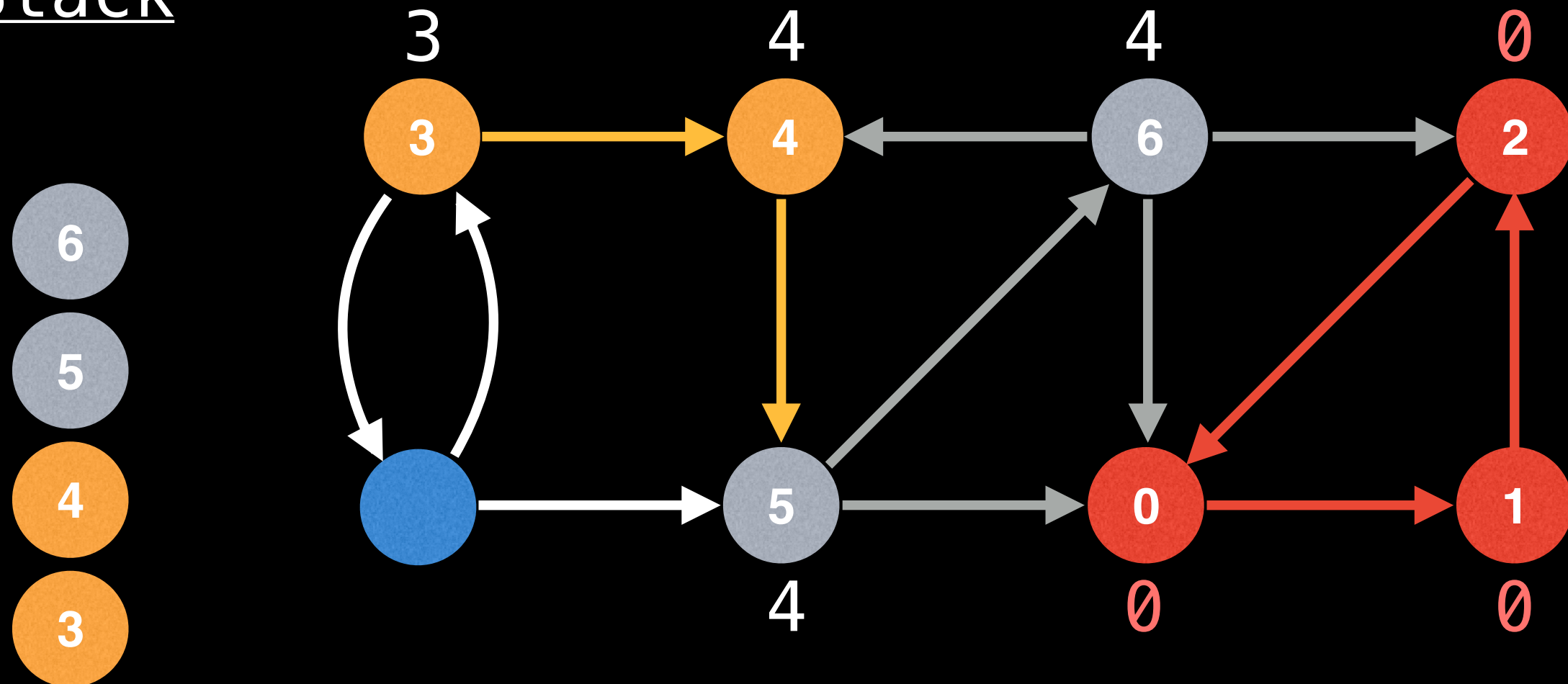
Stack



$\text{lowlink}[5] = \min(\text{lowlink}[5], \text{lowlink}[6])$   
 $= 4$

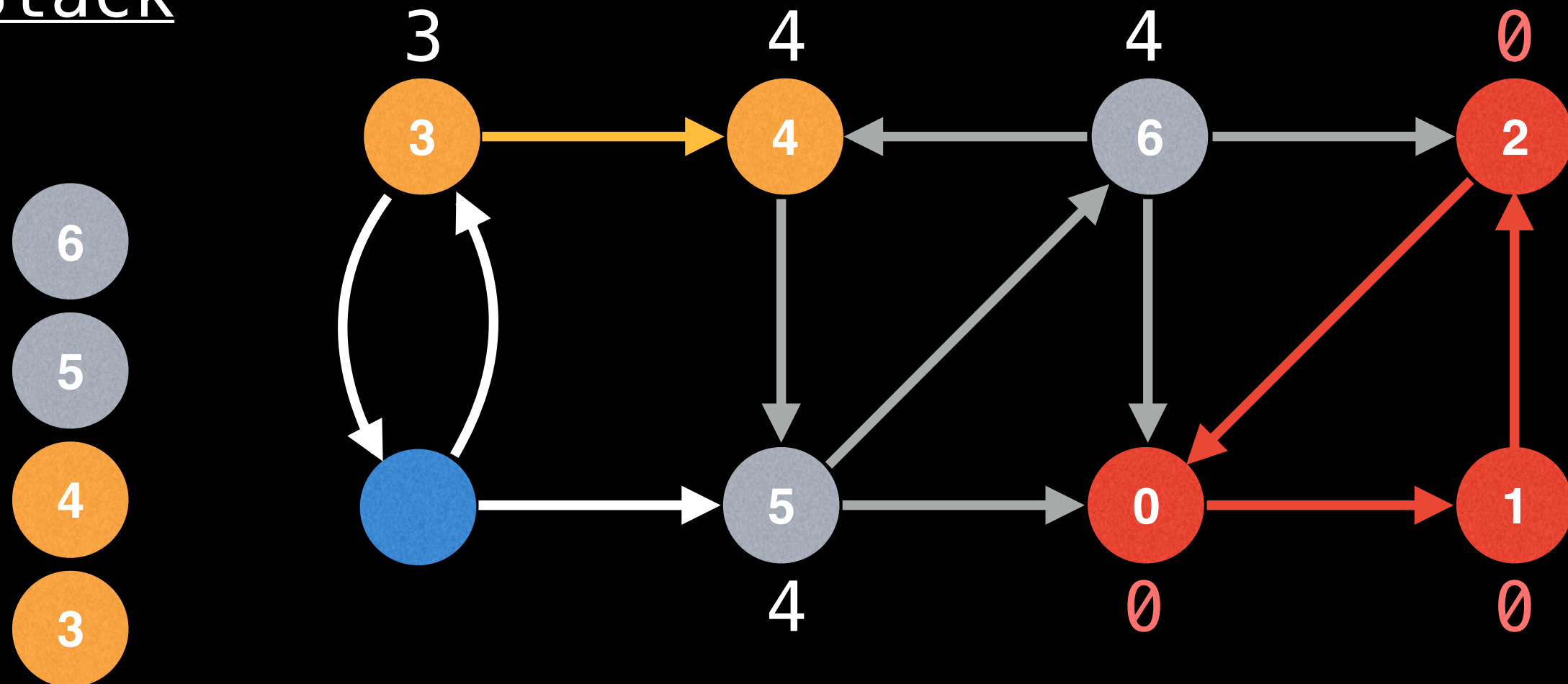
● Unvisited      ● Visiting neighbours      ● Visited all neighbours

Stack



● Unvisited     
 ● Visiting neighbours     
 ● Visited all neighbours

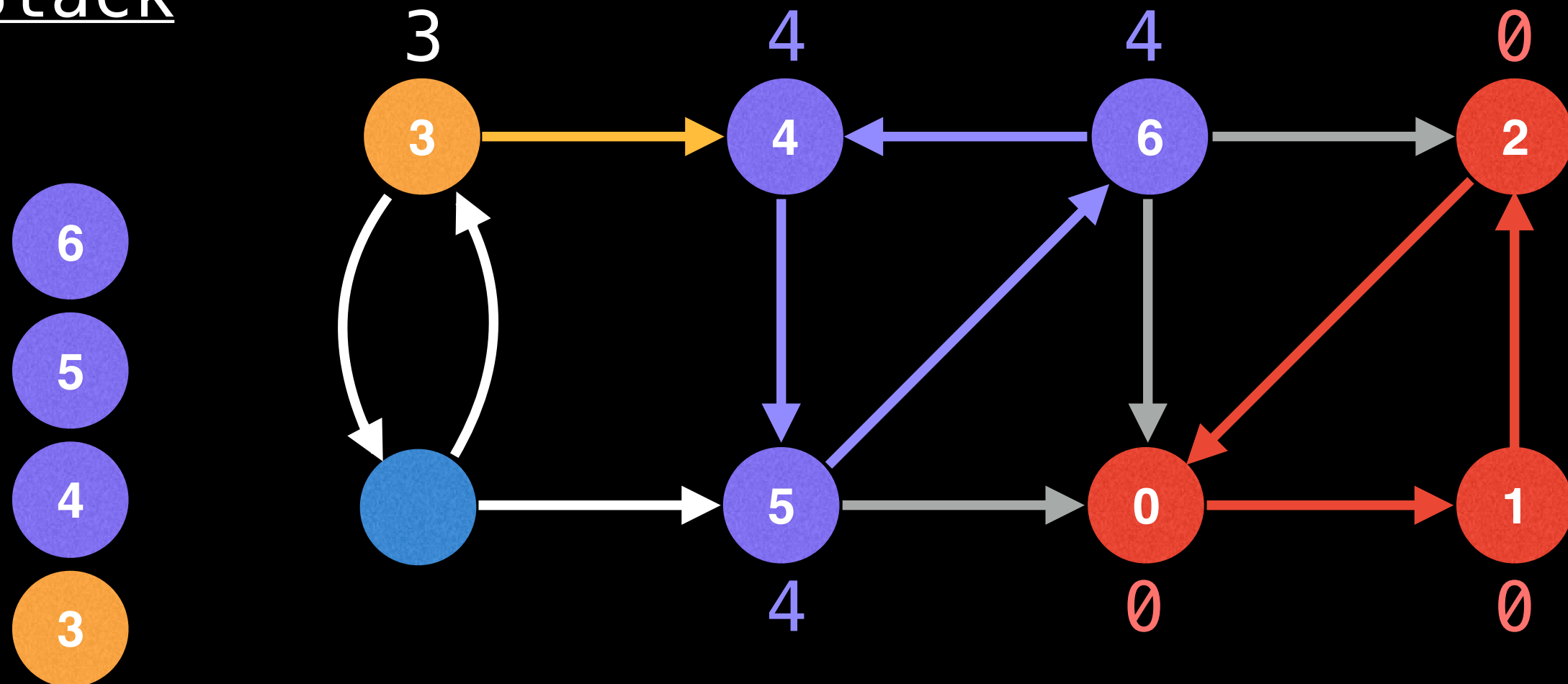
Stack



$$\text{lowlink}[4] = \min(\text{lowlink}[4], \text{lowlink}[5]) = 4$$

● Unvisited     
 ● Visiting neighbours     
 ● Visited all neighbours

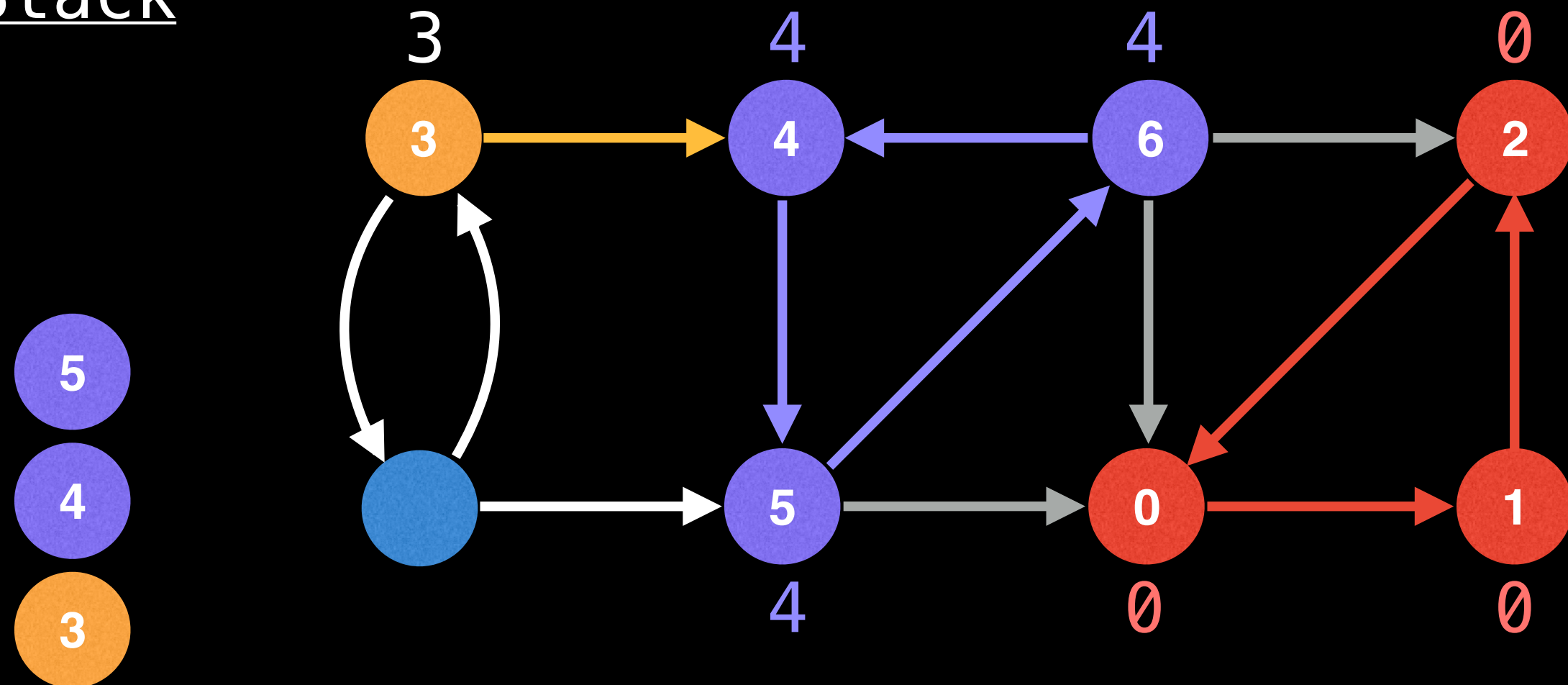
Stack



When a completed SCC is found (current node has visited all its neighbours and its lowlink value equals its id) pop off all associated nodes off the stack.

● Unvisited     
 ● Visiting neighbours     
 ● Visited all neighbours

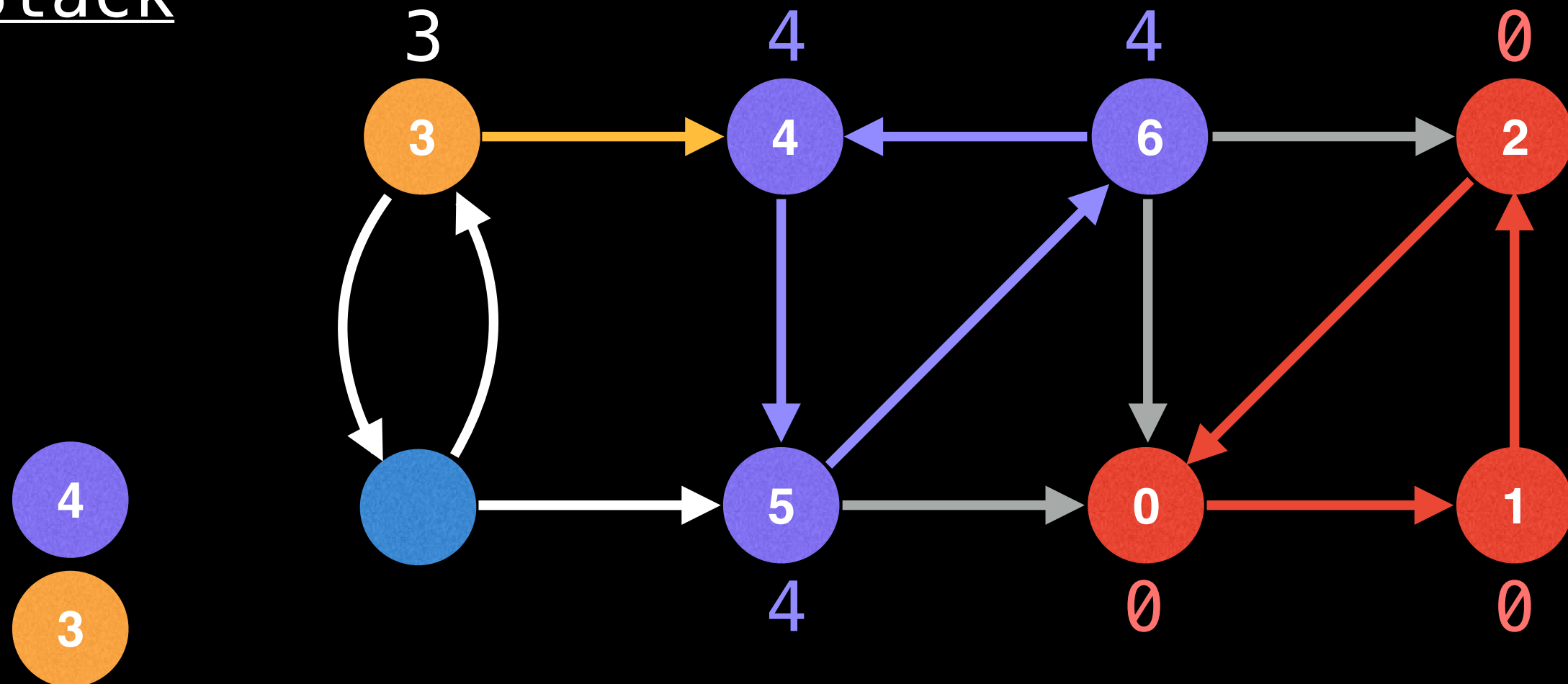
Stack



When a completed SCC is found (current node has visited all its neighbours and its lowlink value equals its id) pop off all associated nodes off the stack.

● Unvisited     
 ● Visiting neighbours     
 ● Visited all neighbours

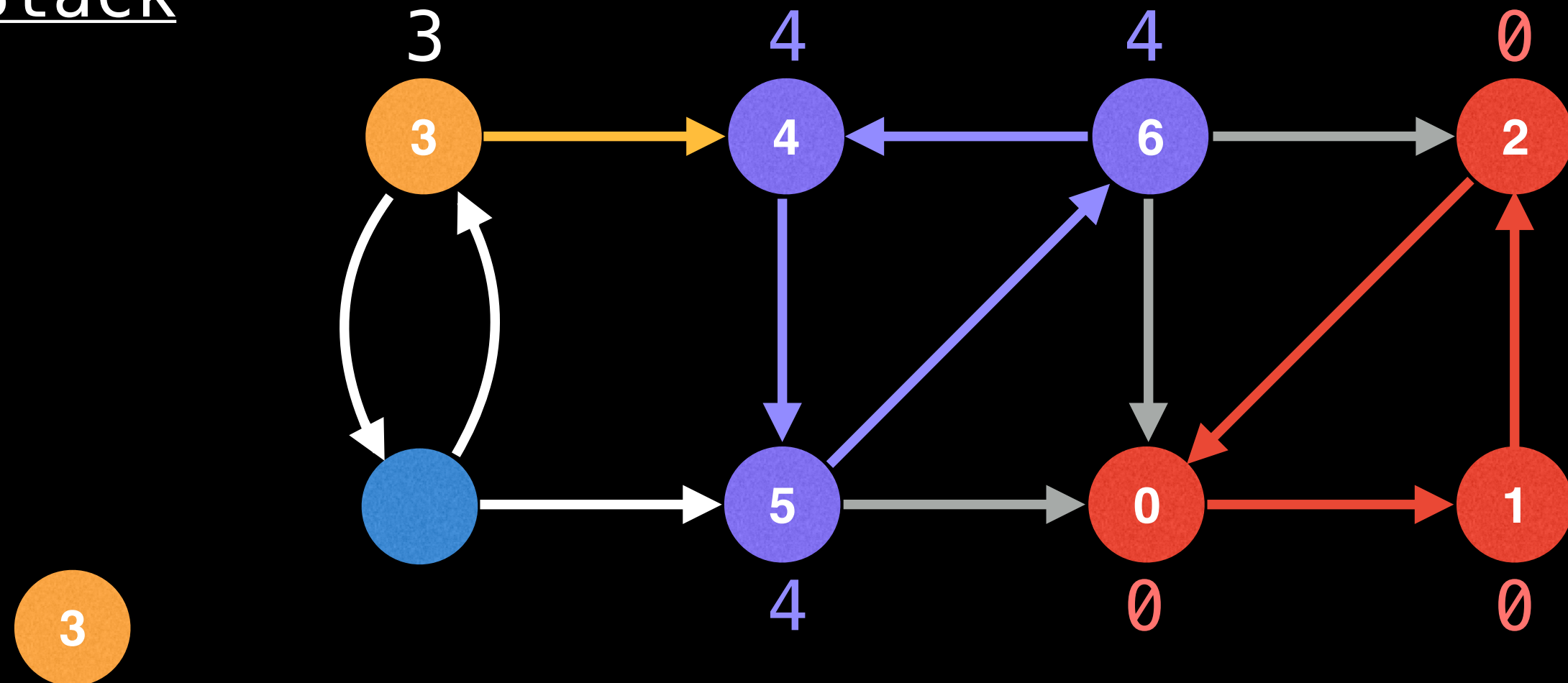
Stack



When a completed SCC is found (current node has visited all its neighbours and its lowlink value equals its id) pop off all associated nodes off the stack.

● Unvisited     
 ● Visiting neighbours     
 ● Visited all neighbours

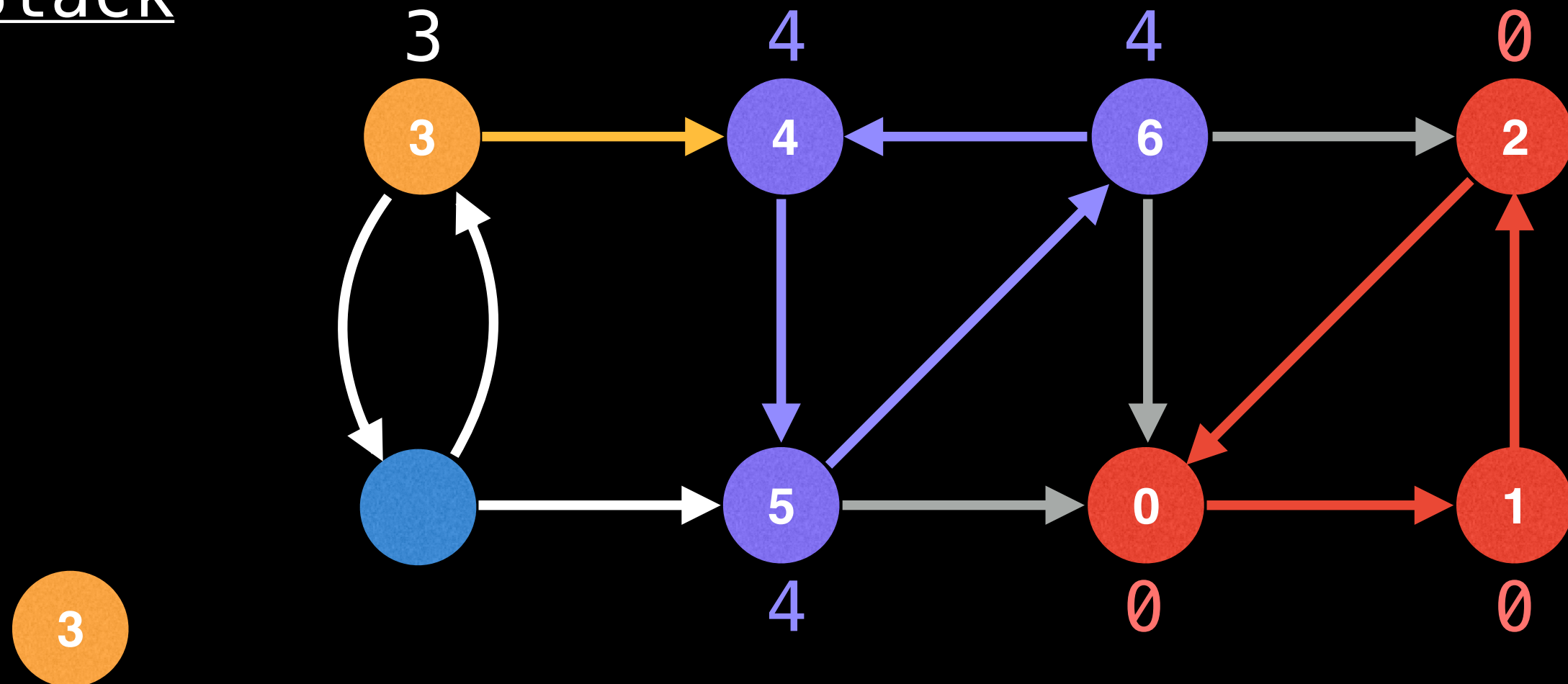
Stack



When a completed SCC is found (current node has visited all its neighbours and its lowlink value equals its id) pop off all associated nodes off the stack.

● Unvisited      ● Visiting neighbours      ● Visited all neighbours

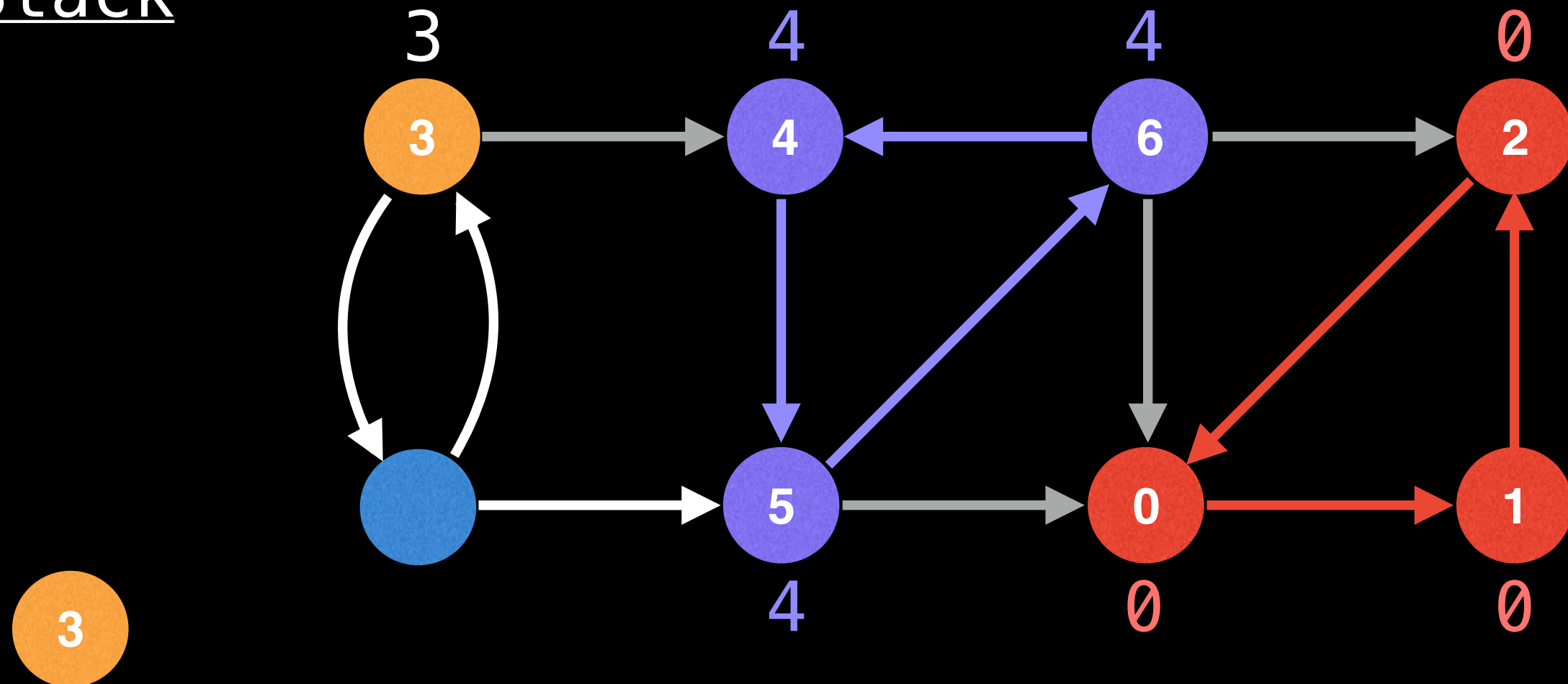
Stack





● Unvisited      ● Visiting neighbours      ● Visited all neighbours

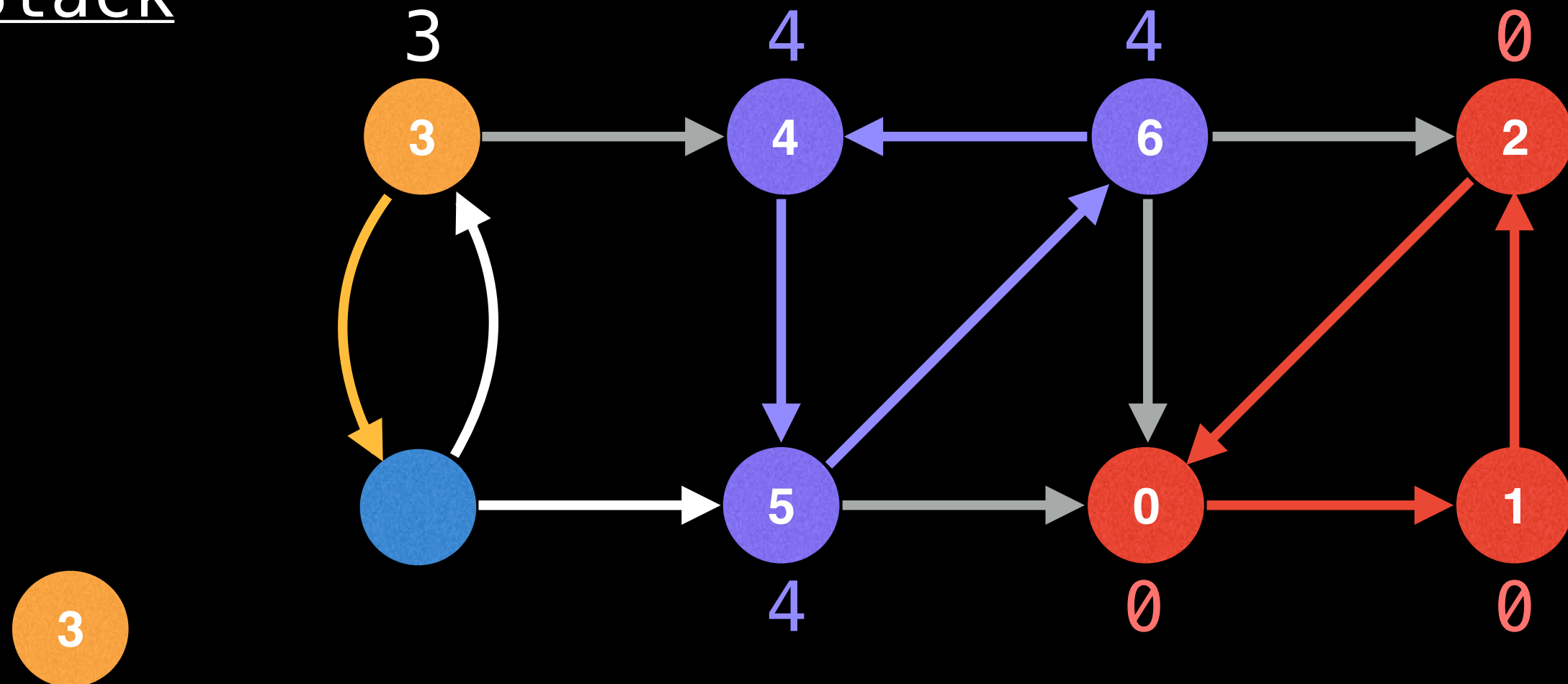
Stack



Node 4 is not on stack so don't min  
with its low-link value.

● Unvisited      ● Visiting neighbours      ● Visited all neighbours

Stack



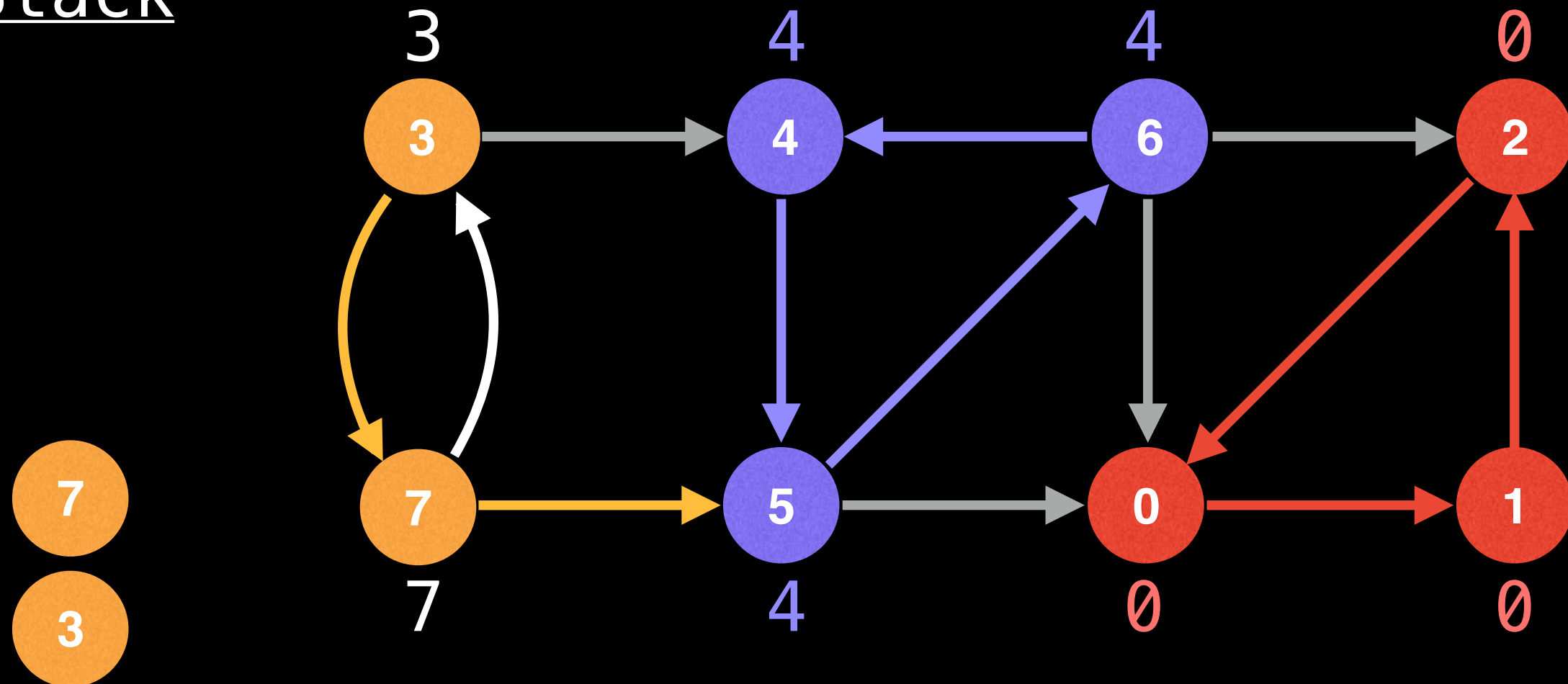


# Stack



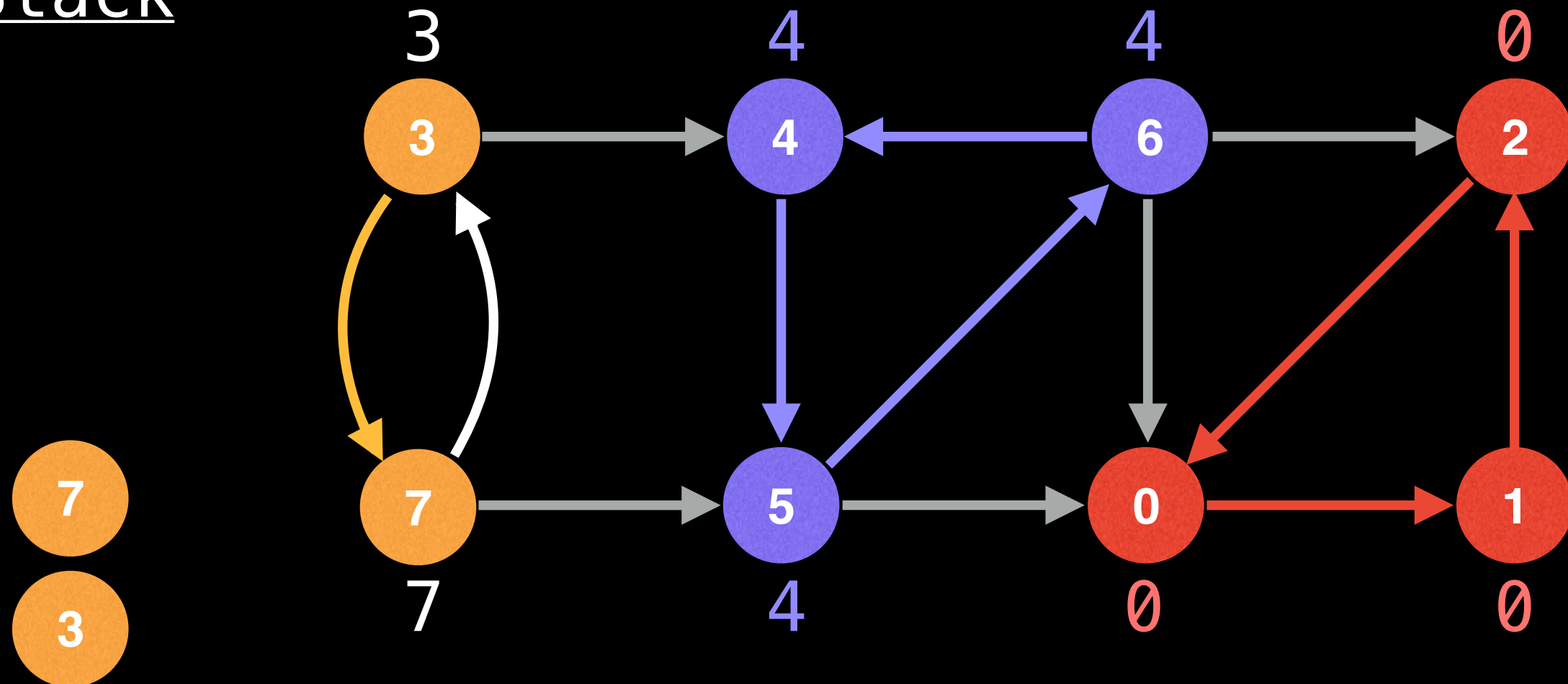
● Unvisited      ● Visiting neighbours      ● Visited all neighbours

Stack



● Unvisited      ● Visiting neighbours      ● Visited all neighbours

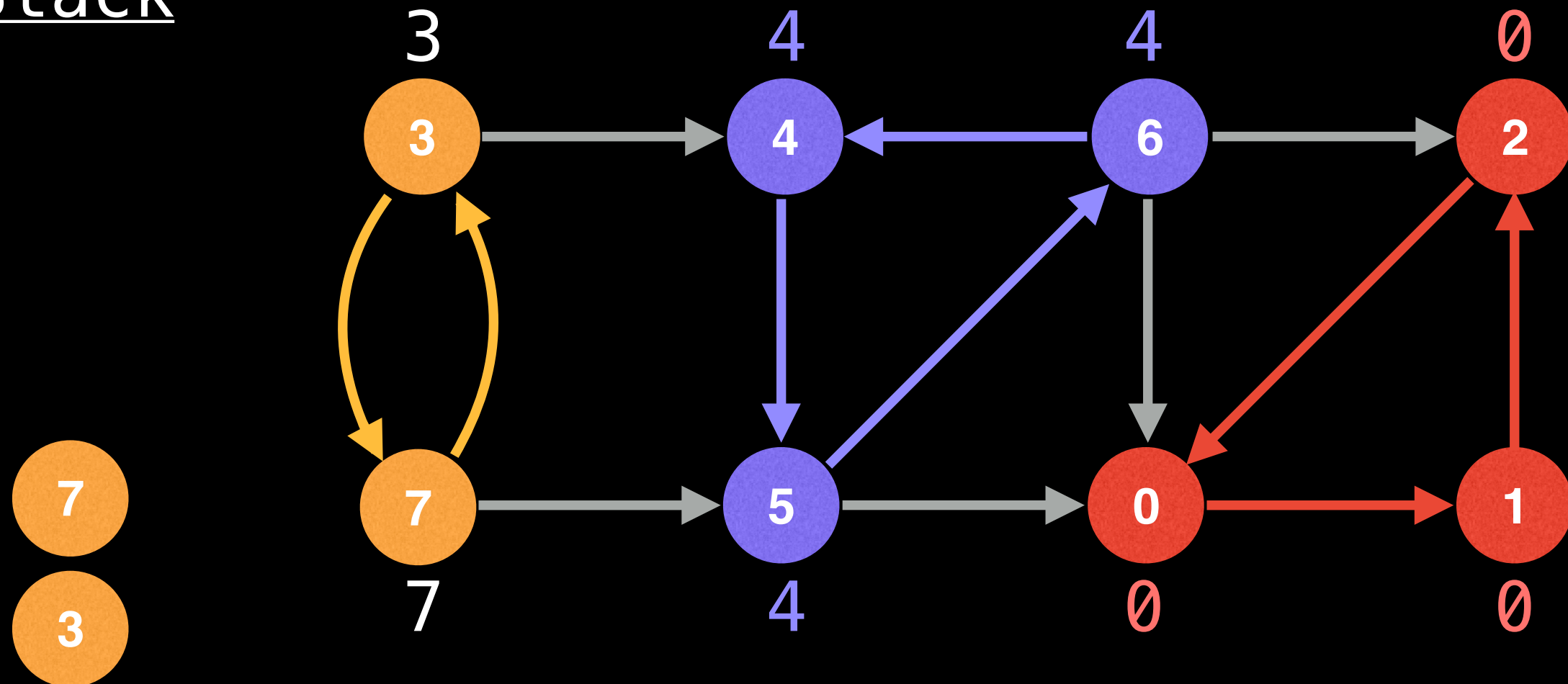
Stack



Node 5 is not on stack so don't min  
with its low-link value.

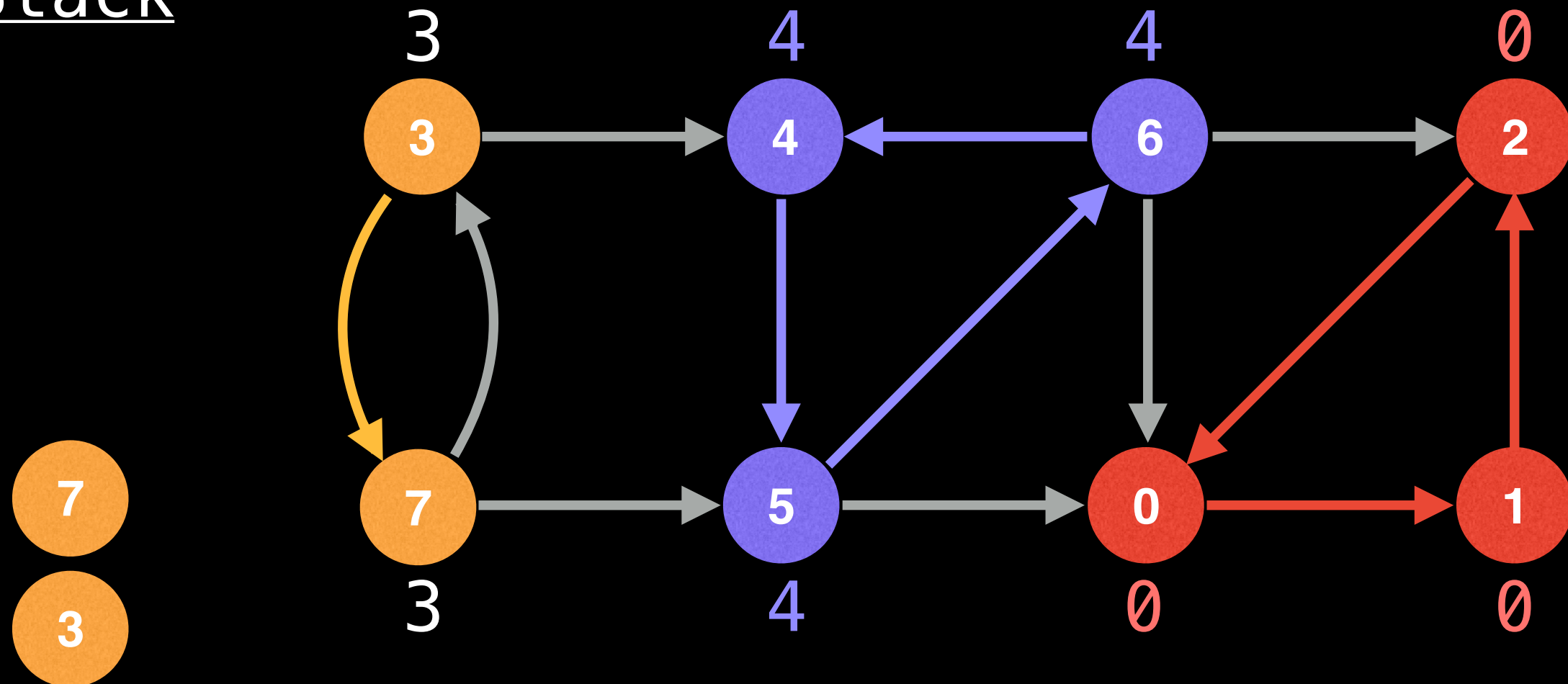
● Unvisited      ● Visiting neighbours      ● Visited all neighbours

Stack



● Unvisited     
 ● Visiting neighbours     
 ● Visited all neighbours

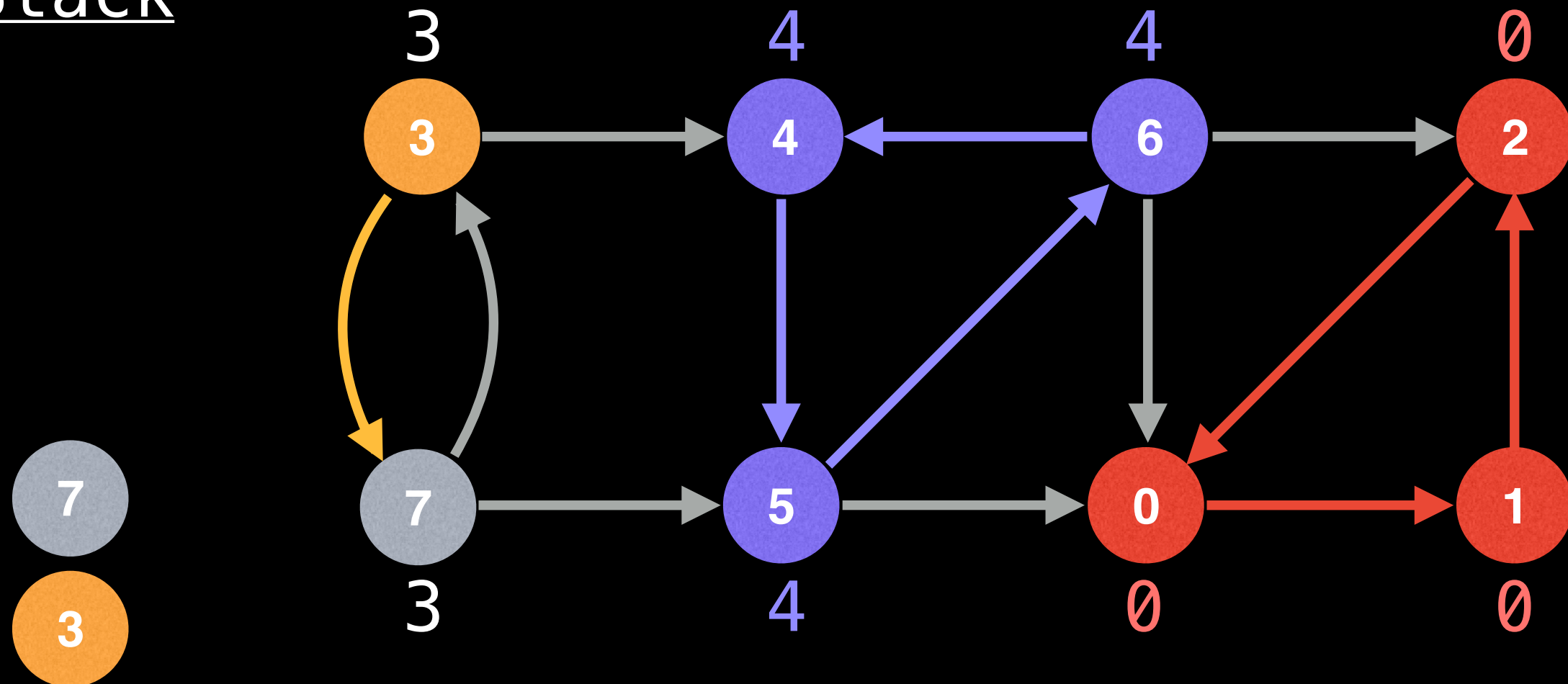
Stack



$\text{lowlink}[6] = \min(\text{lowlink}[6], \text{lowlink}[3])$   
 $= 3$

● Unvisited      ● Visiting neighbours      ● Visited all neighbours

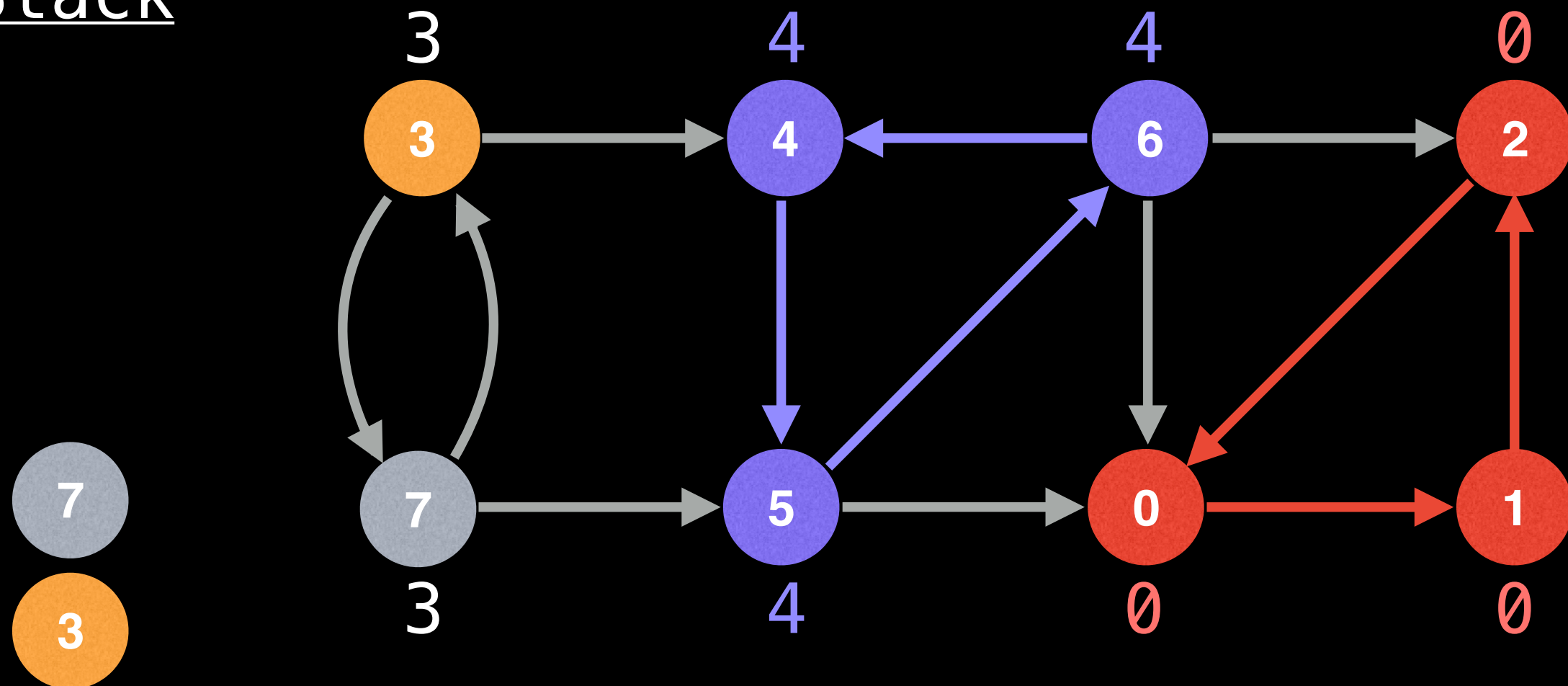
Stack





● Unvisited     
 ● Visiting neighbours     
 ● Visited all neighbours

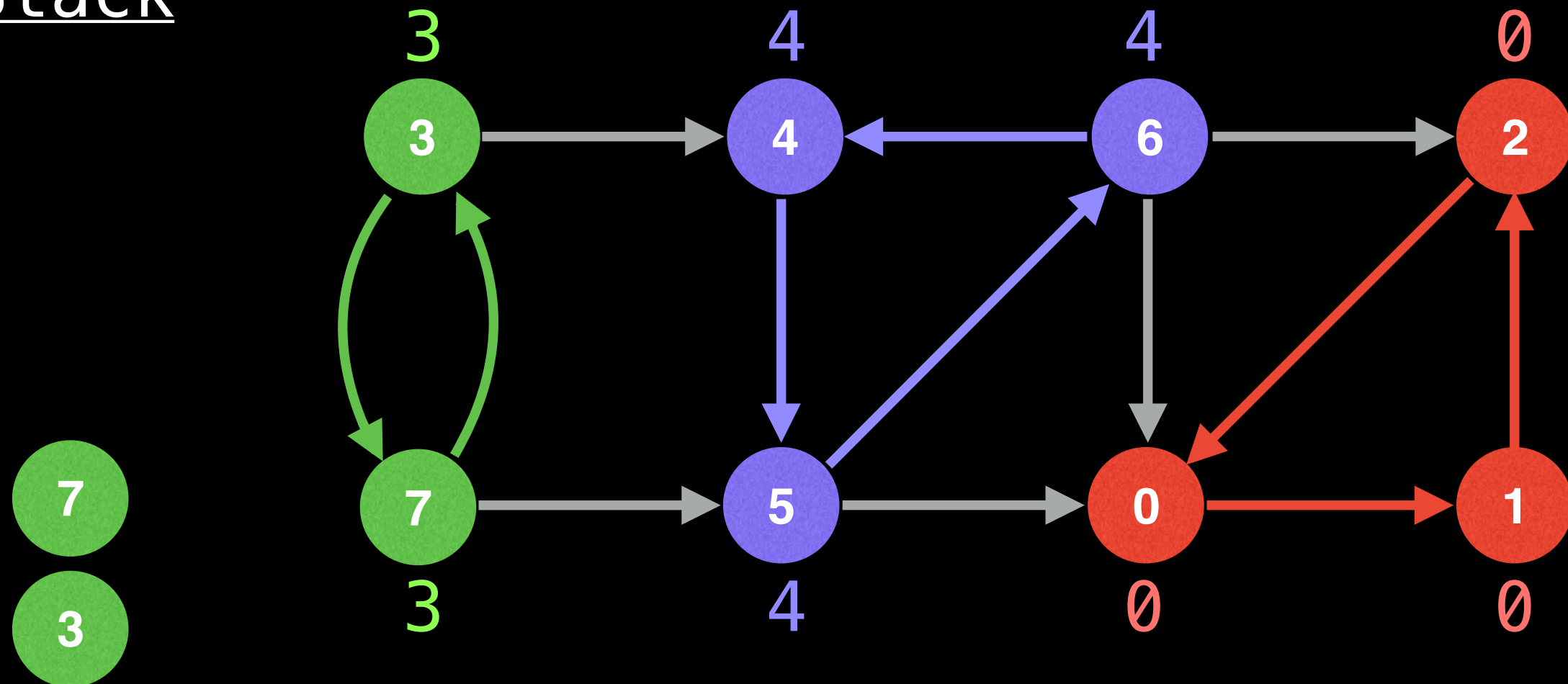
Stack



$\text{lowlink}[3] = \min(\text{lowlink}[3], \text{lowlink}[6])$   
 $= 3$

● Unvisited      ● Visiting neighbours      ● Visited all neighbours

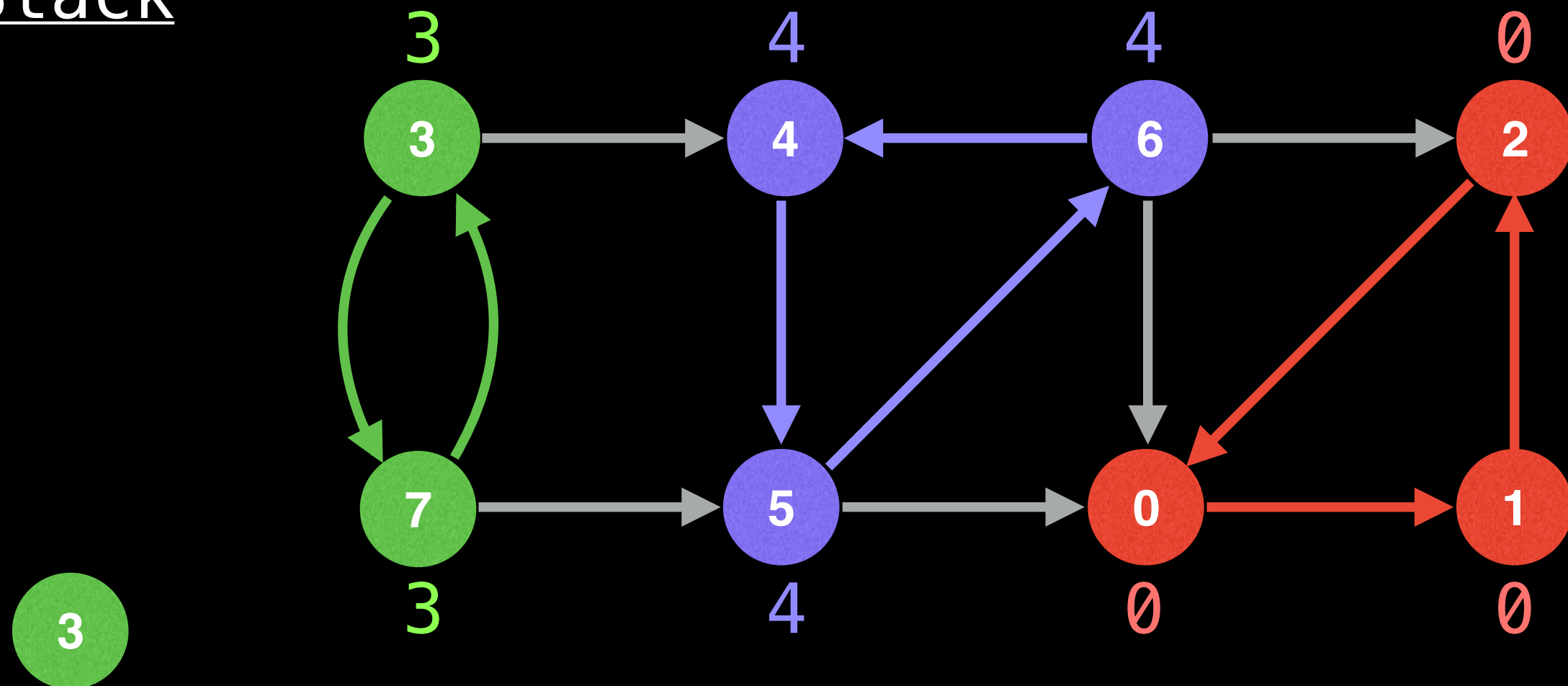
Stack



When a completed SCC is found (current node has visited all its neighbours and its lowlink value equals its id) pop off all associated nodes off the stack.

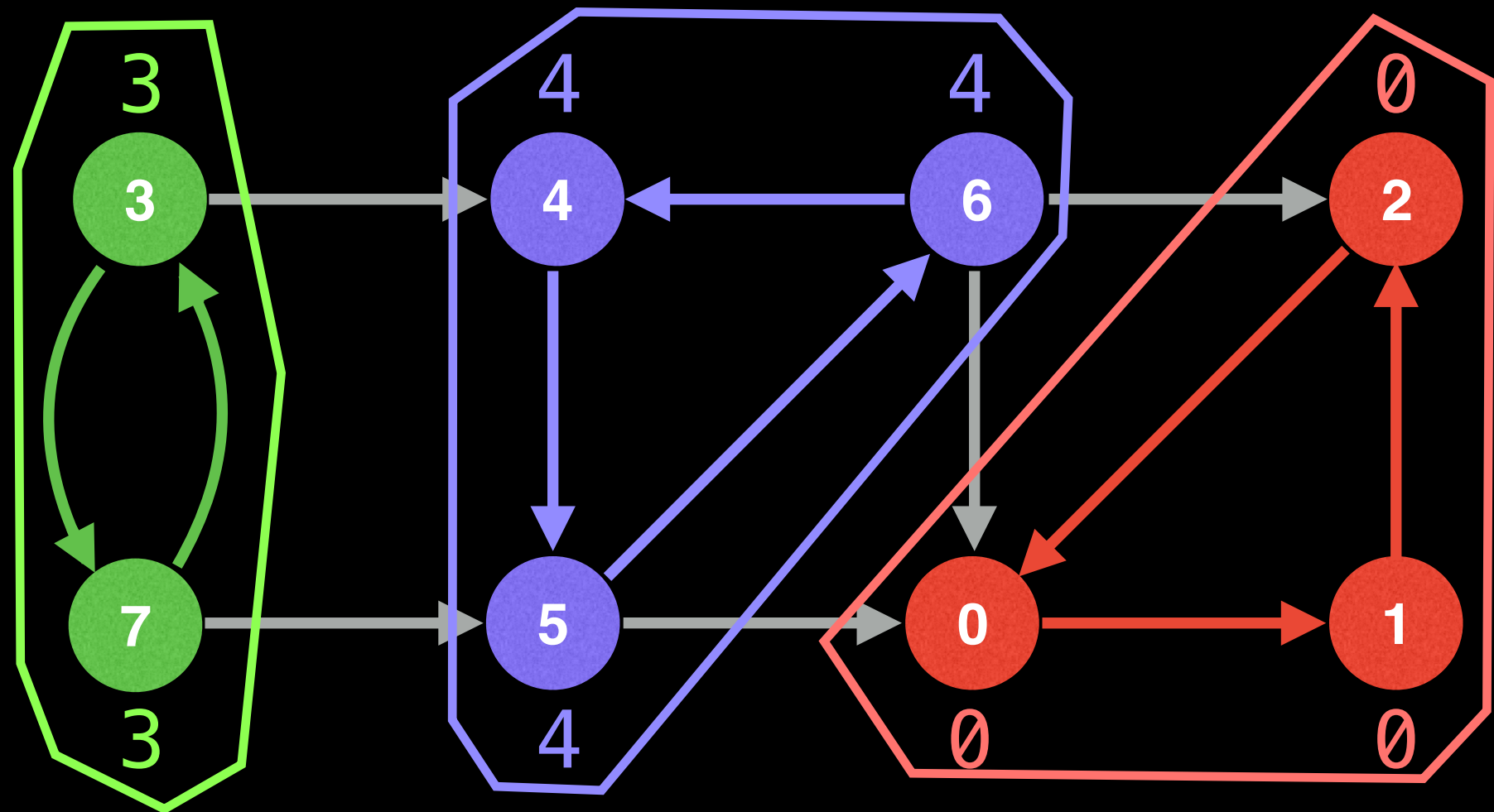
● Unvisited     
 ● Visiting neighbours     
 ● Visited all neighbours

Stack



When a completed SCC is found (current node has visited all its neighbours and its lowlink value equals its id) pop off all associated nodes off the stack.

● Unvisited      ● Visiting neighbours      ● Visited all neighbours



**UNVISITED** = -1

n = number of nodes in graph

g = adjacency list with directed edges

id = 0           # Used to give each node an id

sccCount = 0   # Used to count number of SCCs found

# Index *i* in these arrays represents node *i*

ids = [0, 0, ... 0, 0]           # Length n

low = [0, 0, ... 0, 0]           # Length n

onStack = [**false**, **false**, ..., **false**]   # Length n

stack = an empty stack data structure

**function** findSccs():

**for**(i = 0; i < n; i++): ids[i] = **UNVISITED**

**for**(i = 0; i < n; i++):

**if**(ids[i] == **UNVISITED**):

**dfs**(i)

**return** low

```
UNVISITED = -1
```

```
n = number of nodes in graph
```

```
g = adjacency list with directed edges
```

```
id = 0          # Used to give each node an id
```

```
sccCount = 0    # Used to count number of SCCs found
```

```
# Index i in these arrays represents node i
```

```
ids = [0, 0, ... 0, 0]          # Length n
```

```
low = [0, 0, ... 0, 0]         # Length n
```

```
onStack = [false, false, ..., false] # Length n
```

```
stack = an empty stack data structure
```

```
function findSccs():
```

```
    for(i = 0; i < n; i++): ids[i] = UNVISITED
```

```
    for(i = 0; i < n; i++):
```

```
        if(ids[i] == UNVISITED):
```

```
            dfs(i)
```

```
    return low
```

**UNVISITED** = -1

n = number of nodes in graph

g = adjacency list with directed edges

```
id = 0          # Used to give each node an id
sccCount = 0    # Used to count number of SCCs found
```

# Index *i* in these arrays represents node *i*

ids = [0, 0, ... 0, 0] # Length n

low = [0, 0, ... 0, 0] # Length n

onStack = [false, false, ..., false] # Length n

stack = an empty stack data structure

**function** findSccs():

**for**(i = 0; i < n; i++): ids[i] = **UNVISITED**

**for**(i = 0; i < n; i++):

**if**(ids[i] == **UNVISITED**):

            dfs(i)

**return** low

**UNVISITED** = -1

n = number of nodes in graph

g = adjacency list with directed edges

id = 0           # Used to give each node an id

sccCount = 0   # Used to count number of SCCs found

# Index *i* in these arrays represents node *i*

ids = [0, 0, ... 0, 0]           # Length n

low = [0, 0, ... 0, 0]           # Length n

onStack = [false, false, ..., false] # Length n

stack = an empty stack data structure

**function** findSccs():

**for**(i = 0; i < n; i++): ids[i] = **UNVISITED**

**for**(i = 0; i < n; i++):

**if**(ids[i] == **UNVISITED**):

            dfs(i)

**return** low



**UNVISITED** = -1

n = number of nodes in graph

g = adjacency list with directed edges

id = 0           # Used to give each node an id

sccCount = 0   # Used to count number of SCCs found

# Index *i* in these arrays represents node *i*

ids = [0, 0, ... 0, 0]           # Length n

low = [0, 0, ... 0, 0]           # Length n

onStack = [false, false, ..., false]   # Length n

stack = an empty stack data structure

**function** findSccs():

**for**(i = 0; i < n; i++): ids[i] = **UNVISITED**

**for**(i = 0; i < n; i++):

**if**(ids[i] == **UNVISITED**):

        dfs(i)

**return** low

**UNVISITED** = -1

n = number of nodes in graph

g = adjacency list with directed edges

id = 0                   # Used to give each node an id

sccCount = 0   # Used to count number of SCCs found

# Index *i* in these arrays represents node *i*

ids = [0, 0, ... 0, 0]                   # Length n

low = [0, 0, ... 0, 0]                   # Length n

onStack = [**false**, **false**, ..., **false**]   # Length n

stack = an empty stack data structure

**function** findSccs():

**for**(i = 0; i < n; i++): ids[i] = **UNVISITED**

**for**(i = 0; i < n; i++):

**if**(ids[i] == **UNVISITED**):

**dfs**(i)

**return** low

**UNVISITED** = -1

n = number of nodes in graph

g = adjacency list with directed edges

id = 0           # Used to give each node an id

sccCount = 0   # Used to count number of SCCs found

# Index *i* in these arrays represents node *i*

ids = [0, 0, ... 0, 0]           # Length n

low = [0, 0, ... 0, 0]           # Length n

onStack = [**false**, **false**, ..., **false**] # Length n

stack = an empty stack data structure

**function** findSccs():

**for**(i = 0; i < n; i++): ids[i] = **UNVISITED**

**for**(i = 0; i < n; i++):

**if**(ids[i] == **UNVISITED**):

**dfs**(i)

**return** low

```
function dfs(at):
    stack.push(at)
    onStack[at] = true
    ids[at] = low[at] = id++

# Visit all neighbours & min low-link on callback
for(to : g[at]):
    if(ids[to] == UNVISITED): dfs(to)
    if(onStack[to]): low[at] = min(low[at], low[to])

# After having visited all the neighbours of 'at'
# if we're at the start of a SCC empty the seen
# stack until we're back to the start of the SCC.
if(ids[at] == low[at]):
    for(node = stack.pop();; node = stack.pop()):
        onStack[node] = false
        low[node] = ids[at]
        if(node == at): break
    sccCount++
```

```
function dfs(at):
```

```
    stack.push(at)  
    onStack[at] = true  
    ids[at] = low[at] = id++
```

```
# Visit all neighbours & min low-link on callback
```

```
for(to : g[at]):
```

```
    if(ids[to] == UNVISITED): dfs(to)
```

```
    if(onStack[to]): low[at] = min(low[at], low[to])
```

```
# After having visited all the neighbours of 'at'
```

```
# if we're at the start of a SCC empty the seen
```

```
# stack until we're back to the start of the SCC.
```

```
if(ids[at] == low[at]):
```

```
    for(node = stack.pop();; node = stack.pop()):
```

```
        onStack[node] = false
```

```
        low[node] = ids[at]
```

```
        if(node == at): break
```

```
    sccCount++
```

```
function dfs(at):  
    stack.push(at)  
    onStack[at] = true  
    ids[at] = low[at] = id++
```

```
# Visit all neighbours & min low-link on callback  
for(to : g[at]):
```

```
    if(ids[to] == UNVISITED): dfs(to)  
    if(onStack[to]): low[at] = min(low[at], low[to])
```

```
# After having visited all the neighbours of 'at'  
# if we're at the start of a SCC empty the seen  
# stack until we're back to the start of the SCC.
```

```
if(ids[at] == low[at]):  
    for(node = stack.pop();; node = stack.pop()):  
        onStack[node] = false  
        low[node] = ids[at]  
        if(node == at): break  
    sccCount++
```

```
function dfs(at):  
    stack.push(at)  
    onStack[at] = true  
    ids[at] = low[at] = id++  
  
    # Visit all neighbours & min low-link on callback  
    for(to : g[at]):  
        if(ids[to] == UNVISITED): dfs(to)  
        if(onStack[to]): low[at] = min(low[at], low[to])  
  
    # After having visited all the neighbours of 'at'  
    # if we're at the start of a SCC empty the seen  
    # stack until we're back to the start of the SCC.  
    if(ids[at] == low[at]):  
        for(node = stack.pop();; node = stack.pop()):  
            onStack[node] = false  
            low[node] = ids[at]  
            if(node == at): break  
        sccCount++
```

```
function dfs(at):  
    stack.push(at)  
    onStack[at] = true  
    ids[at] = low[at] = id++  
  
    # Visit all neighbours & min low-link on callback  
    for(to : g[at]):  
        if(ids[to] == UNVISITED): dfs(to)  
        if(onStack[to]): low[at] = min(low[at], low[to])  
  
    # After having visited all the neighbours of 'at'  
    # if we're at the start of a SCC empty the seen  
    # stack until we're back to the start of the SCC.  
    if(ids[at] == low[at]):  
        for(node = stack.pop();; node = stack.pop()):  
            onStack[node] = false  
            low[node] = ids[at]  
            if(node == at): break  
        sccCount++
```



```
function dfs(at):  
    stack.push(at)  
    onStack[at] = true  
    ids[at] = low[at] = id++  
  
    # Visit all neighbours & min low-link on callback  
    for(to : g[at]):  
        if(ids[to] == UNVISITED): dfs(to)  
        if(onStack[to]): low[at] = min(low[at], low[to])  
  
    # After having visited all the neighbours of 'at'  
    # if we're at the start of a SCC empty the seen  
    # stack until we're back to the start of the SCC.  
    if(ids[at] == low[at]):  
        for(node = stack.pop();; node = stack.pop()):  
            onStack[node] = false  
            low[node] = ids[at]  
            if(node == at): break  
        sccCount++
```

```
function dfs(at):  
    stack.push(at)  
    onStack[at] = true  
    ids[at] = low[at] = id++  
  
    # Visit all neighbours & min low-link on callback  
    for(to : g[at]):  
        if(ids[to] == UNVISITED): dfs(to)  
        if(onStack[to]): low[at] = min(low[at], low[to])  
  
    # After having visited all the neighbours of 'at'  
    # if we're at the start of a SCC empty the seen  
    # stack until we're back to the start of the SCC.  
    if(ids[at] == low[at]):  
        for(node = stack.pop();; node = stack.pop()):  
            onStack[node] = false  
            low[node] = ids[at]  
            if(node == at): break  
        sccCount++
```

```
function dfs(at):
    stack.push(at)
    onStack[at] = true
    ids[at] = low[at] = id++

# Visit all neighbours & min low-link on callback
for(to : g[at]):
    if(ids[to] == UNVISITED): dfs(to)
    if(onStack[to]): low[at] = min(low[at], low[to])

# After having visited all the neighbours of 'at'
# if we're at the start of a SCC empty the seen
# stack until we're back to the start of the SCC.
if(ids[at] == low[at]):
    for(node = stack.pop();; node = stack.pop()):
        onStack[node] = false
        low[node] = ids[at]
        if(node == at): break
    sccCount++
```

```
function dfs(at):
    stack.push(at)
    onStack[at] = true
    ids[at] = low[at] = id++

# Visit all neighbours & min low-link on callback
for(to : g[at]):
    if(ids[to] == UNVISITED): dfs(to)
    if(onStack[to]): low[at] = min(low[at], low[to])

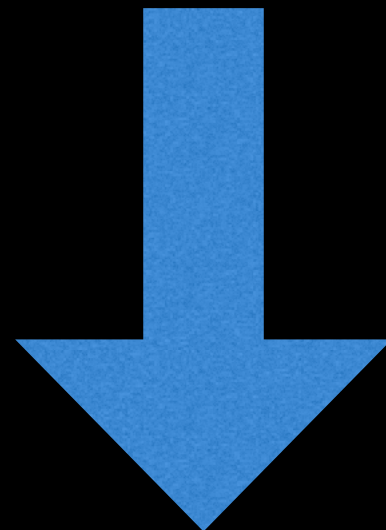
# After having visited all the neighbours of 'at'
# if we're at the start of a SCC empty the seen
# stack until we're back to the start of the SCC.
if(ids[at] == low[at]):
    for(node = stack.pop();; node = stack.pop()):
        onStack[node] = false
        low[node] = ids[at]
        if(node == at): break
    sccCount++
```

# Source Code Link

Slides/source code can be found at the following link:

[github.com/williamfiset/algorithms](https://github.com/williamfiset/algorithms)

Link in the description:





# Tarjan's Algorithm for Finding Strongly Connected Components Source Code

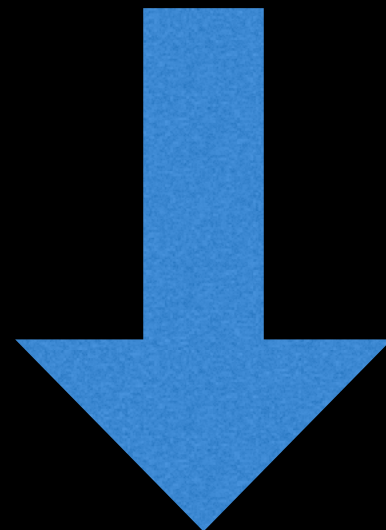
William Fiset

# Source Code Link

Slides/source code can be found at the following link:

[github.com/williamfiset/algorithms](https://github.com/williamfiset/algorithms)

Link in the description:





Deleted slides follow

# Low-Link Values

id

0

1

2

3

4

5

lowlink

$\infty$

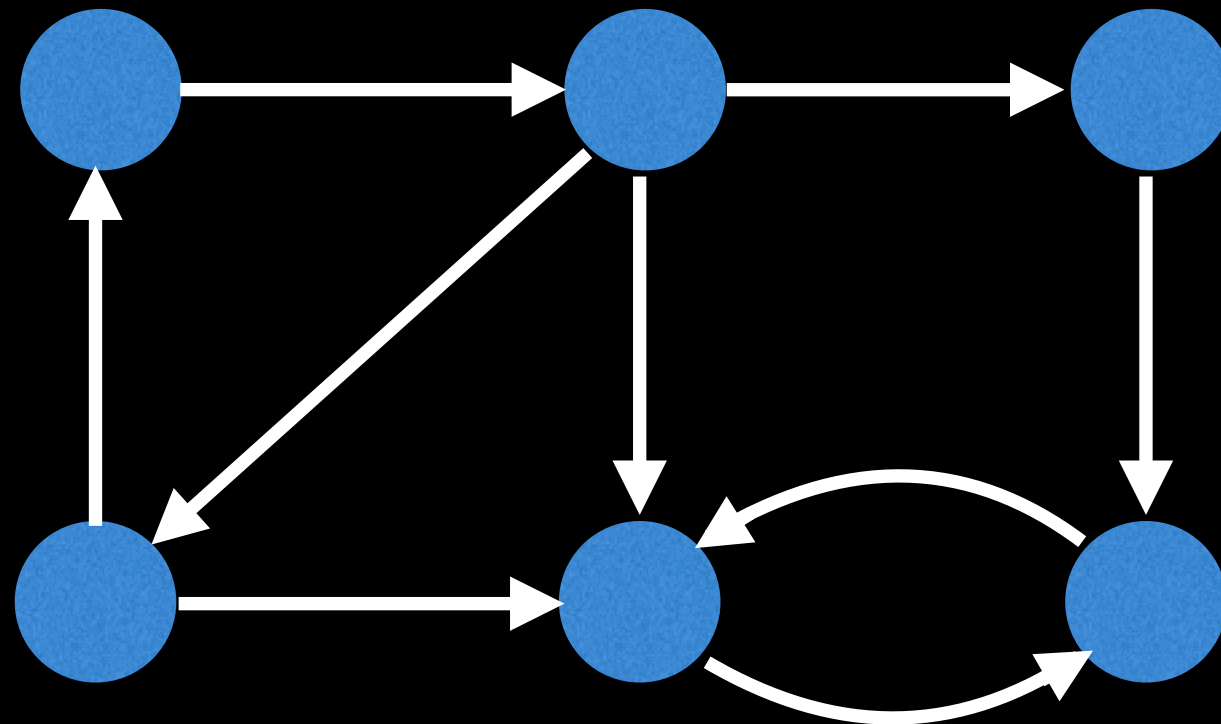
$\infty$

$\infty$

$\infty$

$\infty$

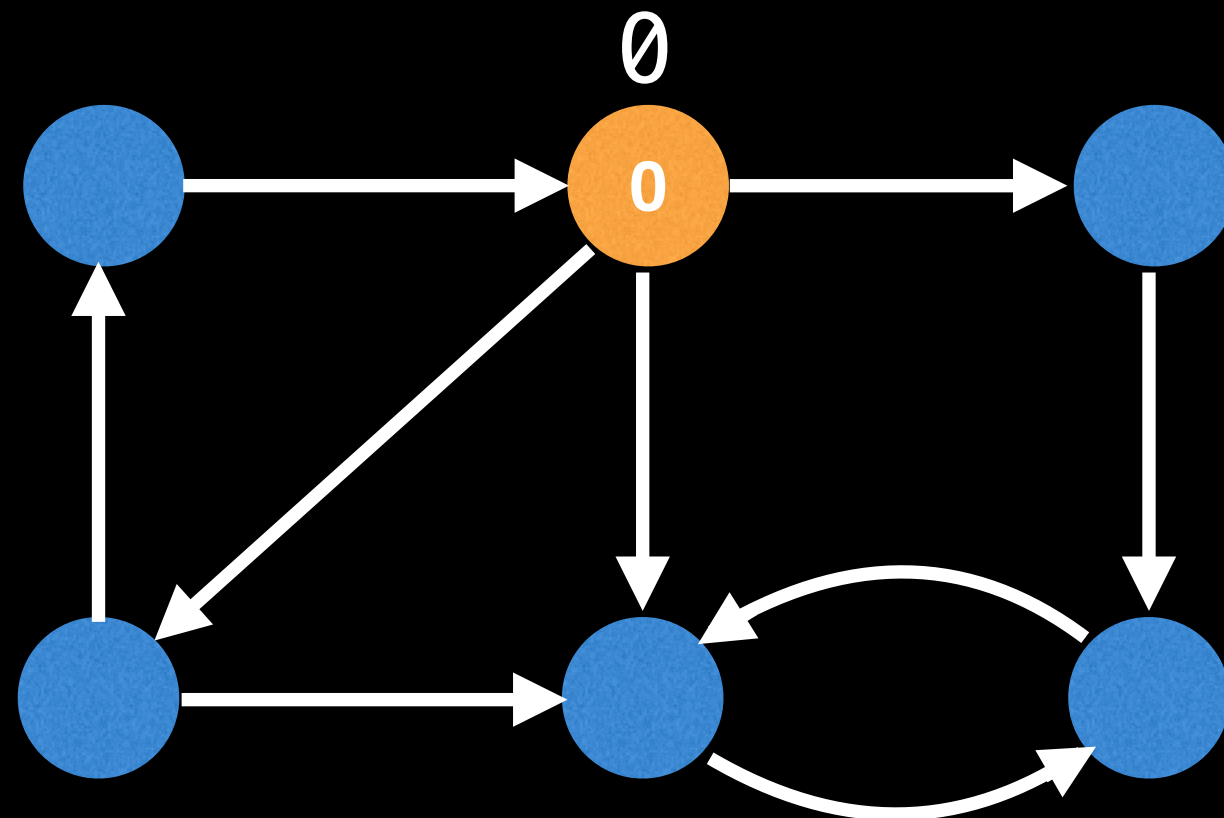
$\infty$



# Low-Link Values

id	0	1	2	3	4	5
lowlink	0	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$

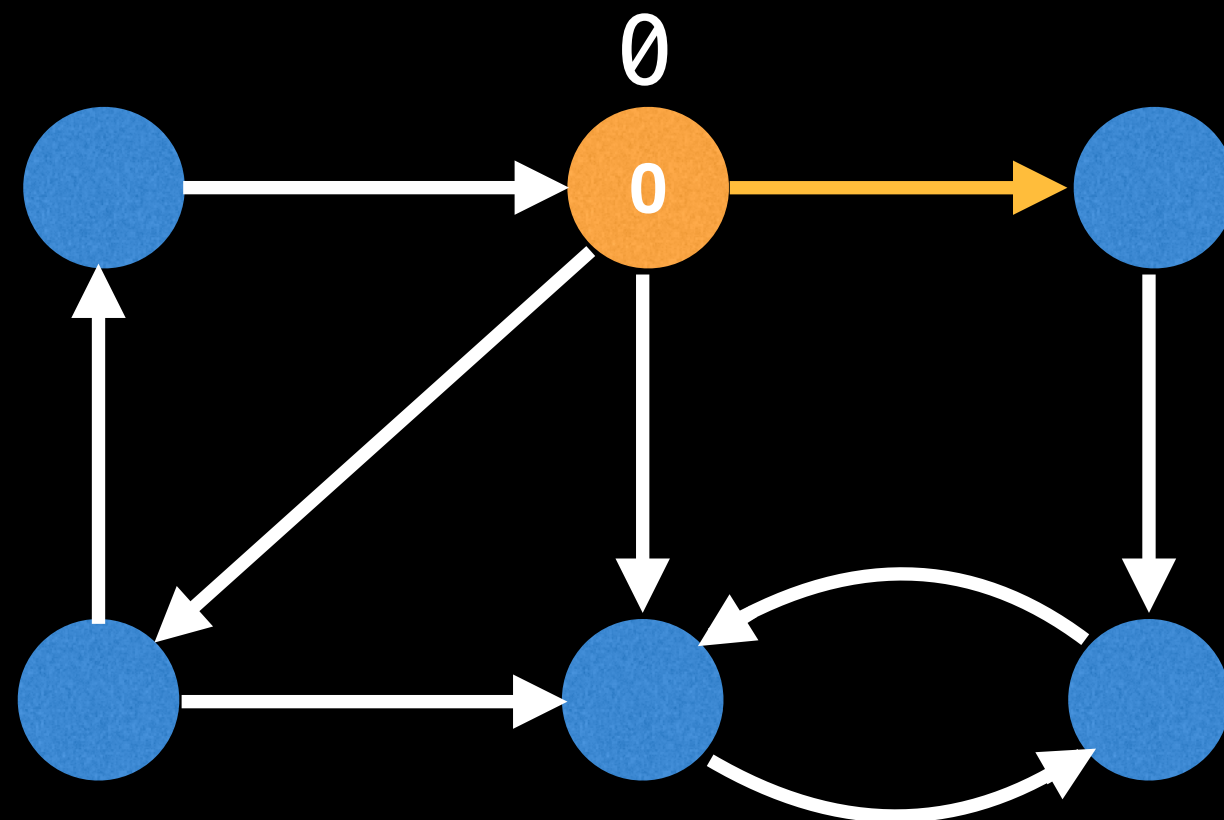
Stack



# Low-Link Values

id	0	1	2	3	4	5
lowlink	0	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$

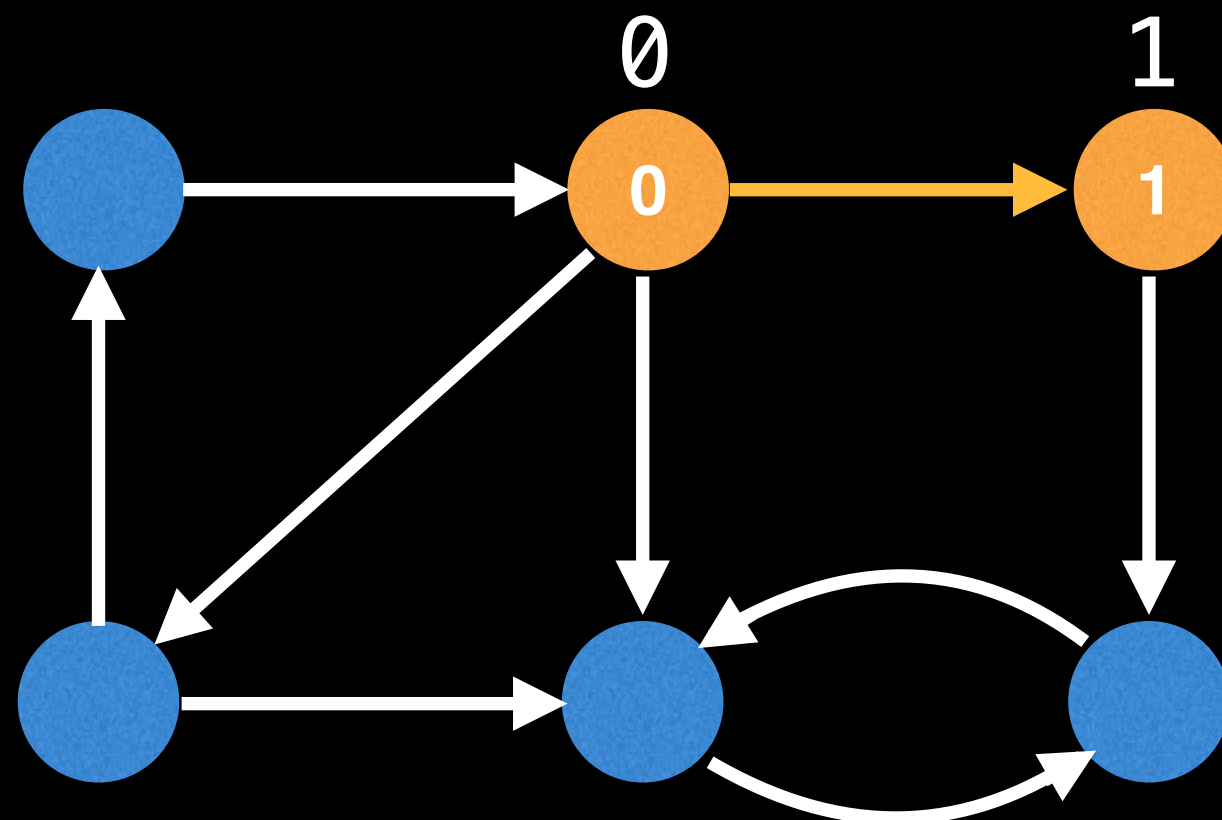
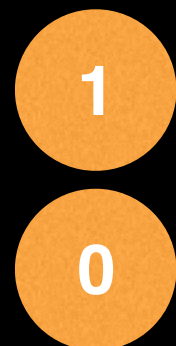
Stack



# Low-Link Values

id	0	1	2	3	4	5
lowlink	0	1	$\infty$	$\infty$	$\infty$	$\infty$

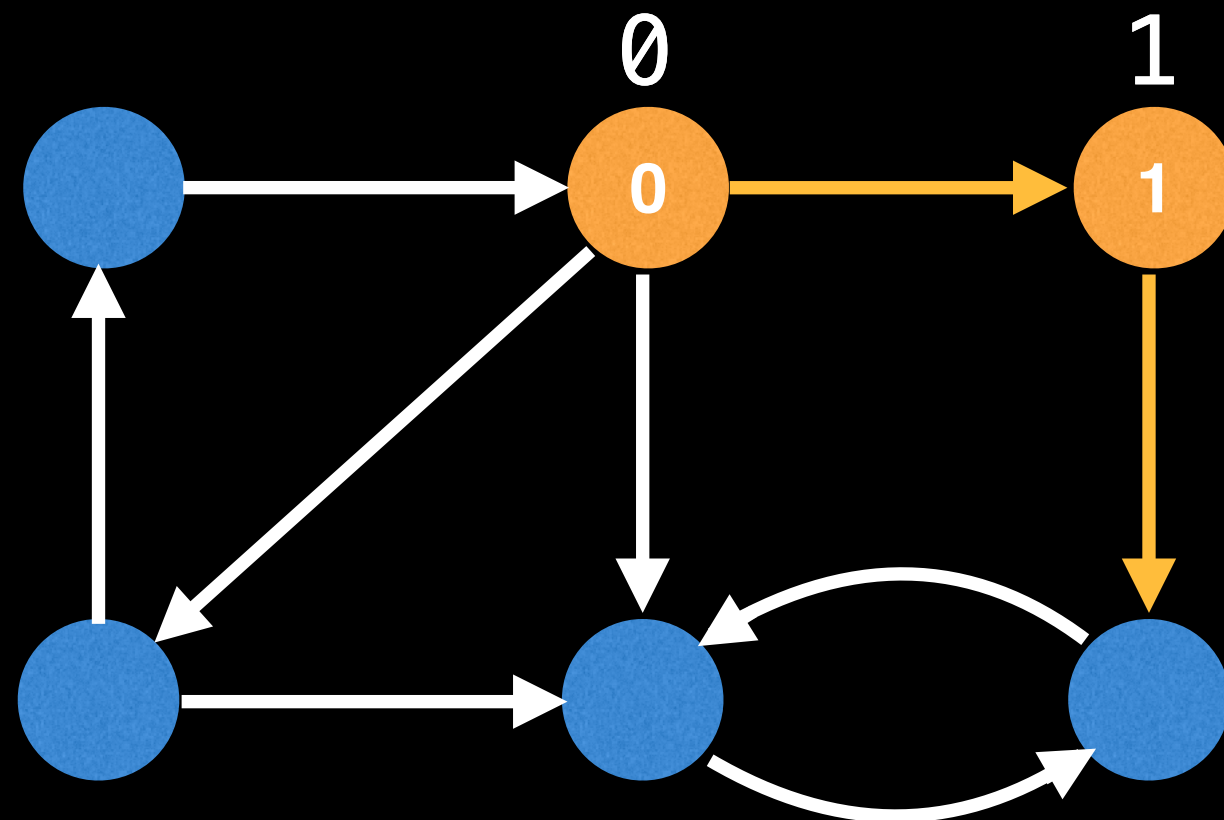
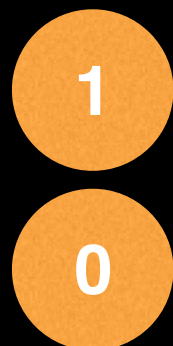
Stack



# Low-Link Values

id	0	1	2	3	4	5
lowlink	0	1	$\infty$	$\infty$	$\infty$	$\infty$

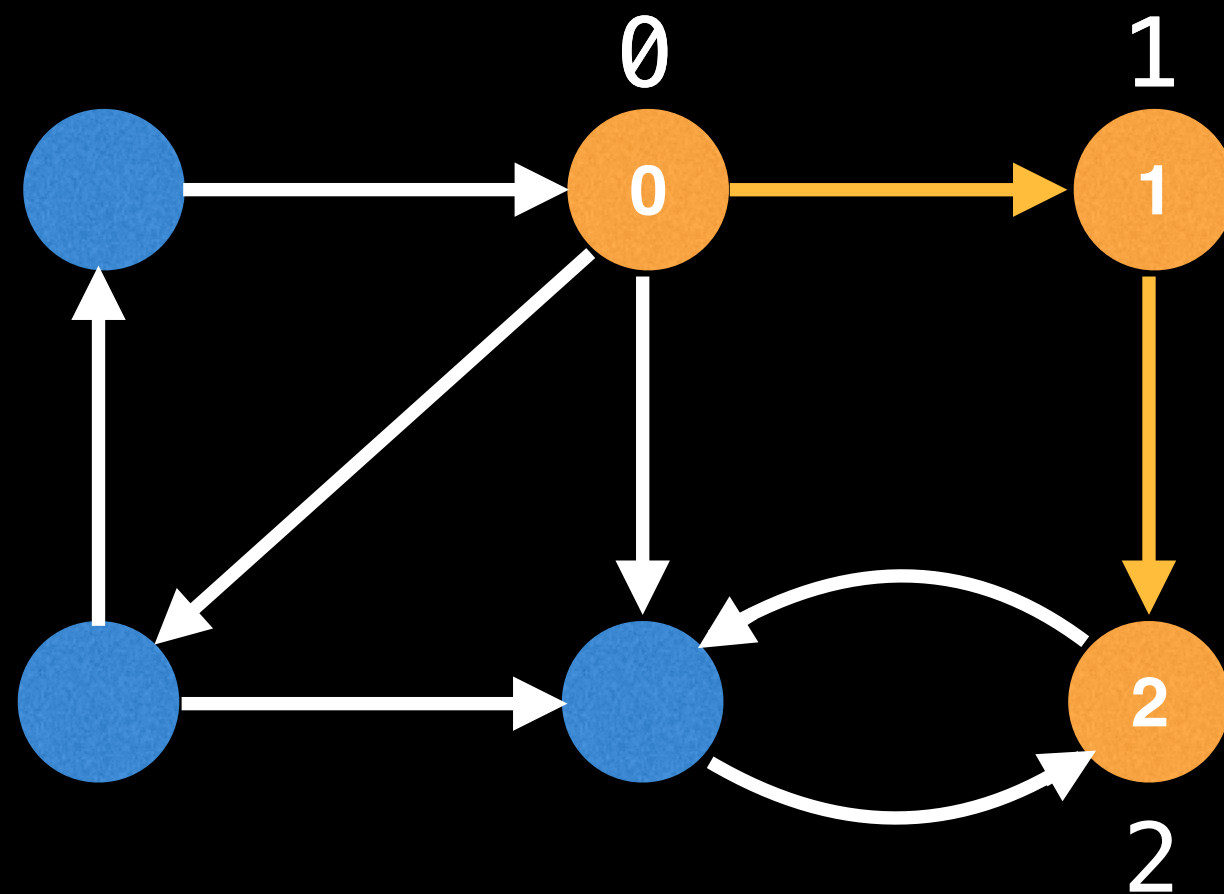
Stack



# Low-Link Values

id	0	1	2	3	4	5
lowlink	0	1	2	$\infty$	$\infty$	$\infty$

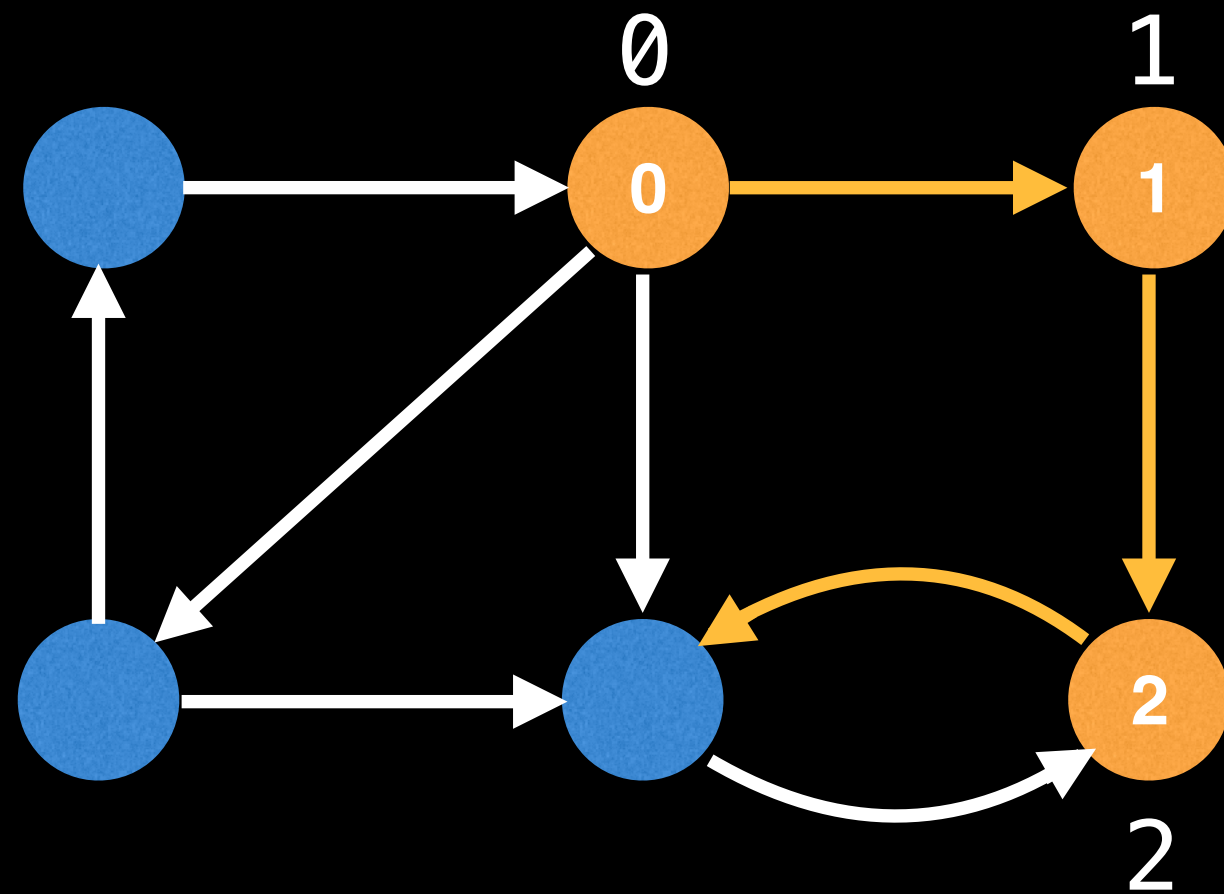
Stack



# Low-Link Values

id	0	1	2	3	4	5
lowlink	0	1	2	$\infty$	$\infty$	$\infty$

Stack

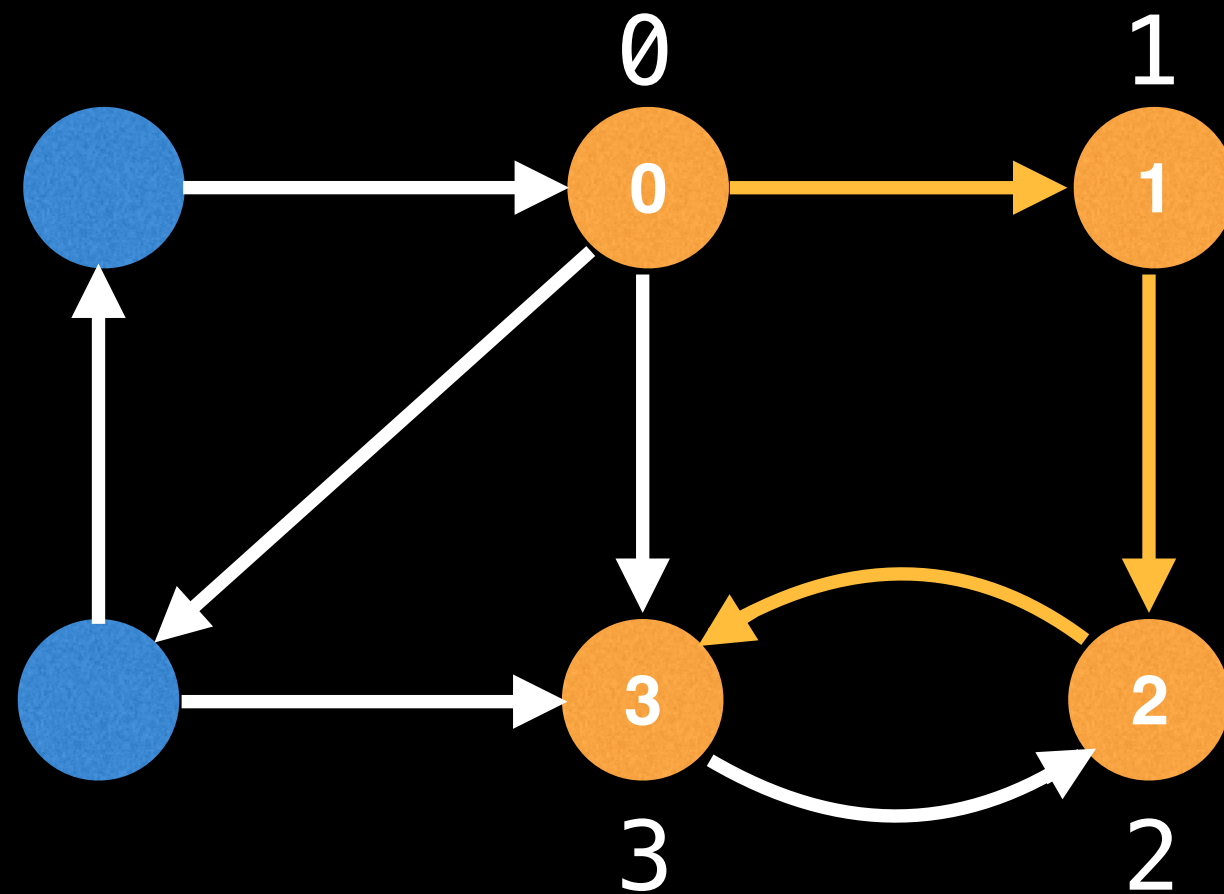
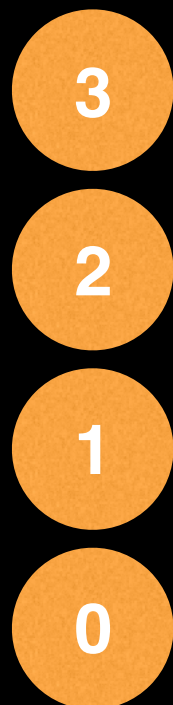




# Low-Link Values

id	0	1	2	3	4	5
lowlink	0	1	2	3	$\infty$	$\infty$

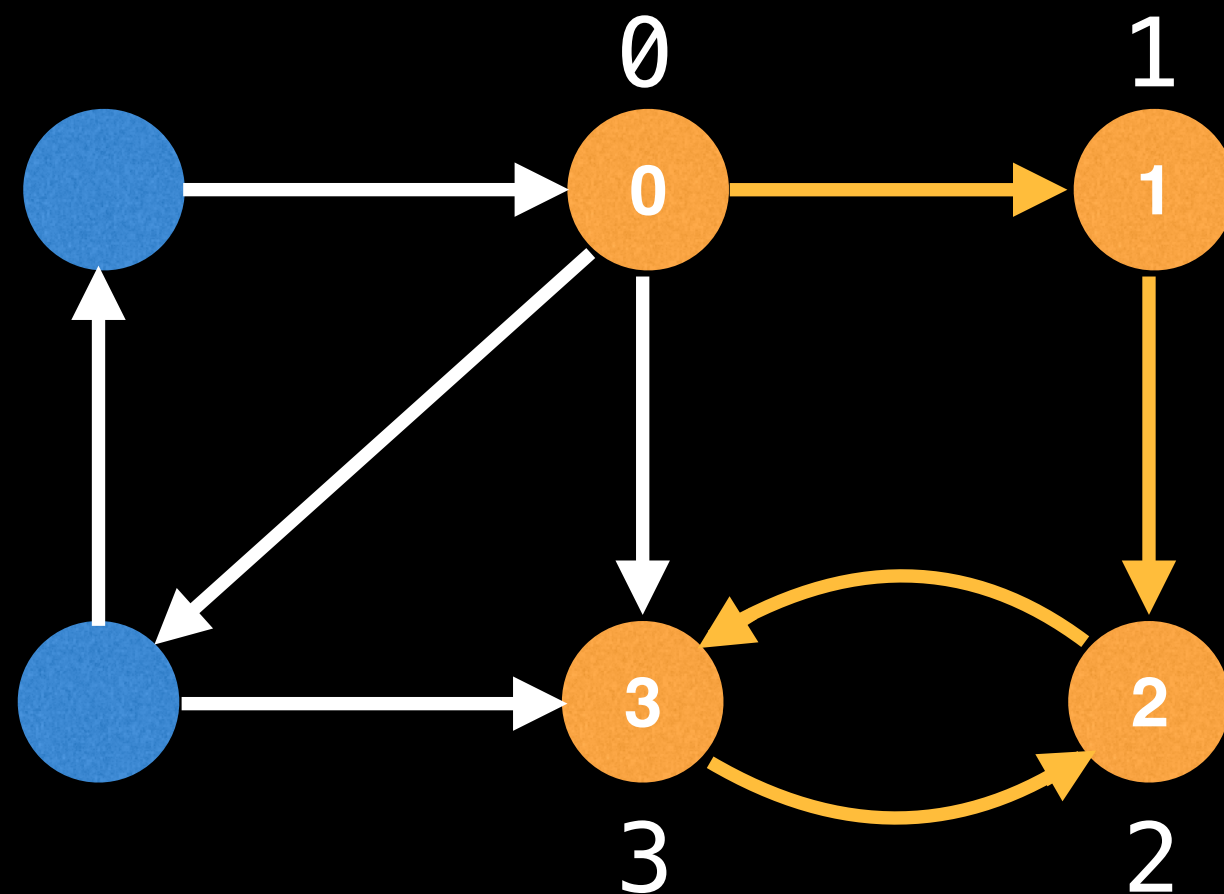
Stack



# Low-Link Values

id	0	1	2	3	4	5
lowlink	0	1	2	3	$\infty$	$\infty$

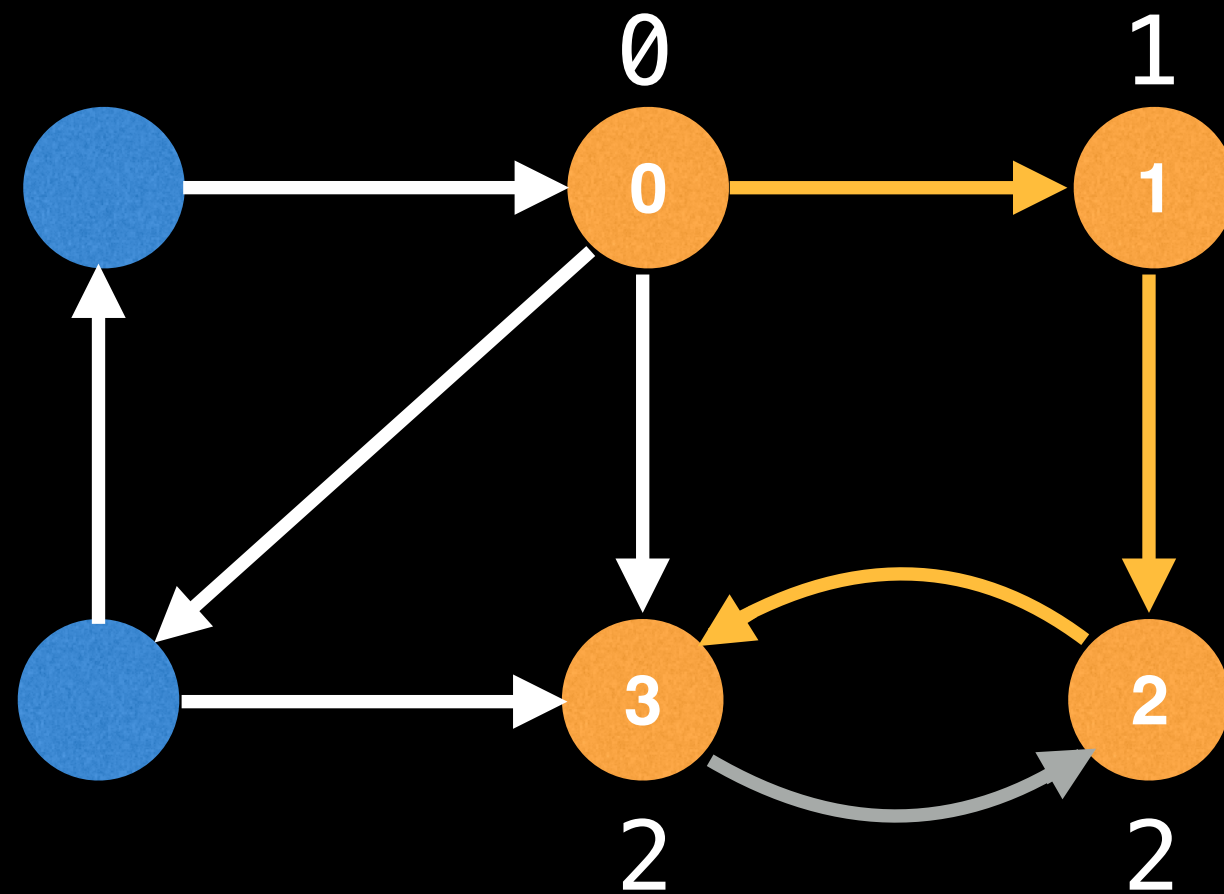
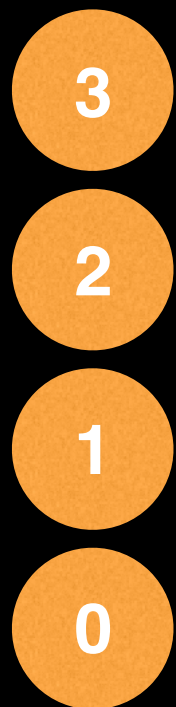
Stack



# Low-Link Values

id	0	1	2	3	4	5
lowlink	0	1	2	2	$\infty$	$\infty$

Stack

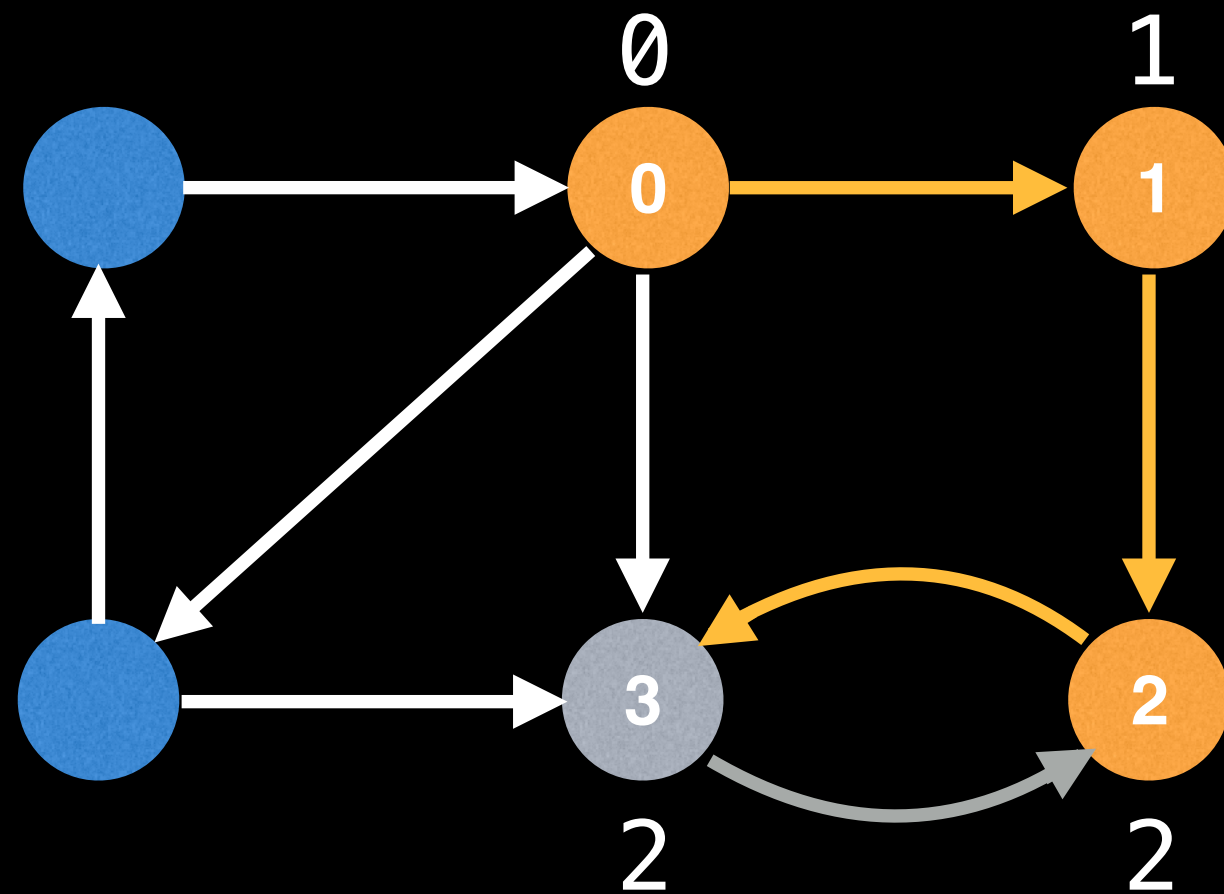
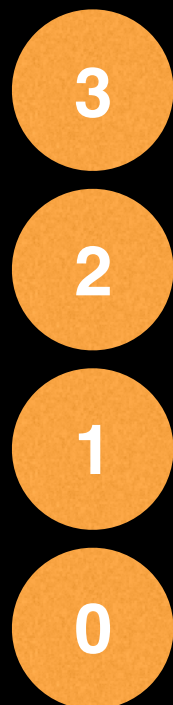


$$\begin{aligned}\text{lowlink}[3] &= \min(\text{lowlink}[3], \text{lowlink}[2]) \\ &= 2\end{aligned}$$

# Low-Link Values

id	0	1	2	3	4	5
lowlink	0	1	2	2	$\infty$	$\infty$

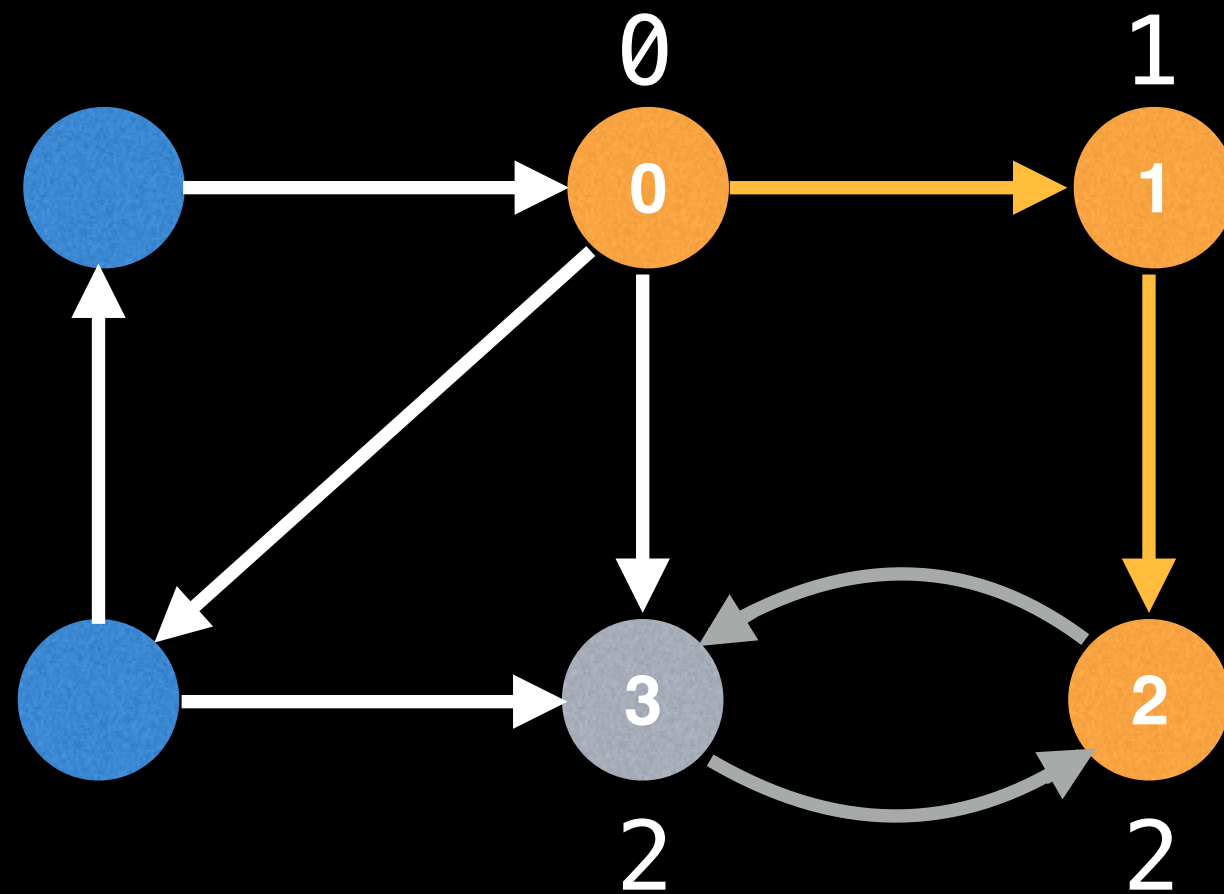
Stack



# Low-Link Values

id	0	1	2	3	4	5
lowlink	0	1	2	2	$\infty$	$\infty$

Stack

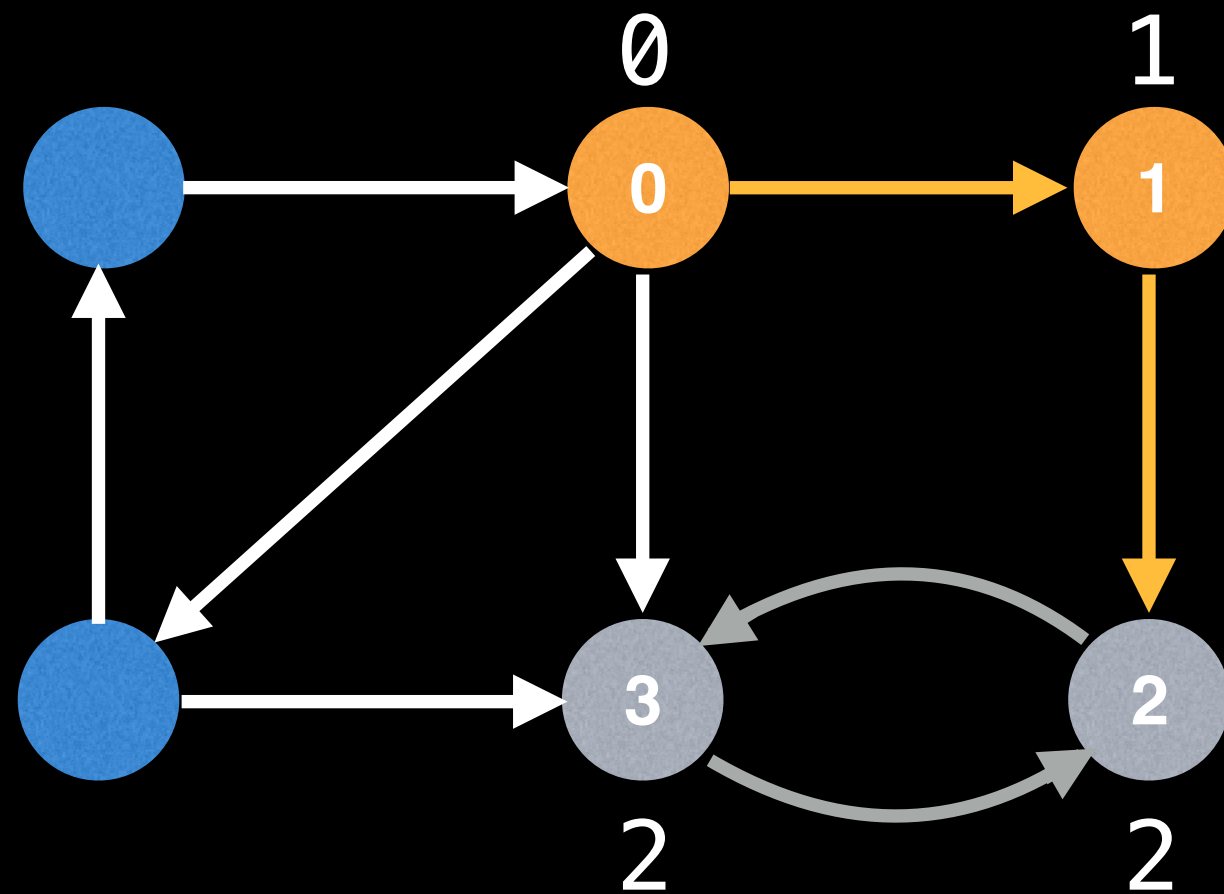


$$\begin{aligned}\text{lowlink}[2] &= \min(\text{lowlink}[2], \text{lowlink}[3]) \\ &= 2\end{aligned}$$

# Low-Link Values

id	0	1	2	3	4	5
lowlink	0	1	2	2	$\infty$	$\infty$

Stack

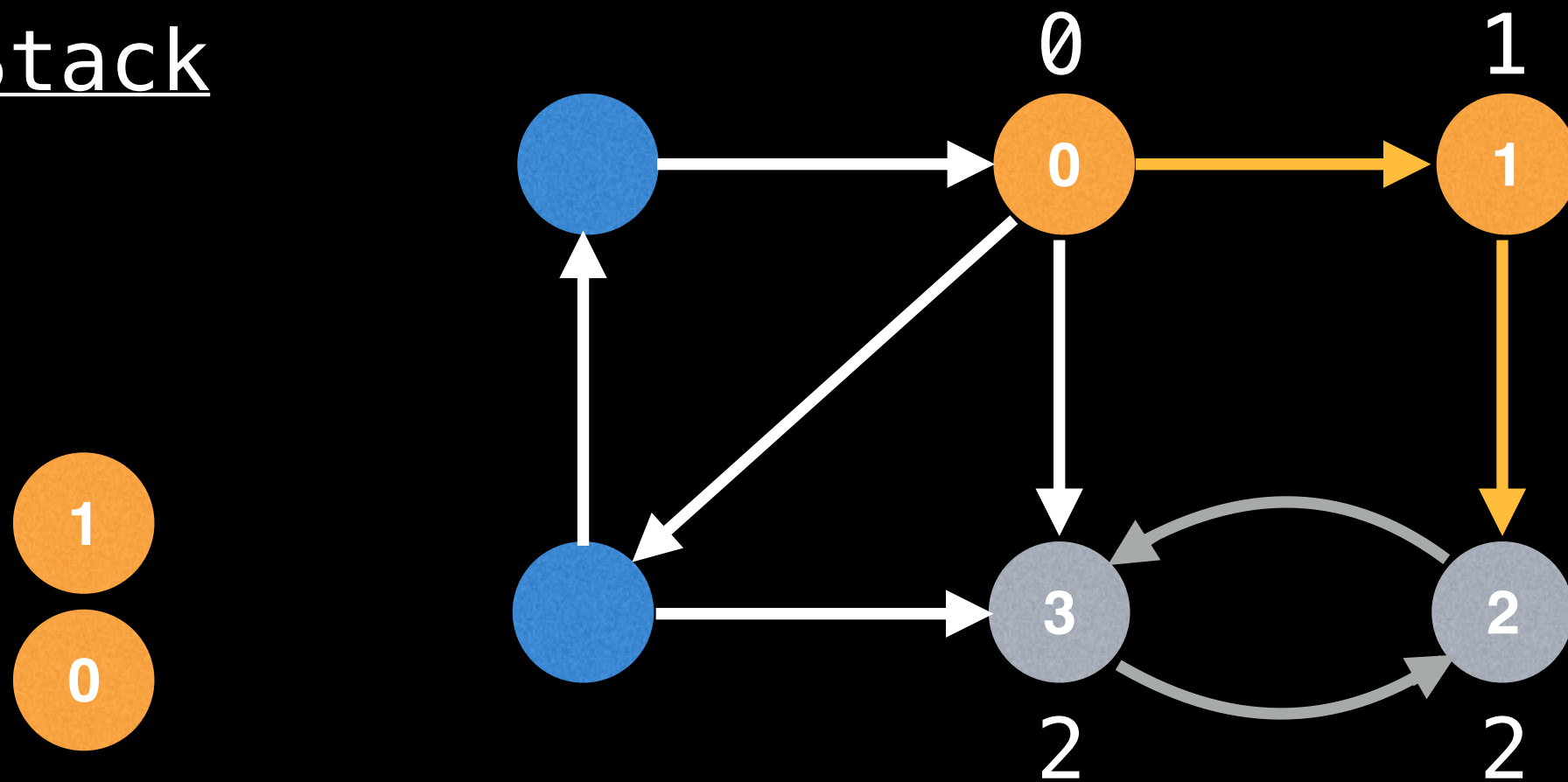


When a completed SCC is found (current node has visited all its neighbours and its lowlink value equals its id) pop off all associated nodes off the stack.

# Low-Link Values

id	0	1	2	3	4	5
lowlink	0	1	2	2	$\infty$	$\infty$

Stack

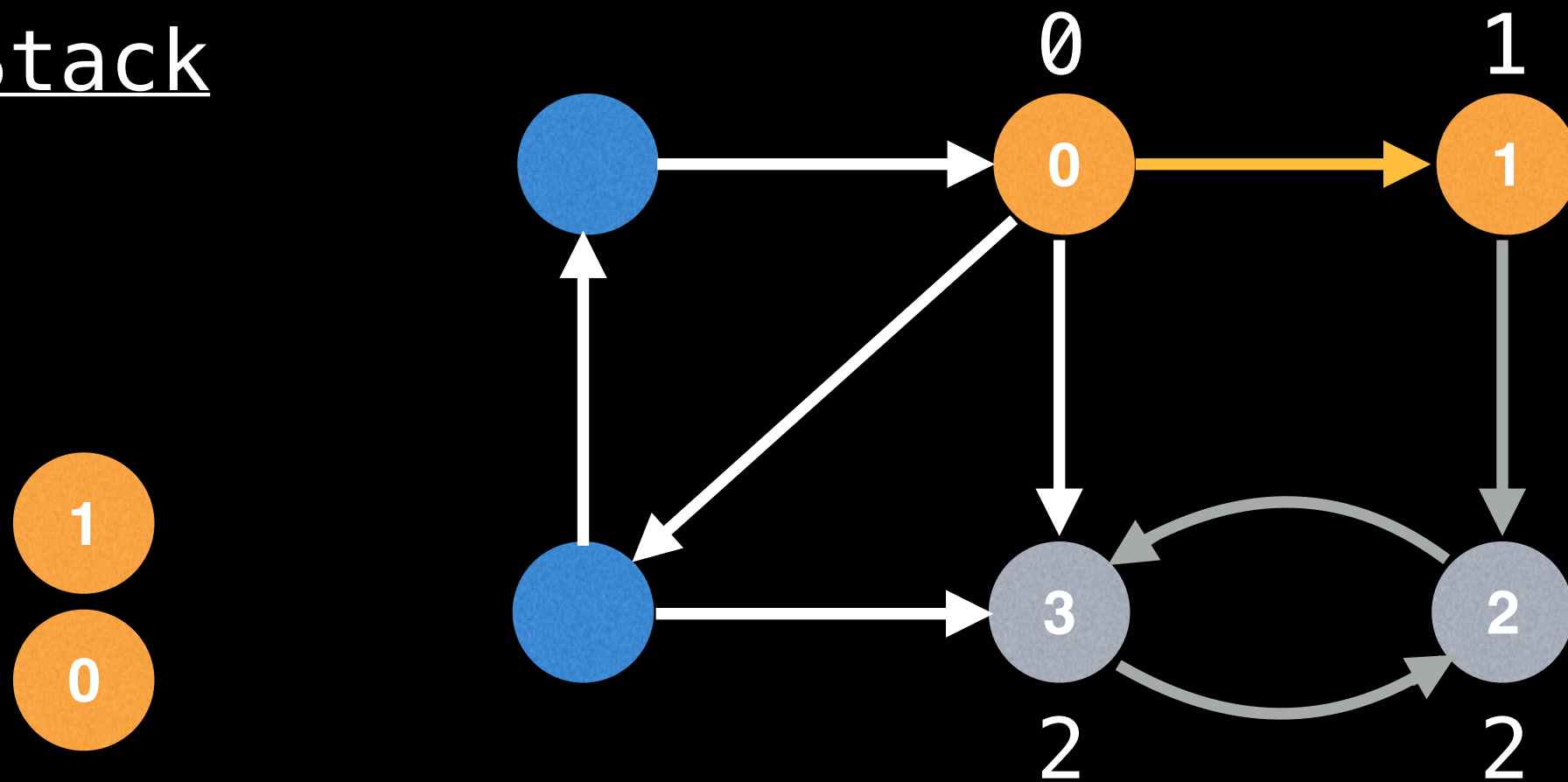


When a completed SCC is found (current node has visited all its neighbours and its lowlink value equals its id) pop off all associated nodes off the stack.

# Low-Link Values

id	0	1	2	3	4	5
lowlink	0	1	2	2	$\infty$	$\infty$

Stack



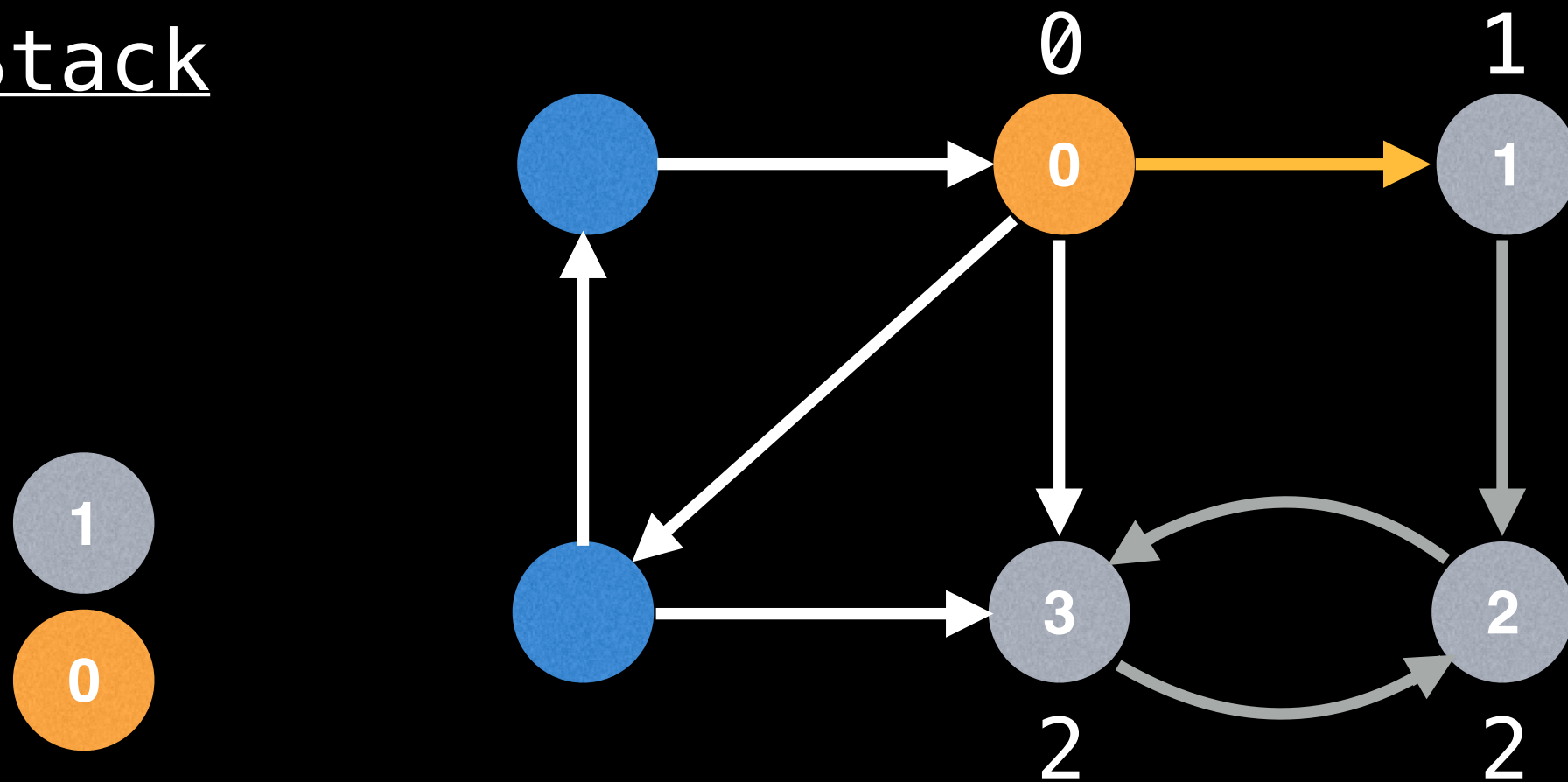
Node 2 is not on stack so don't min  
with its low-link value.



# Low-Link Values

id	0	1	2	3	4	5
lowlink	0	1	2	2	$\infty$	$\infty$

Stack

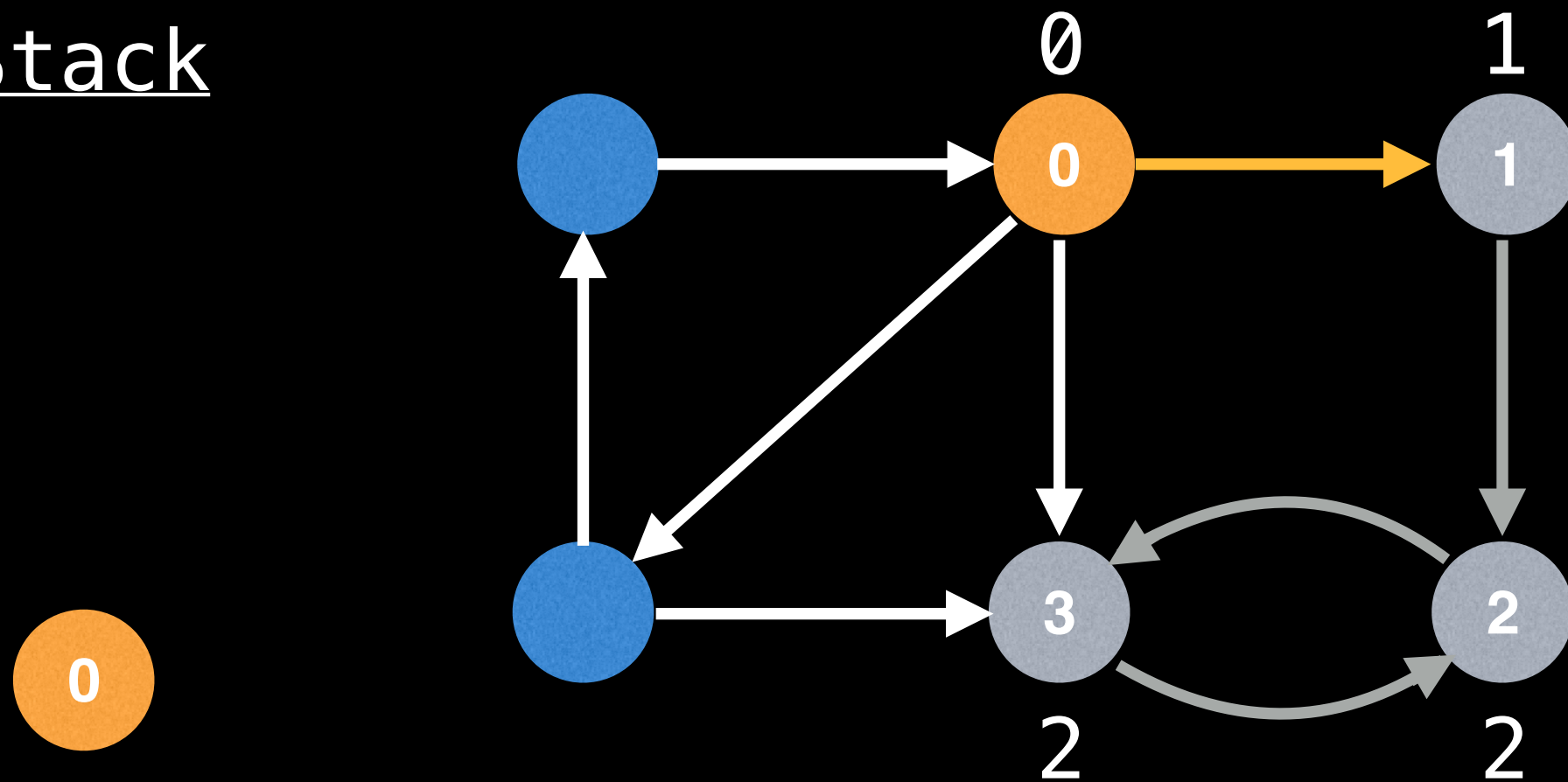


When a completed SCC is found (current node has visited all its neighbours and its lowlink value equals its id) pop off all associated nodes off the stack.

# Low-Link Values

id	0	1	2	3	4	5
lowlink	0	1	2	2	$\infty$	$\infty$

Stack

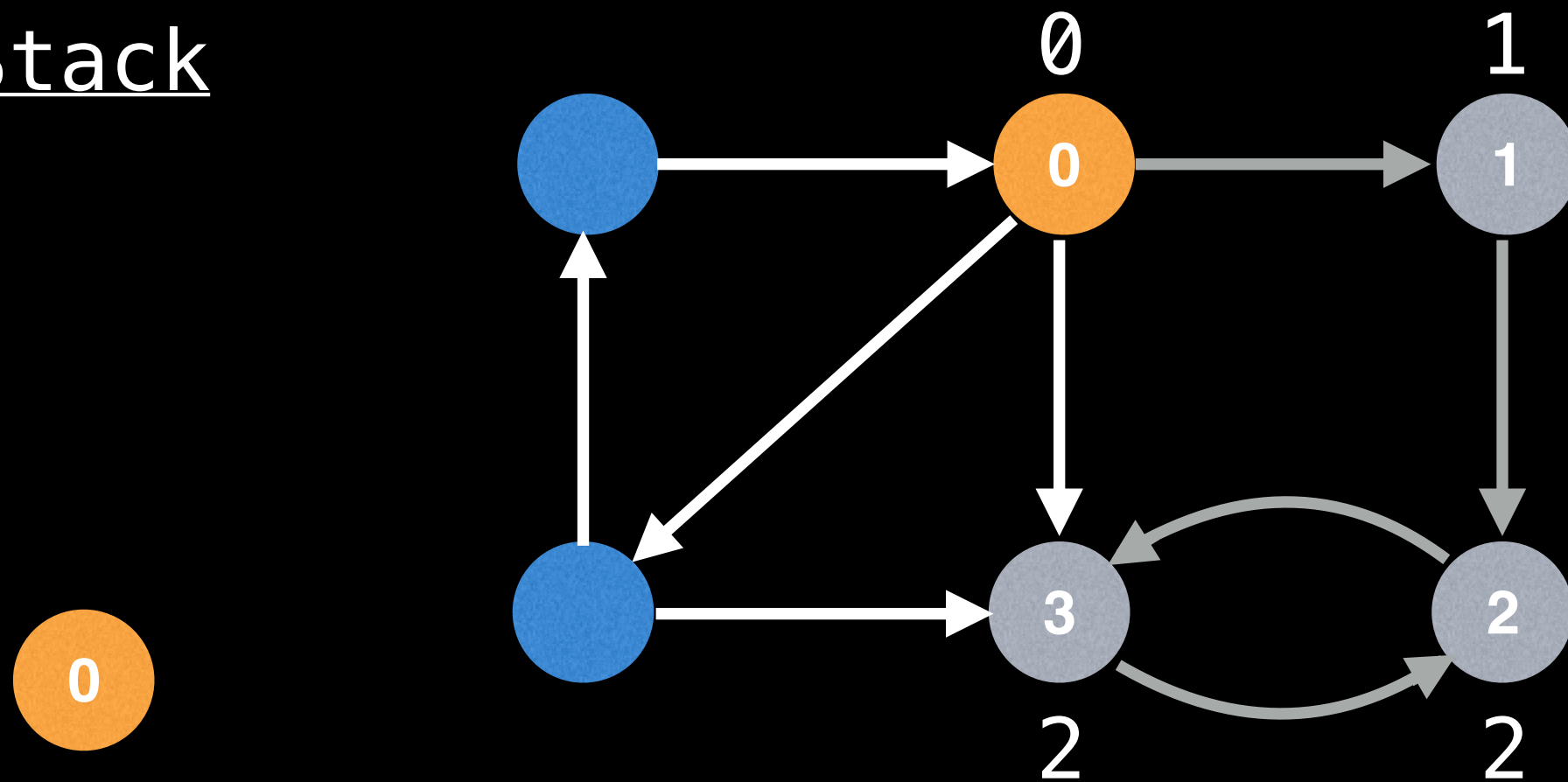


When a completed SCC is found (current node has visited all its neighbours and its lowlink value equals its id) pop off all associated nodes off the stack.

# Low-Link Values

id	0	1	2	3	4	5
lowlink	0	1	2	2	$\infty$	$\infty$

Stack

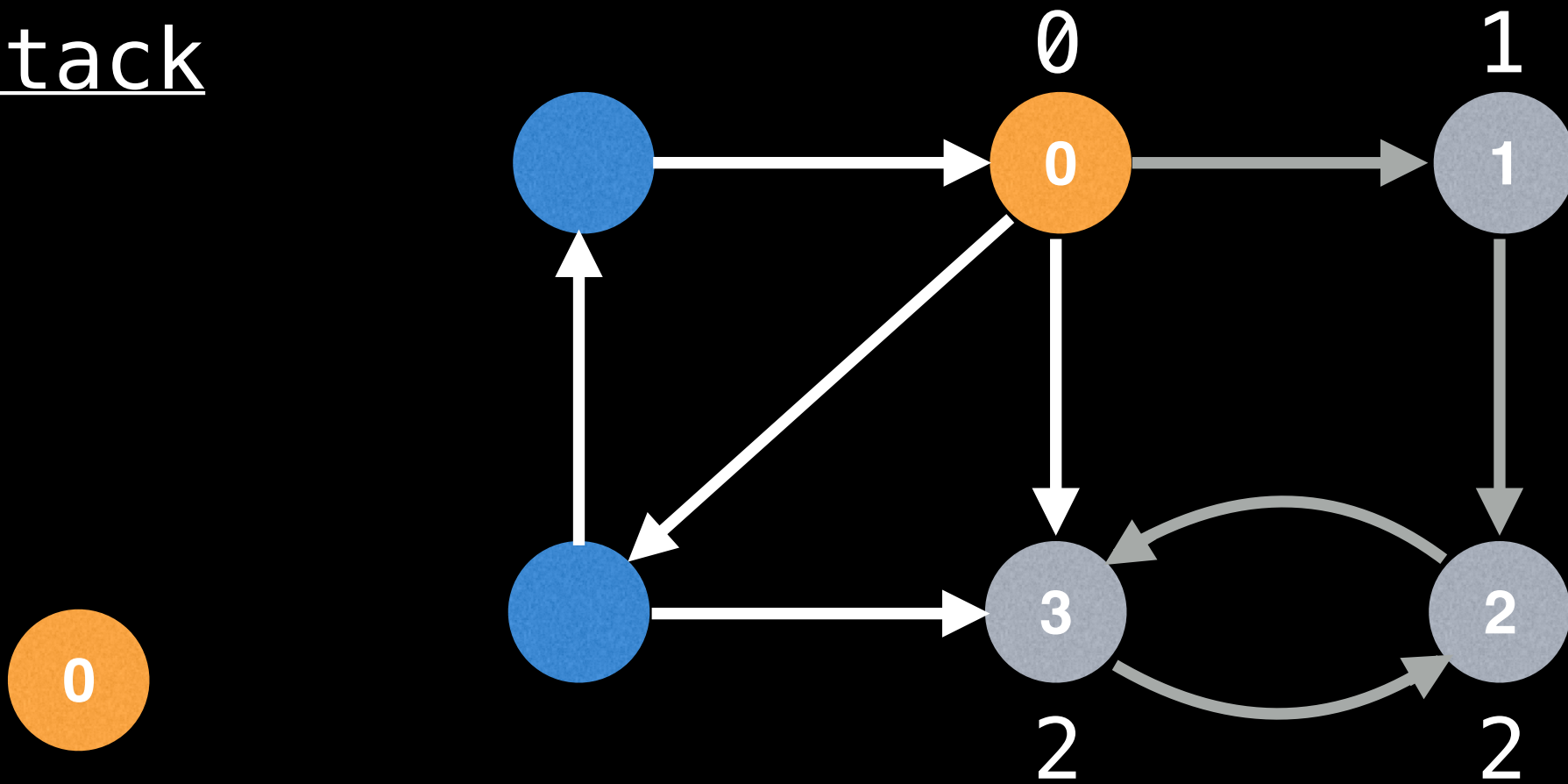


Node 1 is not on stack so don't min  
with its low-link value.

# Low-Link Values

id	0	1	2	3	4	5
lowlink	0	1	2	2	$\infty$	$\infty$

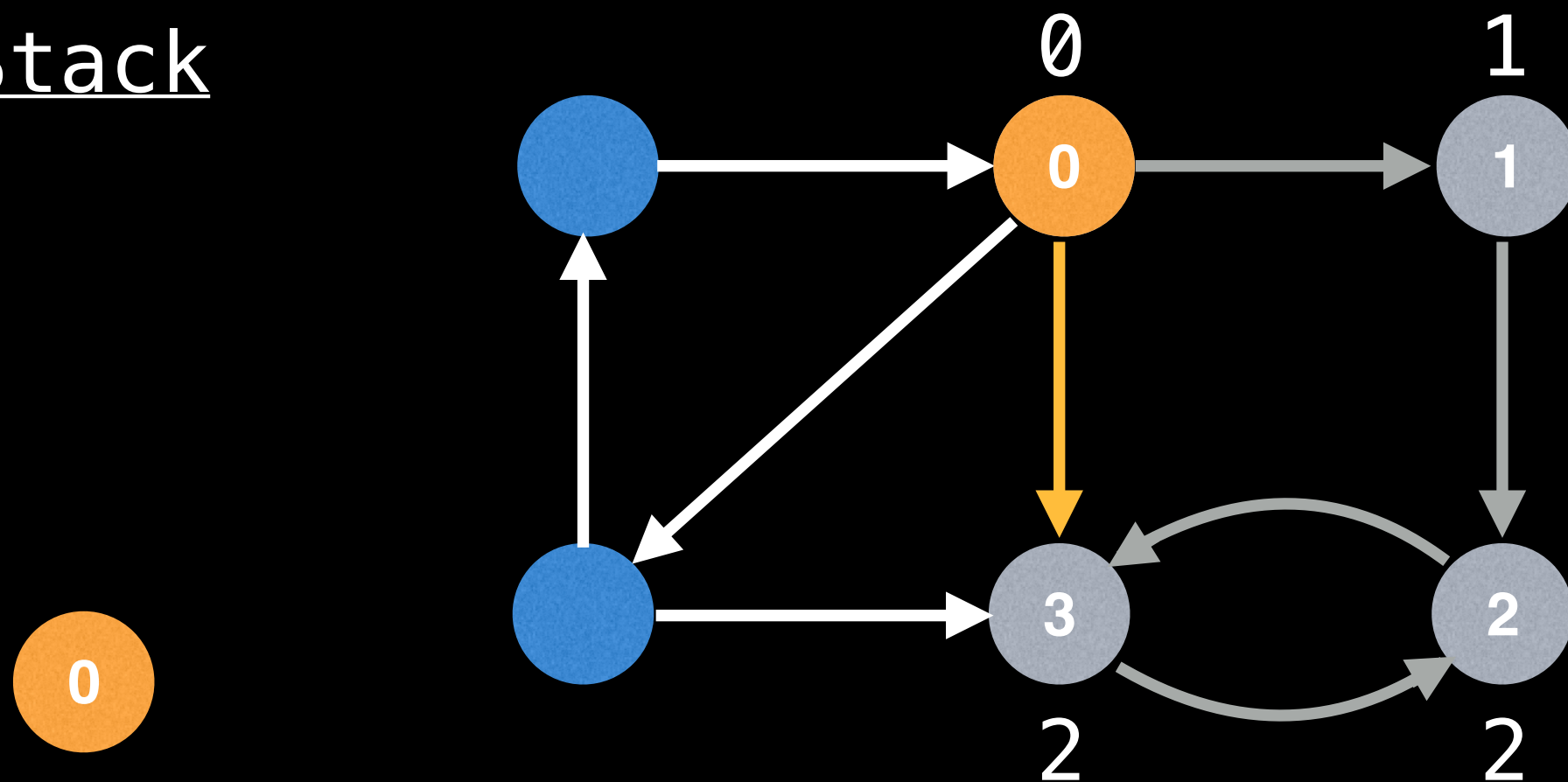
# Stack



# Low-Link Values

id	0	1	2	3	4	5
lowlink	0	1	2	2	$\infty$	$\infty$

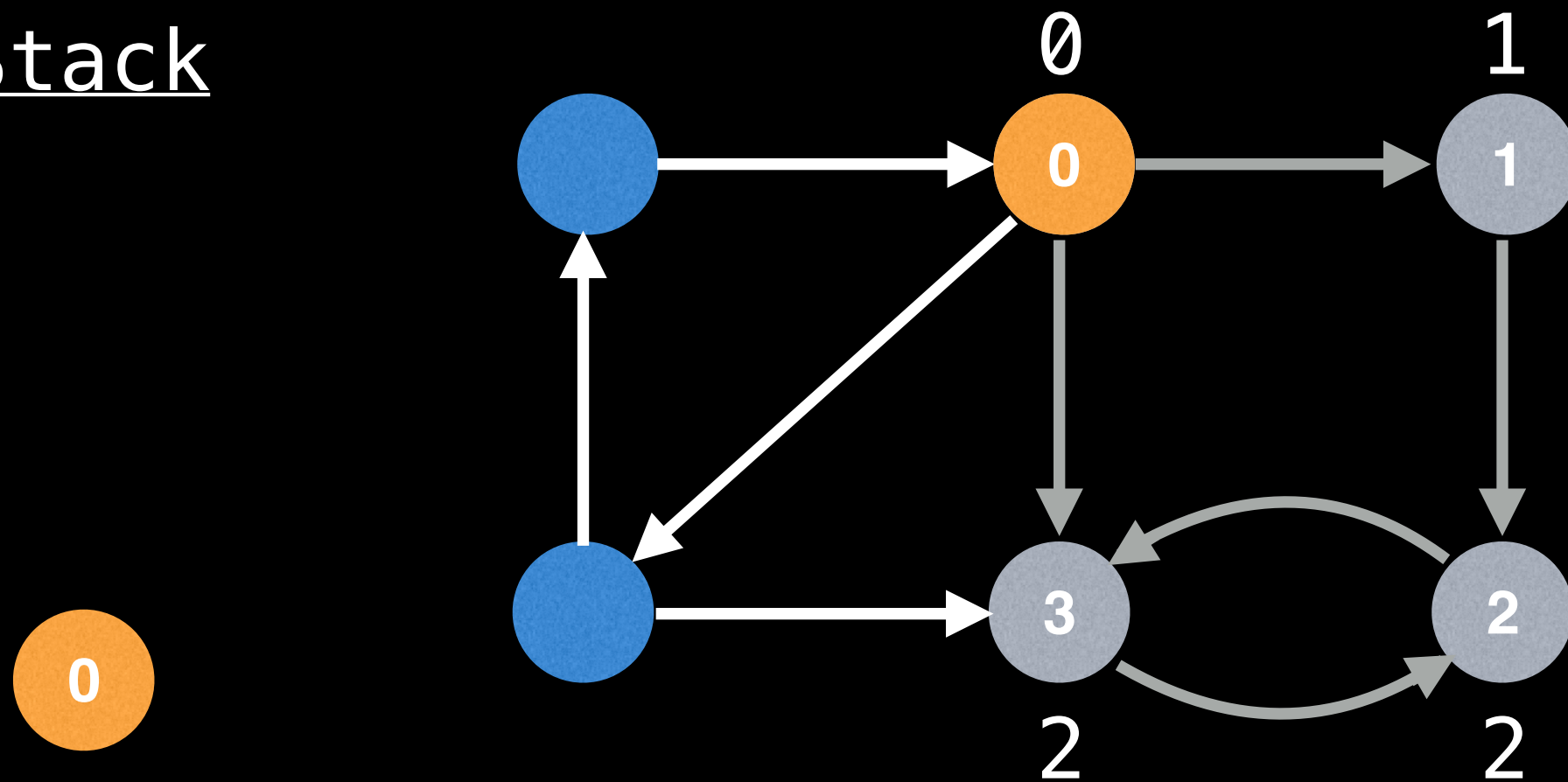
Stack



# Low-Link Values

id	0	1	2	3	4	5
lowlink	0	1	2	2	$\infty$	$\infty$

Stack

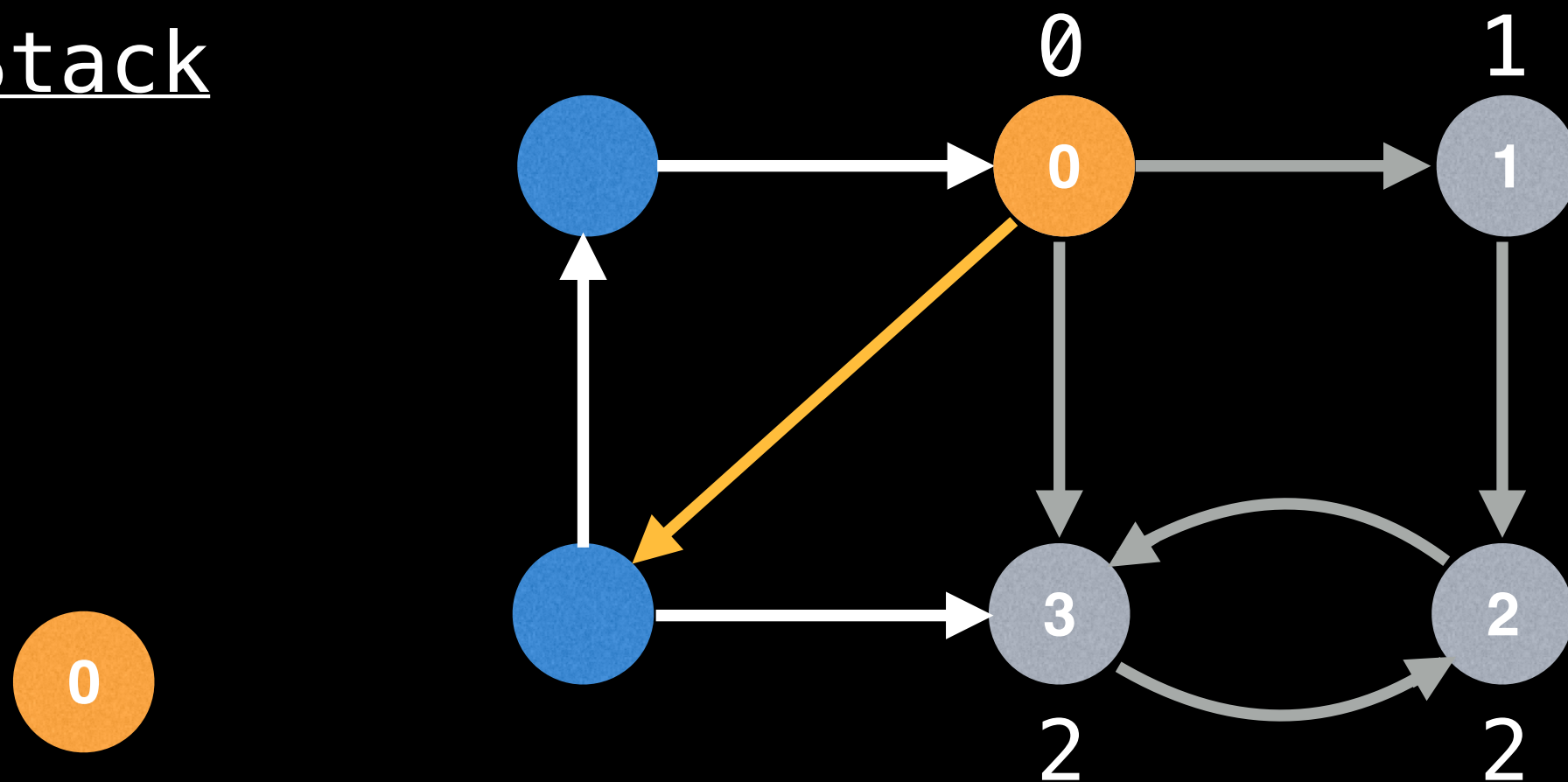


Node 3 is not on stack so don't min  
with its low-link value.

# Low-Link Values

id	0	1	2	3	4	5
lowlink	0	1	2	2	$\infty$	$\infty$

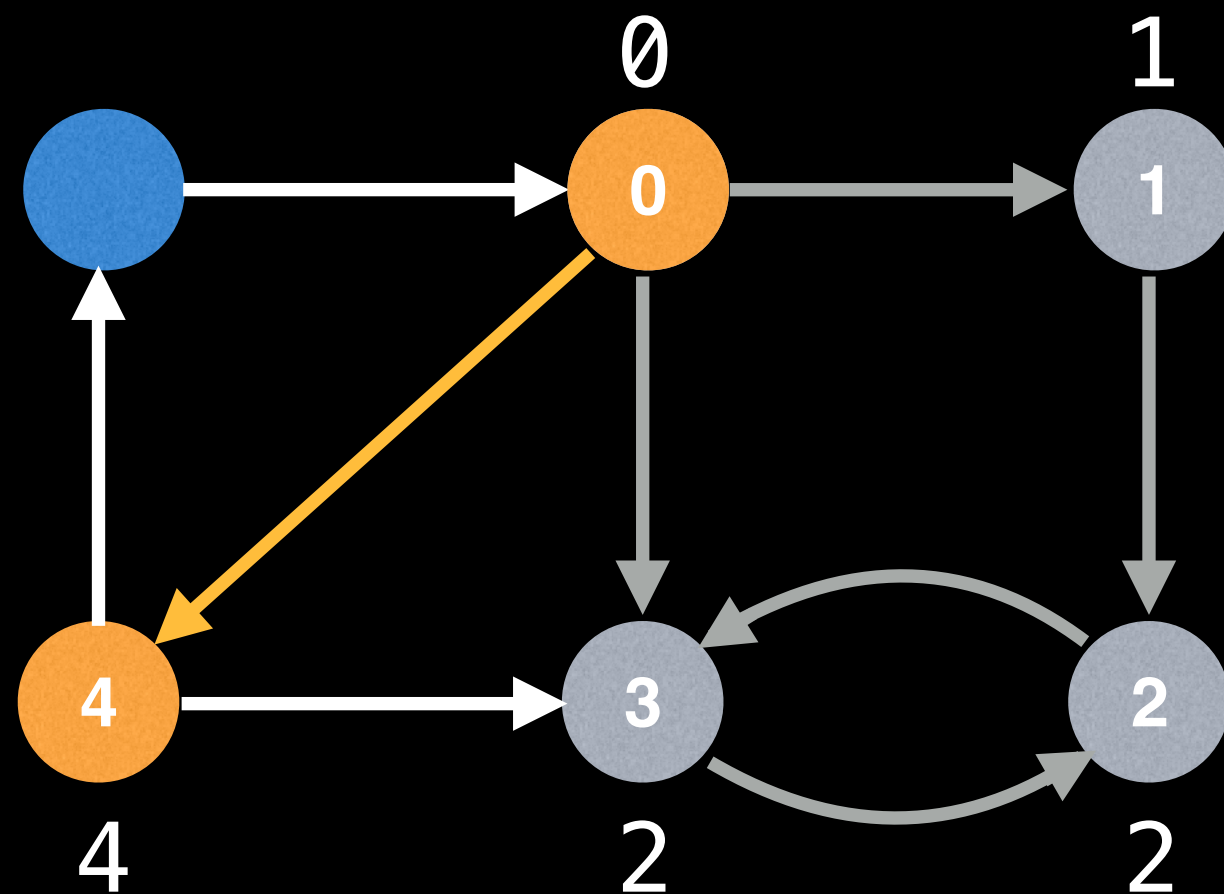
Stack



# Low-Link Values

id	0	1	2	3	4	5
lowlink	0	1	2	2	4	$\infty$

Stack

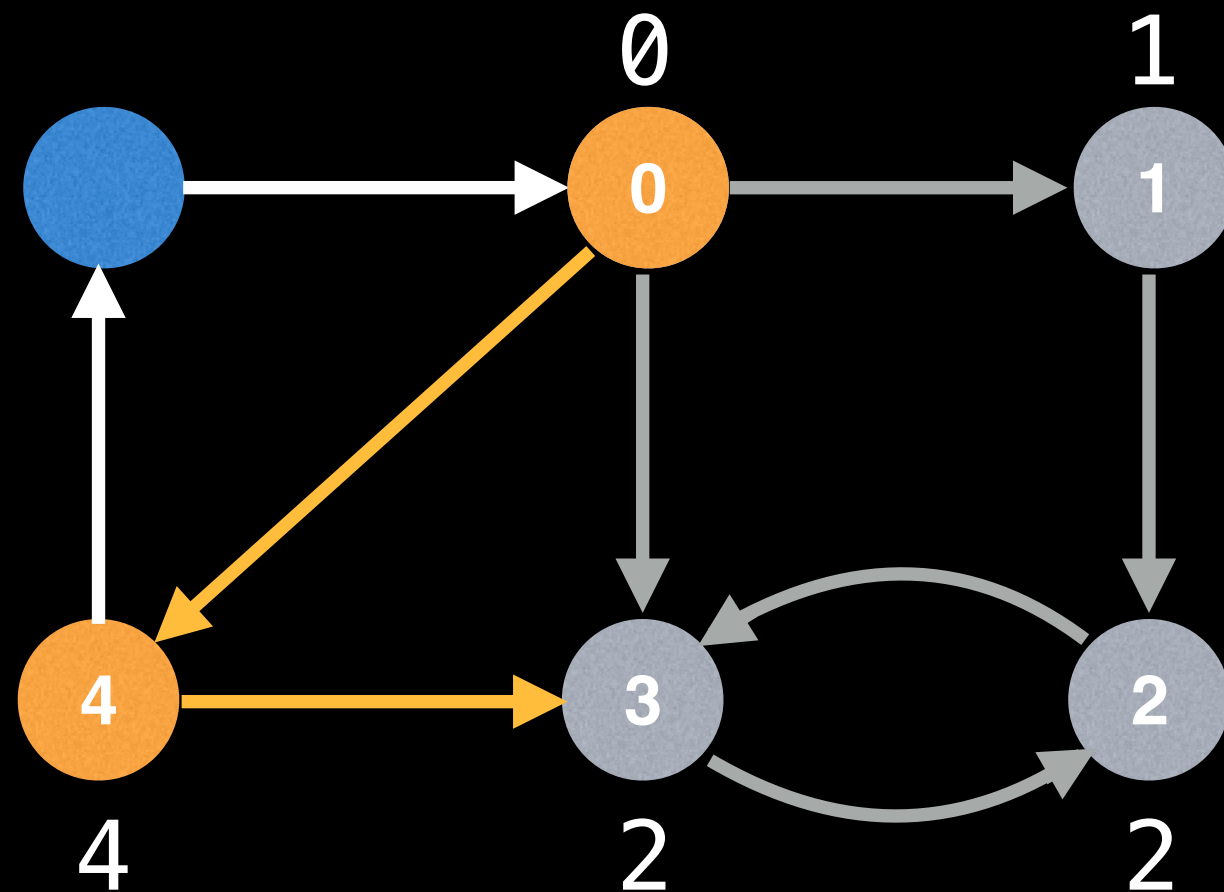




# Low-Link Values

id	0	1	2	3	4	5
lowlink	0	1	2	2	4	$\infty$

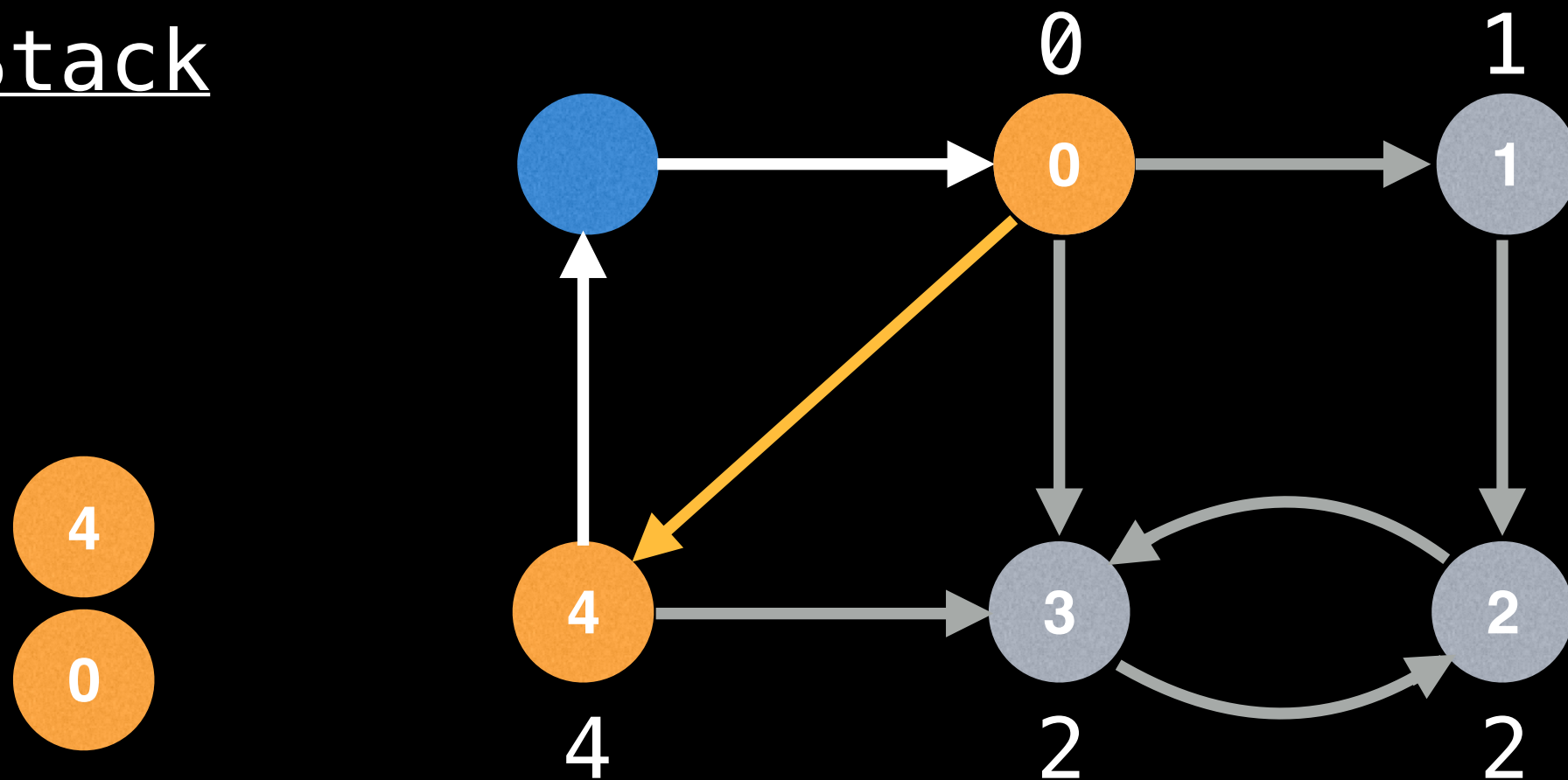
Stack



# Low-Link Values

id	0	1	2	3	4	5
lowlink	0	1	2	2	4	$\infty$

Stack

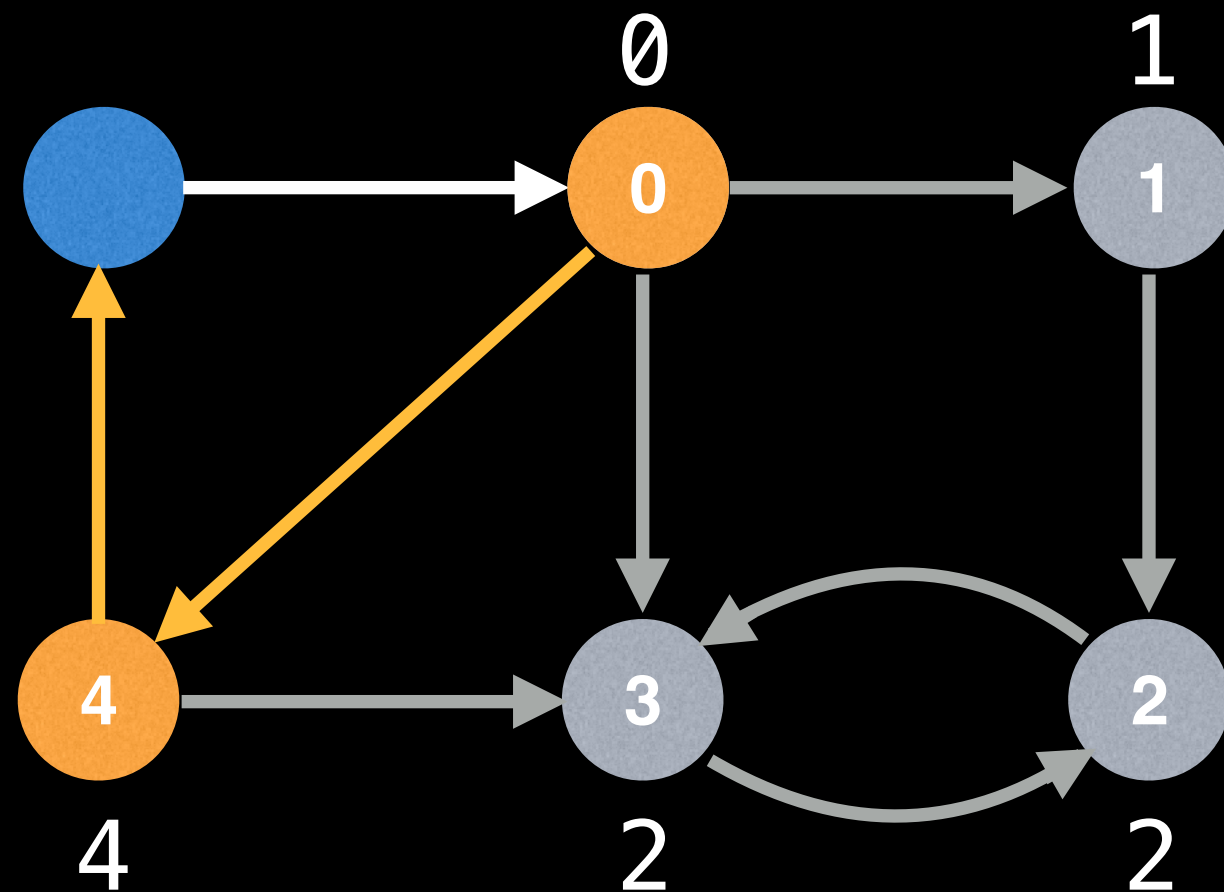


Node 3 is not on stack so don't min  
with its low-link value.

# Low-Link Values

id	0	1	2	3	4	5
lowlink	0	1	2	2	4	$\infty$

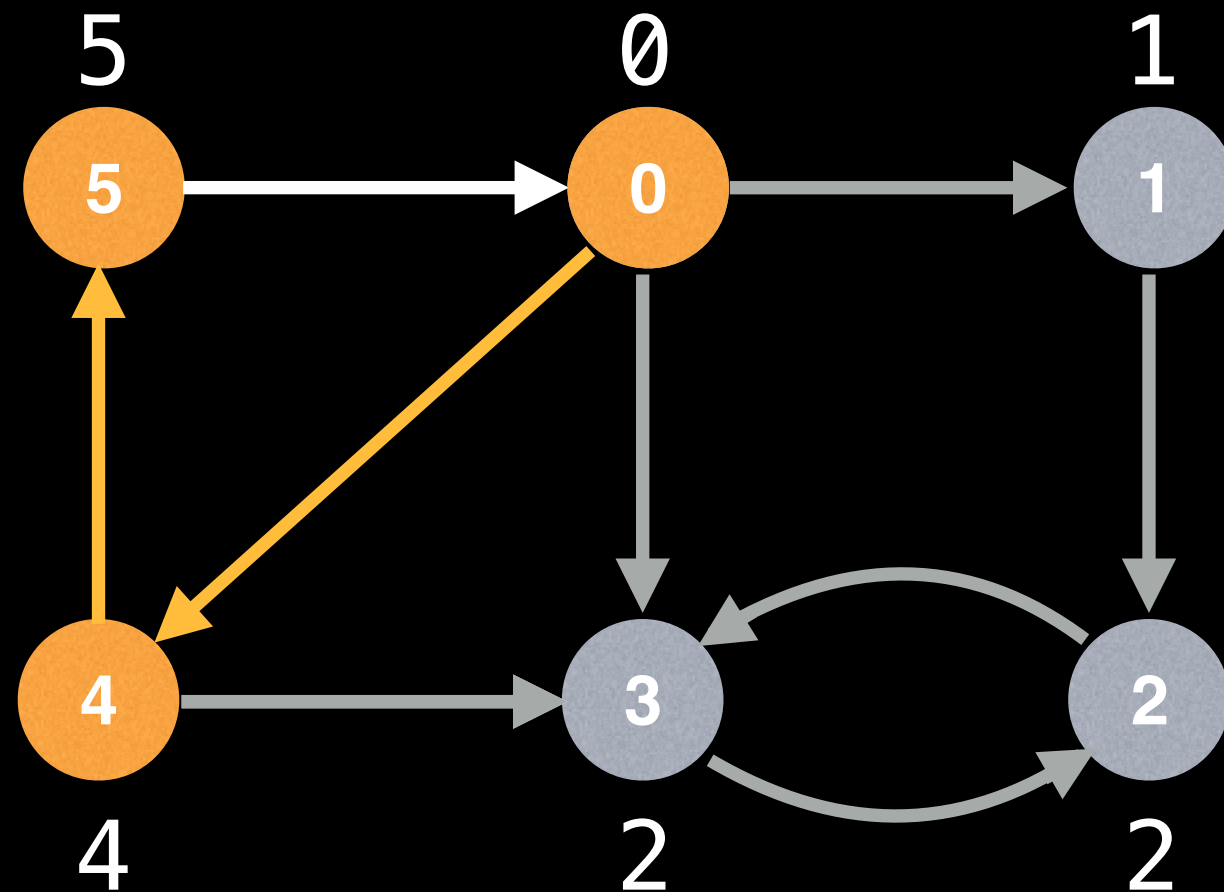
Stack



# Low-Link Values

id	0	1	2	3	4	5
lowlink	0	1	2	2	4	5

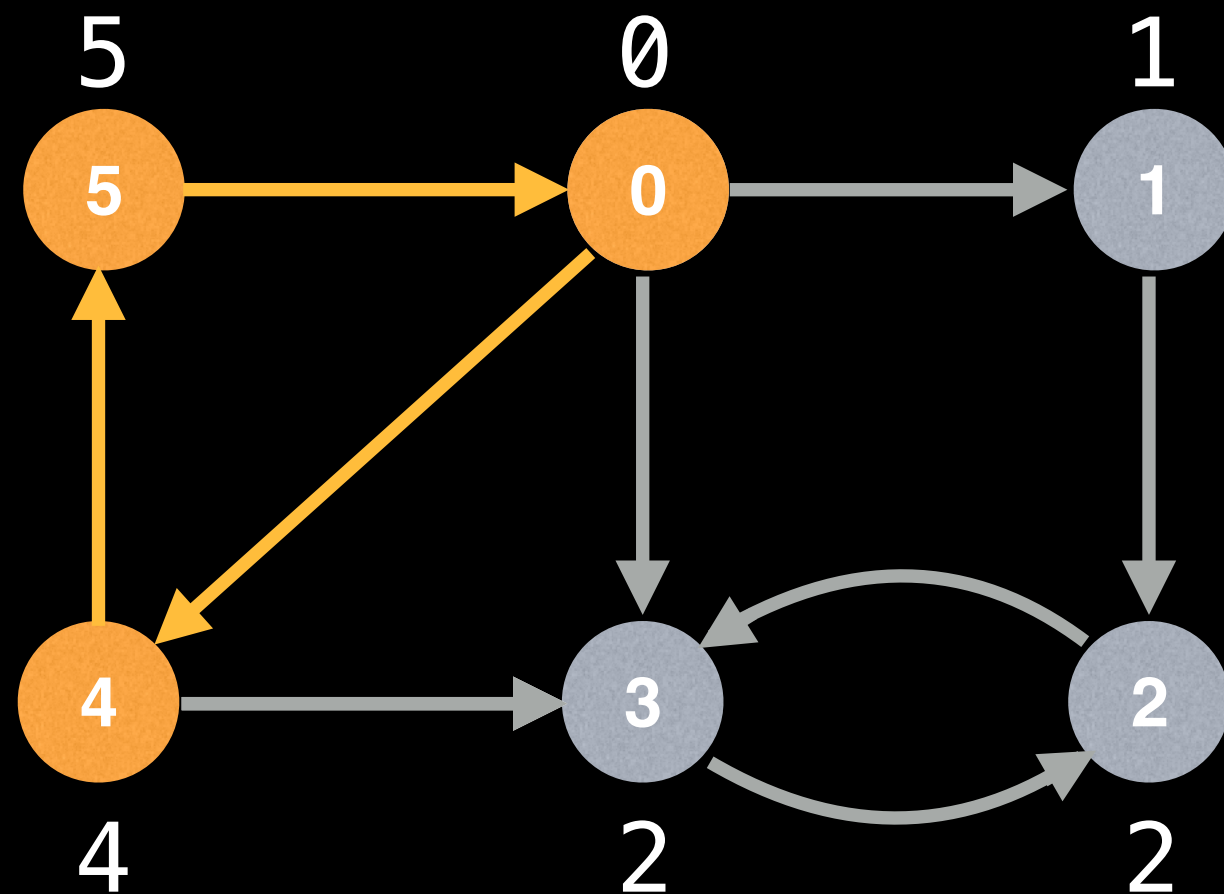
Stack



# Low-Link Values

id	0	1	2	3	4	5
lowlink	0	1	2	2	4	5

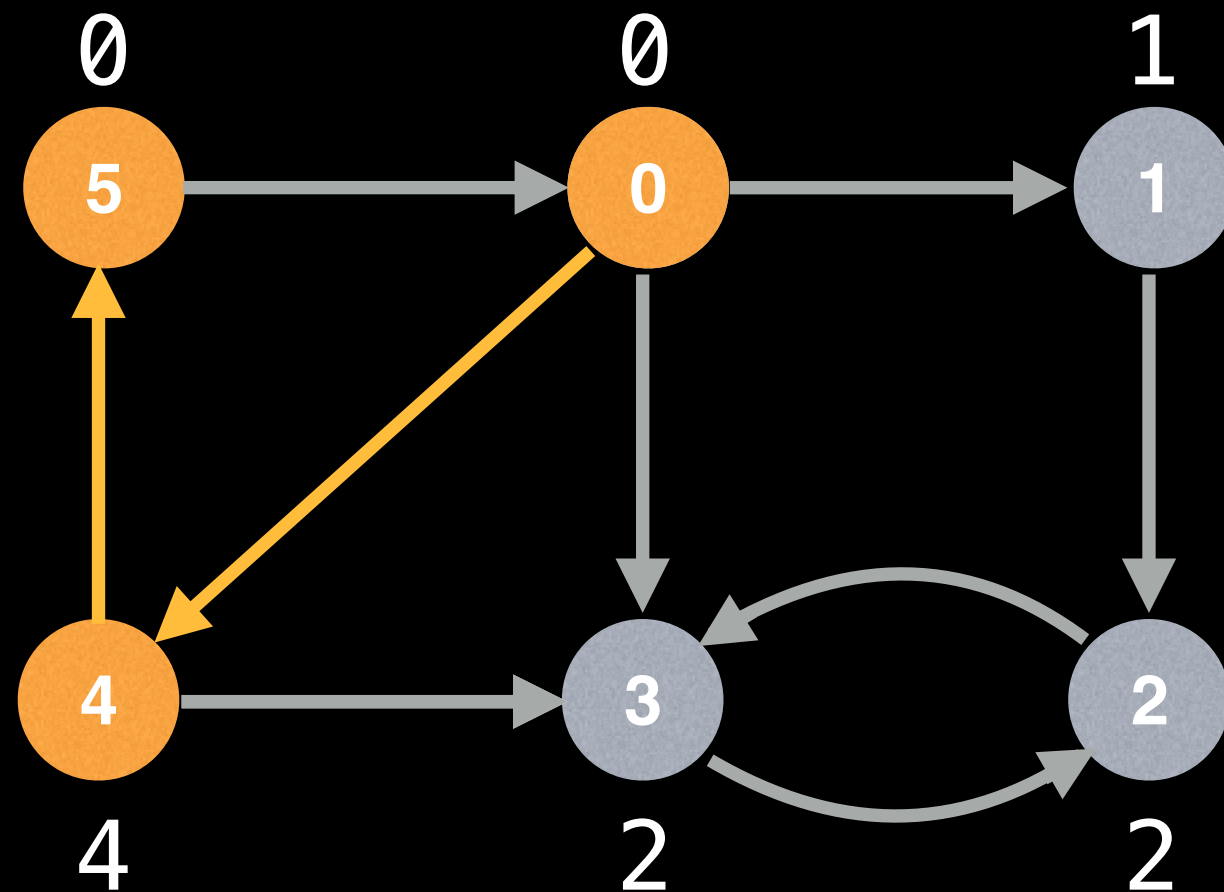
Stack



# Low-Link Values

id	0	1	2	3	4	5
lowlink	0	1	2	2	4	0

Stack

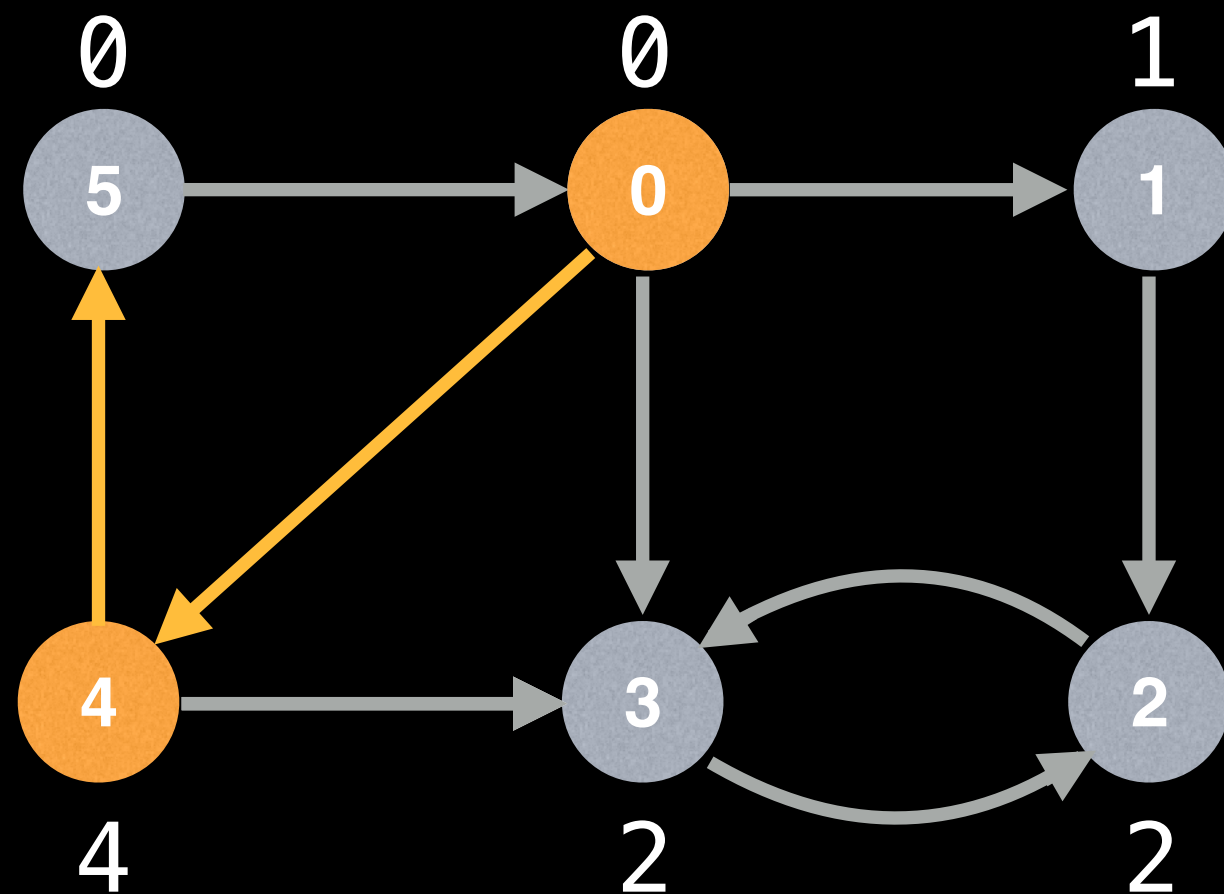


$\text{lowlink}[5] = \min(\text{lowlink}[5], \text{lowlink}[0])$   
 $= 0$

# Low-Link Values

id	0	1	2	3	4	5
lowlink	0	1	2	2	4	0

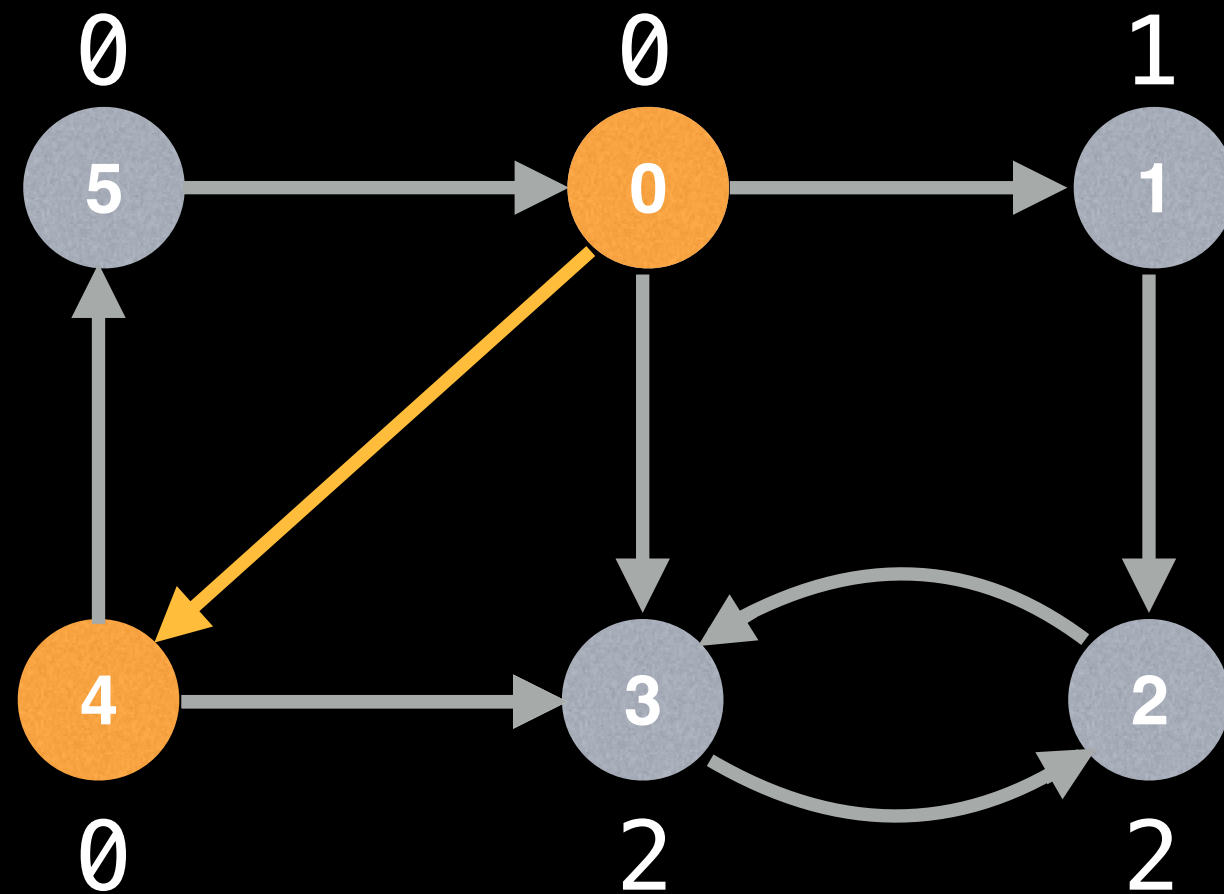
Stack



# Low-Link Values

id	0	1	2	3	4	5
lowlink	0	1	2	2	0	0

Stack



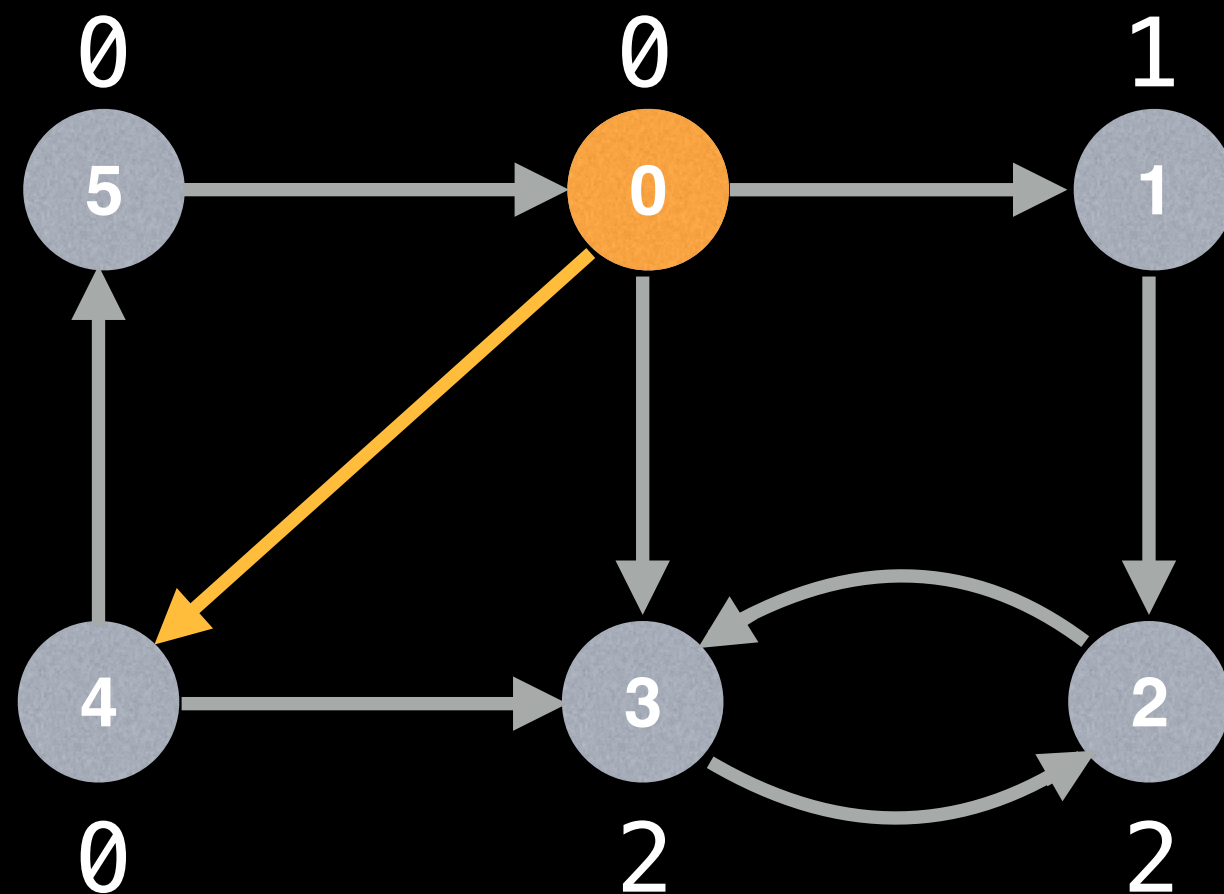
$$\begin{aligned}\text{lowlink}[4] &= \min(\text{lowlink}[4], \text{lowlink}[5]) \\ &= 0\end{aligned}$$



# Low-Link Values

id	0	1	2	3	4	5
lowlink	0	1	2	2	0	0

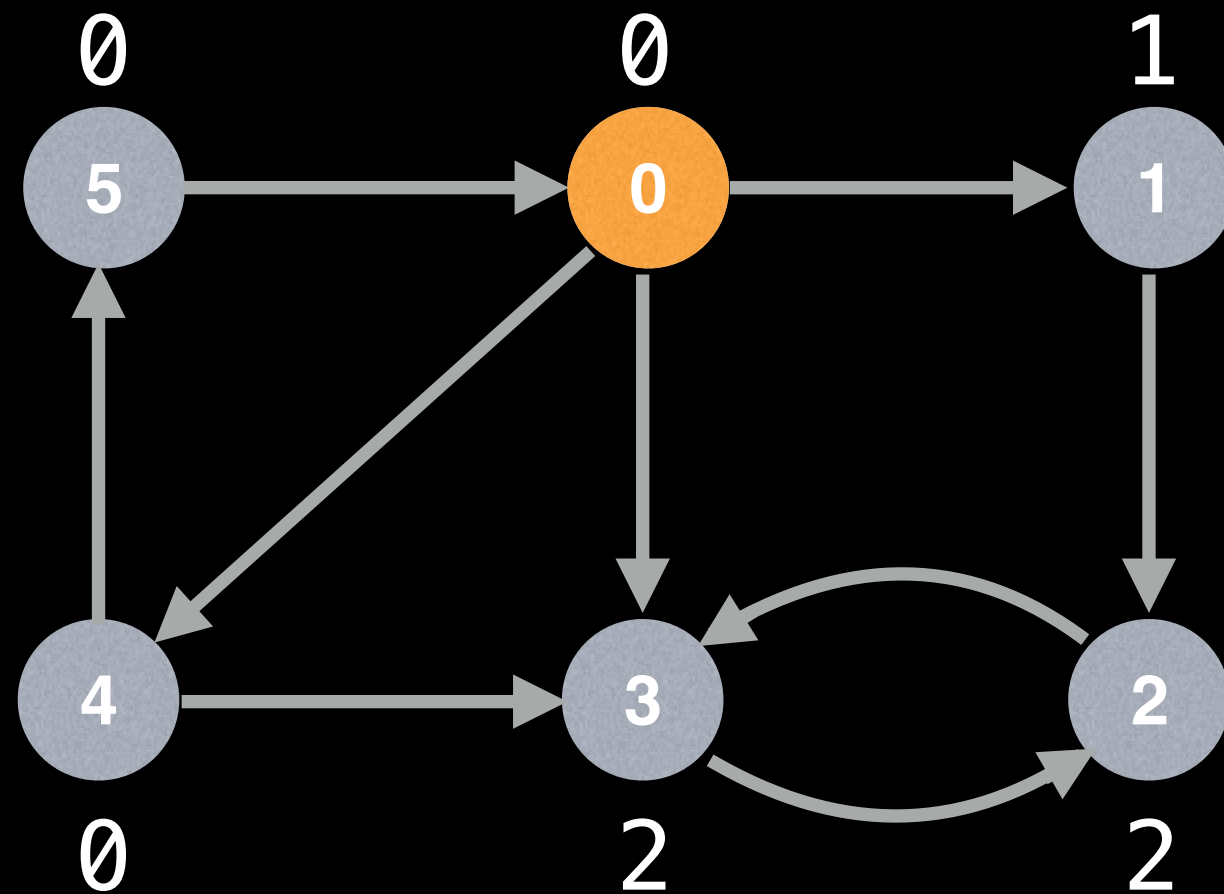
Stack



# Low-Link Values

id	0	1	2	3	4	5
lowlink	0	1	2	2	0	0

Stack

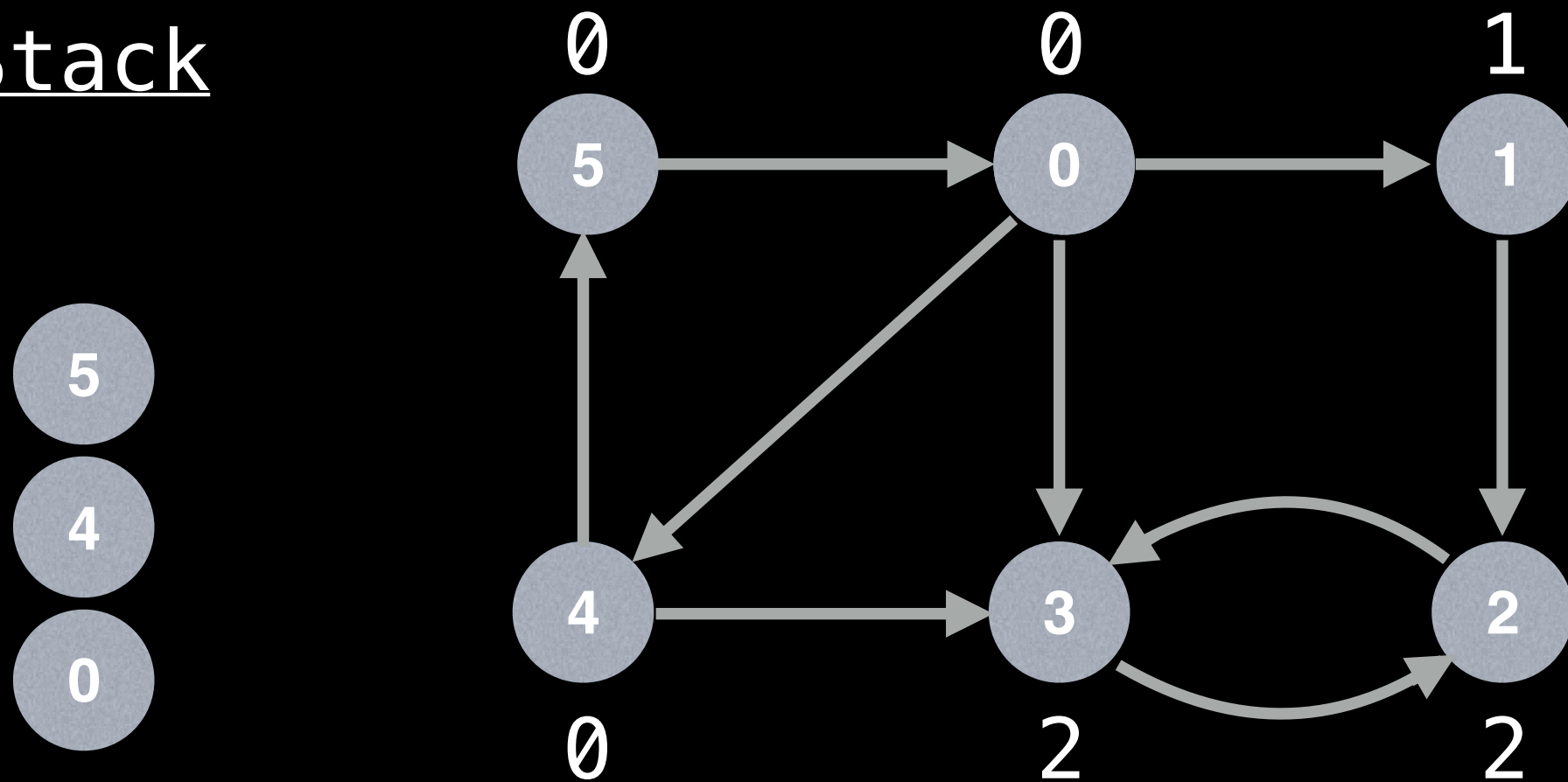


$$\begin{aligned}\text{lowlink}[0] &= \min(\text{lowlink}[0], \text{lowlink}[4]) \\ &= 0\end{aligned}$$

# Low-Link Values

id	0	1	2	3	4	5
lowlink	0	1	2	2	0	0

Stack

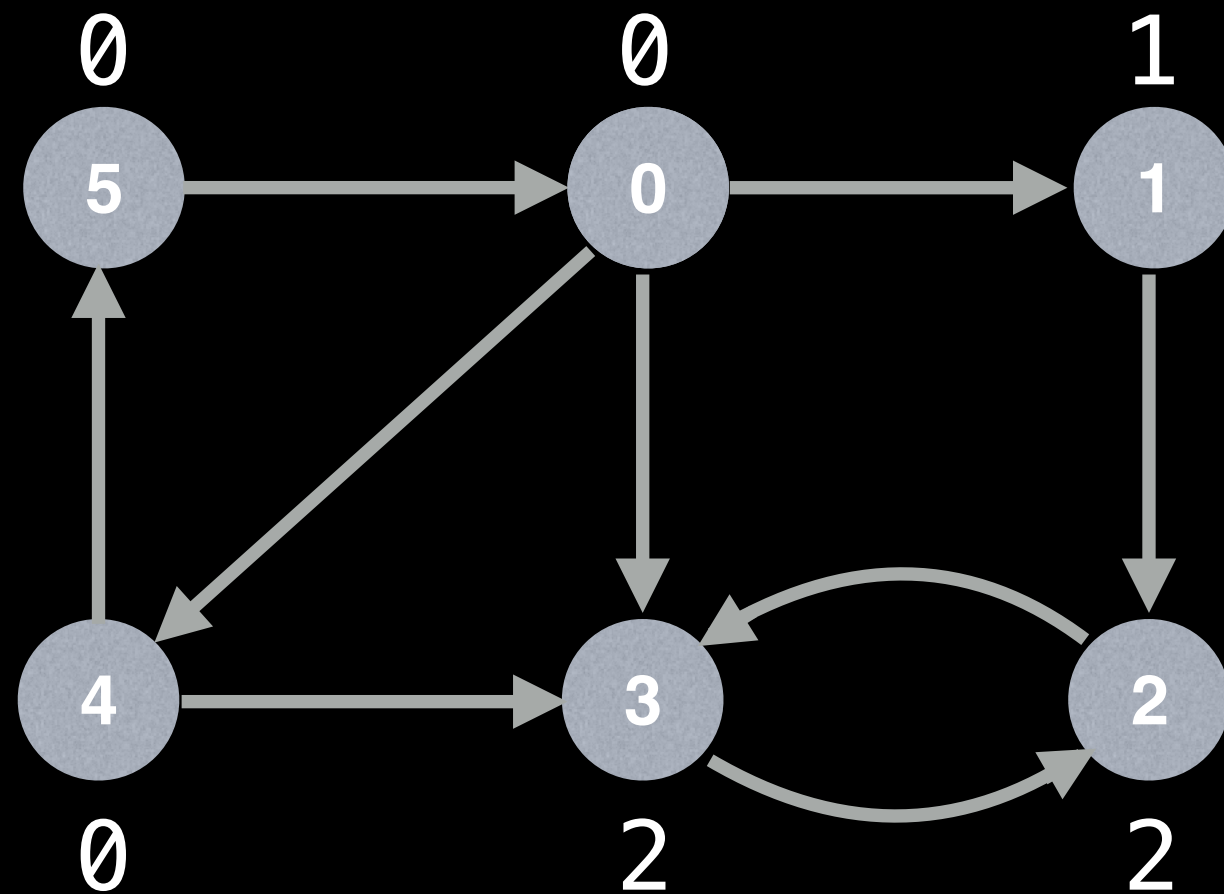


When a completed SCC is found (current node has visited all its neighbours and its lowlink value equals its id) pop off all associated nodes off the stack.

# Low-Link Values

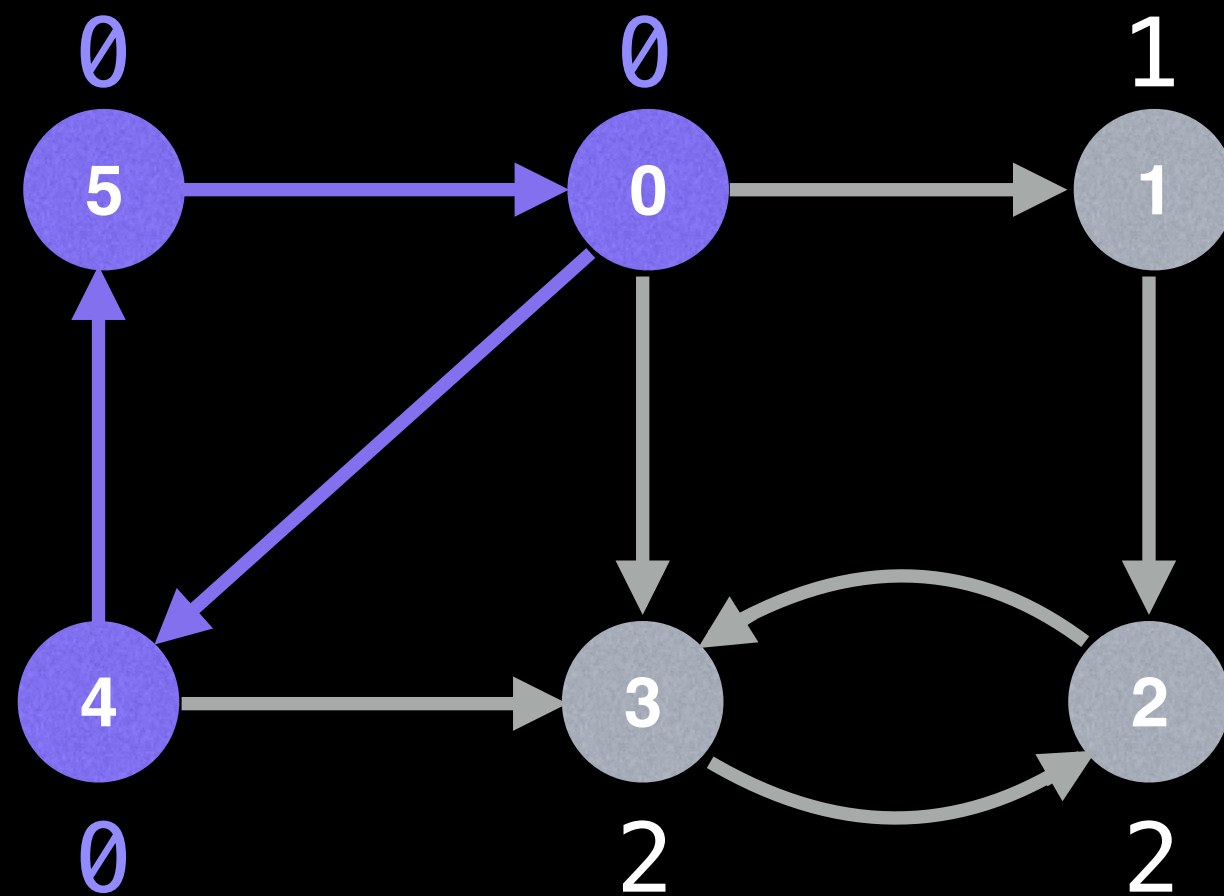
id	0	1	2	3	4	5
lowlink	0	1	2	2	0	0

Stack



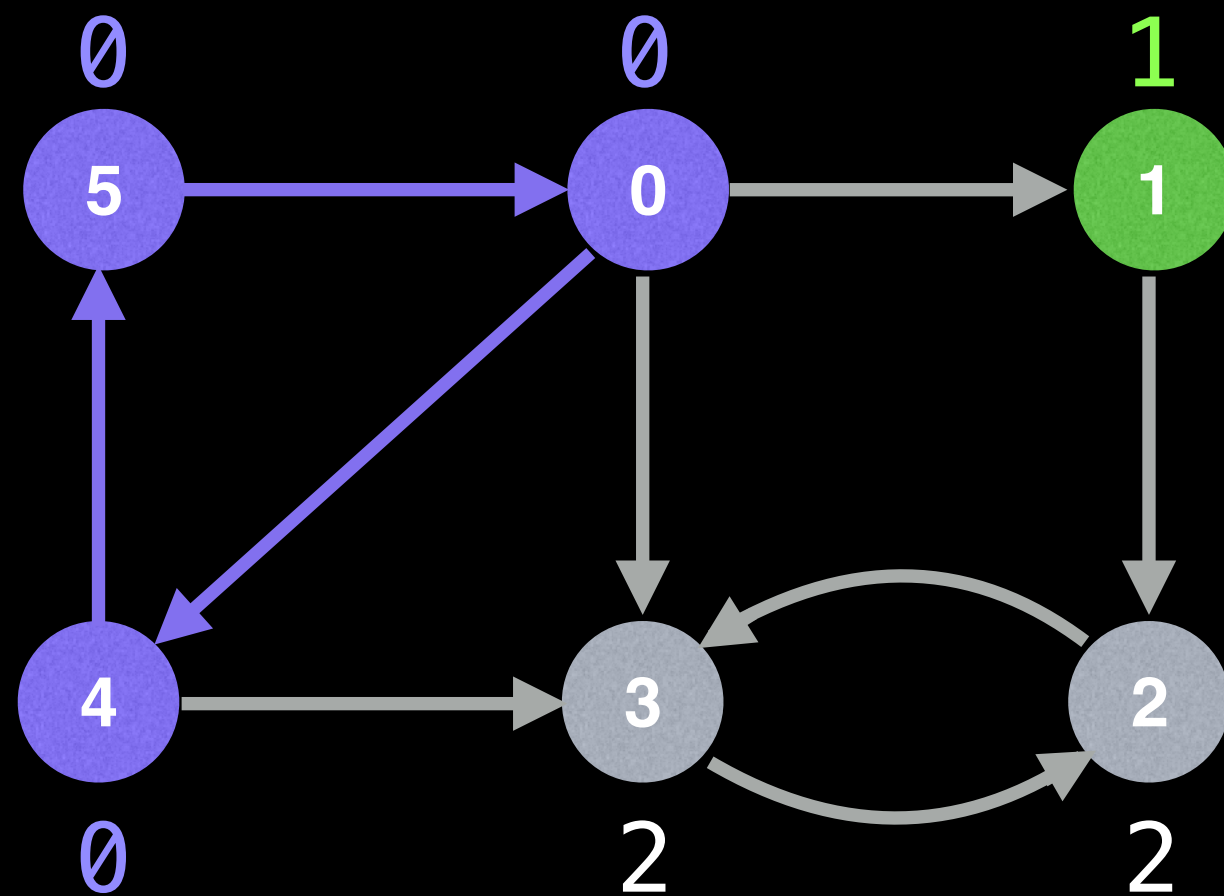
# Low-Link Values

id	0	1	2	3	4	5
lowlink	0	1	2	2	0	0



# Low-Link Values

id	0	1	2	3	4	5
lowlink	0	1	2	2	0	0



# Low-Link Values

id	0	1	2	3	4	5
lowlink	0	1	2	2	0	0

