

# Graph Theory: Breadth First Search

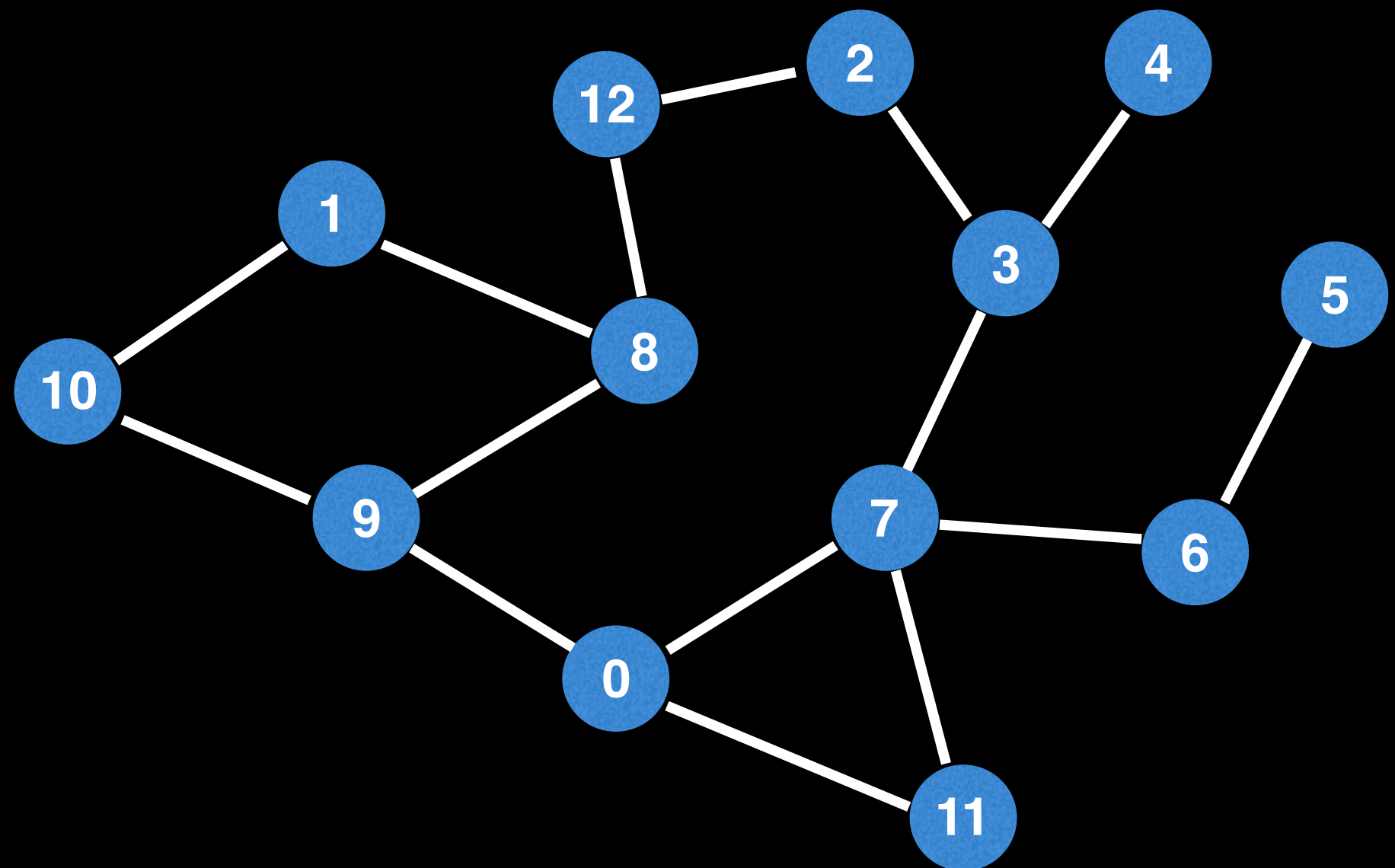
William Fiset

# BFS overview

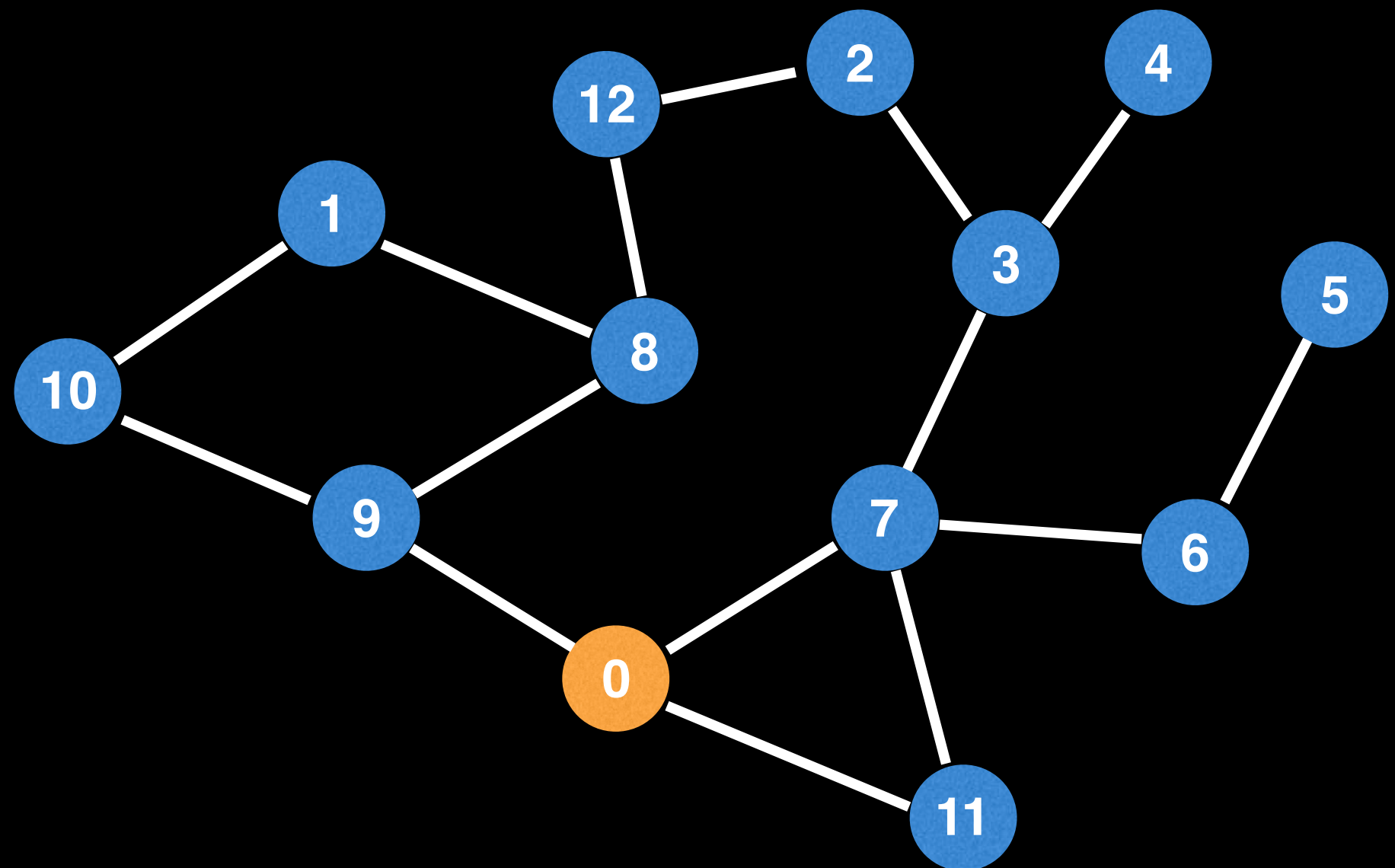
The **Breadth First Search (BFS)** is another fundamental search algorithm used to explore nodes and edges of a graph. It runs with a time complexity of  $O(V+E)$  and is often used as a building block in other algorithms.

The BFS algorithm is particularly useful for one thing: finding the **shortest path on unweighted graphs**.

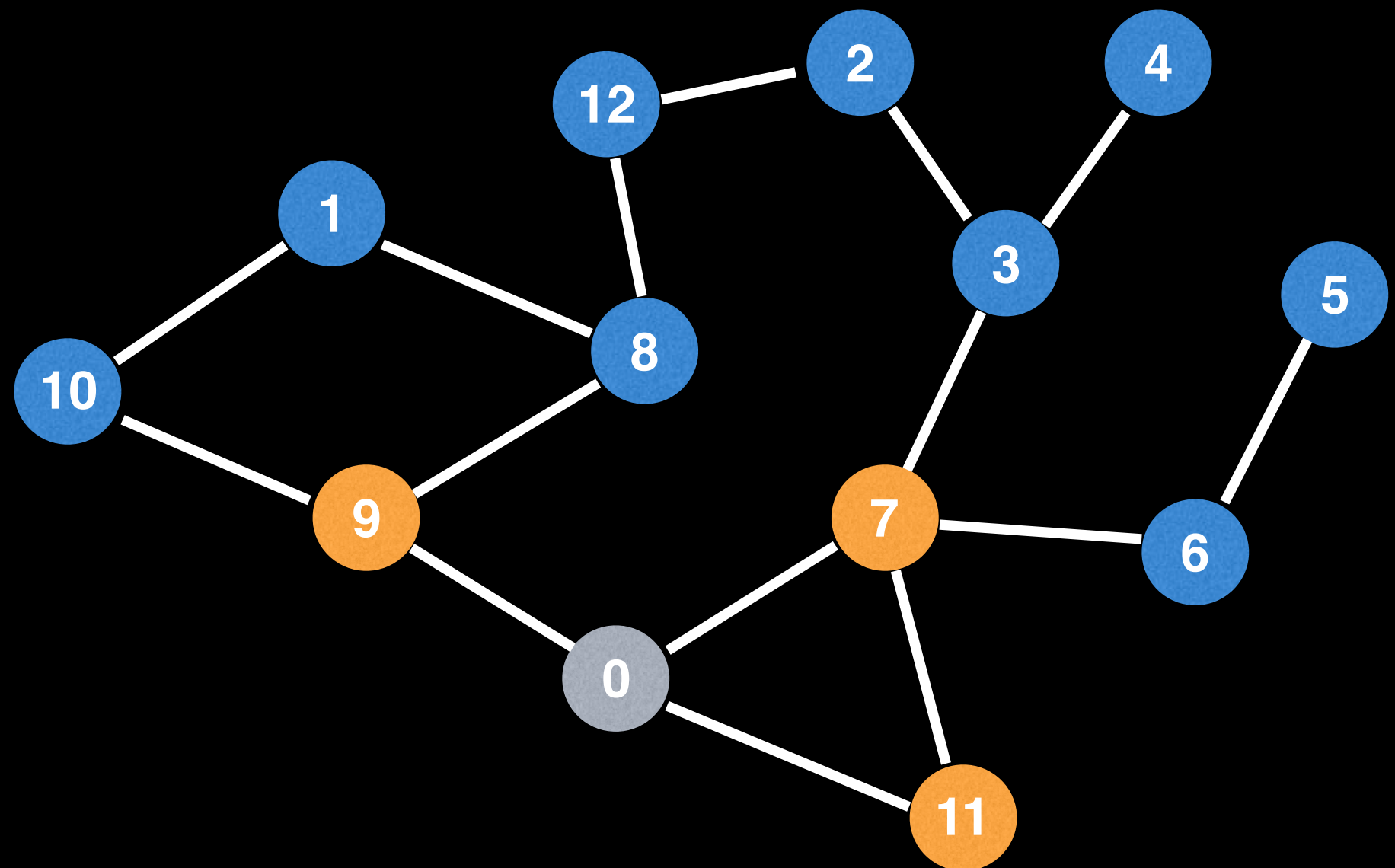
A BFS starts at some arbitrary node of a graph and explores the neighbour nodes first, before moving to the next level neighbours.



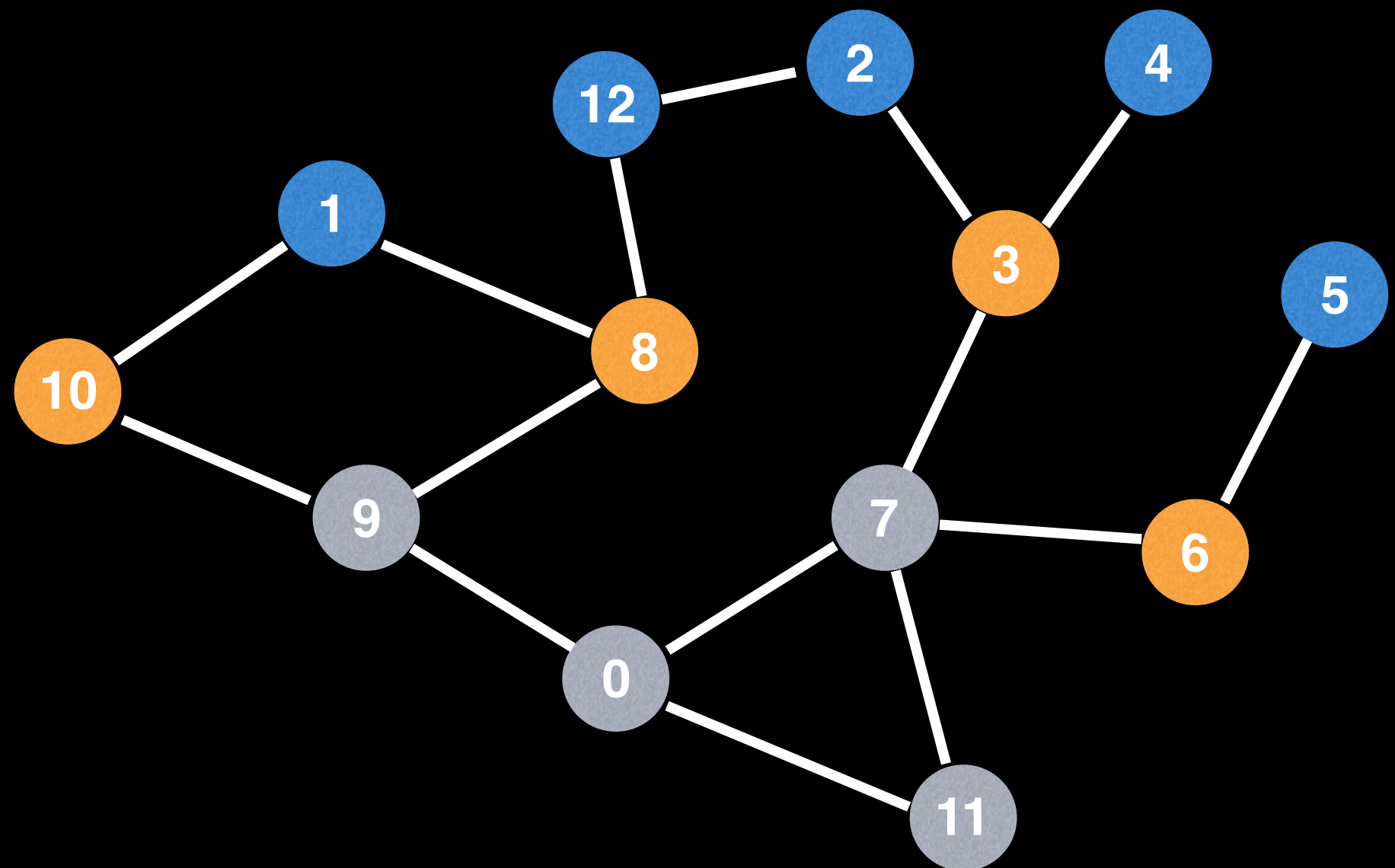
A BFS starts at some arbitrary node of a graph and explores the neighbour nodes first, before moving to the next level neighbours.



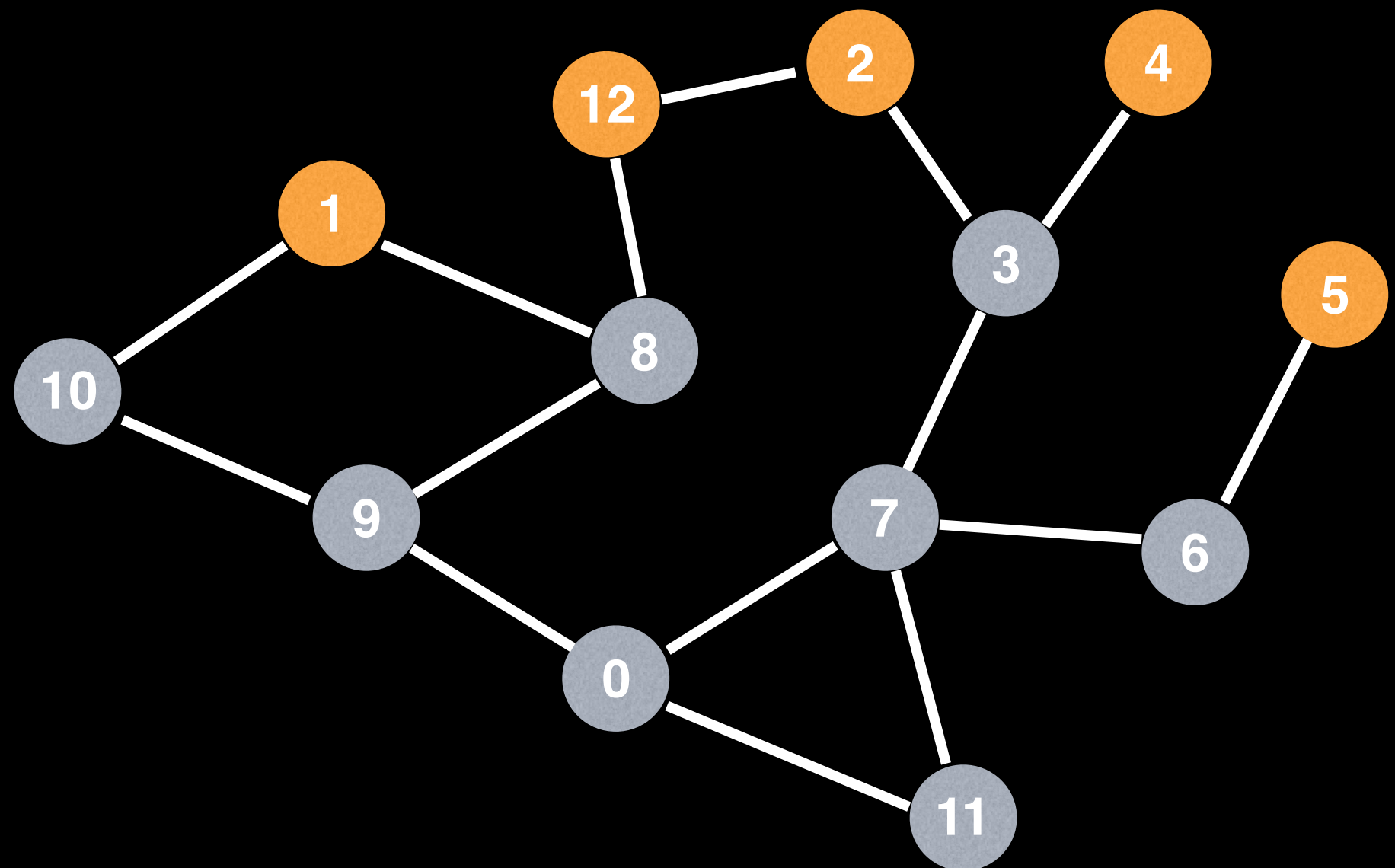
A BFS starts at some arbitrary node of a graph and explores the neighbour nodes first, before moving to the next level neighbours.



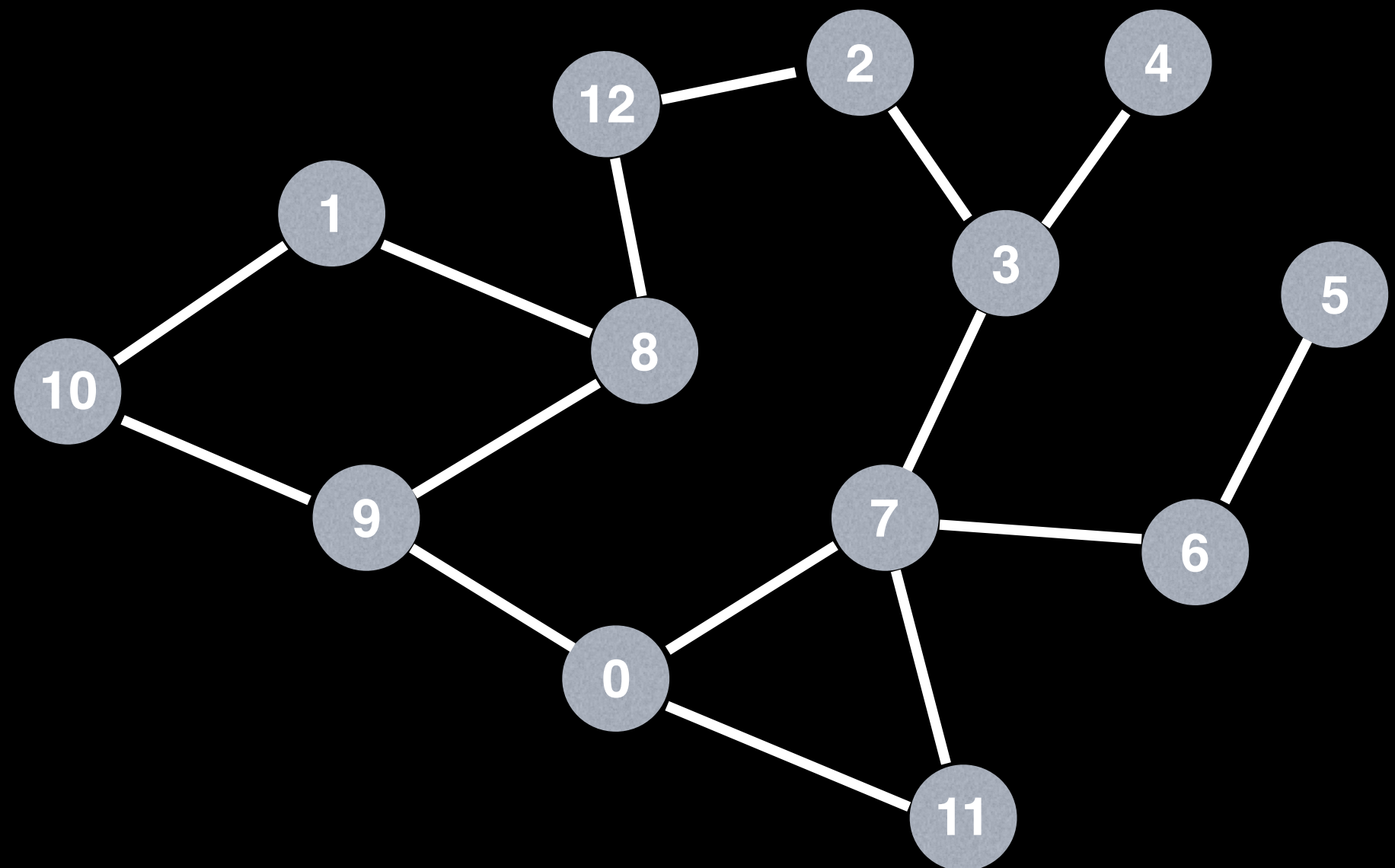
A BFS starts at some arbitrary node of a graph and explores the neighbour nodes first, before moving to the next level neighbours.



A BFS starts at some arbitrary node of a graph and explores the neighbour nodes first, before moving to the next level neighbours.

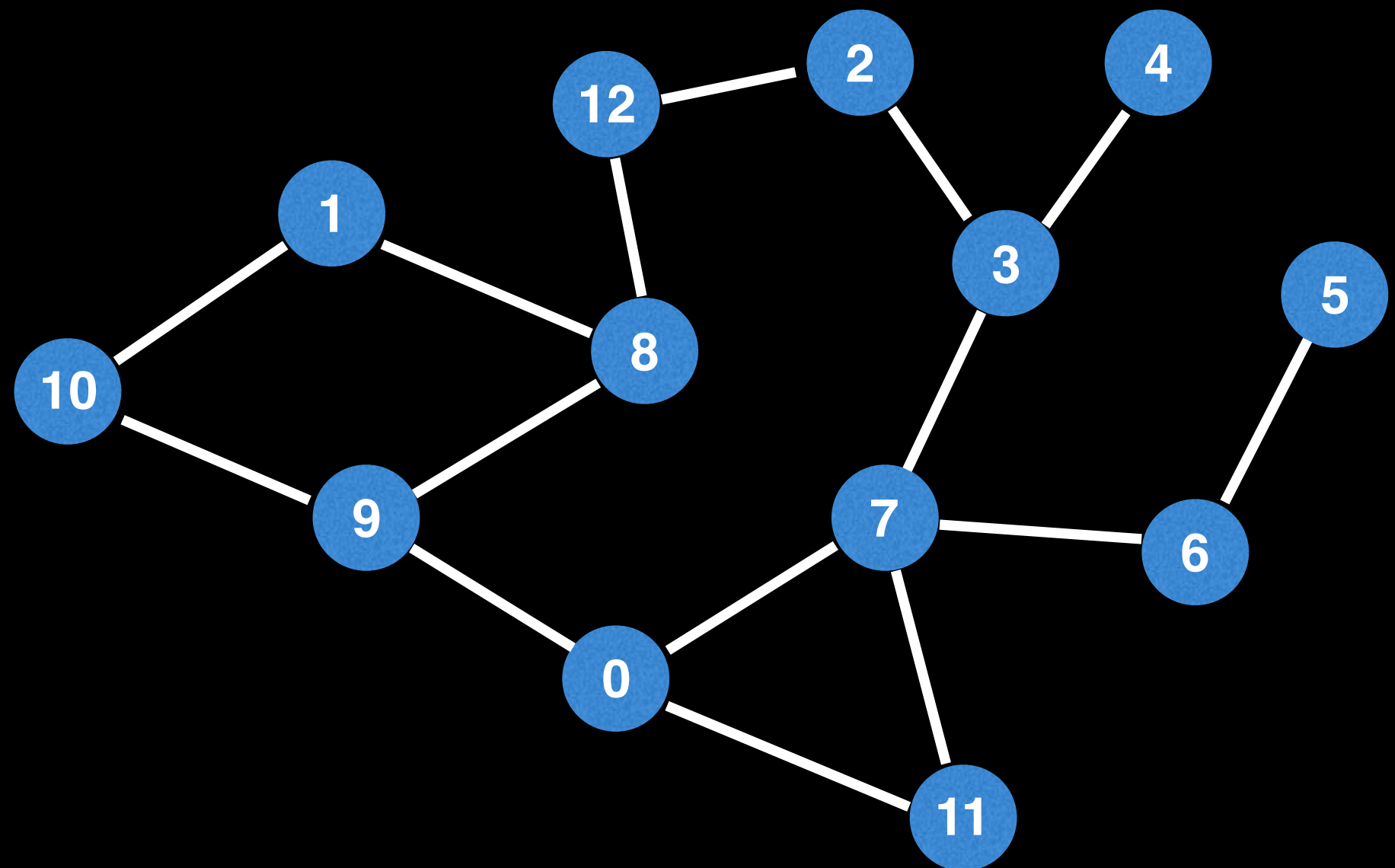


A BFS starts at some arbitrary node of a graph and explores the neighbour nodes first, before moving to the next level neighbours.

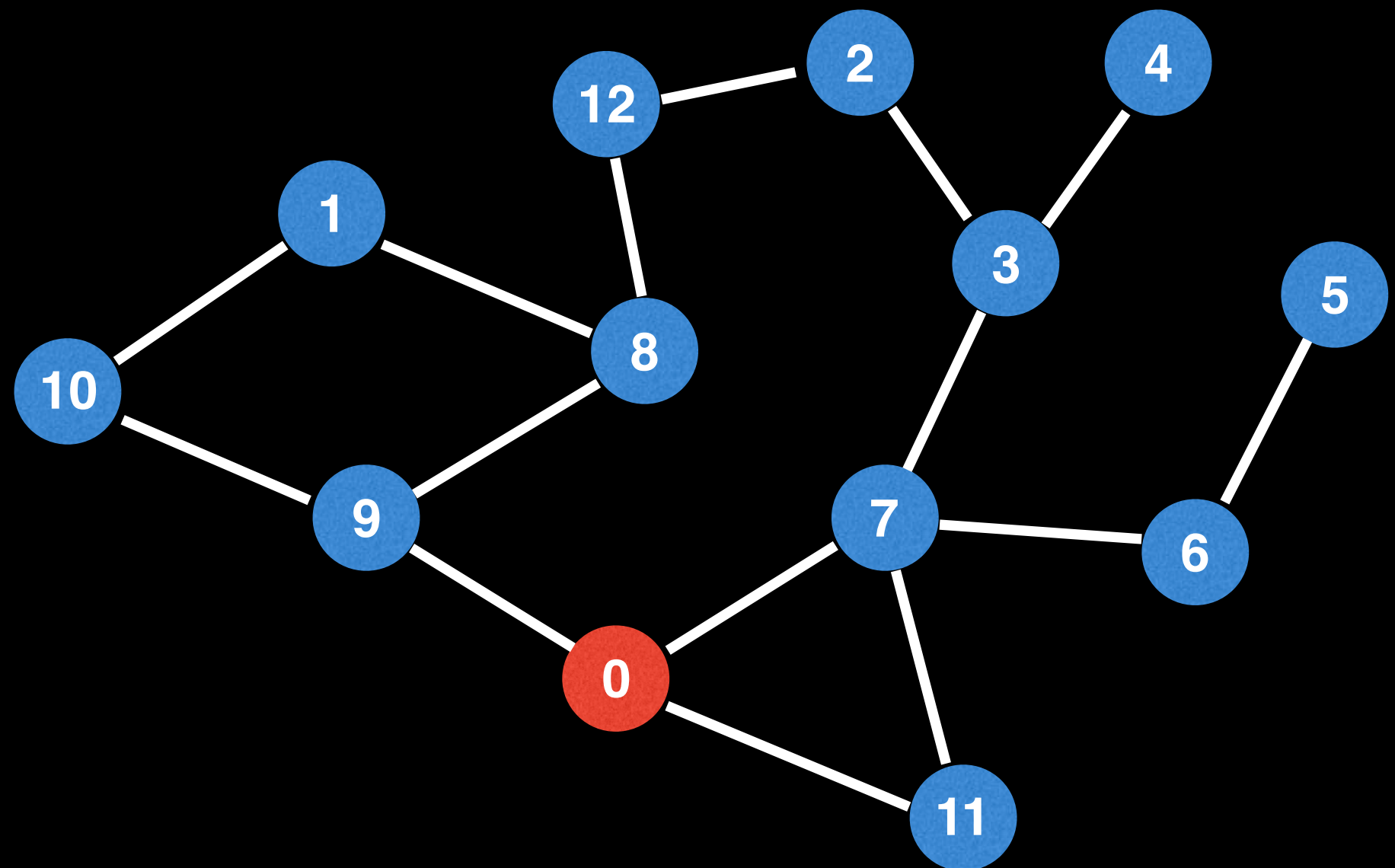




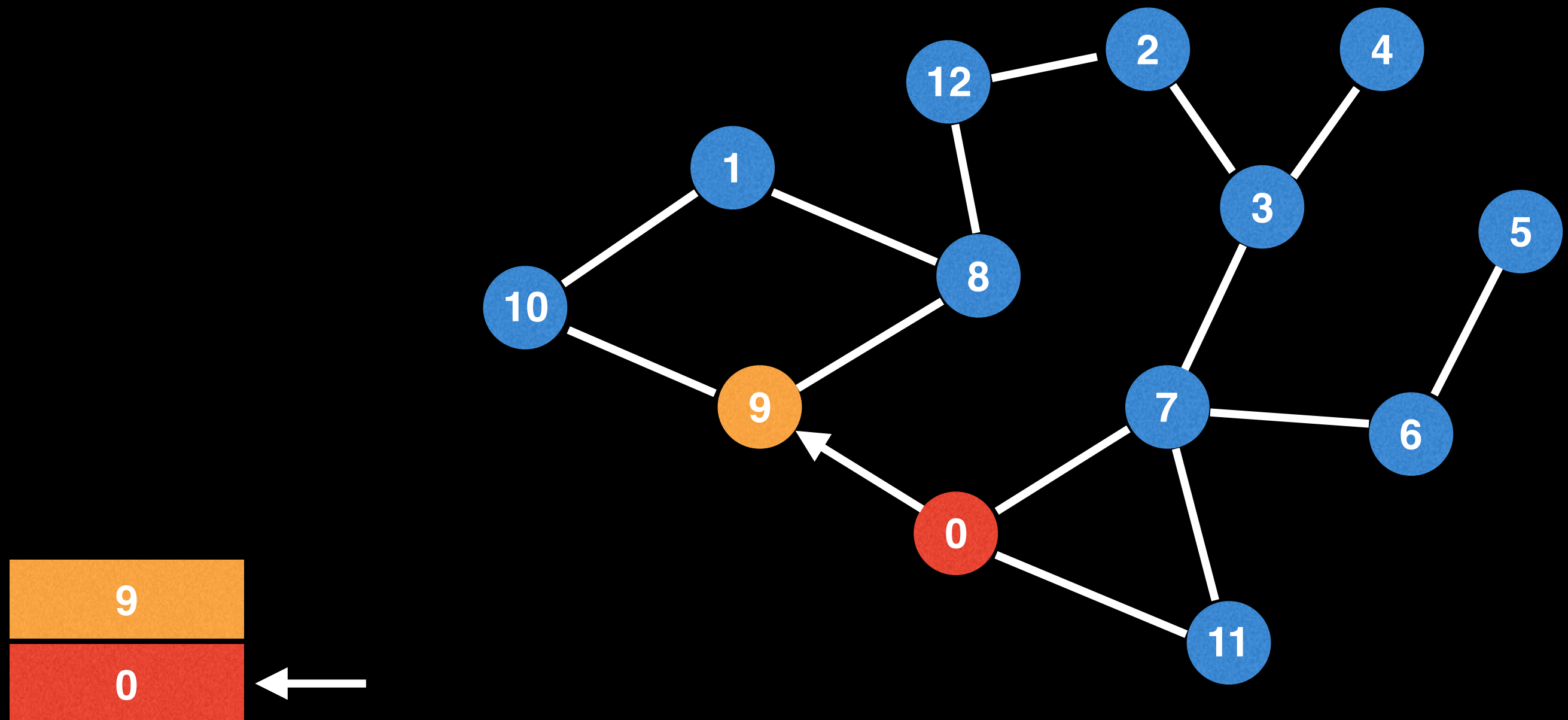
A BFS starts at some arbitrary node of a graph and explores the neighbour nodes first, before moving to the next level neighbours.



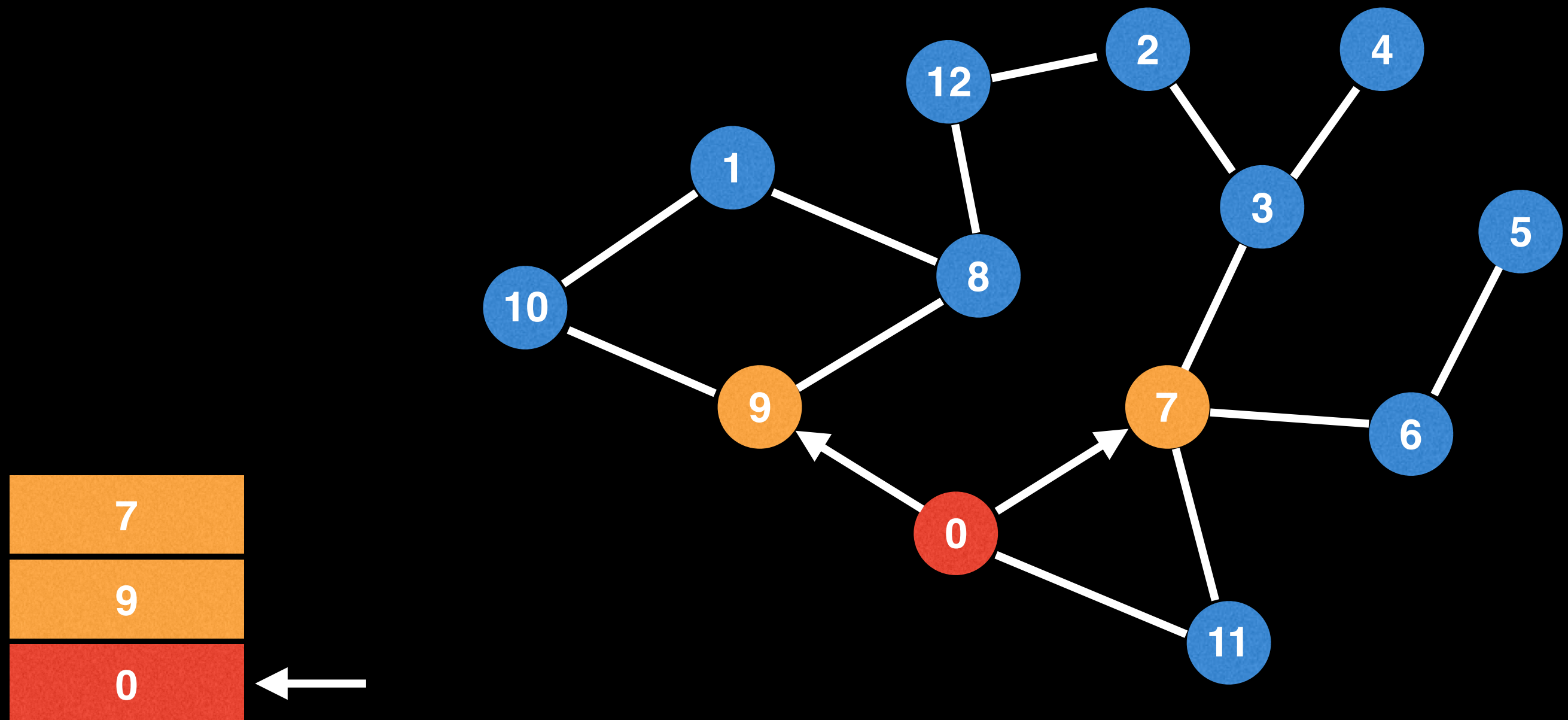
A BFS starts at some arbitrary node of a graph and explores the neighbour nodes first, before moving to the next level neighbours.



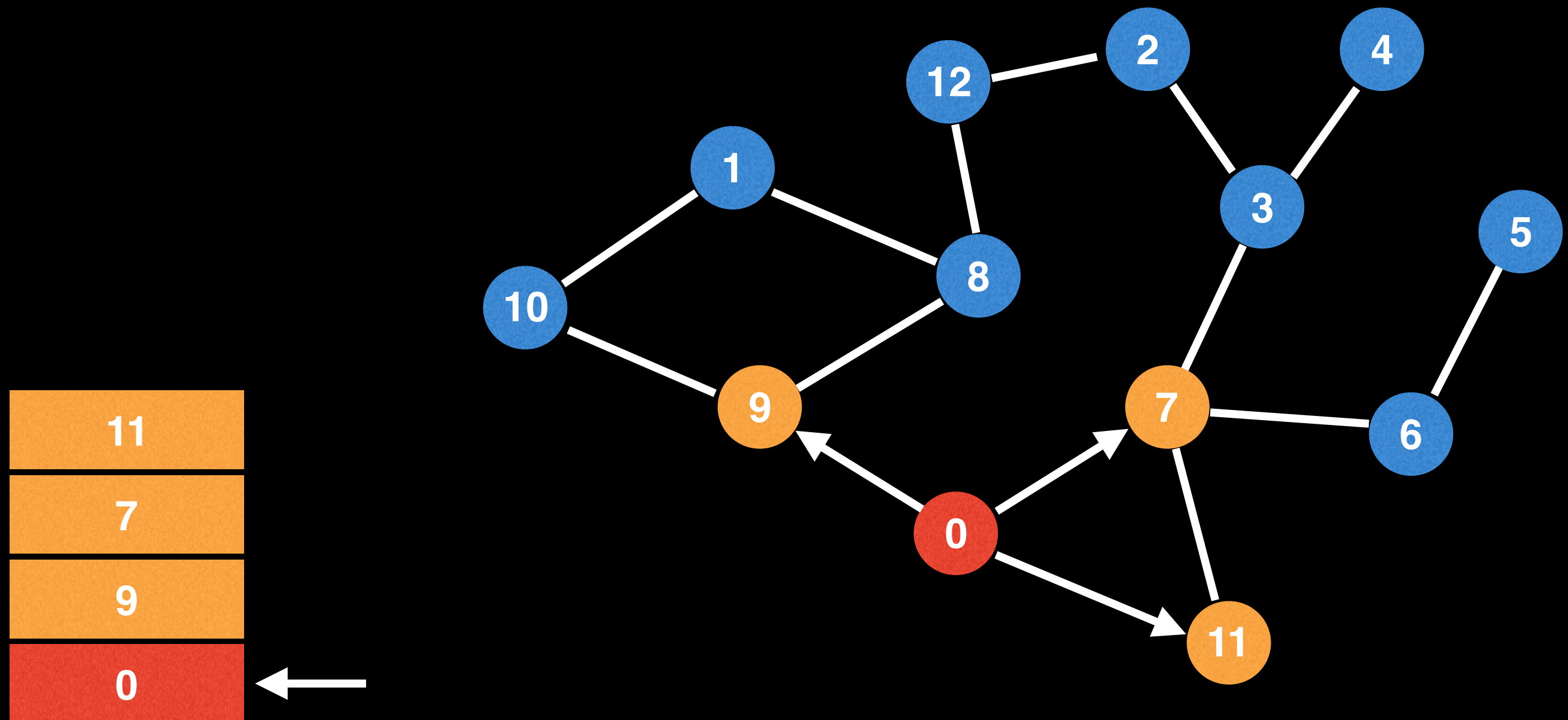
A BFS starts at some arbitrary node of a graph and explores the neighbour nodes first, before moving to the next level neighbours.



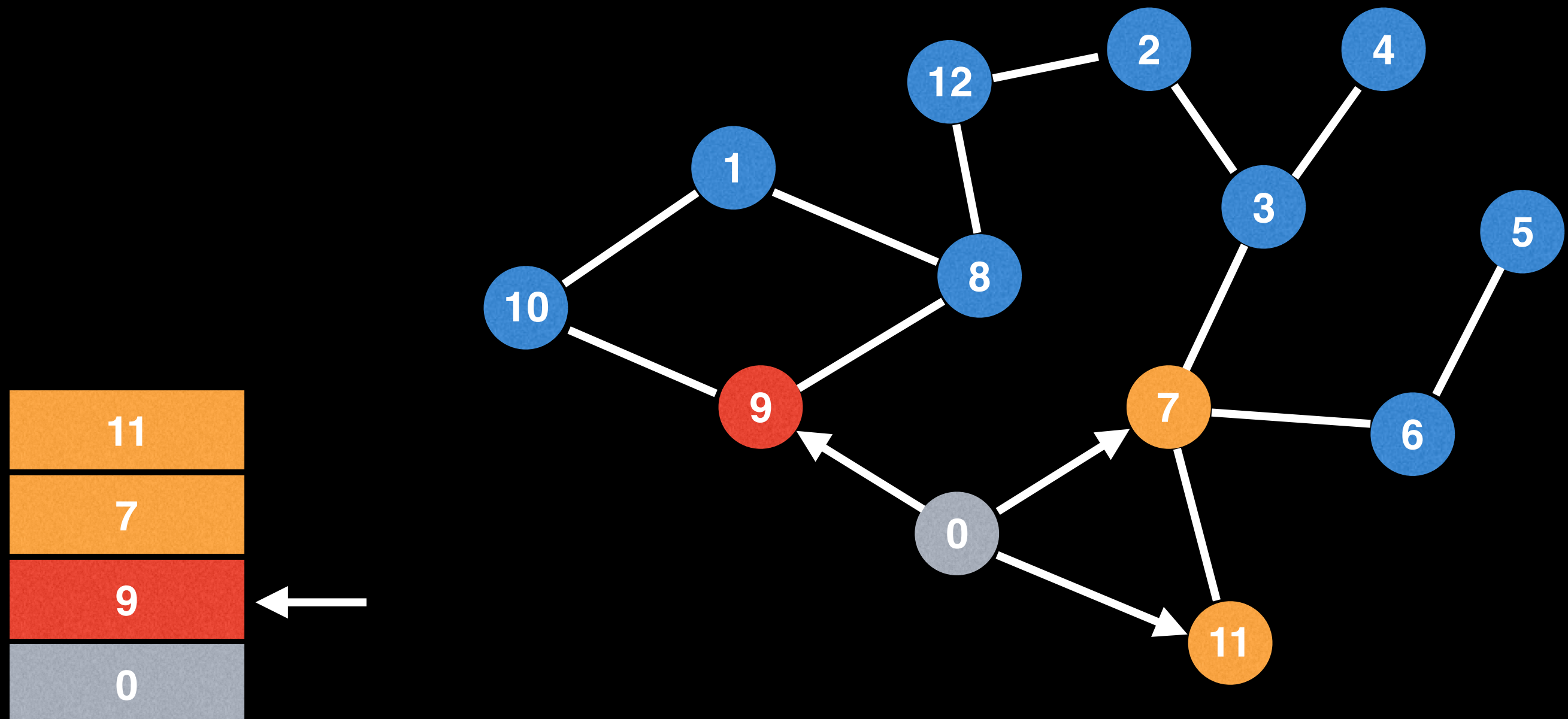
A BFS starts at some arbitrary node of a graph and explores the neighbour nodes first, before moving to the next level neighbours.



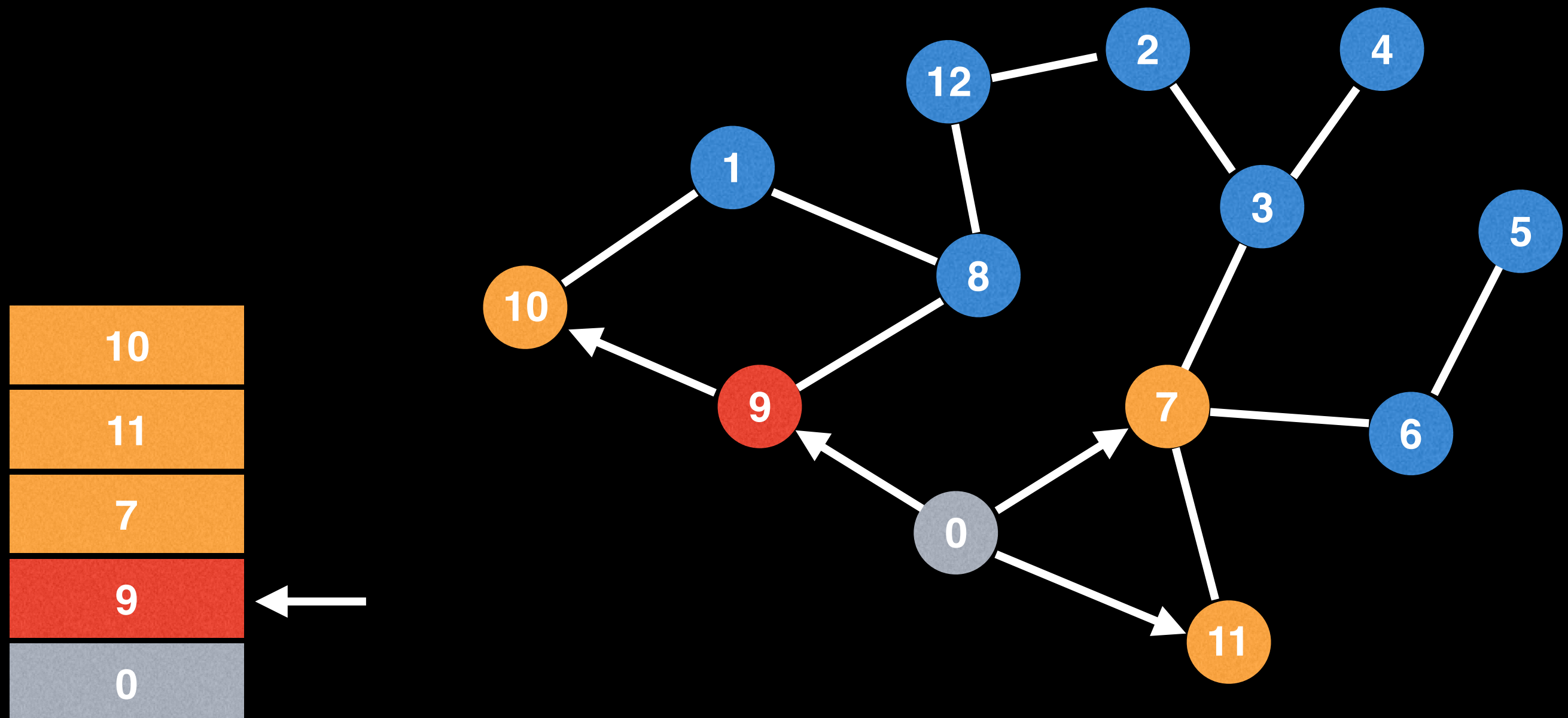
A BFS starts at some arbitrary node of a graph and explores the neighbour nodes first, before moving to the next level neighbours.



A BFS starts at some arbitrary node of a graph and explores the neighbour nodes first, before moving to the next level neighbours.

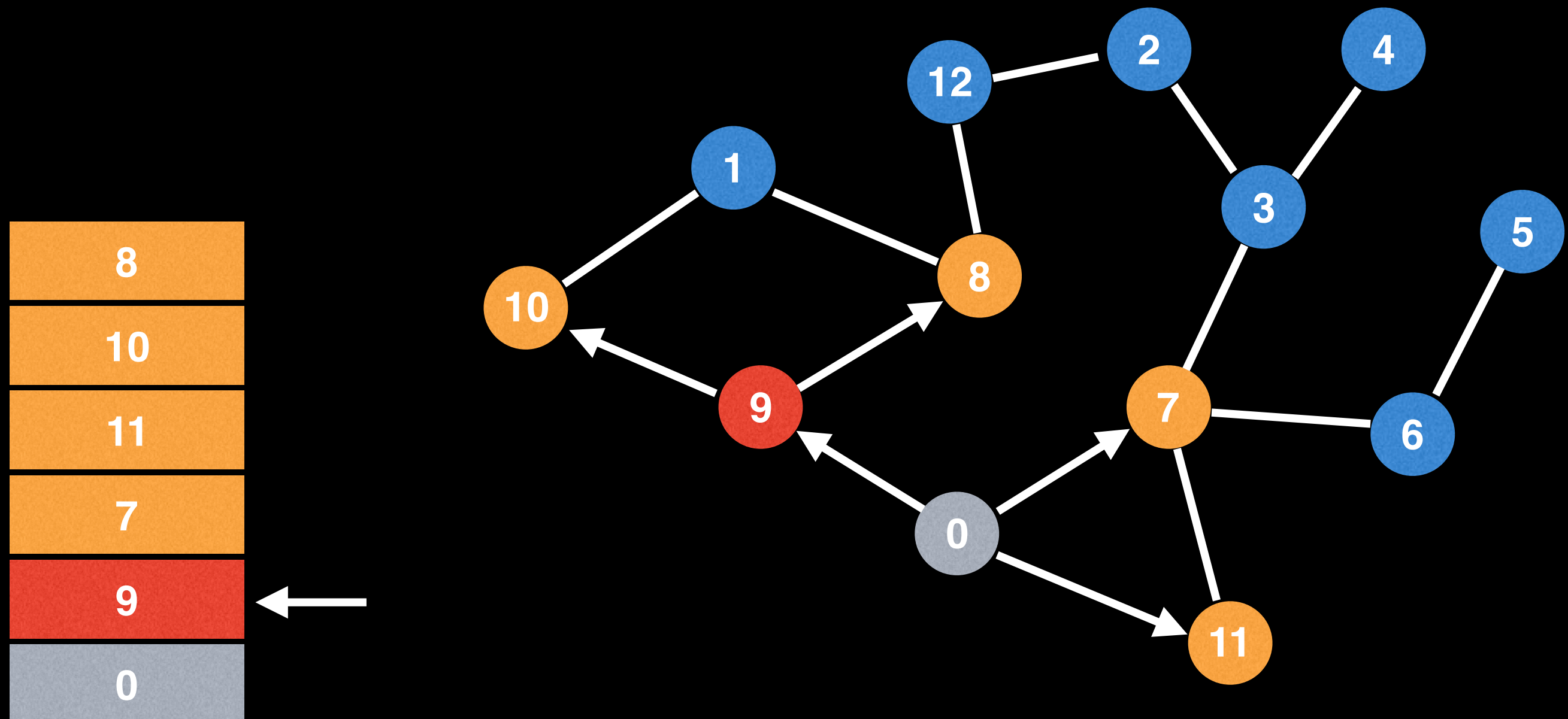


A BFS starts at some arbitrary node of a graph and explores the neighbour nodes first, before moving to the next level neighbours.



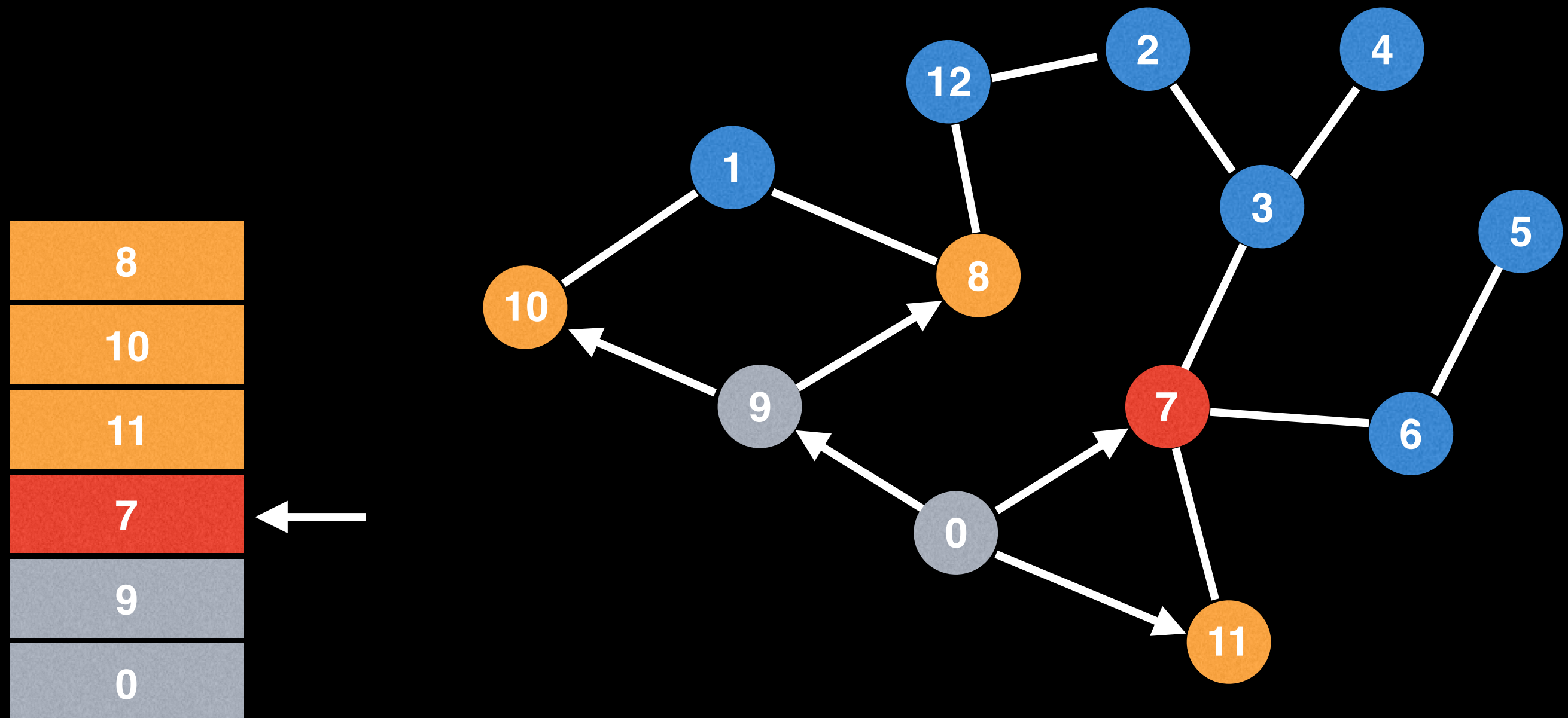


A BFS starts at some arbitrary node of a graph and explores the neighbour nodes first, before moving to the next level neighbours.

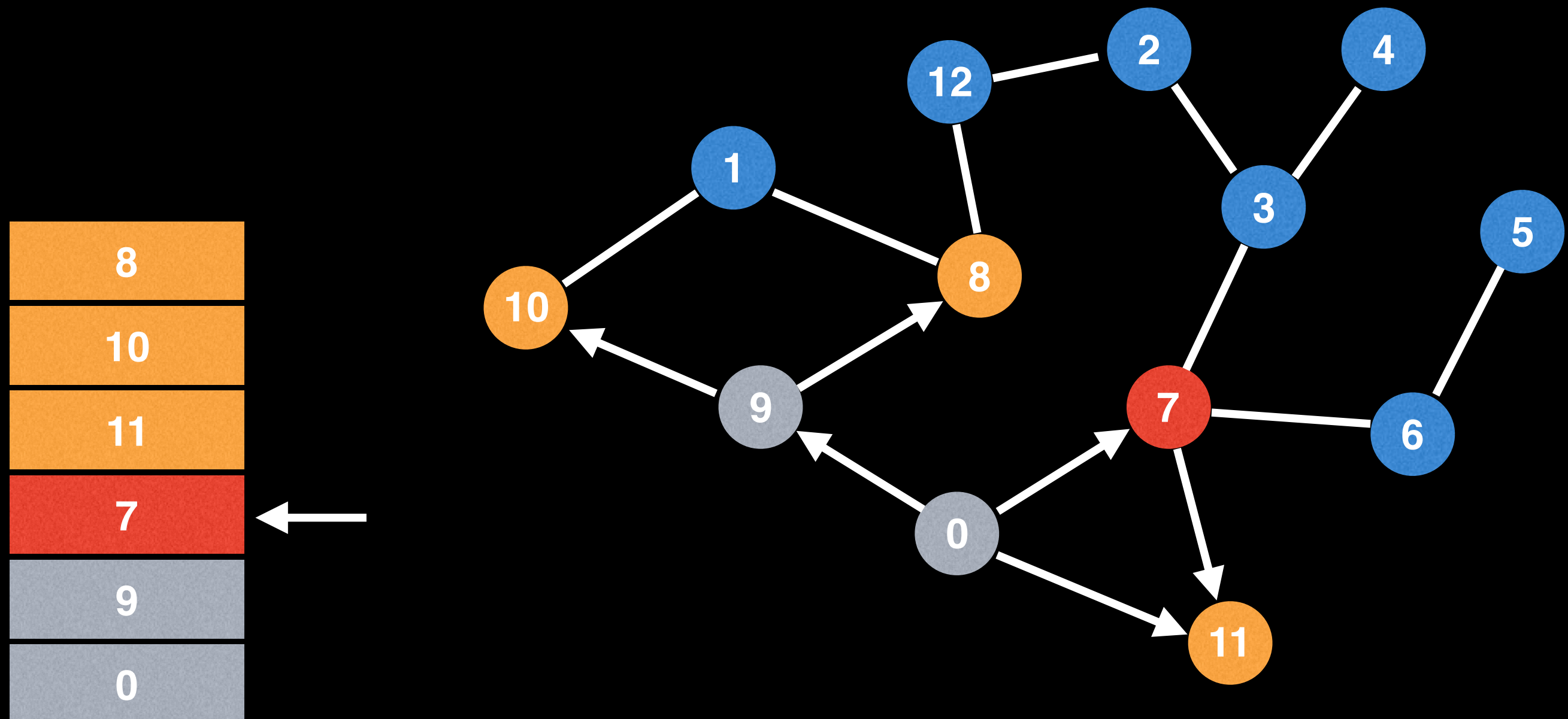




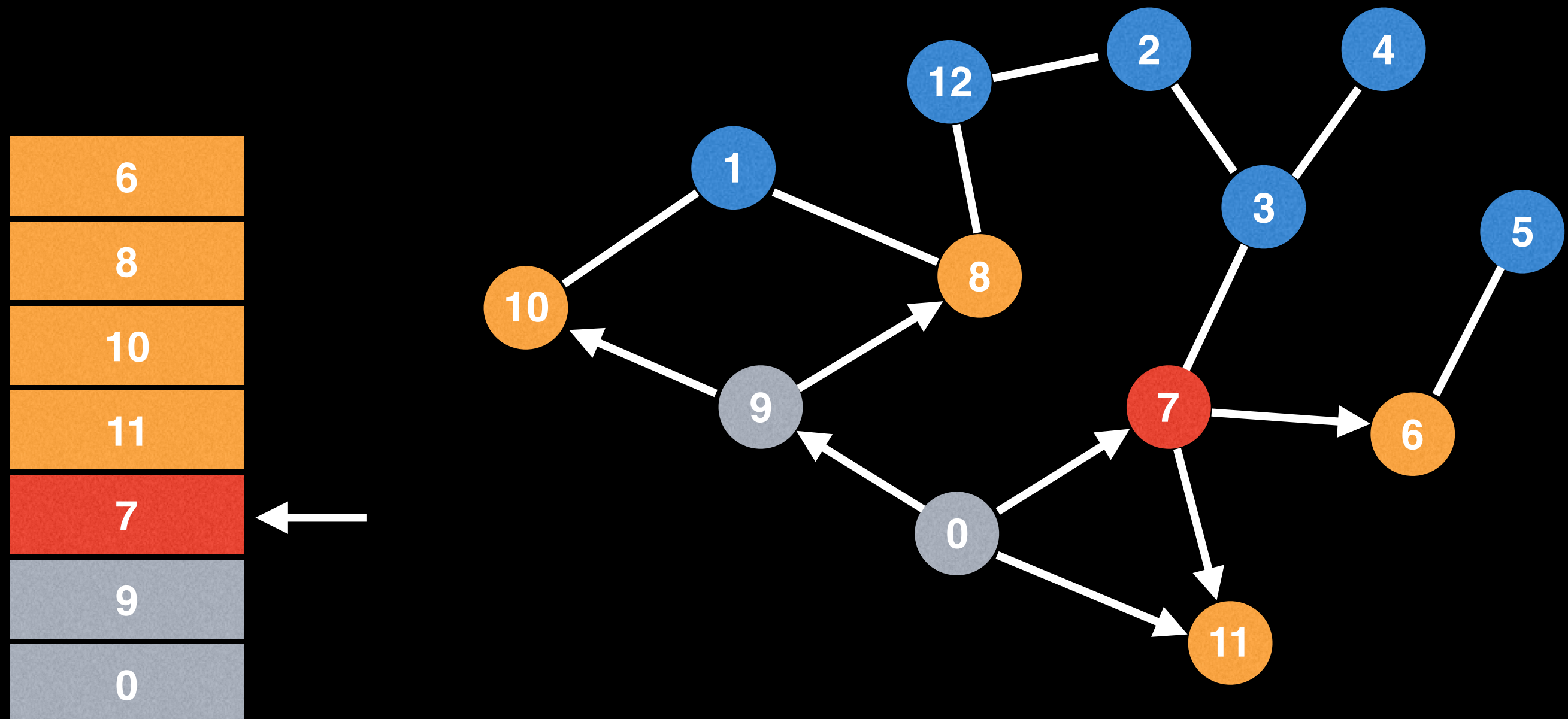
A BFS starts at some arbitrary node of a graph and explores the neighbour nodes first, before moving to the next level neighbours.



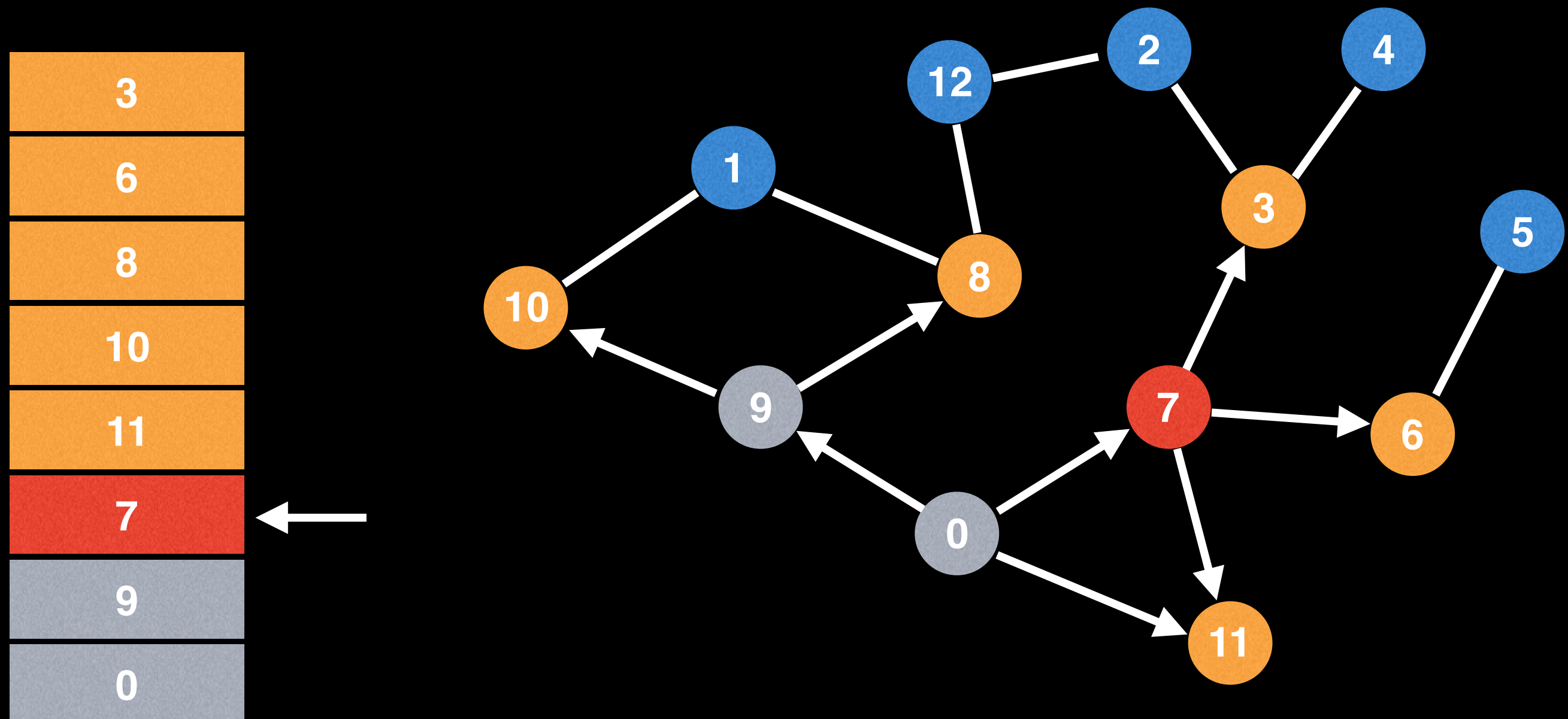
A BFS starts at some arbitrary node of a graph and explores the neighbour nodes first, before moving to the next level neighbours.



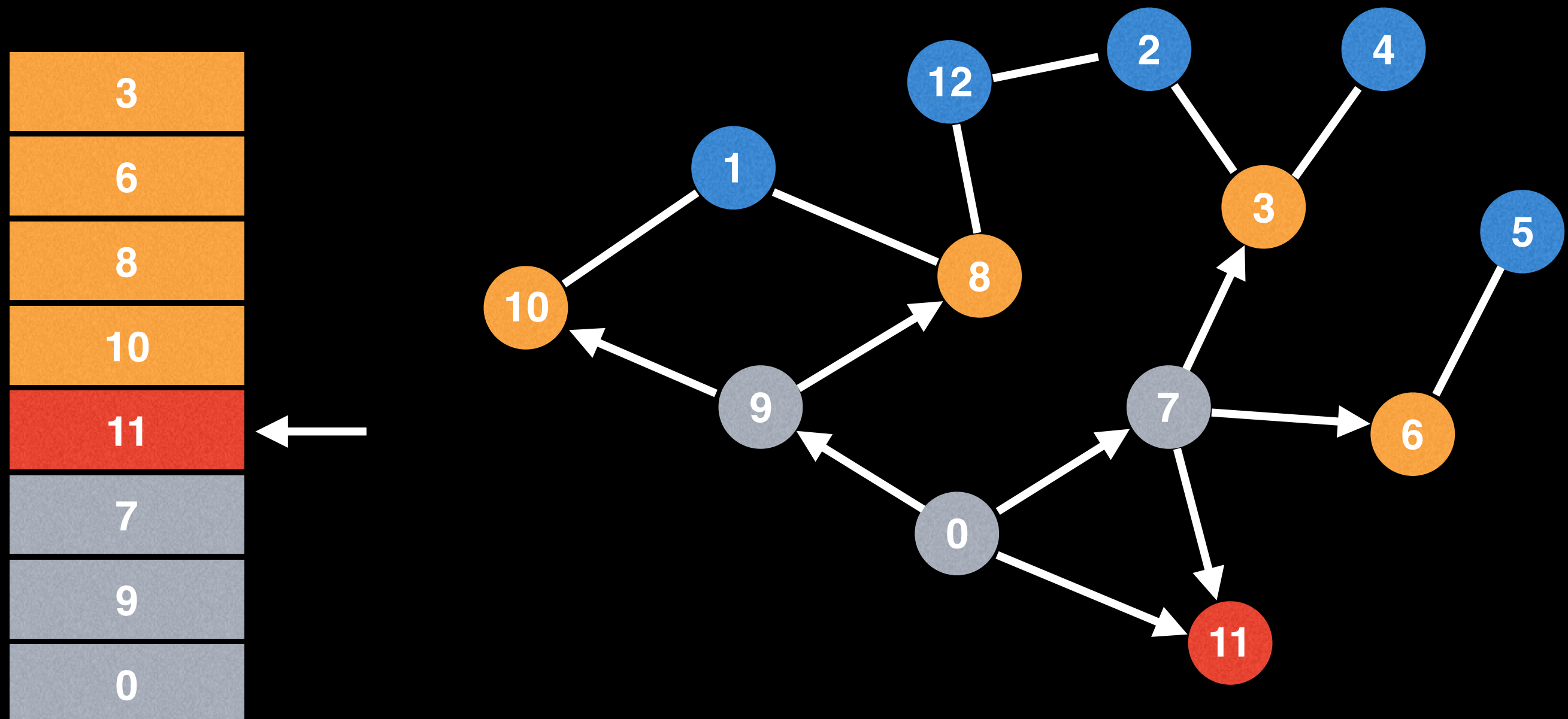
A BFS starts at some arbitrary node of a graph and explores the neighbour nodes first, before moving to the next level neighbours.



A BFS starts at some arbitrary node of a graph and explores the neighbour nodes first, before moving to the next level neighbours.

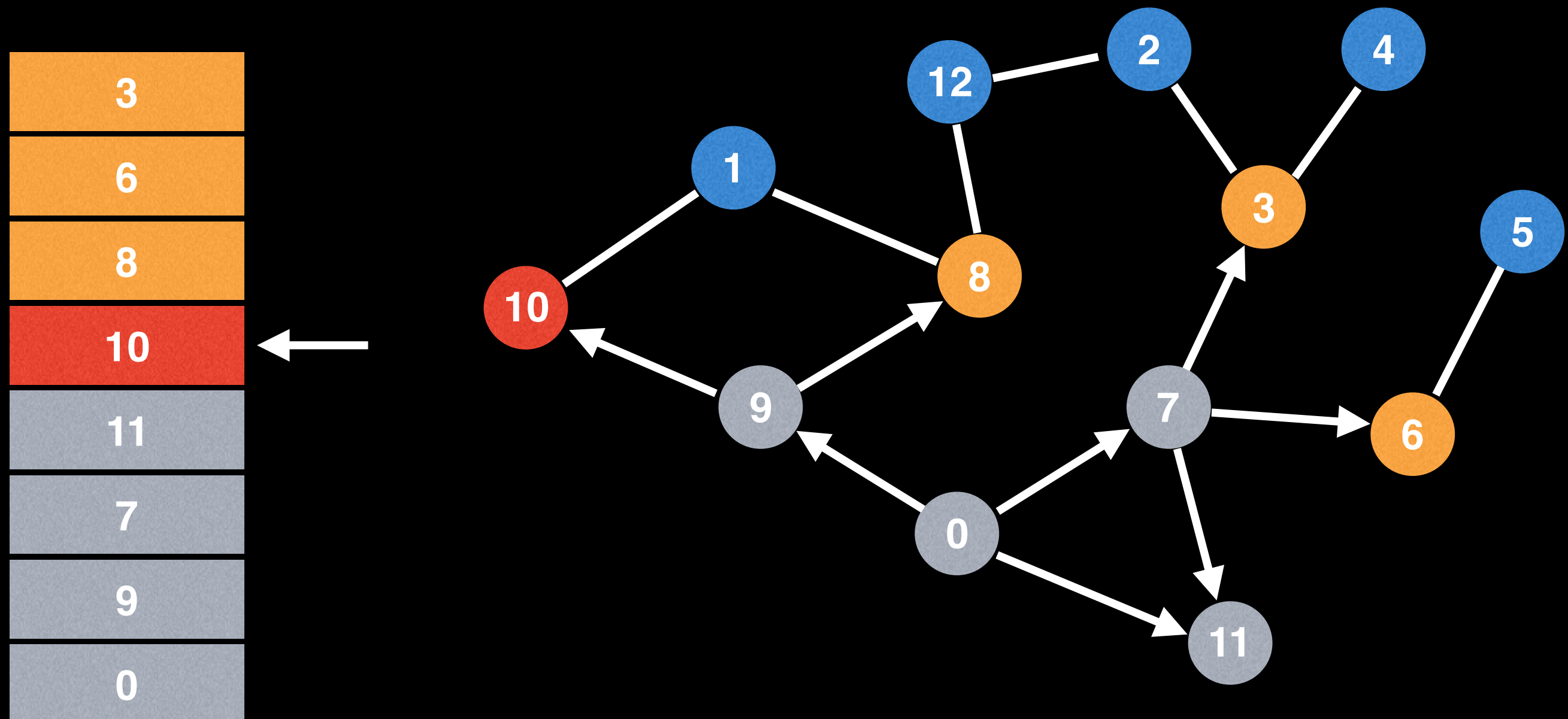


A BFS starts at some arbitrary node of a graph and explores the neighbour nodes first, before moving to the next level neighbours.

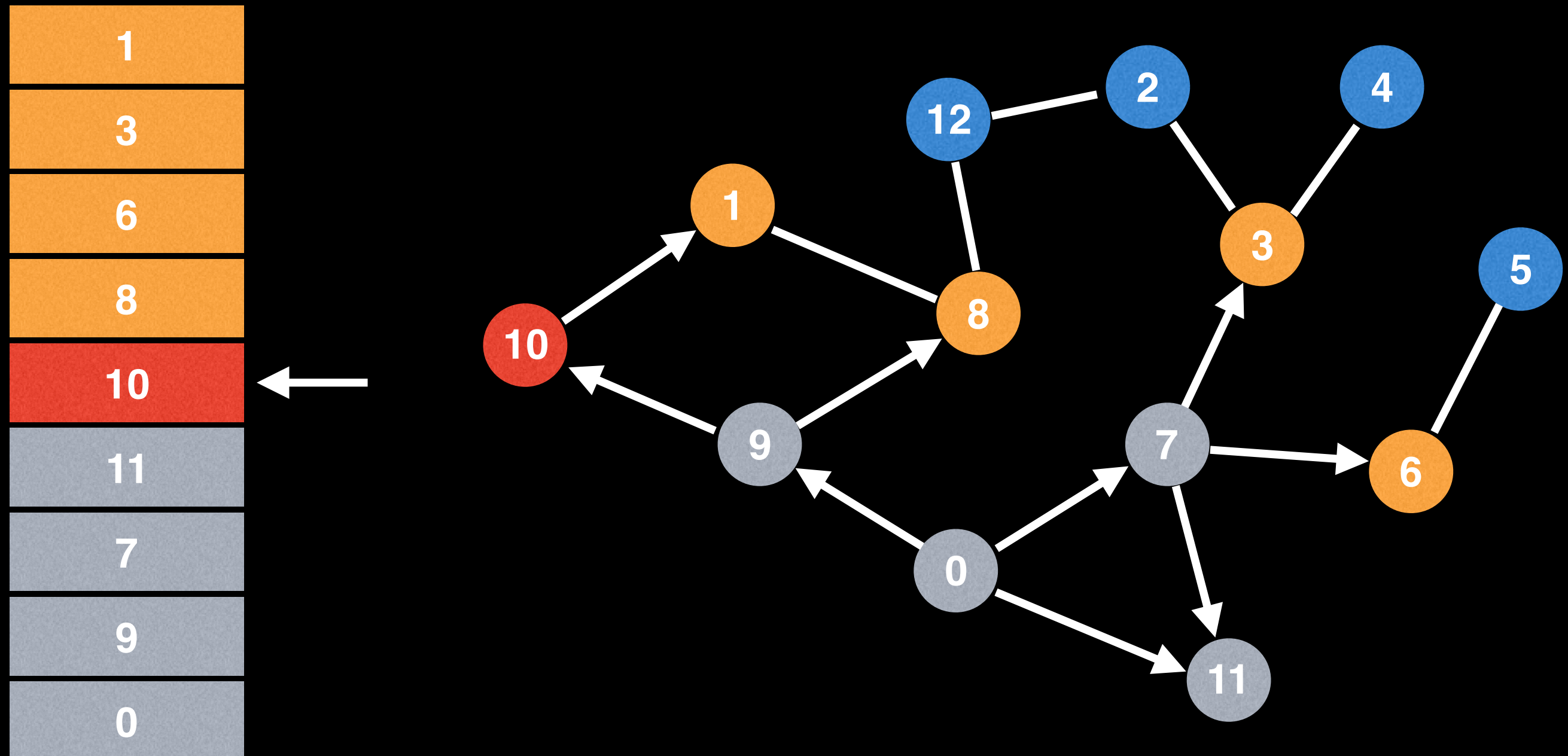




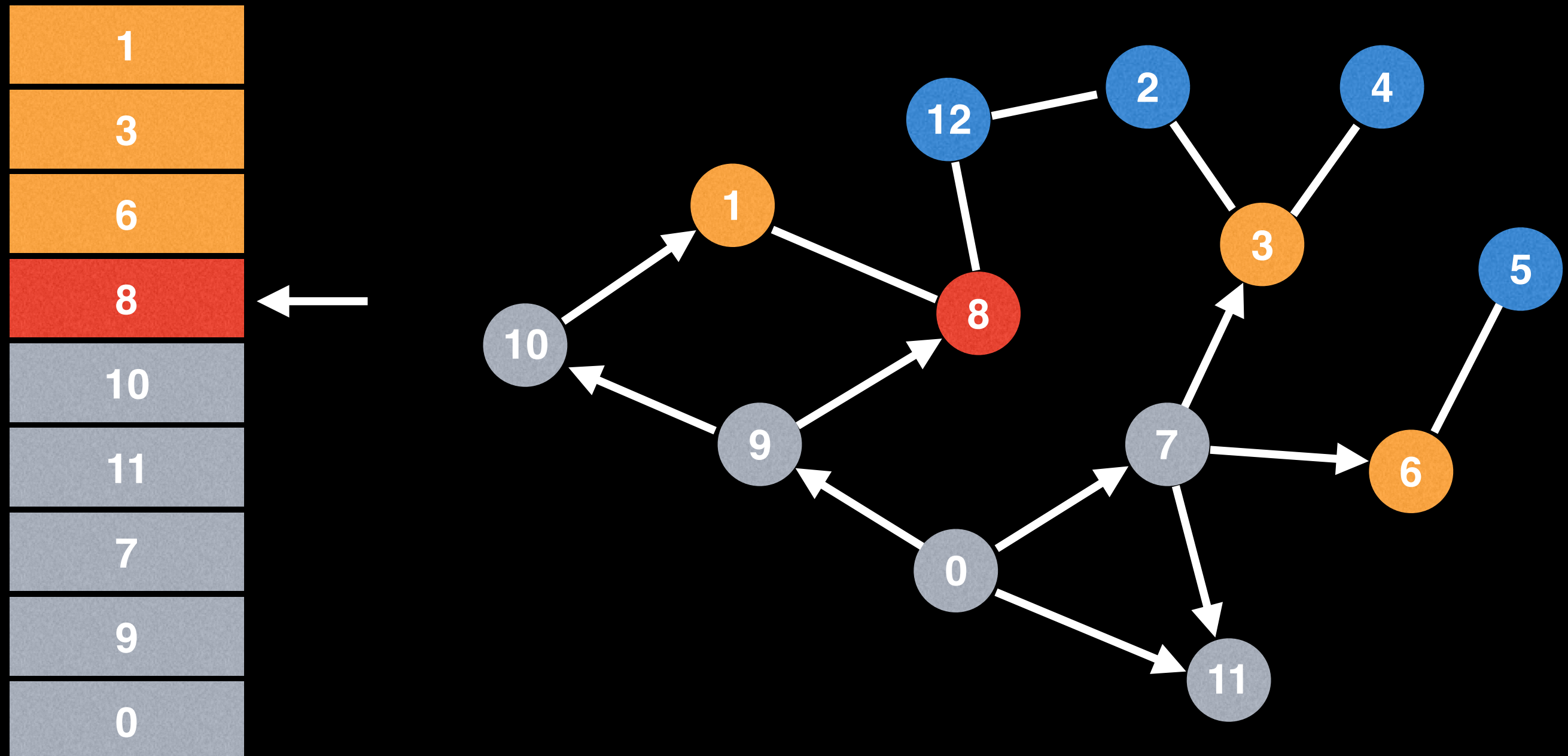
A BFS starts at some arbitrary node of a graph and explores the neighbour nodes first, before moving to the next level neighbours.



A BFS starts at some arbitrary node of a graph and explores the neighbour nodes first, before moving to the next level neighbours.

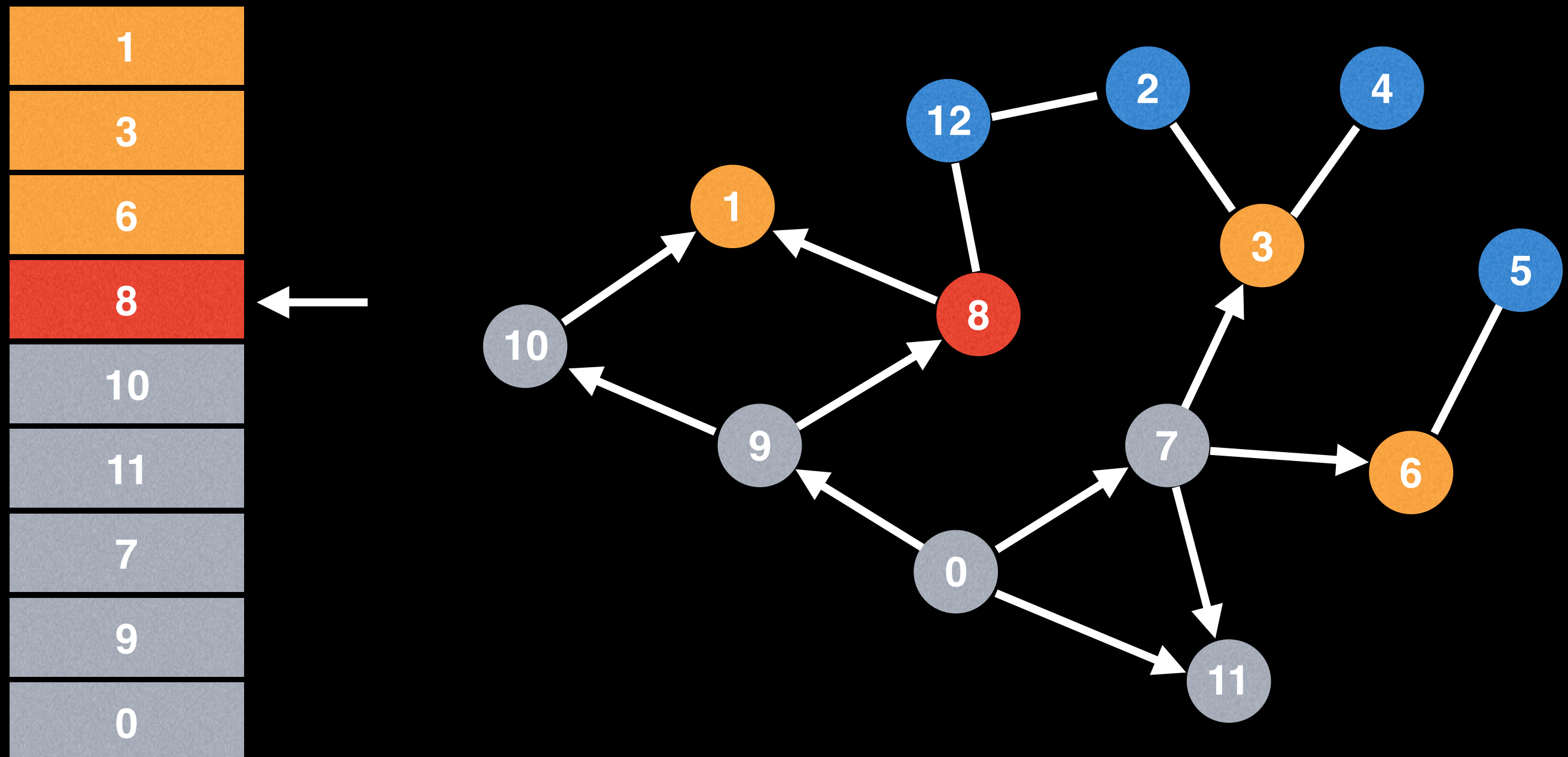


A BFS starts at some arbitrary node of a graph and explores the neighbour nodes first, before moving to the next level neighbours.

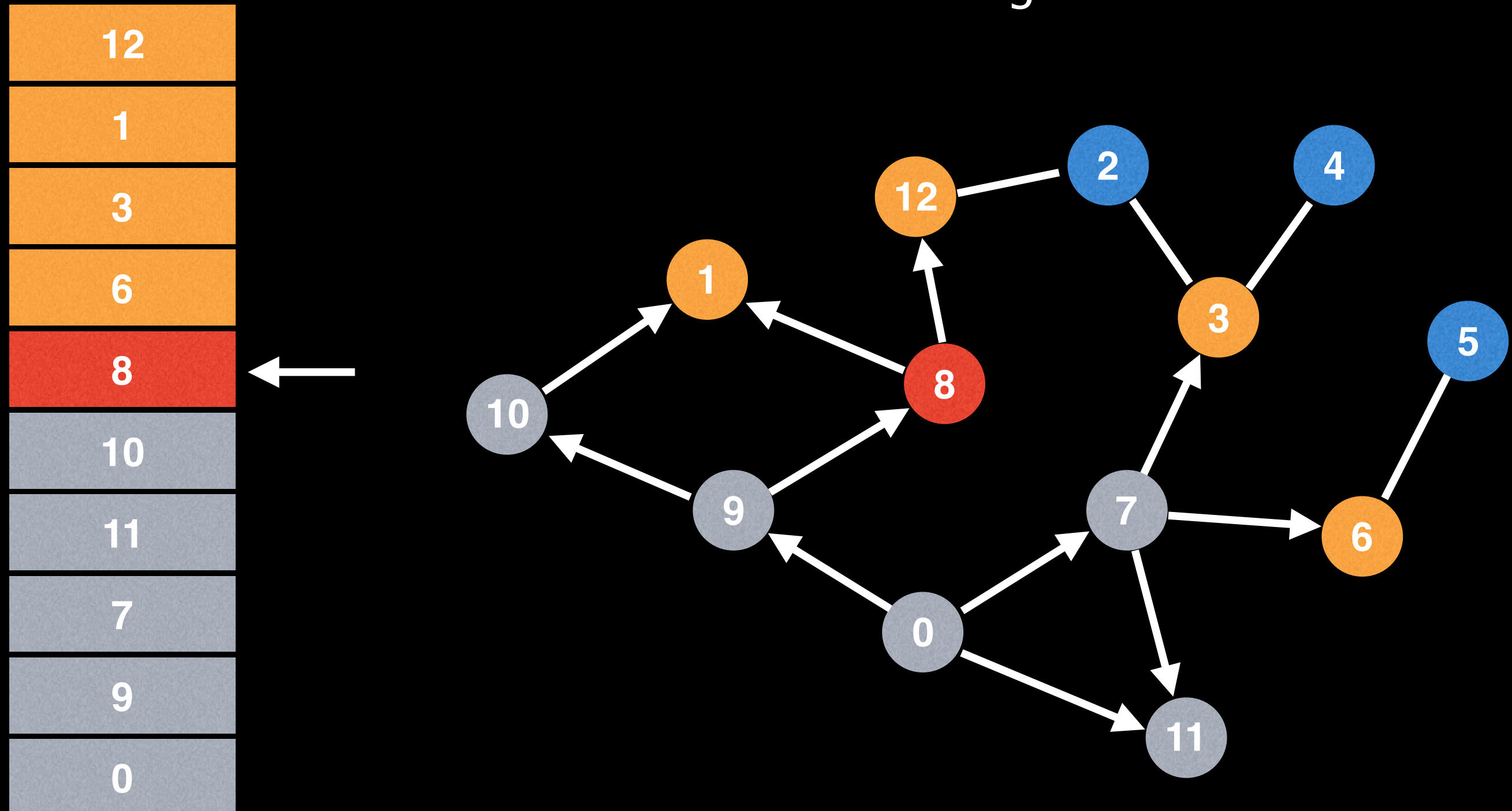




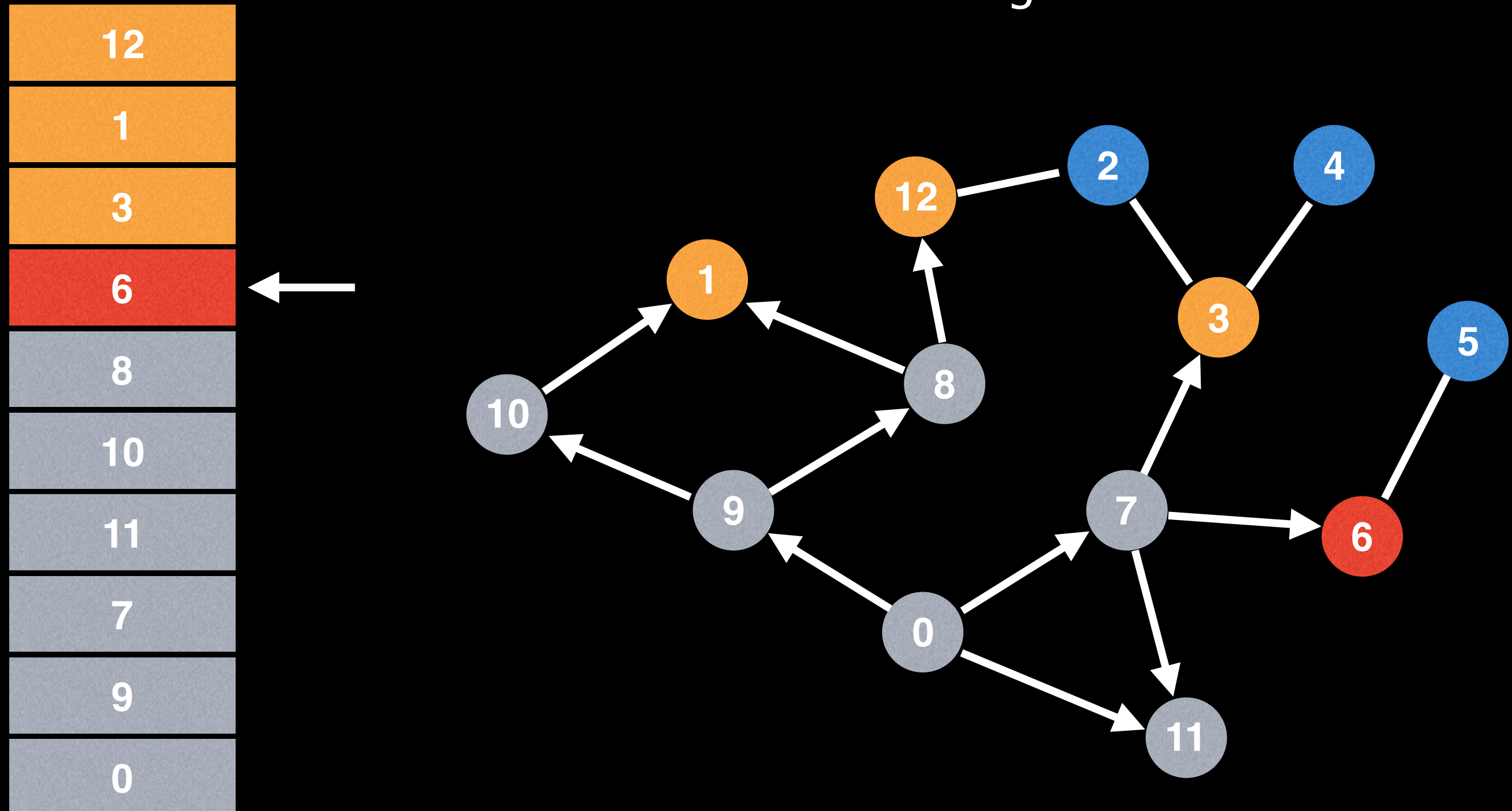
A BFS starts at some arbitrary node of a graph and explores the neighbour nodes first, before moving to the next level neighbours.



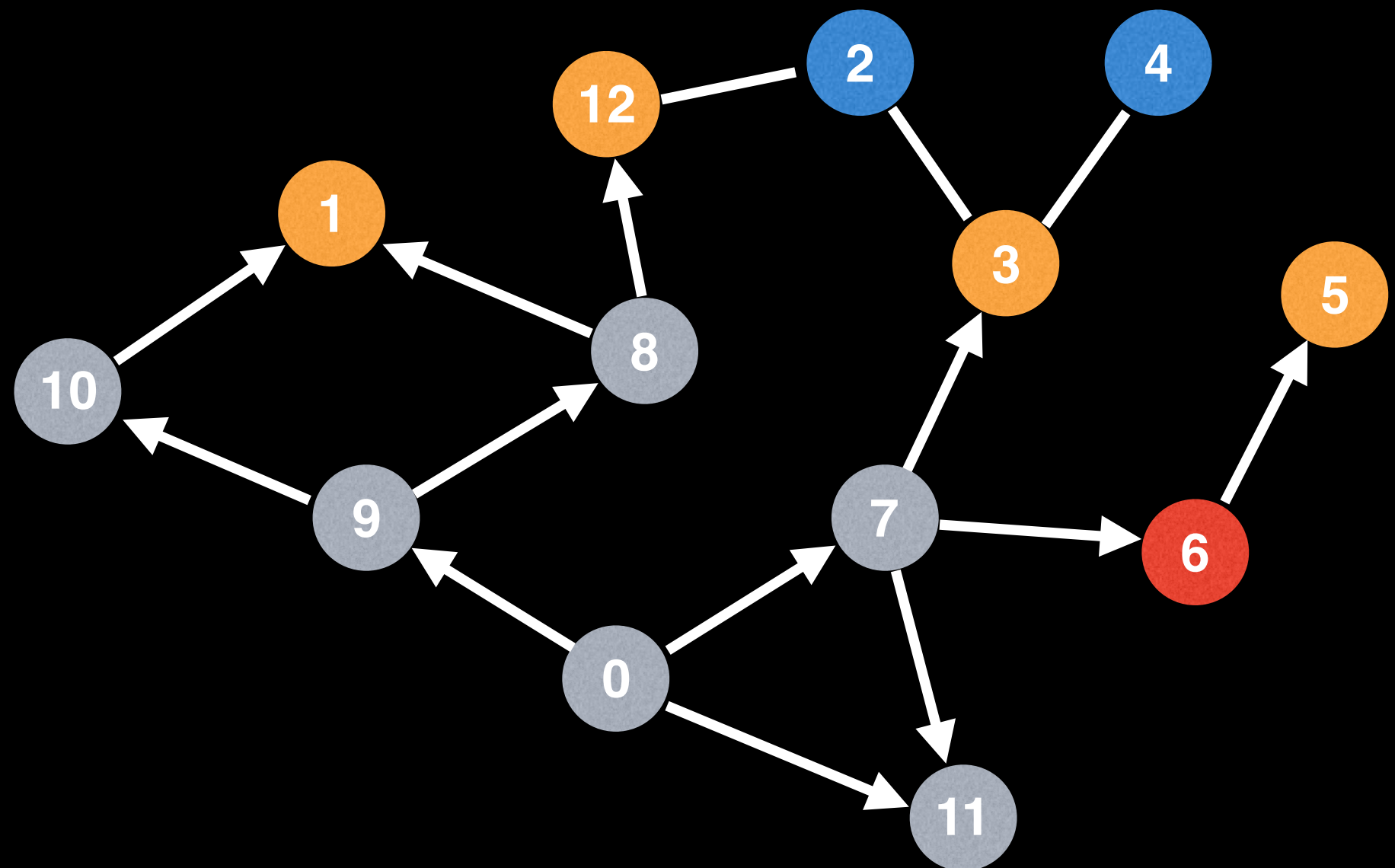
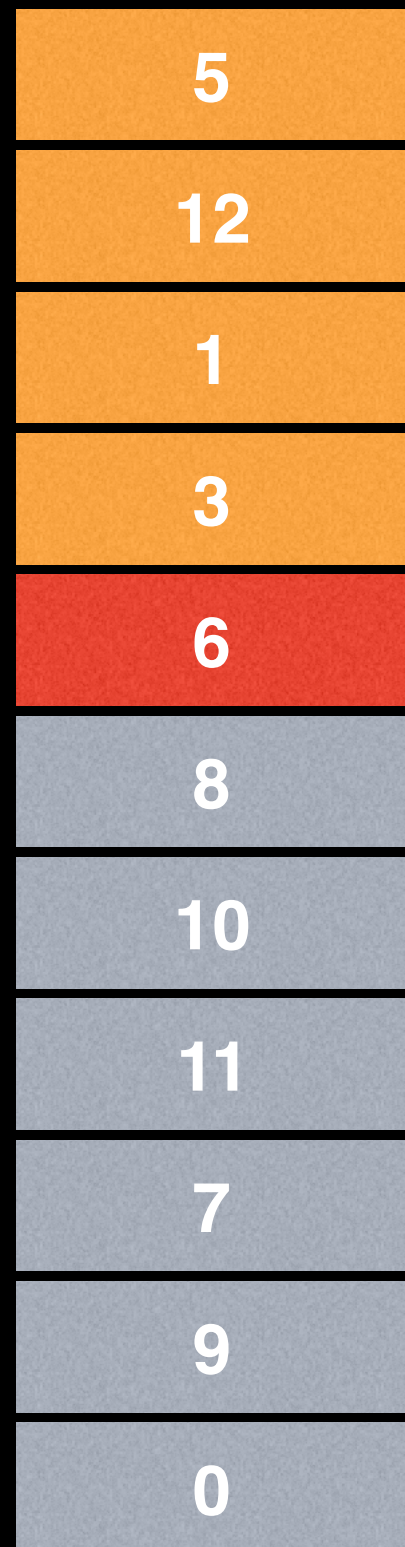
A BFS starts at some arbitrary node of a graph and explores the neighbour nodes first, before moving to the next level neighbours.



A BFS starts at some arbitrary node of a graph and explores the neighbour nodes first, before moving to the next level neighbours.

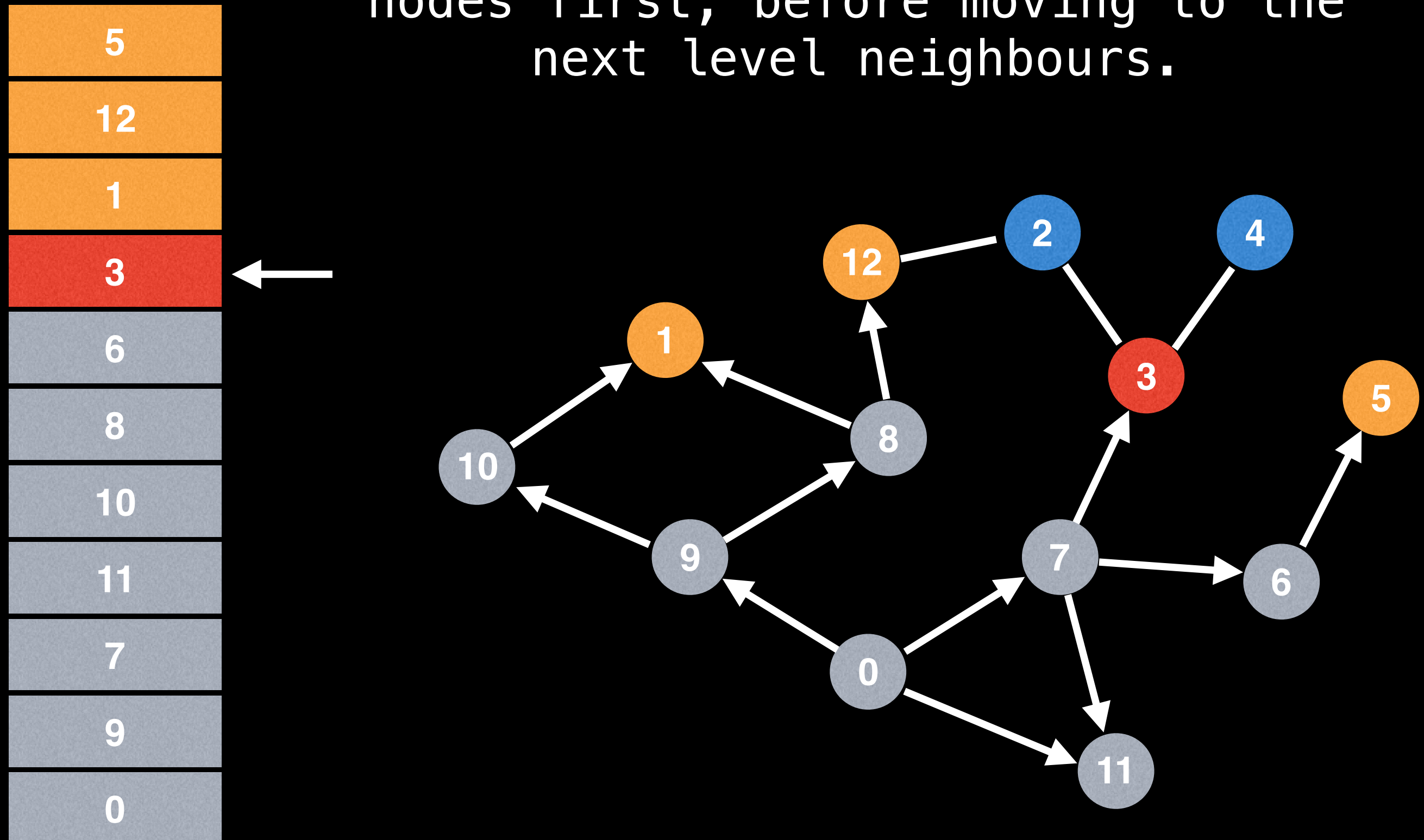


A BFS starts at some arbitrary node of a graph and explores the neighbour nodes first, before moving to the next level neighbours.

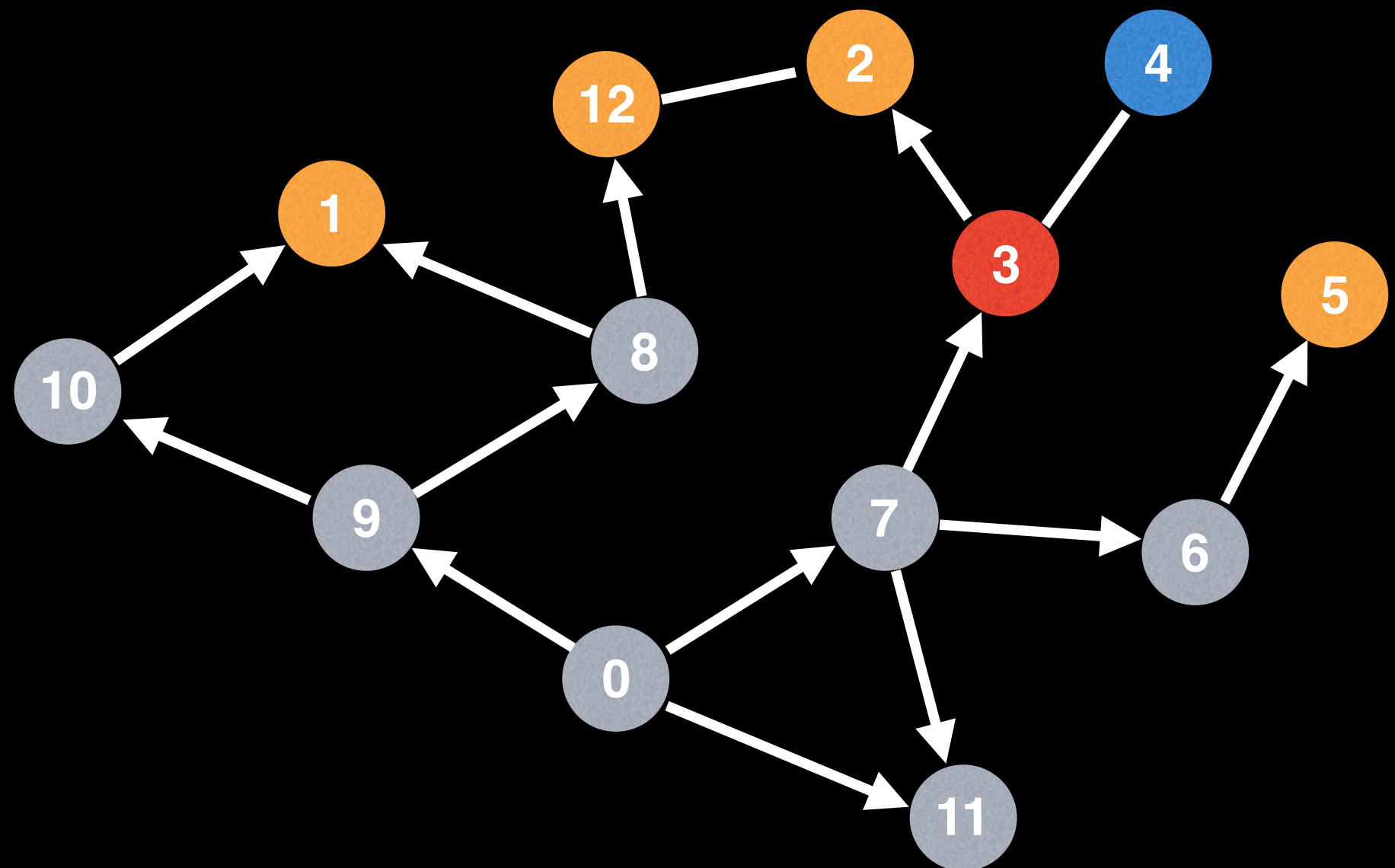
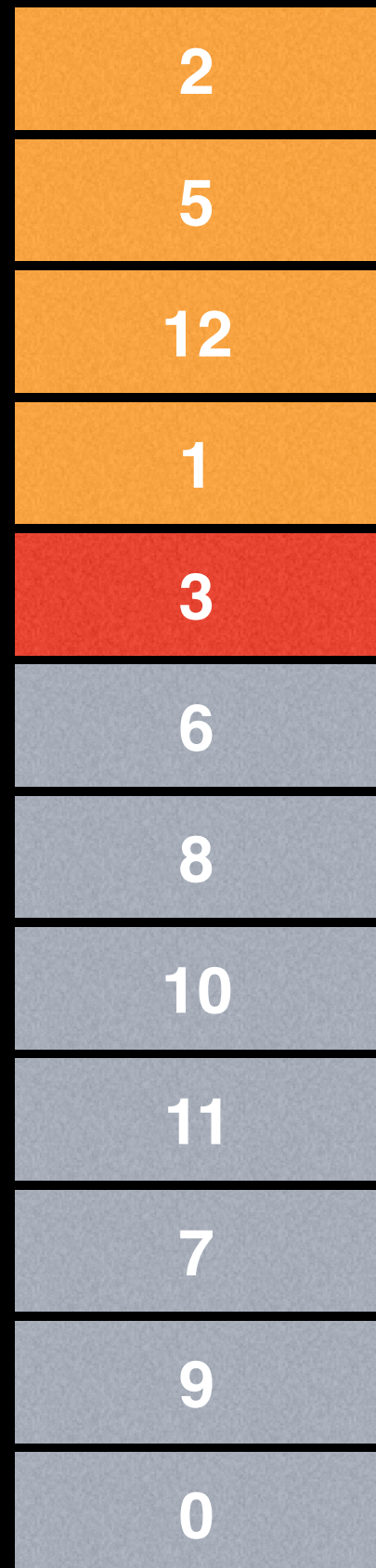




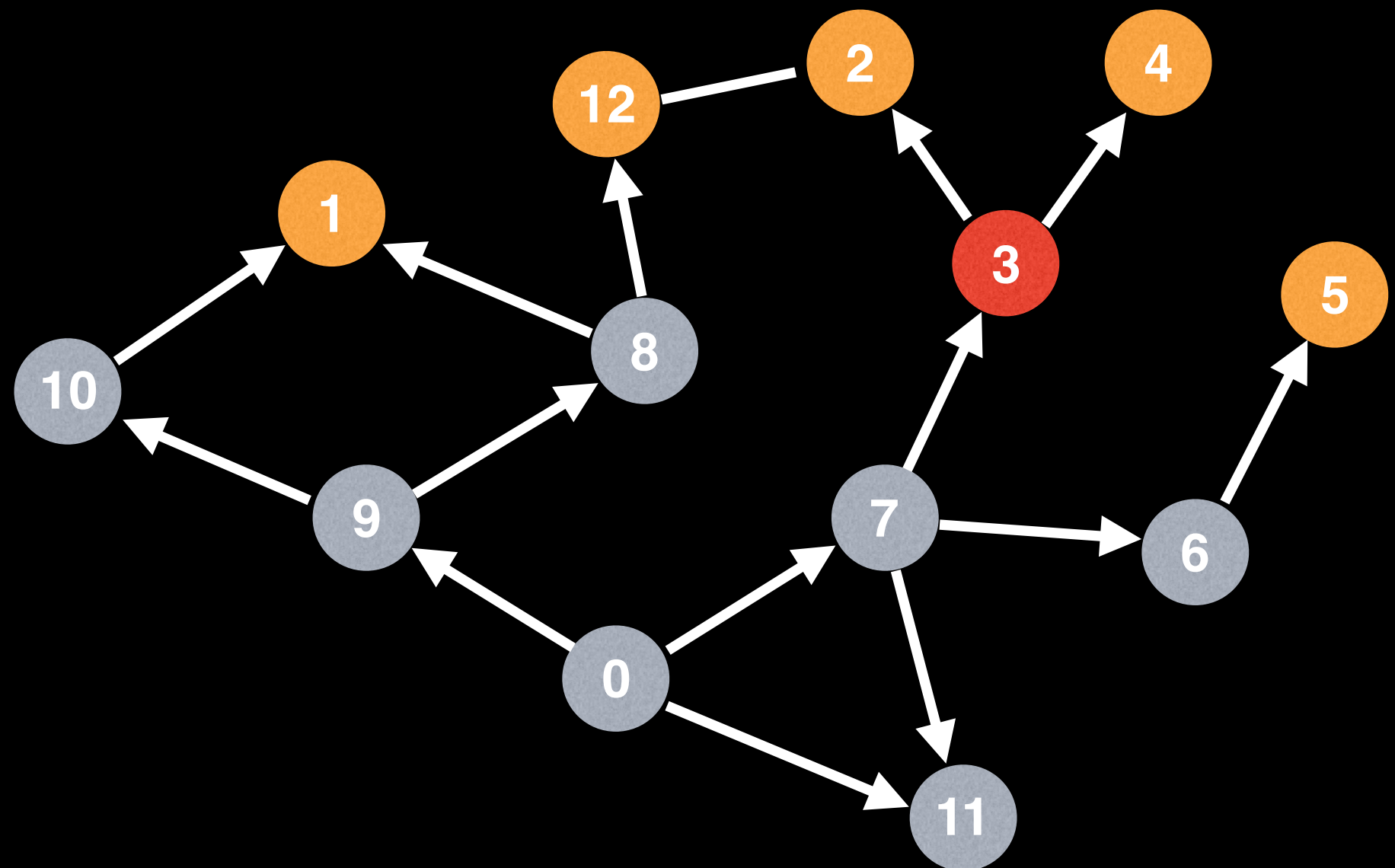
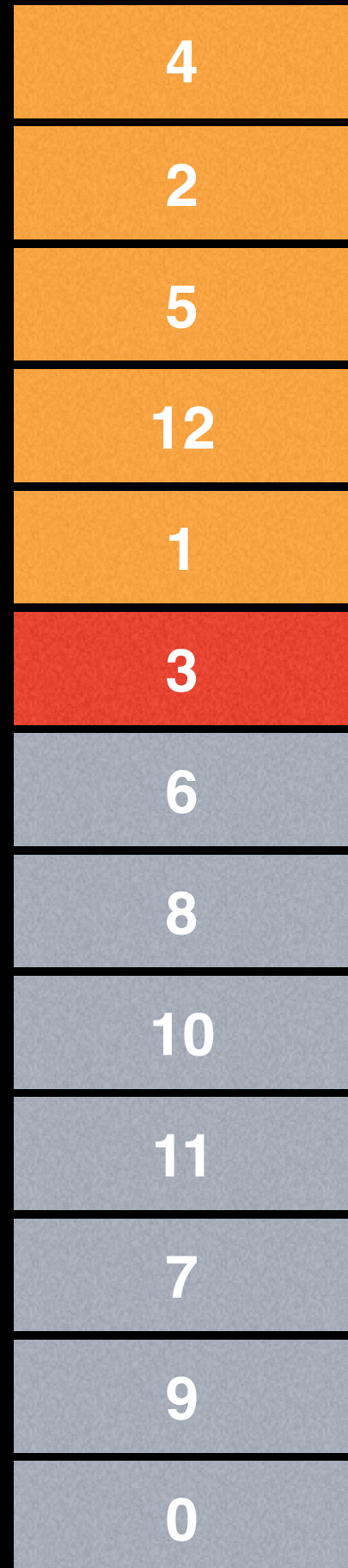
A BFS starts at some arbitrary node of a graph and explores the neighbour nodes first, before moving to the next level neighbours.



A BFS starts at some arbitrary node of a graph and explores the neighbour nodes first, before moving to the next level neighbours.

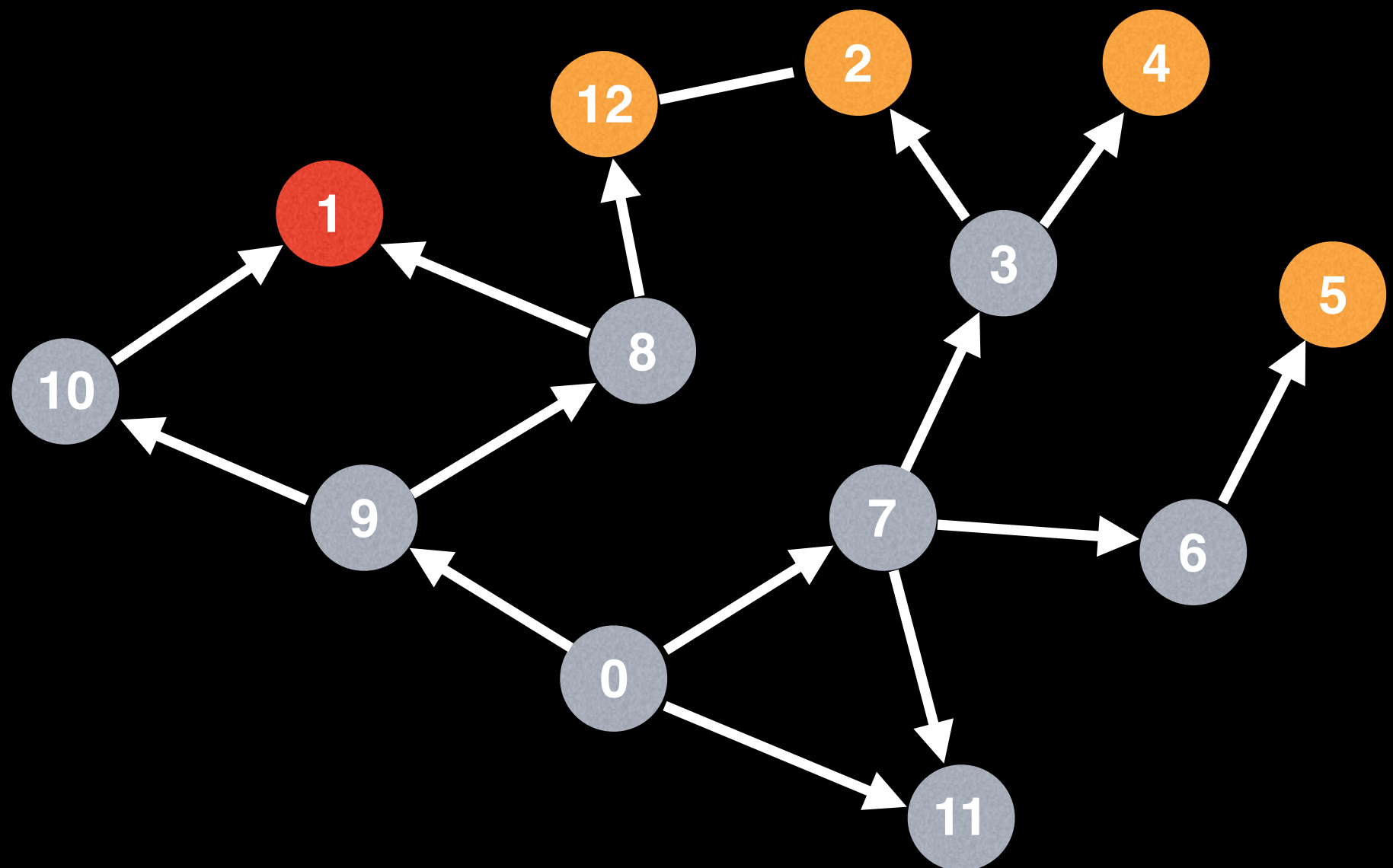


A BFS starts at some arbitrary node of a graph and explores the neighbour nodes first, before moving to the next level neighbours.



A BFS starts at some arbitrary node of a graph and explores the neighbour nodes first, before moving to the next level neighbours.

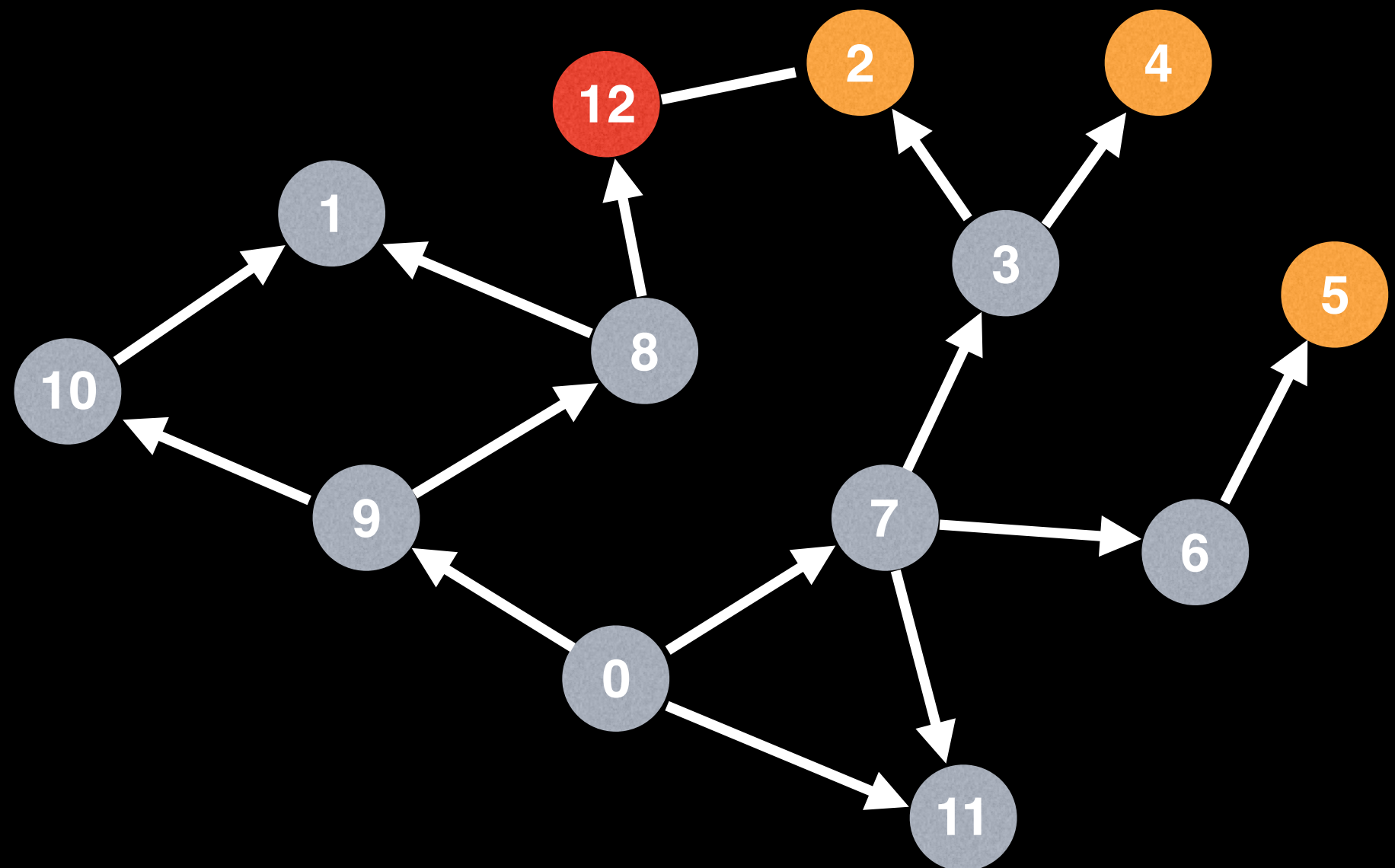
4
2
5
12
1
3
6
8
10
11
7
9
0





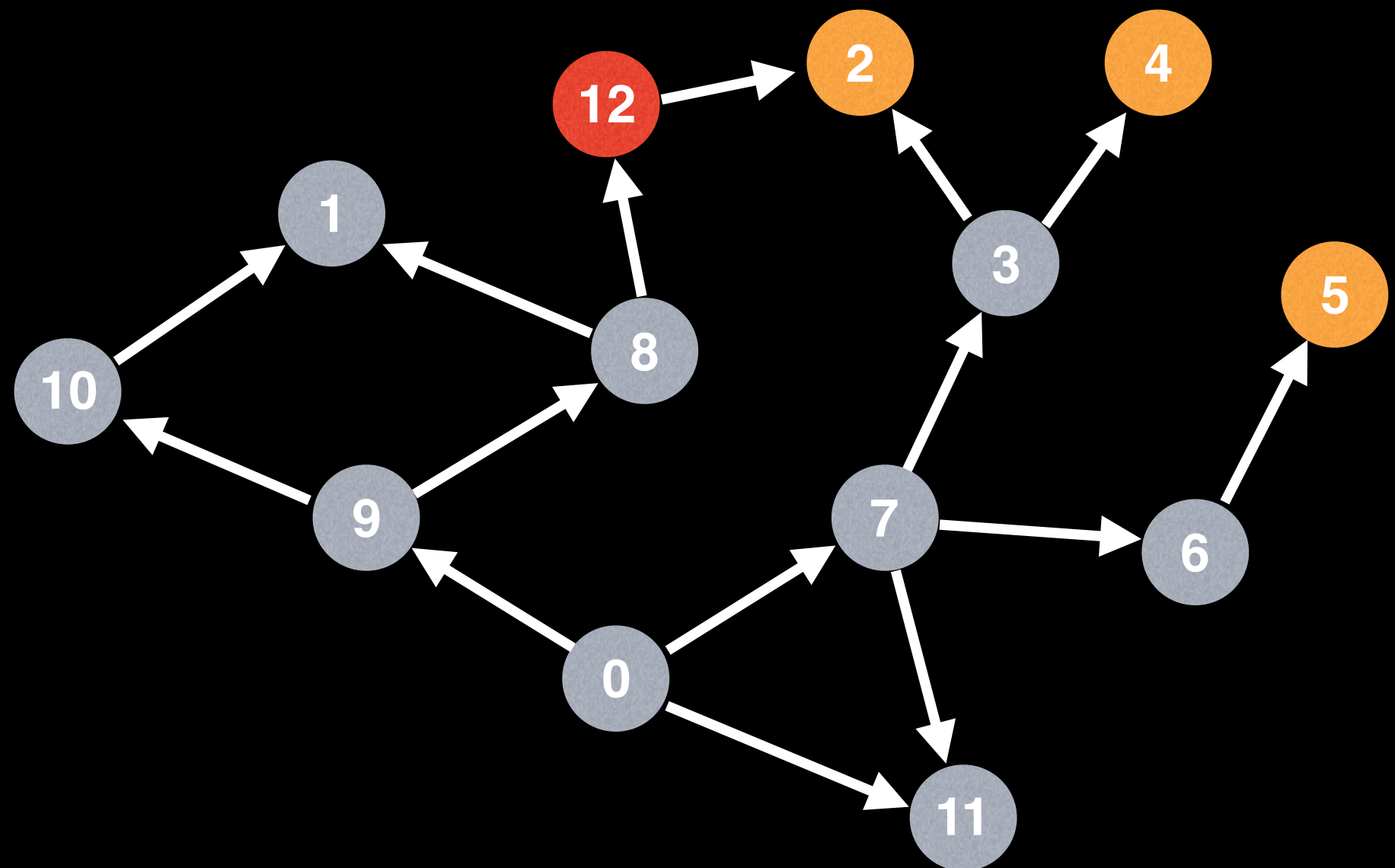
A BFS starts at some arbitrary node of a graph and explores the neighbour nodes first, before moving to the next level neighbours.

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



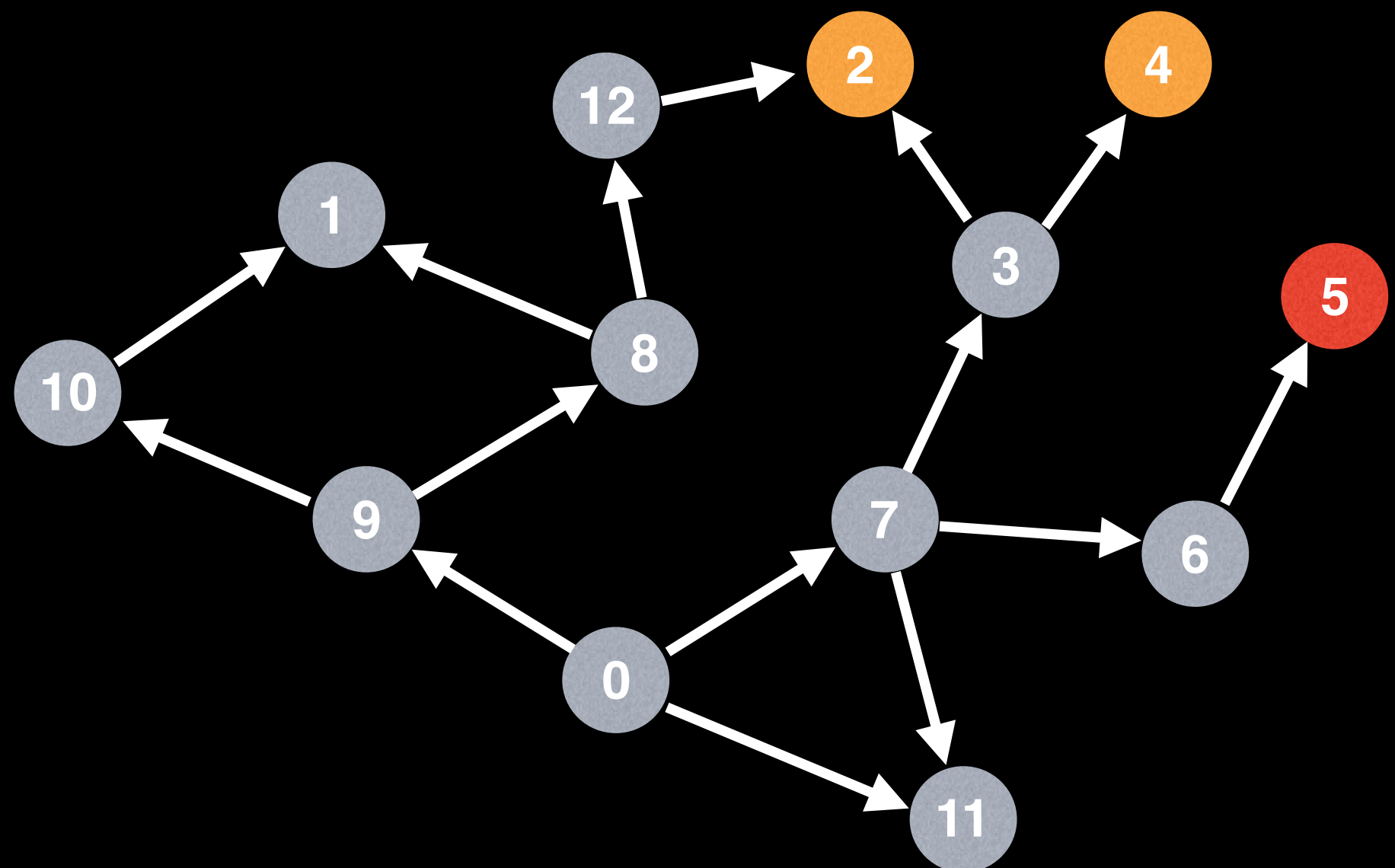
A BFS starts at some arbitrary node of a graph and explores the neighbour nodes first, before moving to the next level neighbours.

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



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

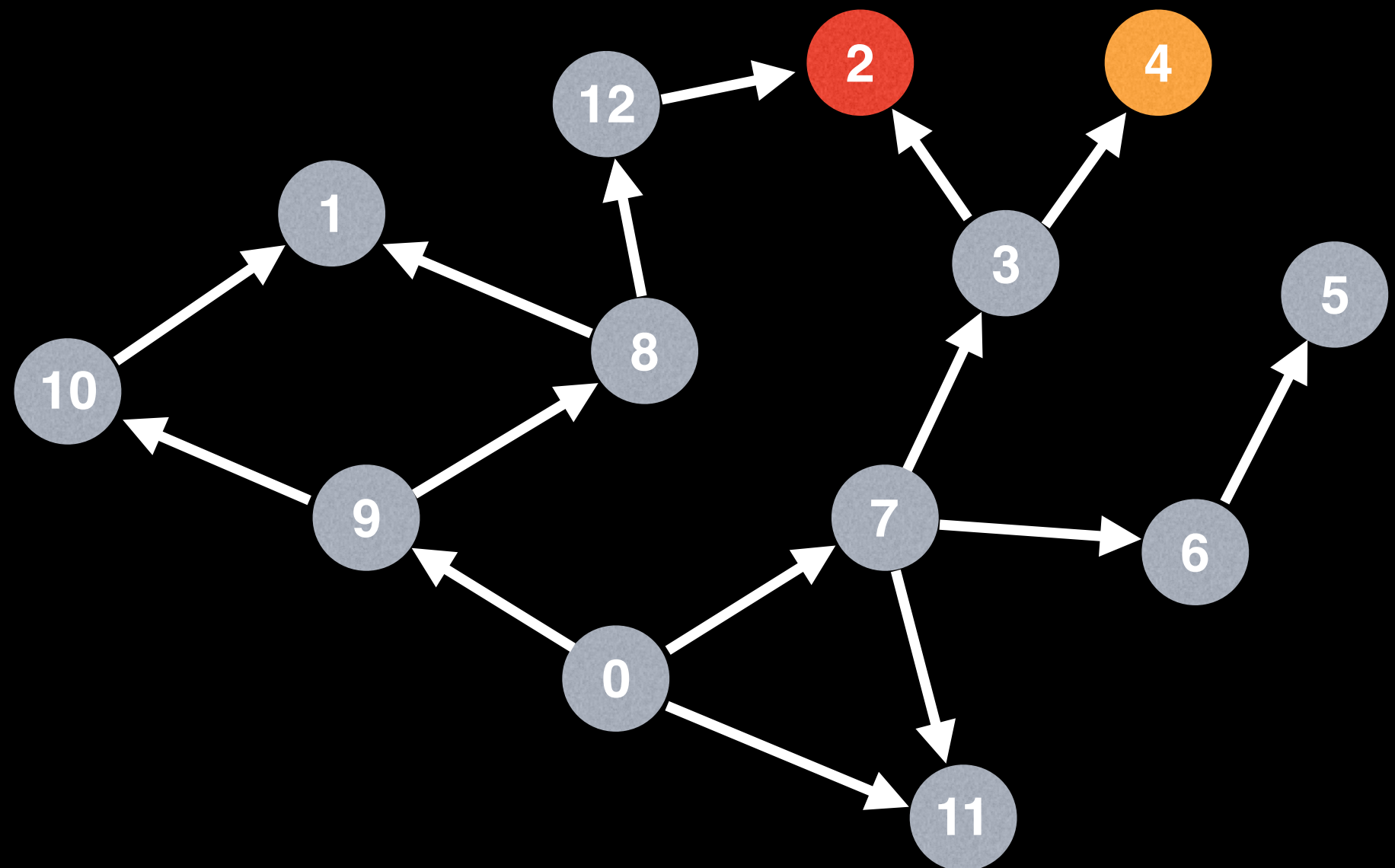
A BFS starts at some arbitrary node of a graph and explores the neighbour nodes first, before moving to the next level neighbours.





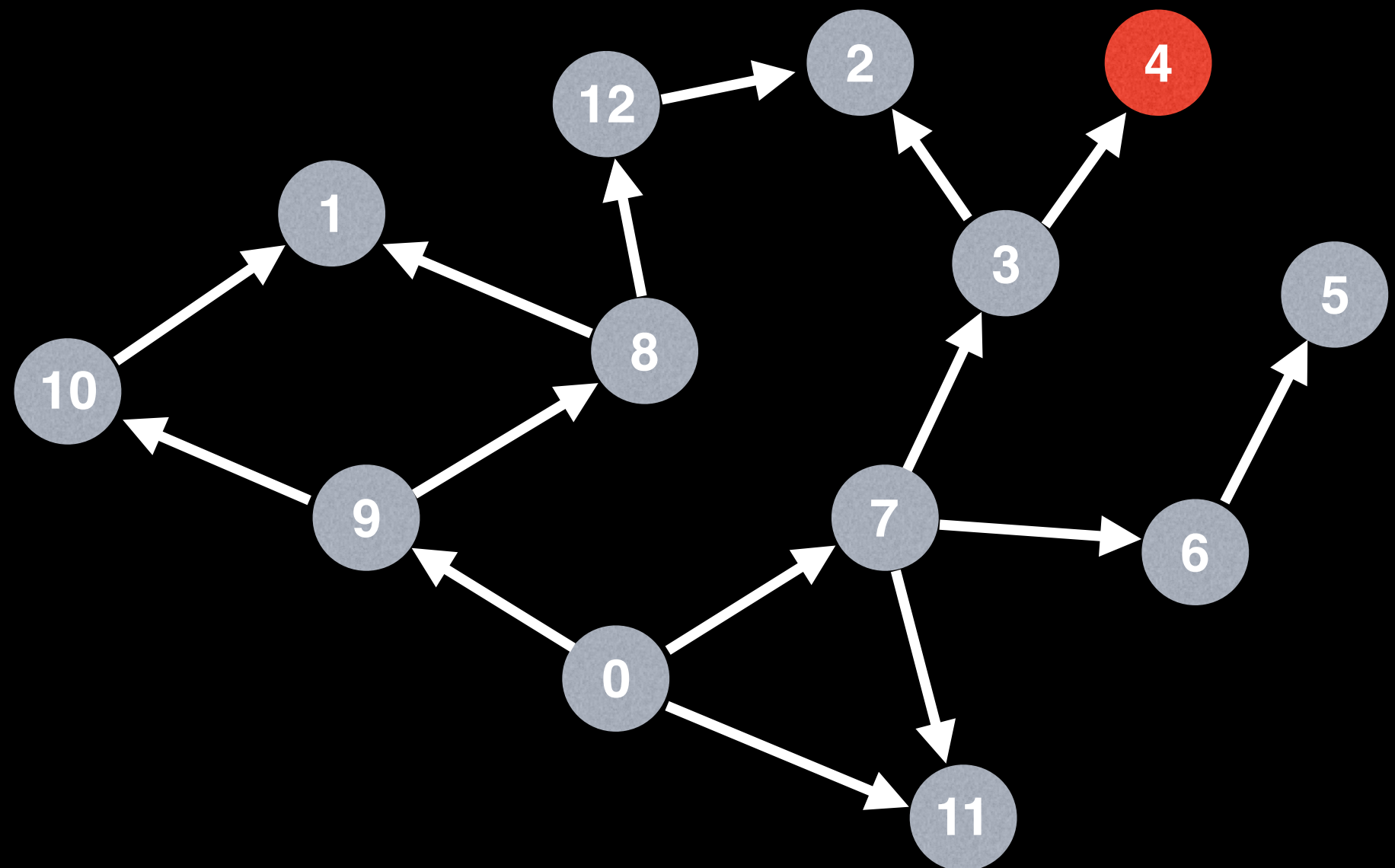
4
2
5
12
1
3
6
8
10
11
7
9
0

A BFS starts at some arbitrary node of a graph and explores the neighbour nodes first, before moving to the next level neighbours.



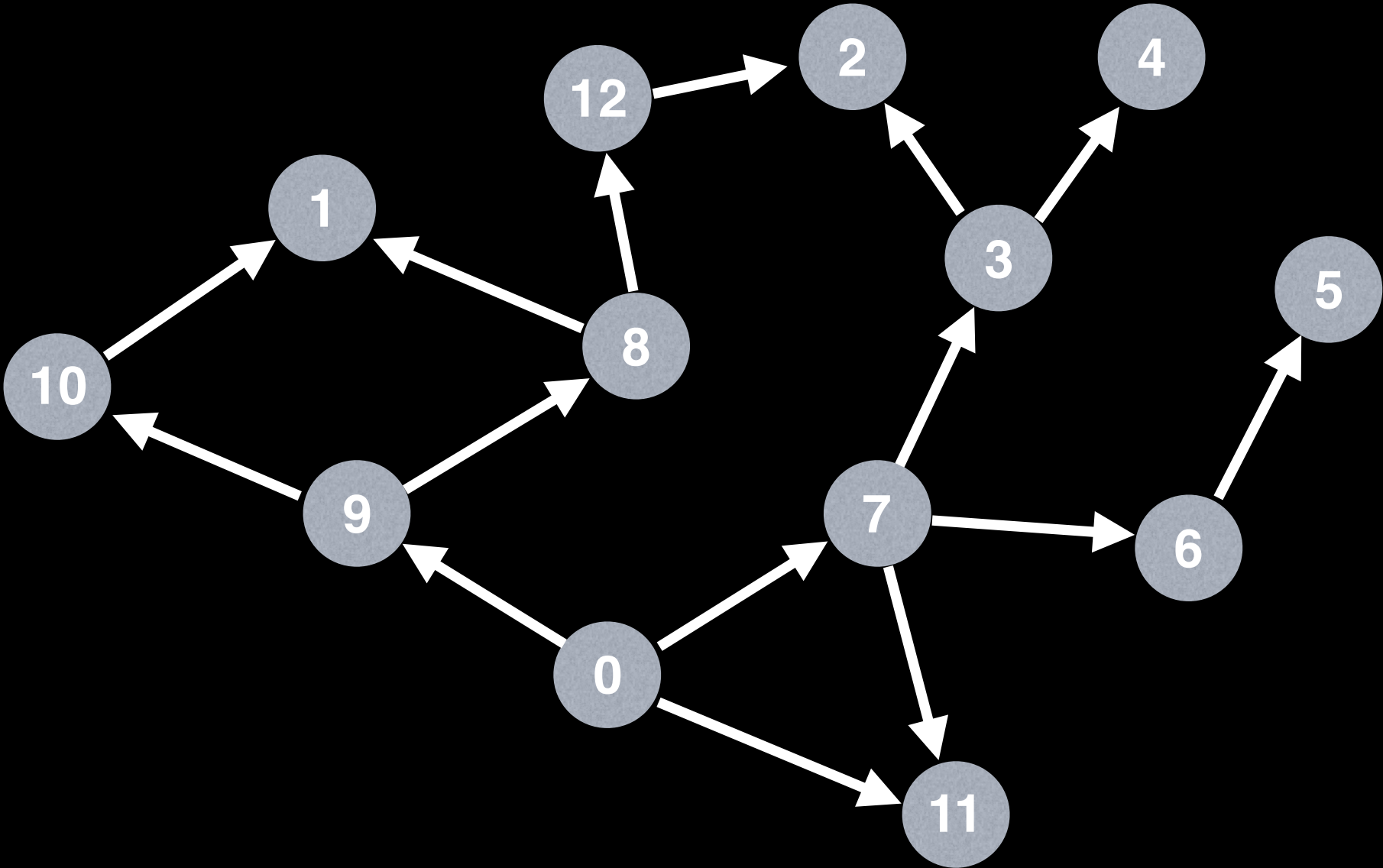
4
2
5
12
1
3
6
8
10
11
7
9
0

← A BFS starts at some arbitrary node of a graph and explores the neighbour nodes first, before moving to the next level neighbours.



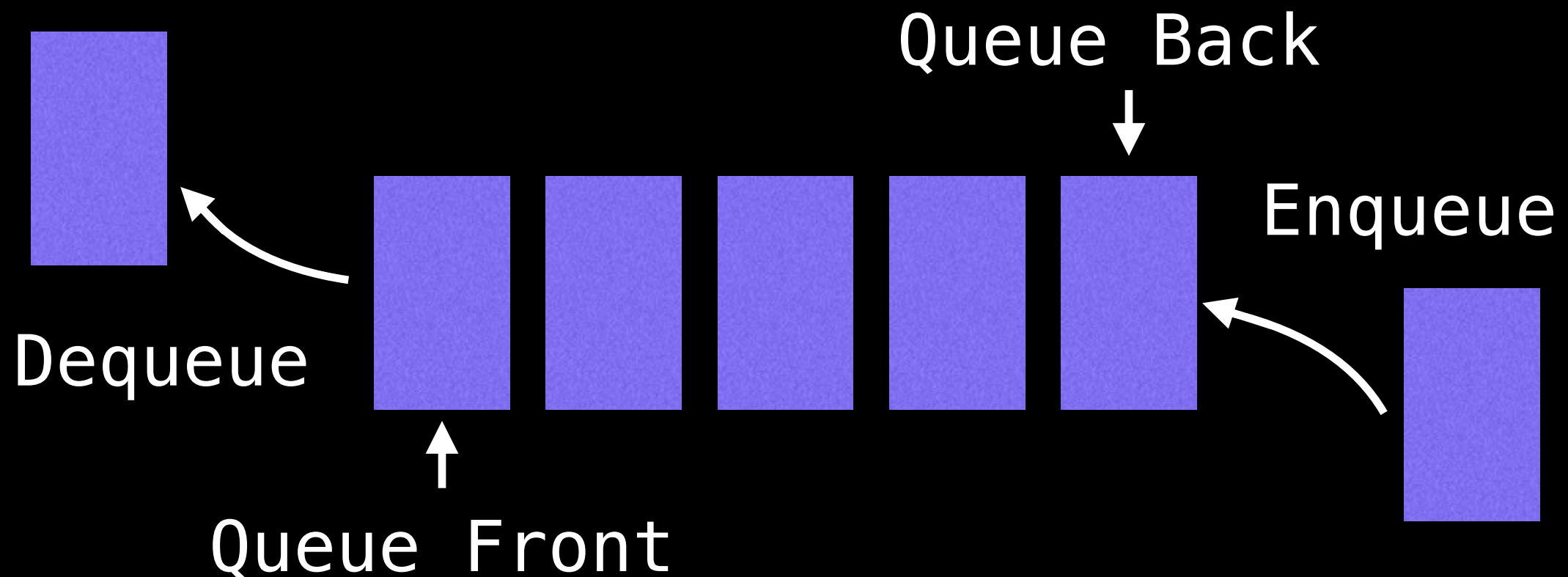
4
2
5
12
1
3
6
8
10
11
7
9
0

A BFS starts at some arbitrary node of a graph and explores the neighbour nodes first, before moving to the next level neighbours.



# Using a Queue

The BFS algorithm uses a queue data structure to track which node to visit next. Upon reaching a new node the algorithm adds it to the queue to visit it later. The queue data structure works just like a real world queue such as a waiting line at a restaurant. People can either enter the waiting line (**enqueue**) or get seated (**dequeue**).



```
# Global/class scope variables
n = number of nodes in the graph
g = adjacency list representing unweighted graph

# s = start node, e = end node, and  $0 \leq e, s < n$ 
function bfs(s, e):

    # Do a BFS starting at node s
    prev = solve(s)

    # Return reconstructed path from s  $\rightarrow$  e
    return reconstructPath(s, e, prev)
```



```
# Global/class scope variables  
n = number of nodes in the graph  
g = adjacency list representing unweighted graph
```

```
# s = start node, e = end node, and  $0 \leq e, s < n$   
function bfs(s, e):
```

```
    # Do a BFS starting at node s  
    prev = solve(s)
```

```
    # Return reconstructed path from s  $\rightarrow$  e  
    return reconstructPath(s, e, prev)
```

```
# Global/class scope variables
n = number of nodes in the graph
g = adjacency list representing unweighted graph
```

```
# s = start node, e = end node, and  $0 \leq e, s < n$ 
function bfs(s, e):
```

```
    # Do a BFS starting at node s
    prev = solve(s)
```

```
    # Return reconstructed path from s  $\rightarrow$  e
    return reconstructPath(s, e, prev)
```

```
# Global/class scope variables
n = number of nodes in the graph
g = adjacency list representing unweighted graph

# s = start node, e = end node, and  $0 \leq e, s < n$ 
function bfs(s, e):

    # Do a BFS starting at node s
    prev = solve(s)

    # Return reconstructed path from s  $\rightarrow$  e
    return reconstructPath(s, e, prev)
```

```
function solve(s):  
    q = queue data structure with enqueue and dequeue  
    q.enqueue(s)  
  
    visited = [false, ..., false] # size n  
    visited[s] = true  
  
    prev = [null, ..., null] # size n  
    while !q.isEmpty():  
        node = q.dequeue()  
        neighbours = g.get(node)  
  
        for(next : neighbours):  
            if !visited[next]:  
                q.enqueue(next)  
                visited[next] = true  
                prev[next] = node  
    return prev
```

```
function solve(s):
```

```
q = queue data structure with enqueue and dequeue  
q.enqueue(s)
```

```
visited = [false, ..., false] # size n  
visited[s] = true
```

```
prev = [null, ..., null] # size n
```

```
while !q.isEmpty():  
    node = q.dequeue()  
    neighbours = g.get(node)
```

```
    for(next : neighbours):  
        if !visited[next]:  
            q.enqueue(next)  
            visited[next] = true  
            prev[next] = node
```

```
return prev
```

```
function solve(s):  
    q = queue data structure with enqueue and dequeue  
    q.enqueue(s)  
  
    visited = [false, ..., false] # size n  
    visited[s] = true  
  
    prev = [null, ..., null] # size n  
    while !q.isEmpty():  
        node = q.dequeue()  
        neighbours = g.get(node)  
  
        for(next : neighbours):  
            if !visited[next]:  
                q.enqueue(next)  
                visited[next] = true  
                prev[next] = node  
    return prev
```

```
function solve(s):  
    q = queue data structure with enqueue and dequeue  
    q.enqueue(s)  
  
    visited = [false, ..., false] # size n  
    visited[s] = true  
  
    prev = [null, ..., null] # size n  
    while !q.isEmpty():  
        node = q.dequeue()  
        neighbours = g.get(node)  
  
        for(next : neighbours):  
            if !visited[next]:  
                q.enqueue(next)  
                visited[next] = true  
                prev[next] = node  
    return prev
```

```
function solve(s):  
    q = queue data structure with enqueue and dequeue  
    q.enqueue(s)  
  
    visited = [false, ..., false] # size n  
    visited[s] = true  
  
    prev = [null, ..., null] # size n  
    while !q.isEmpty():  
        node = q.dequeue()  
        neighbours = g.get(node)  
  
        for(next : neighbours):  
            if !visited[next]:  
                q.enqueue(next)  
                visited[next] = true  
                prev[next] = node  
    return prev
```



```
function solve(s):  
    q = queue data structure with enqueue and dequeue  
    q.enqueue(s)  
  
    visited = [false, ..., false] # size n  
    visited[s] = true  
  
    prev = [null, ..., null] # size n  
    while !q.isEmpty():  
        node = q.dequeue()  
        neighbours = g.get(node)  
  
        for(next : neighbours):  
            if !visited[next]:  
                q.enqueue(next)  
                visited[next] = true  
                prev[next] = node  
    return prev
```

```
function solve(s):  
    q = queue data structure with enqueue and dequeue  
    q.enqueue(s)  
  
    visited = [false, ..., false] # size n  
    visited[s] = true  
  
    prev = [null, ..., null] # size n  
    while !q.isEmpty():  
        node = q.dequeue()  
        neighbours = g.get(node)  
  
        for(next : neighbours):  
            if !visited[next]:  
                q.enqueue(next)  
                visited[next] = true  
                prev[next] = node  
    return prev
```

```
function solve(s):  
    q = queue data structure with enqueue and dequeue  
    q.enqueue(s)  
  
    visited = [false, ..., false] # size n  
    visited[s] = true  
  
    prev = [null, ..., null] # size n  
    while !q.isEmpty():  
        node = q.dequeue()  
        neighbours = g.get(node)  
  
        for(next : neighbours):  
            if !visited[next]:  
                q.enqueue(next)  
                visited[next] = true  
                prev[next] = node  
    return prev
```

```
function solve(s):  
    q = queue data structure with enqueue and dequeue  
    q.enqueue(s)  
  
    visited = [false, ..., false] # size n  
    visited[s] = true  
  
    prev = [null, ..., null] # size n  
    while !q.isEmpty():  
        node = q.dequeue()  
        neighbours = g.get(node)  
  
        for(next : neighbours):  
            if !visited[next]:  
                q.enqueue(next)  
                visited[next] = true  
                prev[next] = node  
    return prev
```

```
# Global/class scope variables
n = number of nodes in the graph
g = adjacency list representing unweighted graph

# s = start node, e = end node, and  $0 \leq e, s < n$ 
function bfs(s, e):

    # Do a BFS starting at node s
    prev = solve(s)

    # Return reconstructed path from s  $\rightarrow$  e
    return reconstructPath(s, e, prev)
```

```
# Global/class scope variables
n = number of nodes in the graph
g = adjacency list representing unweighted graph

# s = start node, e = end node, and  $0 \leq e, s < n$ 
function bfs(s, e):

    # Do a BFS starting at node s
    prev = solve(s)

    # Return reconstructed path from s  $\rightarrow$  e
    return reconstructPath(s, e, prev)
```

```
function reconstructPath(s, e, prev):  
  
    # Reconstruct path going backwards from e  
    path = []  
    for(at = e; at != null; at = prev[at]):  
        path.add(at)  
  
    path.reverse()  
  
    # If s and e are connected return the path  
    if path[0] == s:  
        return path  
    return []
```



```
function reconstructPath(s, e, prev):
```

```
# Reconstruct path going backwards from e
path = []
for(at = e; at != null; at = prev[at]):
    path.add(at)
```

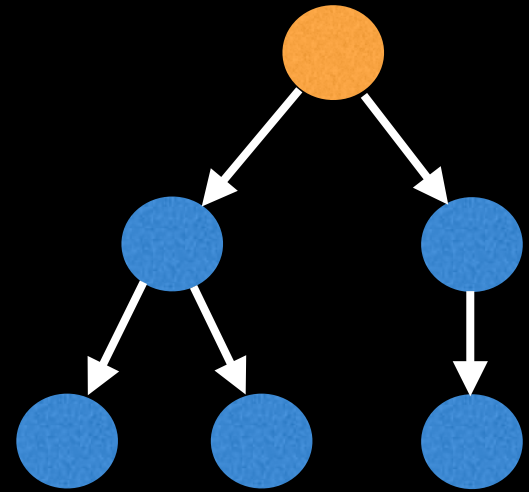
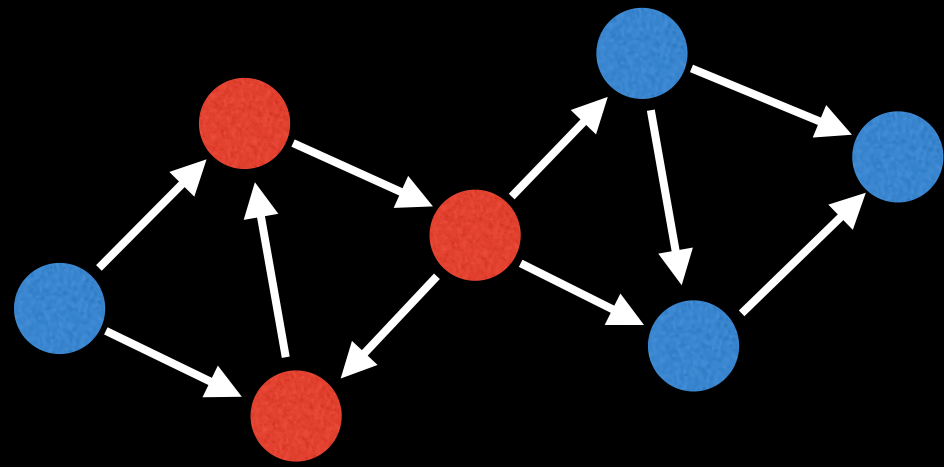
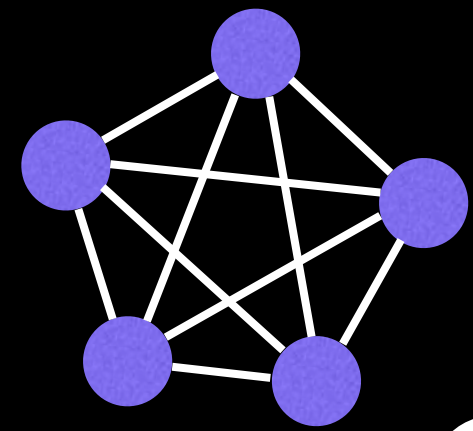
```
path.reverse()
```

```
# If s and e are connected return the path
if path[0] == s:
    return path
return []
```

```
function reconstructPath(s, e, prev):  
  
    # Reconstruct path going backwards from e  
    path = []  
    for(at = e; at != null; at = prev[at]):  
        path.add(at)  
  
    path.reverse()  
  
    # If s and e are connected return the path  
    if path[0] == s:  
        return path  
    return []
```

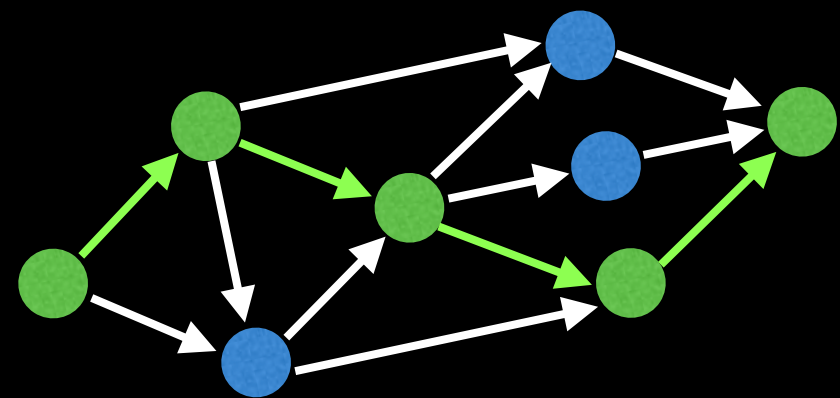
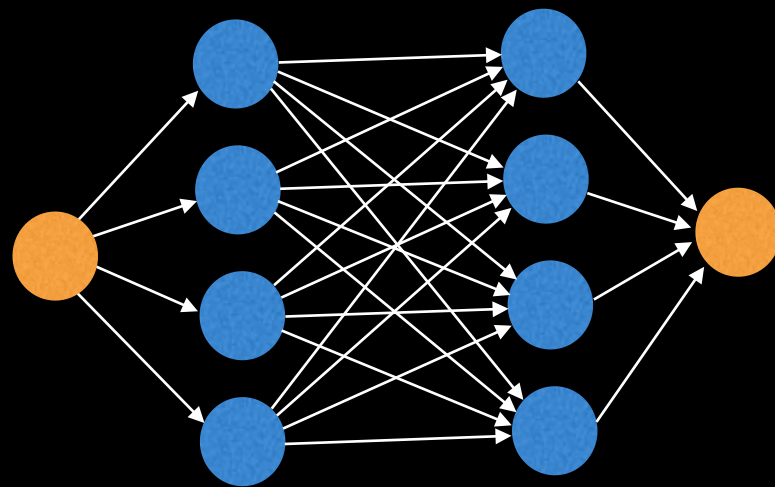
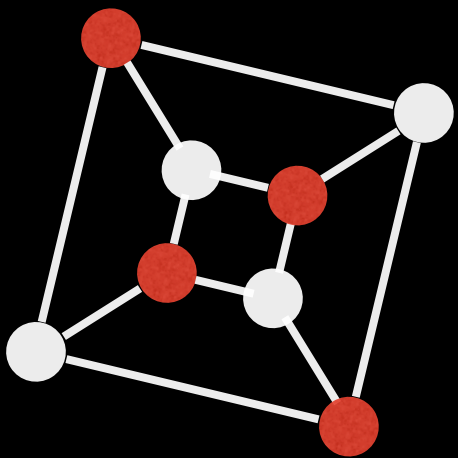
```
function reconstructPath(s, e, prev):  
  
    # Reconstruct path going backwards from e  
    path = []  
    for(at = e; at != null; at = prev[at]):  
        path.add(at)  
  
    path.reverse()  
  
    # If s and e are connected return the path  
    if path[0] == s:  
        return path  
    return []
```





# Graph Theory

## Video Series





# BFS Shortest Path on a Grid

William Fiset

# BFS Video

Please watch **Breadth First Search Algorithm** to understand the basics of a BFS before proceeding.

Link should be in the description below.

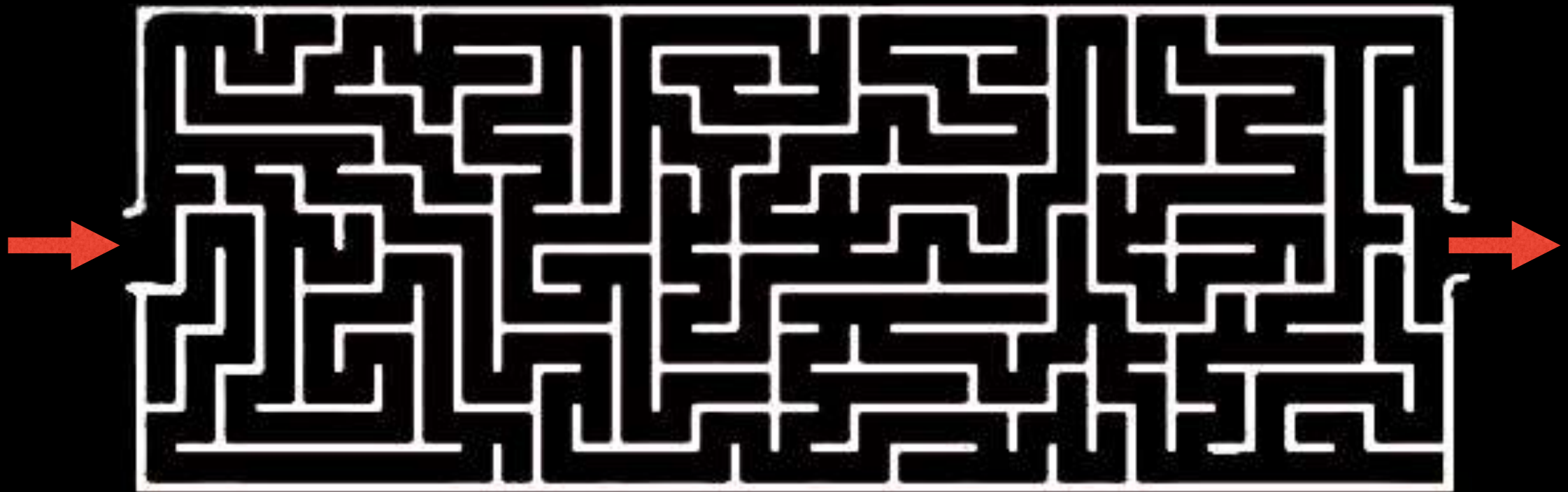
# Motivation

Many problems in graph theory can be represented using a grid. Grids are a form of **implicit graph** because we can determine a node's neighbours based on our location within the grid.

# Motivation

Many problems in graph theory can be represented using a grid. Grids are a form of **implicit graph** because we can determine a node's neighbours based on our location within the grid.

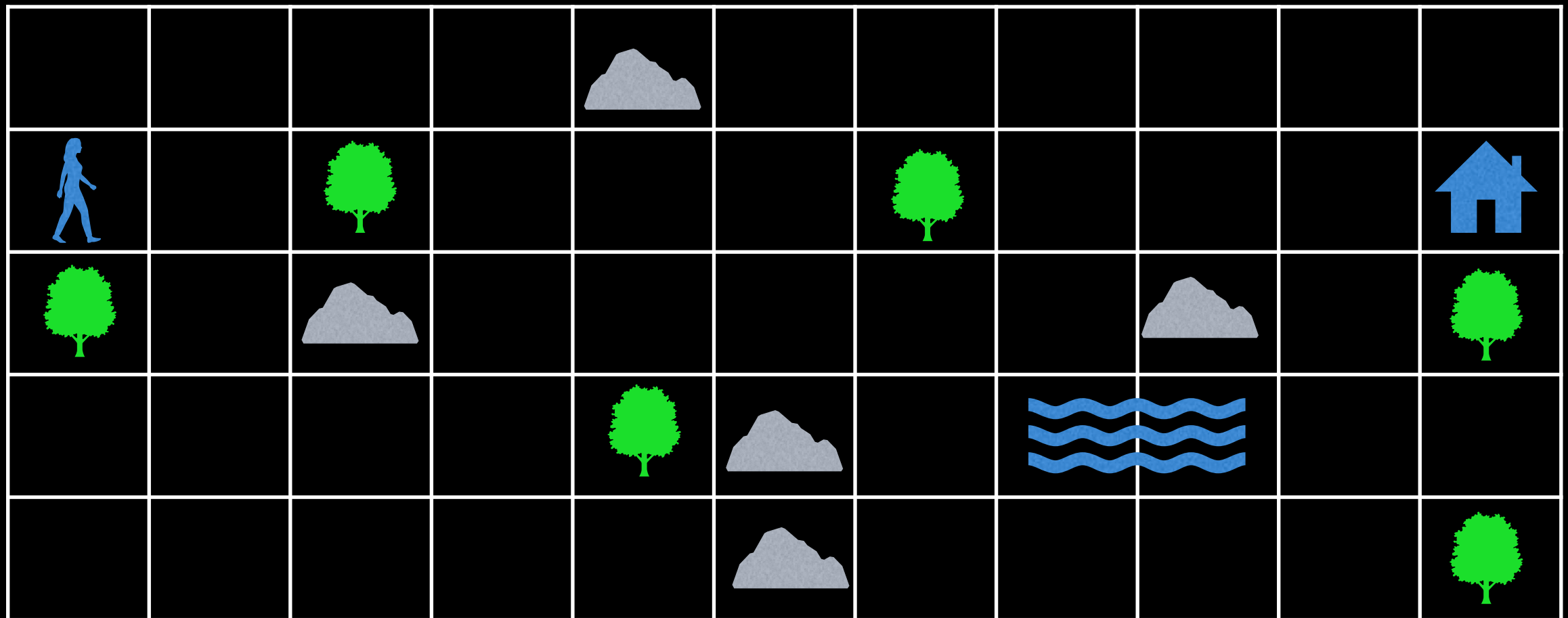
A type of problem that involves finding a path through a grid is solving a maze:



# Motivation

Many problems in graph theory can be represented using a grid. Grids are a form of **implicit graph** because we can determine a node's neighbours based on our location within the grid.

Another example could be routing through obstacles (trees, rivers, rocks, etc) to get to a location:

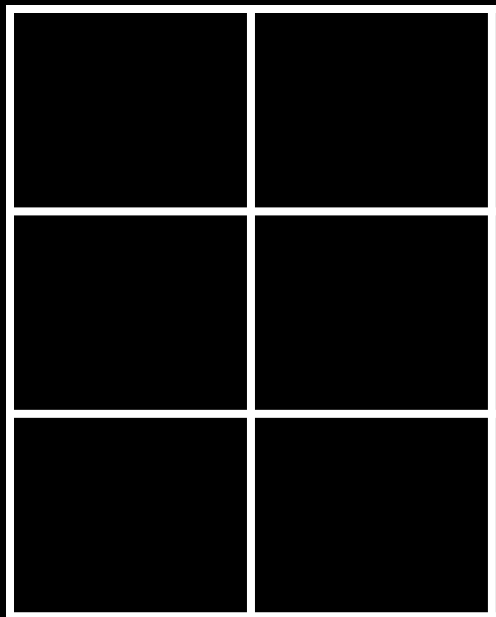




# Graph Theory on Grids

A common approach to solving graph theory problems on grids is to first convert the grid to a familiar format such as an **adjacency list/matrix**.

Empty Grid



**IMPORTANT:** Assume the grid is unweighted and cells connect left, right, up and down.

# Graph Theory on Grids

A common approach to solving graph theory problems on grids is to first convert the grid to a familiar format such as an **adjacency list/matrix**.

Empty Grid

0	1
2	3
4	5

First label the cells in the grid with numbers  $[0, n)$  where  $n = \text{\#rows} \times \text{\#columns}$

**IMPORTANT:** Assume the grid is unweighted and cells connect left, right, up and down.

# Graph Theory on Grids

A common approach to solving graph theory problems on grids is to first convert the grid to a familiar format such as an **adjacency list/matrix**.

Adjacency List:

Adjacency Matrix:

0 1 2 3 4 5

0

1

2

3

4

5


Empty Grid:

0	1
2	3
4	5

**IMPORTANT:** Assume the grid is unweighted and cells connect left, right, up and down.

# Graph Theory on Grids

A common approach to solving graph theory problems on grids is to first convert the grid to a familiar format such as an **adjacency list/matrix**.

Empty Grid:

0	→	1
↓		
2		3
4		5

Adjacency List:

0 → [1, 2]

Adjacency Matrix:

	0	1	2	3	4	5
0	0	1	1	0	0	0
1						
2						
3						
4						
5						

**IMPORTANT:** Assume the grid is unweighted and cells connect left, right, up and down.

# Graph Theory on Grids

A common approach to solving graph theory problems on grids is to first convert the grid to a familiar format such as an **adjacency list/matrix**.

Empty Grid:

0	1
2	3
4	5

Adjacency List:

0 → [1, 2]  
1 → [0, 3]

Adjacency Matrix:

	0	1	2	3	4	5
0	0	1	1	0	0	0
1	1	0	0	1	0	0
2						
3						
4						
5						

**IMPORTANT:** Assume the grid is unweighted and cells connect left, right, up and down.

# Graph Theory on Grids

A common approach to solving graph theory problems on grids is to first convert the grid to a familiar format such as an **adjacency list/matrix**.

Empty Grid:

0	1
2	3
4	5

Adjacency List:

0 → [1, 2]  
1 → [0, 3]  
2 → [0, 3, 4]

Adjacency Matrix:

	0	1	2	3	4	5
0	0	1	1	0	0	0
1	1	0	0	1	0	0
2	1	0	0	1	1	0
3						
4						
5						

**IMPORTANT:** Assume the grid is unweighted and cells connect left, right, up and down.



# Graph Theory on Grids

A common approach to solving graph theory problems on grids is to first convert the grid to a familiar format such as an **adjacency list/matrix**.

Empty Grid:

0	1
2	3
4	5

Adjacency List:

0 → [1, 2]  
1 → [0, 3]  
2 → [0, 3, 4]  
3 → [1, 2, 5]

Adjacency Matrix:

	0	1	2	3	4	5
0	0	1	1	0	0	0
1	1	0	0	1	0	0
2	1	0	0	1	1	0
3	0	1	1	0	0	1
4						
5						

**IMPORTANT:** Assume the grid is unweighted and cells connect left, right, up and down.

# Graph Theory on Grids

A common approach to solving graph theory problems on grids is to first convert the grid to a familiar format such as an **adjacency list/matrix**.

Empty Grid:

0	1
2	3
4	5

Adjacency List:

0 → [1, 2]  
1 → [0, 3]  
2 → [0, 3, 4]  
3 → [1, 2, 5]  
4 → [2, 5]

Adjacency Matrix:

	0	1	2	3	4	5
0	0	1	1	0	0	0
1	1	0	0	1	0	0
2	1	0	0	1	1	0
3	0	1	1	0	0	1
4	0	0	1	0	0	1
5						

**IMPORTANT:** Assume the grid is unweighted and cells connect left, right, up and down.

# Graph Theory on Grids

A common approach to solving graph theory problems on grids is to first convert the grid to a familiar format such as an **adjacency list/matrix**.

Empty Grid:

0	1
2	3
4	5

Adjacency List:

0 → [1, 2]  
1 → [0, 3]  
2 → [0, 3, 4]  
3 → [1, 2, 5]  
4 → [2, 5]  
5 → [3, 4]

Adjacency Matrix:

	0	1	2	3	4	5
0	0	1	1	0	0	0
1	1	0	0	1	0	0
2	1	0	0	1	1	0
3	0	1	1	0	0	1
4	0	0	1	0	0	1
5	0	0	0	1	1	0

**IMPORTANT:** Assume the grid is unweighted and cells connect left, right, up and down.

# Graph Theory on Grids

A common approach to solving graph theory problems on grids is to first convert the grid to a familiar format such as an **adjacency list/matrix**.

Empty Grid:

0	1
2	3
4	5

Adjacency List:

0 → [1, 2]  
1 → [0, 3]  
2 → [0, 3, 4]  
3 → [1, 2, 5]  
4 → [2, 5]  
5 → [3, 4]

Adjacency Matrix:

	0	1	2	3	4	5
0	0	1	1	0	0	0
1	1	0	0	1	0	0
2	1	0	0	1	1	0
3	0	1	1	0	0	1
4	0	0	1	0	0	1
5	0	0	0	1	1	0

**IMPORTANT:** Assume the grid is unweighted and cells connect left, right, up and down.

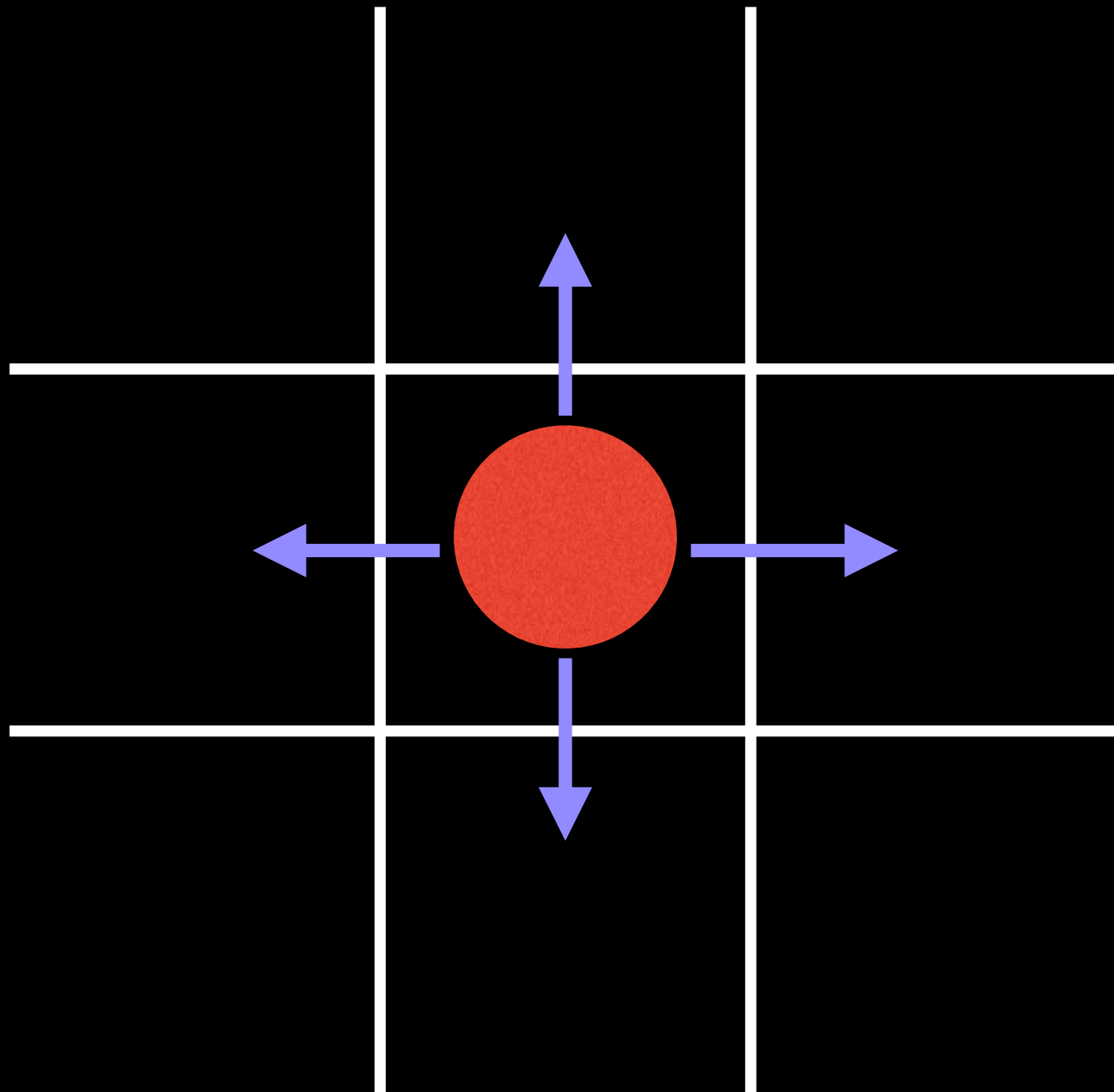
# Graph Theory on Grids

Once we have an adjacency list/matrix we can run whatever specialized graph algorithm to solve our problem such as: shortest path, connected components, etc...

However, **transformations between graph representations can usually be avoided** due to the structure of a grid.

# Direction Vectors

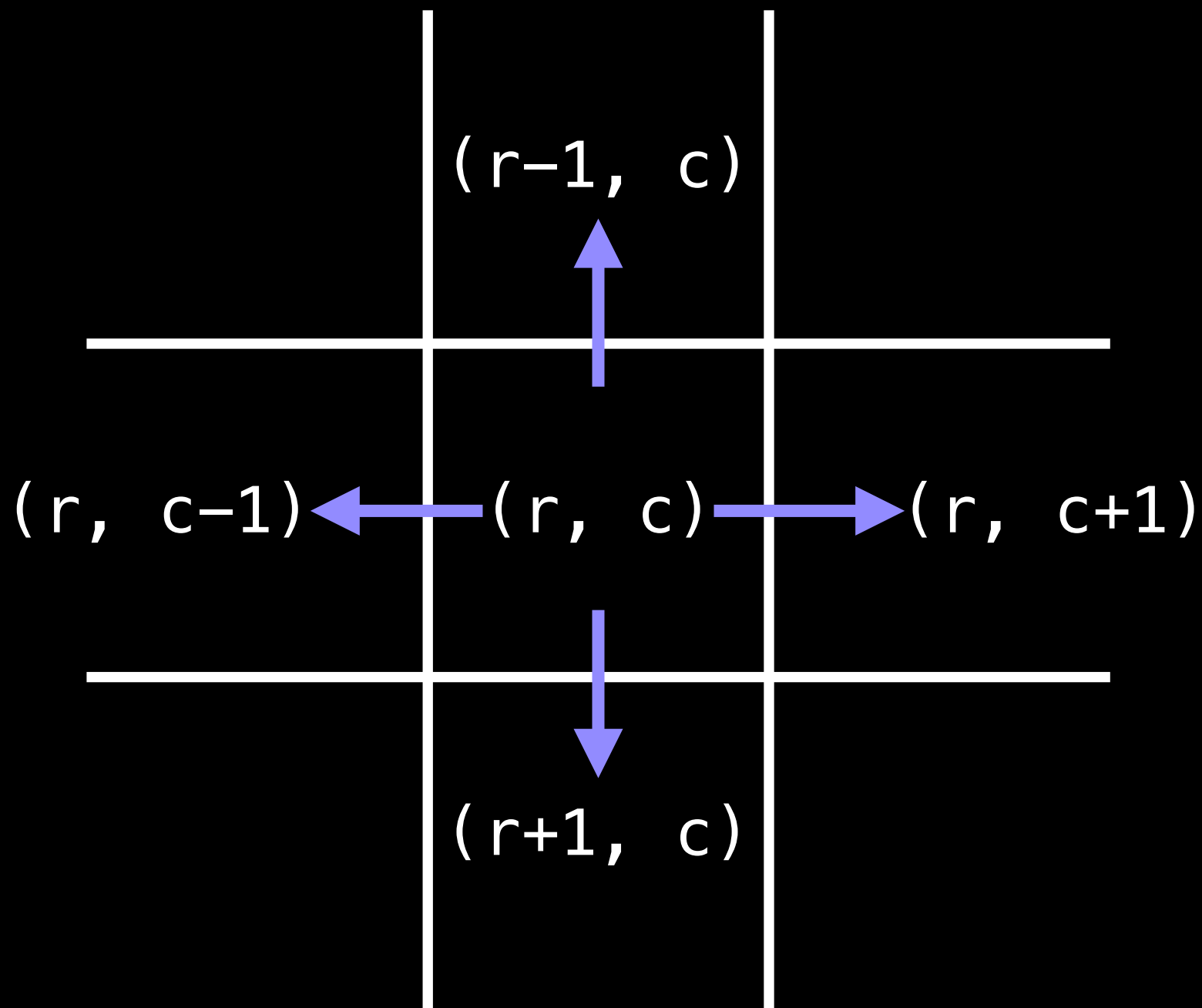
Due to the structure of a grid, if we are at the **red ball** in the middle we know we can move left, right, up and down to reach adjacent cells:





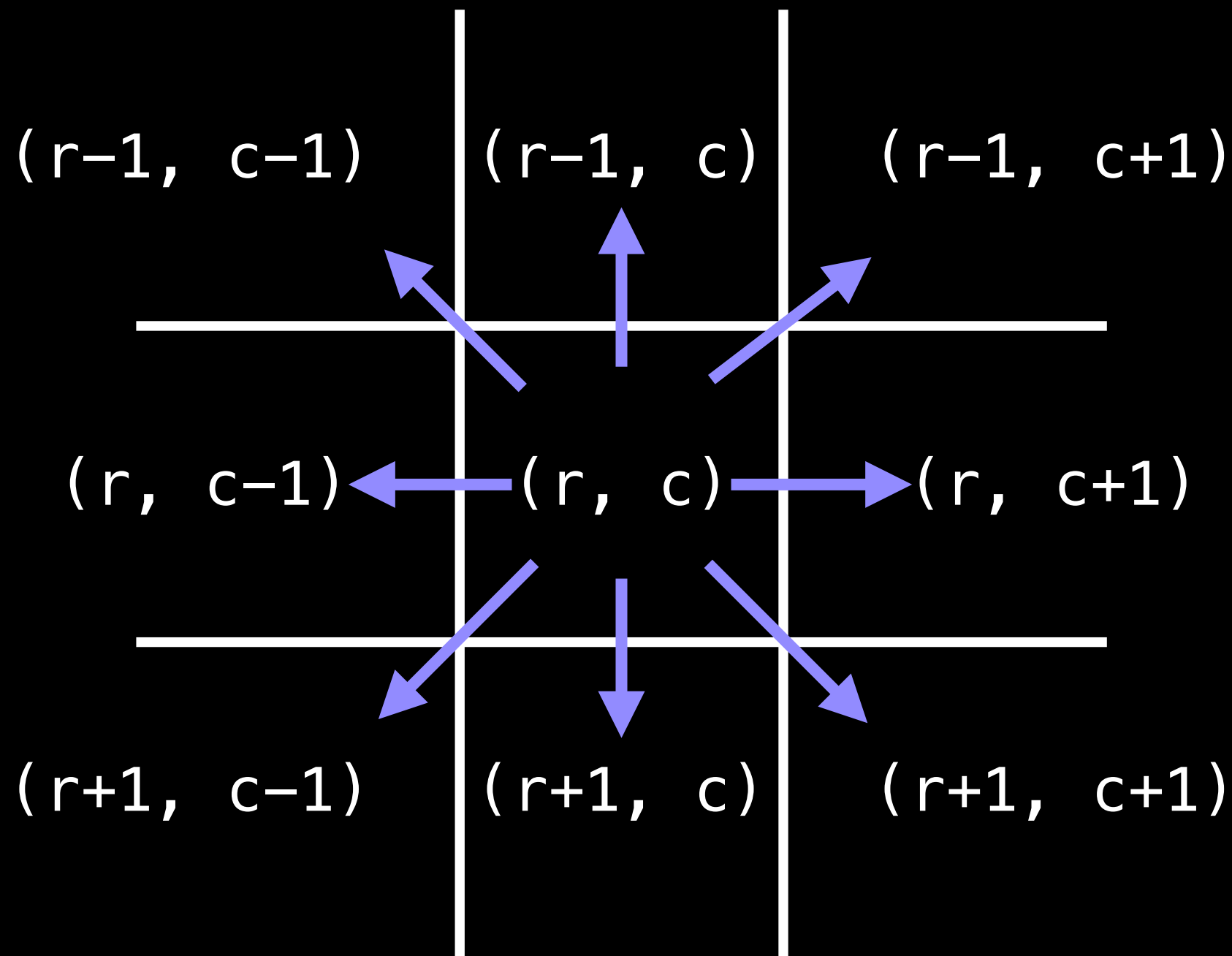
# Direction Vectors

Mathematically, if the **red ball** is at the row-column coordinate  $(r, c)$  we can add the row vectors  $[-1, 0]$ ,  $[1, 0]$ ,  $[0, 1]$ , and  $[0, -1]$  to reach adjacent cells.



# Direction Vectors

If the problem you are trying to solve allows moving diagonally then you can also include the row vectors:  $[-1, -1]$ ,  $[-1, 1]$ ,  $[1, 1]$ ,  $[1, -1]$



# Direction Vectors

This makes it very easy to access neighbouring cells from the current row-column position:

```
# Define the direction vectors for  
# north, south, east and west.
```

```
dr = [-1, +1, 0, 0]  
dc = [ 0, 0, +1, -1]
```

```
for(i = 0; i < 4; i++):
```

```
    rr = r + dr[i]
```

```
    cc = c + dc[i]
```

```
# Skip invalid cells. Assume R and
```

```
# C for the number of rows and columns
```

```
if rr < 0 or cc < 0: continue
```

```
if rr >= R or cc >= C: continue
```

```
 #(rr, cc) is a neighbouring cell of (r, c)
```

# Direction Vectors

This makes it very easy to access neighbouring cells from the current row-column position:

```
# Define the direction vectors for  
# north, south, east and west.  
dr = [-1, +1, 0, 0]  
dc = [ 0, 0, +1, -1]
```

```
for(i = 0; i < 4; i++):  
    rr = r + dr[i]  
    cc = c + dc[i]  
    # Skip invalid cells. Assume R and  
    # C for the number of rows and columns  
    if rr < 0 or cc < 0: continue  
    if rr >= R or cc >= C: continue  
    #(rr, cc) is a neighbouring cell of (r, c)
```

# Direction Vectors

This makes it very easy to access neighbouring cells from the current row-column position:

```
# Define the direction vectors for  
# north, south, east and west.
```

```
dr = [-1, +1, 0, 0]  
dc = [ 0, 0, +1, -1]
```

```
for(i = 0; i < 4; i++):
```

```
    rr = r + dr[i]
```

```
    cc = c + dc[i]
```

```
# Skip invalid cells. Assume R and  
# C for the number of rows and columns
```

```
if rr < 0 or cc < 0: continue
```

```
if rr >= R or cc >= C: continue
```

```
    #(rr, cc) is a neighbouring cell of (r, c)
```

# Direction Vectors

This makes it very easy to access neighbouring cells from the current row-column position:

```
# Define the direction vectors for  
# north, south, east and west.
```

```
dr = [-1, +1, 0, 0]  
dc = [ 0, 0, +1, -1]
```

```
for(i = 0; i < 4; i++):
```

```
    rr = r + dr[i]  
    cc = c + dc[i]
```

```
# Skip invalid cells. Assume R and  
# C for the number of rows and columns
```

```
if rr < 0 or cc < 0: continue
```

```
if rr >= R or cc >= C: continue
```

```
 #(rr, cc) is a neighbouring cell of (r, c)
```

# Direction Vectors

This makes it very easy to access neighbouring cells from the current row-column position:

```
# Define the direction vectors for  
# north, south, east and west.
```

```
dr = [-1, +1, 0, 0]  
dc = [ 0,  0, +1, -1]
```

```
for(i = 0; i < 4; i++):  
    rr = r + dr[i]  
    cc = c + dc[i]
```

```
# Skip invalid cells. Assume R and  
# C for the number of rows and columns
```

```
if rr < 0 or cc < 0: continue
```

```
if rr >= R or cc >= C: continue
```

```
##(rr, cc) is a neighbouring cell of (r, c)
```



# Direction Vectors

This makes it very easy to access neighbouring cells from the current row-column position:

```
# Define the direction vectors for  
# north, south, east and west.
```

```
dr = [-1, +1, 0, 0]  
dc = [ 0, 0, +1, -1]
```

```
for(i = 0; i < 4; i++):
```

```
    rr = r + dr[i]
```

```
    cc = c + dc[i]
```

```
# Skip invalid cells. Assume R and
```

```
# C for the number of rows and columns
```

```
if rr < 0 or cc < 0: continue
```

```
if rr >= R or cc >= C: continue
```

```
#(rr, cc) is a neighbouring cell of (r, c)
```

# Direction Vectors

This makes it very easy to access neighbouring cells from the current row-column position:

```
# Define the direction vectors for  
# north, south, east and west.
```

```
dr = [-1, +1, 0, 0]  
dc = [ 0, 0, +1, -1]
```

```
for(i = 0; i < 4; i++):  
    rr = r + dr[i]  
    cc = c + dc[i]  
    # Skip invalid cells. Assume R and  
    # C for the number of rows and columns  
    if rr < 0 or cc < 0: continue  
    if rr >= R or cc >= C: continue  
    #(rr, cc) is a neighbouring cell of (r, c)
```

# Dungeon Problem Statement

You are trapped in a 2D dungeon and need to find the quickest way out! The dungeon is composed of unit cubes which may or may not be filled with rock. It takes one minute to move one unit north, south, east, west. You cannot move diagonally and the maze is surrounded by solid rock on all sides.

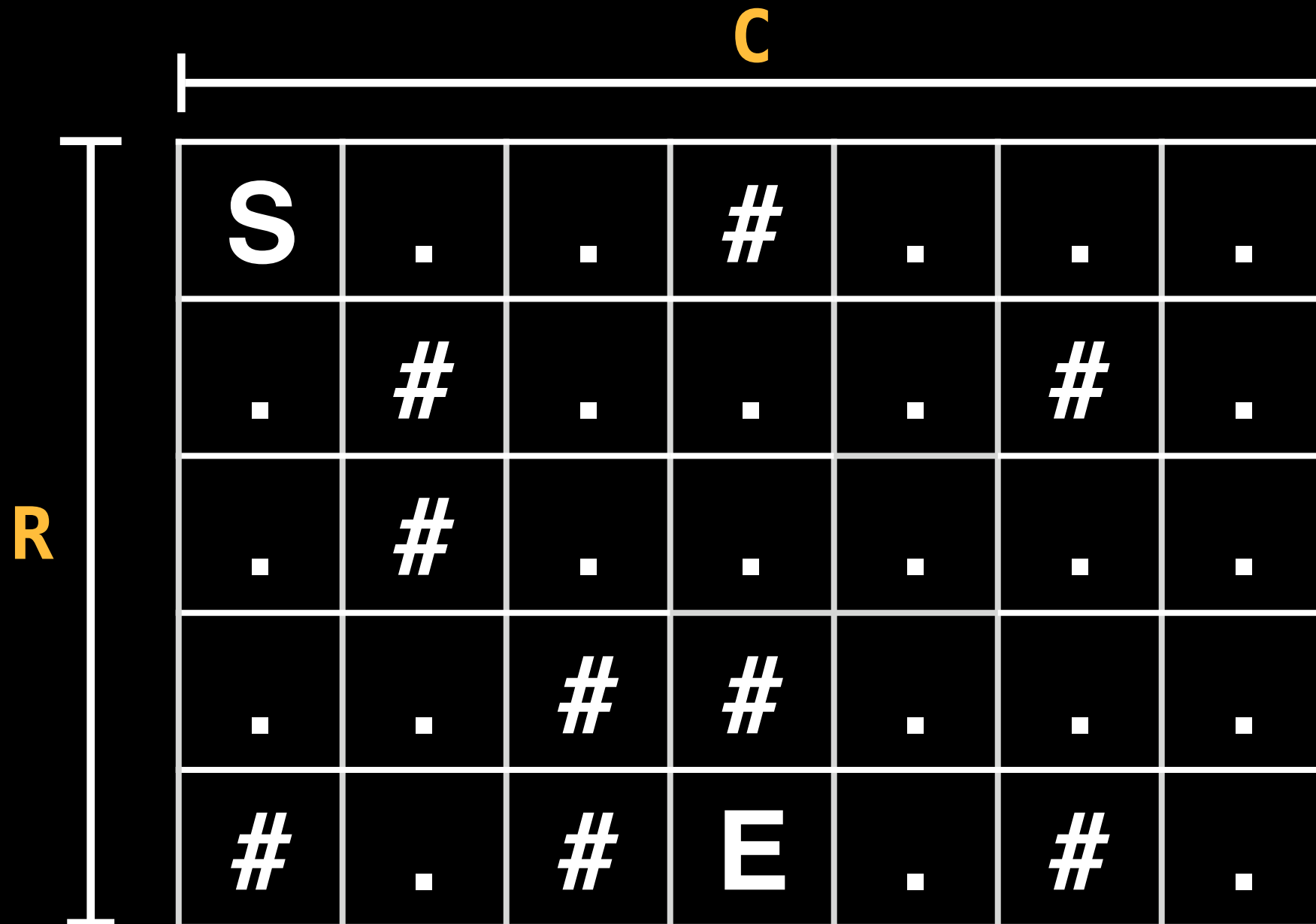
Is an escape possible?  
If yes, how long will  
it take?



This is an easier version of the “Dungeon Master” problem on Kattis: [open.kattis.com/problems/dungeon](https://open.kattis.com/problems/dungeon). Link in description.

# Dungeon Problem Statement

The dungeon has a size of **R** x **C** and you start at cell 'S' and there's an exit at cell 'E'. A cell full of rock is indicated by a '#' and empty cells are represented by a '.'

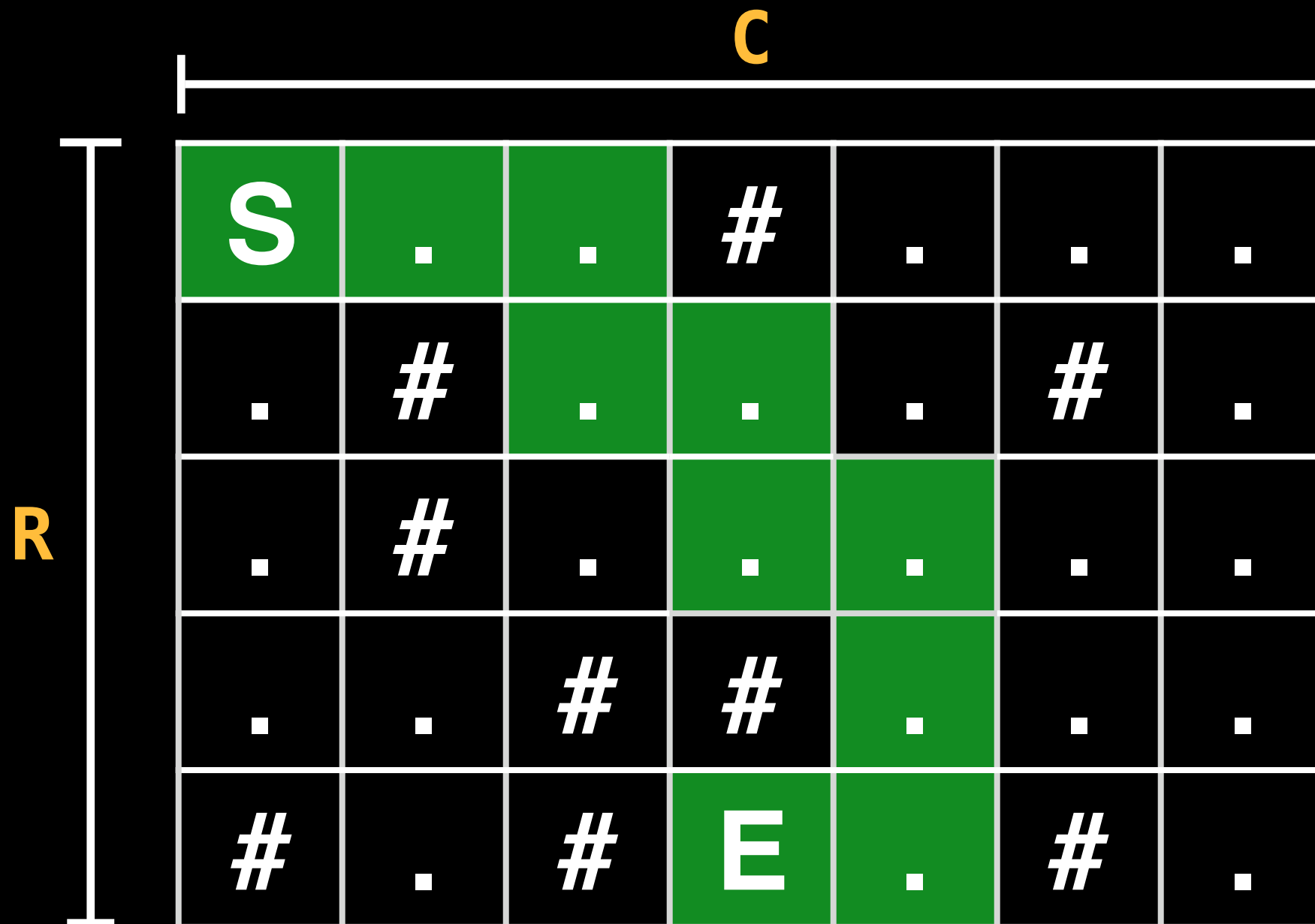


A 5x7 grid representing a dungeon. The grid is labeled with dimensions: **R** (rows) and **C** (columns). The grid contains the following characters:

S	.	.	#	.	.	.
.	#	.	.	.	#	.
.	#	.	.	.	.	.
.	.	#	#	.	.	.
#	.	#	E	.	#	.

# Dungeon Problem Statement

The dungeon has a size of **R** x **C** and you start at cell 'S' and there's an exit at cell 'E'. A cell full of rock is indicated by a '#' and empty cells are represented by a '.'.



	0	1	2	3	4	5	6
0	S	.	.	#	.	.	.
1	.	#	.	.	.	#	.
2	.	#	.	.	.	.	.
3	.	.	#	#	.	.	.
4	#	.	#	E	.	#	.

Start at the start node coordinate by adding (sr, sc) to the queue.

	0	1	2	3	4	5	6
0	(0,0)	.	.	#	.	.	.
1	.	#	.	.	.	#	.
2	.	#	.	.	.	.	.
3	.	.	#	#	.	.	.
4	#	.	#	E	.	#	.

(0, 0)



(1, 0)
(0, 1)
(0, 0)

	0	1	2	3	4	5	6
0	(0,0)	(0,1)	.	#	.	.	.
1	(1,0)	#	.	.	.	#	.
2	.	#	.	.	.	.	.
3	.	.	#	#	.	.	.
4	#	.	#	E	.	#	.

(2, 0)
(0, 2)
(1, 0)
(0, 1)
(0, 0)

	0	1	2	3	4	5	6
0	(0,0)	(0,1)	(0,2)	#	.	.	.
1	(1,0)	#	.	.	.	#	.
2	(2,0)	#	.	.	.	.	.
3	.	.	#	#	.	.	.
4	#	.	#	E	.	#	.

(3, 0)
(1, 2)
(2, 0)
(0, 2)
(1, 0)
(0, 1)
(0, 0)

	0	1	2	3	4	5	6
0	(0,0)	(0,1)	(0,2)	#	.	.	.
1	(1,0)	#	(1,2)	.	.	#	.
2	(2,0)	#	.	.	.	.	.
3	(3,0)	.	#	#	.	.	.
4	#	.	#	E	.	#	.

(3, 1)
(2, 2)
(1, 3)
(3, 0)
(1, 2)
(2, 0)
(0, 2)
(1, 0)
(0, 1)
(0, 0)

	0	1	2	3	4	5	6
0	(0,0)	(0,1)	(0,2)	#	.	.	.
1	(1,0)	#	(1,2)	(1,3)	.	#	.
2	(2,0)	#	(2,2)	.	.	.	.
3	(3,0)	(3,1)	#	#	.	.	.
4	#	.	#	E	.	#	.

(4, 1)
(2, 3)
(1, 4)
(3, 1)
(2, 2)
(1, 3)
(3, 0)
(1, 2)
(2, 0)
(0, 2)
(1, 0)
(0, 1)
(0, 0)

	0	1	2	3	4	5	6
0	(0,0)	(0,1)	(0,2)	#	.	.	.
1	(1,0)	#	(1,2)	(1,3)	(1,4)	#	.
2	(2,0)	#	(2,2)	(2,3)	.	.	.
3	(3,0)	(3,1)	#	#	.	.	.
4	#	(4,1)	#	E	.	#	.

(0, 4)
(2, 4)
(4, 1)
(2, 3)
(1, 4)
(3, 1)
(2, 2)
(1, 3)
(3, 0)
(1, 2)
(2, 0)
(0, 2)
(1, 0)
(0, 1)
(0, 0)

	0	1	2	3	4	5	6
0	(0,0)	(0,1)	(0,2)	#	(0,4)	.	.
1	(1,0)	#	(1,2)	(1,3)	(1,4)	#	.
2	(2,0)	#	(2,2)	(2,3)	(2,4)	.	.
3	(3,0)	(3,1)	#	#	.	.	.
4	#	(4,1)	#	E	.	#	.

(0, 5)
(3, 4)
2, 5)
(0, 4)
(2, 4)
(4, 1)
(2, 3)
(1, 4)
(3, 1)
(2, 2)
(1, 3)
(3, 0)
(1, 2)
(2, 0)
(0, 2)
(1, 0)
(0, 1)
(0, 0)

	0	1	2	3	4	5	6
0	(0,0)	(0,1)	(0,2)	#	(0,4)	(0,5)	.
1	(1,0)	#	(1,2)	(1,3)	(1,4)	#	.
2	(2,0)	#	(2,2)	(2,3)	(2,4)	(2,5)	.
3	(3,0)	(3,1)	#	#	(3,4)	.	.
4	#	(4,1)	#	E	.	#	.



(0, 5)
(3, 4)
2, 5)
(0, 4)
(2, 4)
(4, 1)
(2, 3)
(1, 4)
(3, 1)
(2, 2)
(1, 3)
(3, 0)
(1, 2)
(2, 0)
(0, 2)
(1, 0)
(0, 1)
(0, 0)

(0, 6)
(4, 4)
(3, 5)
(2, 6)

	0	1	2	3	4	5	6
0	(0,0)	(0,1)	(0,2)	#	(0,4)	(0,5)	(0,6)
1	(1,0)	#	(1,2)	(1,3)	(1,4)	#	.
2	(2,0)	#	(2,2)	(2,3)	(2,4)	(2,5)	(2,6)
3	(3,0)	(3,1)	#	#	(3,4)	(3,5)	.
4	#	(4,1)	#	E	(4,4)	#	.

(0, 5)
(3, 4)
2, 5)
(0, 4)
(2, 4)
(4, 1)
(2, 3)
(1, 4)
(3, 1)
(2, 2)
(1, 3)
(3, 0)
(1, 2)
(2, 0)
(0, 2)
(1, 0)
(0, 1)
(0, 0)

(4, 3)
(1, 6)
(3,6)
(0, 6)
(4, 4)
(3, 5)
(2, 6)

	0	1	2	3	4	5	6
0	(0,0)	(0,1)	(0,2)	#	(0,4)	(0,5)	(0,6)
1	(1,0)	#	(1,2)	(1,3)	(1,4)	#	(1,6)
2	(2,0)	#	(2,2)	(2,3)	(2,4)	(2,5)	(2,6)
3	(3,0)	(3,1)	#	#	(3,4)	(3,5)	(3,6)
4	#	(4,1)	#	(4,3)	(4,4)	#	.

We have reached the end, and if we had a 2D prev matrix we could regenerate the path by retracing our steps.

(0, 5)
(3, 4)
2, 5)
(0, 4)
(2, 4)
(4, 1)
(2, 3)
(1, 4)
(3, 1)
(2, 2)
(1, 3)
(3, 0)
(1, 2)
(2, 0)
(0, 2)
(1, 0)
(0, 1)
(0, 0)

(4, 3)
(1, 6)
(3,6)
(0, 6)
(4, 4)
(3, 5)
(2, 6)

	0	1	2	3	4	5	6
0	(0,0)	(0,1)	(0,2)	#	(0,4)	(0,5)	(0,6)
1	(1,0)	#	(1,2)	(1,3)	(1,4)	#	(1,6)
2	(2,0)	#	(2,2)	(2,3)	(2,4)	(2,5)	(2,6)
3	(3,0)	(3,1)	#	#	(3,4)	(3,5)	(3,6)
4	#	(4,1)	#	(4,3)	(4,4)	#	.

# Alternative State representation

So far we have been storing the next x-y position in the queue as an (x, y) pair. This works well but requires either an array or an object wrapper to store the coordinate values. In practice, this requires a lot of packing and unpacking of values to and from the queue.

Let's take a look at an alternative approach which also scales well in higher dimensions and (IMHO) requires less setup effort...

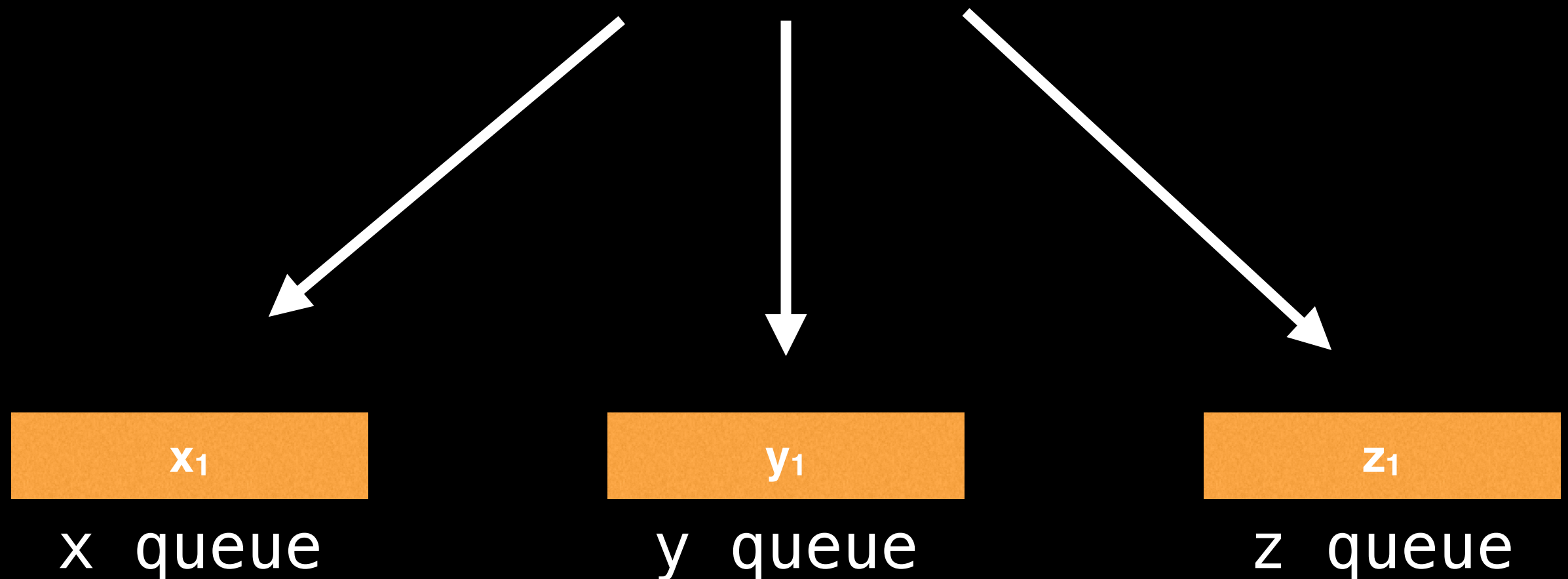
# Alternative State representation

An alternative approach is to **use one queue for each dimension**, so in a 3D grid you would have one queue for each of the x, y, and z dimensions.

# Alternative State representation

An alternative approach is to **use one queue for each dimension**, so in a 3D grid you would have one queue for each of the x, y, and z dimensions.

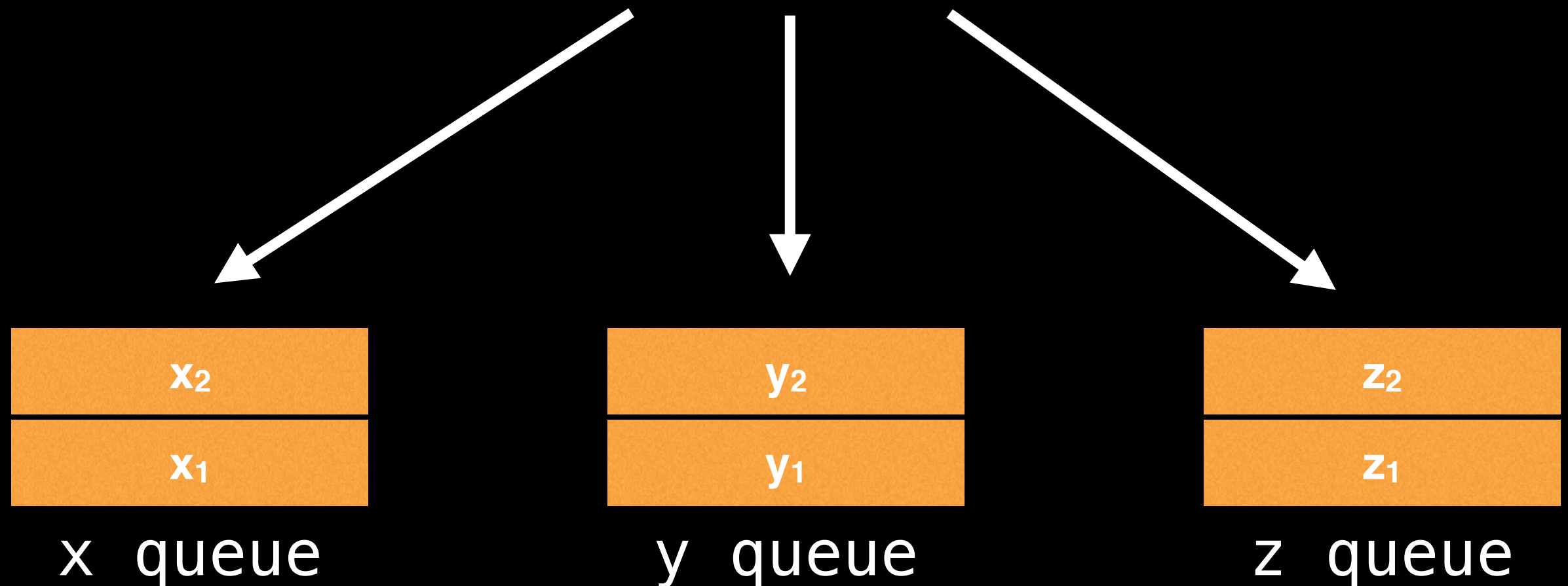
Enqueuing  $(x_1, y_1, z_1)$



# Alternative State representation

An alternative approach is to **use one queue for each dimension**, so in a 3D grid you would have one queue for each of the x, y, and z dimensions.

Enqueueing ( $x_2, y_2, z_2$ )

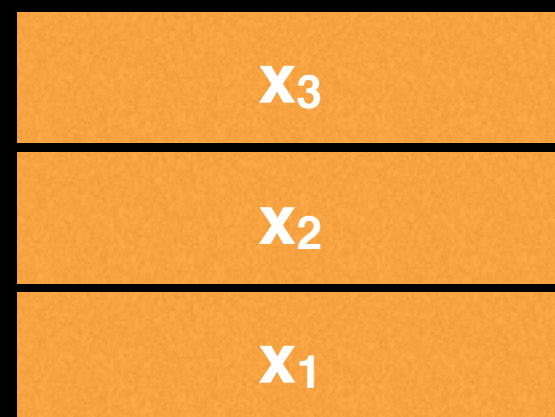




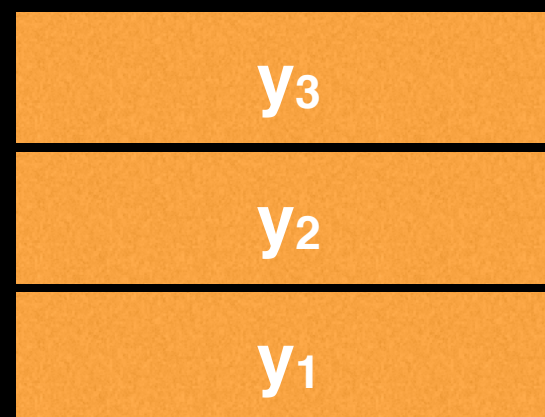
# Alternative State representation

An alternative approach is to **use one queue for each dimension**, so in a 3D grid you would have one queue for each of the  $x$ ,  $y$ , and  $z$  dimensions.

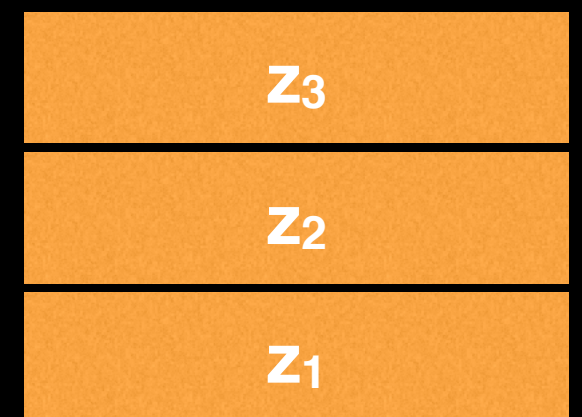
Enqueuing  $(x_3, y_3, z_3)$



$x$  queue



$y$  queue

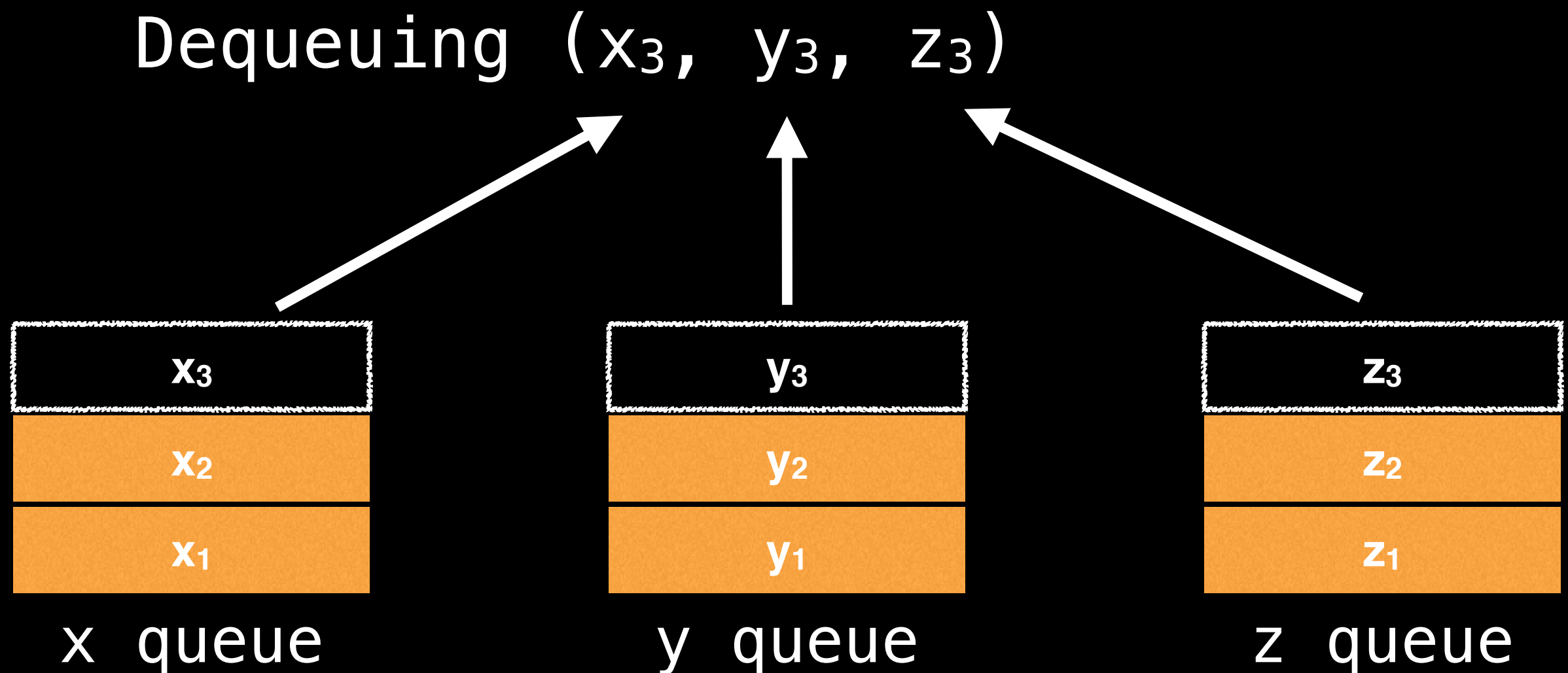


$z$  queue



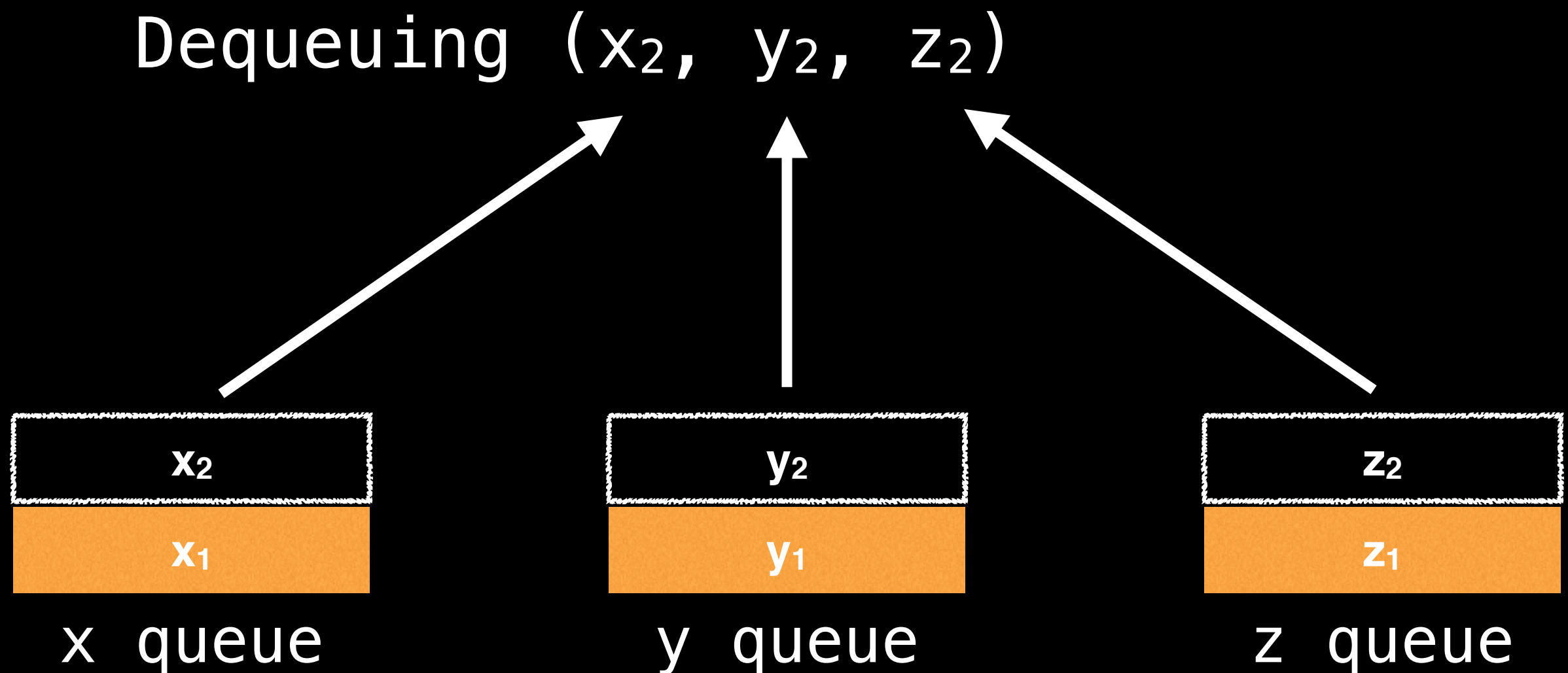
# Alternative State representation

An alternative approach is to **use one queue for each dimension**, so in a 3D grid you would have one queue for each of the  $x$ ,  $y$ , and  $z$  dimensions.



# Alternative State representation

An alternative approach is to **use one queue for each dimension**, so in a 3D grid you would have one queue for each of the  $x$ ,  $y$ , and  $z$  dimensions.



# Alternative State representation

An alternative approach is to **use one queue for each dimension**, so in a 3D grid you would have one queue for each of the x, y, and z dimensions.



$x_1$

x queue

$y_1$

y queue

$z_1$

z queue

```
# Global/class scope variables
R, C = ... # R = number of rows, C = number of columns
m = ...   # Input character matrix of size R x C
sr, sc = ... # 'S' symbol row and column values
rq, cq = ... # Empty Row Queue (RQ) and Column Queue (CQ)

# Variables used to track the number of steps taken.
move_count = 0
nodes_left_in_layer = 1
nodes_in_next_layer = 0

# Variable used to track whether the 'E' character
# ever gets reached during the BFS.
reached_end = false

# R x C matrix of false values used to track whether
# the node at position (i, j) has been visited.
visited = ...

# North, south, east, west direction vectors.
dr = [-1, +1, 0, 0]
dc = [ 0,  0, +1, -1]
```

```
# Global/class scope variables
R, C = ... # R = number of rows, C = number of columns
m = ...   # Input character matrix of size R x C

sr, sc = ... # 'S' symbol row and column values
rq, cq = ... # Empty Row Queue (RQ) and Column Queue (CQ)

# Variables used to track the number of steps taken.
move_count = 0
nodes_left_in_layer = 1
nodes_in_next_layer = 0

# Variable used to track whether the 'E' character
# ever gets reached during the BFS.
reached_end = false

# R x C matrix of false values used to track whether
# the node at position (i, j) has been visited.
visited = ...

# North, south, east, west direction vectors.
dr = [-1, +1, 0, 0]
dc = [ 0,  0, +1, -1]
```

```
# Global/class scope variables
R, C = ... # R = number of rows, C = number of columns
m = ...   # Input character matrix of size R x C
sr, sc = ... # 'S' symbol row and column values
rq, cq = ... # Empty Row Queue (RQ) and Column Queue (CQ)

# Variables used to track the number of steps taken.
move_count = 0
nodes_left_in_layer = 1
nodes_in_next_layer = 0

# Variable used to track whether the 'E' character
# ever gets reached during the BFS.
reached_end = false

# R x C matrix of false values used to track whether
# the node at position (i, j) has been visited.
visited = ...

# North, south, east, west direction vectors.
dr = [-1, +1, 0, 0]
dc = [ 0,  0, +1, -1]
```

```
# Global/class scope variables
R, C = ... # R = number of rows, C = number of columns
m = ...    # Input character matrix of size R x C
sr, sc = ... # 'S' symbol row and column values
rq, cq = ... # Empty Row Queue (RQ) and Column Queue (CQ)
```

```
# Variables used to track the number of steps taken.
move_count = 0
nodes_left_in_layer = 1
nodes_in_next_layer = 0
```

```
# Variable used to track whether the 'E' character
# ever gets reached during the BFS.
reached_end = false
```

```
# R x C matrix of false values used to track whether
# the node at position (i, j) has been visited.
visited = ...
```

```
# North, south, east, west direction vectors.
dr = [-1, +1, 0, 0]
dc = [ 0, 0, +1, -1]
```

```
# Global/class scope variables
R, C = ... # R = number of rows, C = number of columns
m = ...    # Input character matrix of size R x C
sr, sc = ... # 'S' symbol row and column values
rq, cq = ... # Empty Row Queue (RQ) and Column Queue (CQ)
```

```
# Variables used to track the number of steps taken.
move_count = 0
nodes_left_in_layer = 1
nodes_in_next_layer = 0
```

```
# Variable used to track whether the 'E' character
# ever gets reached during the BFS.
reached_end = false
```

```
# R x C matrix of false values used to track whether
# the node at position (i, j) has been visited.
visited = ...
```

```
# North, south, east, west direction vectors.
dr = [-1, +1, 0, 0]
dc = [ 0, 0, +1, -1]
```



```
# Global/class scope variables
R, C = ... # R = number of rows, C = number of columns
m = ...   # Input character matrix of size R x C
sr, sc = ... # 'S' symbol row and column values
rq, cq = ... # Empty Row Queue (RQ) and Column Queue (CQ)
```

```
# Variables used to track the number of steps taken.
move_count = 0
nodes_left_in_layer = 1
nodes_in_next_layer = 0
```

```
# Variable used to track whether the 'E' character
# ever gets reached during the BFS.
reached_end = false
```

```
# R x C matrix of false values used to track whether
# the node at position (i, j) has been visited.
visited = ...
```

```
# North, south, east, west direction vectors.
dr = [-1, +1, 0, 0]
dc = [ 0, 0, +1, -1]
```

```
# Global/class scope variables
R, C = ... # R = number of rows, C = number of columns
m = ...    # Input character matrix of size R x C
sr, sc = ... # 'S' symbol row and column values
rq, cq = ... # Empty Row Queue (RQ) and Column Queue (CQ)
```

```
# Variables used to track the number of steps taken.
```

```
move_count = 0
nodes_left_in_layer = 1
nodes_in_next_layer = 0
```

```
# Variable used to track whether the 'E' character
# ever gets reached during the BFS.
```

```
reached_end = false
```

```
# R x C matrix of false values used to track whether
# the node at position (i, j) has been visited.
```

```
visited = ...
```

```
# North, south, east, west direction vectors.
```

```
dr = [-1, +1, 0, 0]
dc = [ 0,  0, +1, -1]
```

```
# Global/class scope variables
R, C = ... # R = number of rows, C = number of columns
m = ...    # Input character matrix of size R x C
sr, sc = ... # 'S' symbol row and column values
rq, cq = ... # Empty Row Queue (RQ) and Column Queue (CQ)
```

```
# Variables used to track the number of steps taken.
```

```
move_count = 0
nodes_left_in_layer = 1
nodes_in_next_layer = 0
```

```
# Variable used to track whether the 'E' character
# ever gets reached during the BFS.
```

```
reached_end = false
```

```
# R x C matrix of false values used to track whether
# the node at position (i, j) has been visited.
```

```
visited = ...
```

```
# North, south, east, west direction vectors.
```

```
dr = [-1, +1, 0, 0]
```

```
dc = [ 0, 0, +1, -1]
```

```
function solve():  
    rq.enqueue(sr)  
    cq.enqueue(sc)  
    visited[sr][sc] = true  
    while rq.size() > 0: # or cq.size() > 0  
        r = rq.dequeue()  
        c = cq.dequeue()  
        if m[r][c] == 'E':  
            reached_end = true  
            break  
        explore_neighbours(r, c)  
        nodes_left_in_layer--  
        if nodes_left_in_layer == 0:  
            nodes_left_in_layer = nodes_in_next_layer  
            nodes_in_next_layer = 0  
            move_count++  
    if reached_end:  
        return move_count  
    return -1
```

```
function solve():  
    rq.enqueue(sr)  
    cq.enqueue(sc)  
    visited[sr][sc] = true  
    while rq.size() > 0: # or cq.size() > 0  
        r = rq.dequeue()  
        c = cq.dequeue()  
        if m[r][c] == 'E':  
            reached_end = true  
            break  
        explore_neighbours(r, c)  
        nodes_left_in_layer--  
        if nodes_left_in_layer == 0:  
            nodes_left_in_layer = nodes_in_next_layer  
            nodes_in_next_layer = 0  
            move_count++  
    if reached_end:  
        return move_count  
    return -1
```

```
function solve():  
    rq.enqueue(sr)  
    cq.enqueue(sc)  
    visited[sr][sc] = true  
    while rq.size() > 0: # or cq.size() > 0  
        r = rq.dequeue()  
        c = cq.dequeue()  
        if m[r][c] == 'E':  
            reached_end = true  
            break  
        explore_neighbours(r, c)  
        nodes_left_in_layer--  
        if nodes_left_in_layer == 0:  
            nodes_left_in_layer = nodes_in_next_layer  
            nodes_in_next_layer = 0  
            move_count++  
    if reached_end:  
        return move_count  
    return -1
```

```
function solve():  
    rq.enqueue(sr)  
    cq.enqueue(sc)  
    visited[sr][sc] = true  
    while rq.size() > 0: # or cq.size() > 0  
        r = rq.dequeue()  
        c = cq.dequeue()  
        if m[r][c] == 'E':  
            reached_end = true  
            break  
        explore_neighbours(r, c)  
        nodes_left_in_layer--  
        if nodes_left_in_layer == 0:  
            nodes_left_in_layer = nodes_in_next_layer  
            nodes_in_next_layer = 0  
            move_count++  
    if reached_end:  
        return move_count  
    return -1
```

```
function solve():  
    rq.enqueue(sr)  
    cq.enqueue(sc)  
    visited[sr][sc] = true  
    while rq.size() > 0: # or cq.size() > 0  
        r = rq.dequeue()  
        c = cq.dequeue()  
        if m[r][c] == 'E':  
            reached_end = true  
            break  
        explore_neighbours(r, c)  
        nodes_left_in_layer--  
        if nodes_left_in_layer == 0:  
            nodes_left_in_layer = nodes_in_next_layer  
            nodes_in_next_layer = 0  
            move_count++  
    if reached_end:  
        return move_count  
    return -1
```



```
function solve():
    rq.enqueue(sr)
    cq.enqueue(sc)
    visited[sr][sc] = true
    while rq.size() > 0: # or cq.size() > 0
        r = rq.dequeue()
        c = cq.dequeue()
        if m[r][c] == 'E':
            reached_end = true
            break
        explore_neighbours(r, c)
        nodes_left_in_layer--
        if nodes_left_in_layer == 0:
            nodes_left_in_layer = nodes_in_next_layer
            nodes_in_next_layer = 0
            move_count++
    if reached_end:
        return move_count
    return -1
```

```
function solve():  
    rq.enqueue(sr)  
    cq.enqueue(sc)  
    visited[sr][sc] = true  
    while rq.size() > 0: # or cq.size() > 0  
        r = rq.dequeue()  
        c = cq.dequeue()  
        if m[r][c] == 'E':  
            reached_end = true  
            break  
        explore_neighbours(r, c)  
        nodes_left_in_layer--  
        if nodes_left_in_layer == 0:  
            nodes_left_in_layer = nodes_in_next_layer  
            nodes_in_next_layer = 0  
            move_count++  
    if reached_end:  
        return move_count  
    return -1
```

```
function explore_neighbours(r, c):  
    for(i = 0; i < 4; i++):  
        rr = r + dr[i]  
        cc = c + dc[i]  
  
        # Skip out of bounds locations  
        if rr < 0 or cc < 0: continue  
        if rr >= R or cc >= C: continue  
  
        # Skip visited locations or blocked cells  
        if visited[rr][cc]: continue  
        if m[rr][cc] == '#': continue  
  
        rq.enqueue(rr)  
        cq.enqueue(cc)  
        visited[rr][cc] = true  
        nodes_in_next_layer++
```

```
function explore_neighbours(r, c):
```

```
    for(i = 0; i < 4; i++):
```

```
        rr = r + dr[i]
```

```
        cc = c + dc[i]
```

```
    # Skip out of bounds locations
```

```
    if rr < 0 or cc < 0: continue
```

```
    if rr >= R or cc >= C: continue
```

```
    # Skip visited locations or blocked cells
```

```
    if visited[rr][cc]: continue
```

```
    if m[rr][cc] == '#': continue
```

```
    rq.enqueue(rr)
```

```
    cq.enqueue(cc)
```

```
    visited[rr][cc] = true
```

```
    nodes_in_next_layer++
```

```
function explore_neighbours(r, c):  
    for(i = 0; i < 4; i++):  
        rr = r + dr[i]  
        cc = c + dc[i]  
  
        # Skip out of bounds locations  
        if rr < 0 or cc < 0: continue  
        if rr >= R or cc >= C: continue  
  
        # Skip visited locations or blocked cells  
        if visited[rr][cc]: continue  
        if m[rr][cc] == '#': continue  
  
        rq.enqueue(rr)  
        cq.enqueue(cc)  
        visited[rr][cc] = true  
        nodes_in_next_layer++
```

```
function explore_neighbours(r, c):  
    for(i = 0; i < 4; i++):  
        rr = r + dr[i]  
        cc = c + dc[i]  
  
        # Skip out of bounds locations  
        if rr < 0 or cc < 0: continue  
        if rr >= R or cc >= C: continue  
  
        # Skip visited locations or blocked cells  
        if visited[rr][cc]: continue  
        if m[rr][cc] == '#': continue  
  
        rq.enqueue(rr)  
        cq.enqueue(cc)  
        visited[rr][cc] = true  
        nodes_in_next_layer++
```

```
function explore_neighbours(r, c):  
    for(i = 0; i < 4; i++):  
        rr = r + dr[i]  
        cc = c + dc[i]
```

```
# Skip out of bounds locations  
if rr < 0 or cc < 0: continue  
if rr >= R or cc >= C: continue
```

```
# Skip visited locations or blocked cells  
if visited[rr][cc]: continue  
if m[rr][cc] == '#': continue
```

```
rq.enqueue(rr)  
cq.enqueue(cc)  
visited[rr][cc] = true  
nodes_in_next_layer++
```

```
function explore_neighbours(r, c):  
    for(i = 0; i < 4; i++):  
        rr = r + dr[i]  
        cc = c + dc[i]  
  
        # Skip out of bounds locations  
        if rr < 0 or cc < 0: continue  
        if rr >= R or cc >= C: continue
```

```
# Skip visited locations or blocked cells  
if visited[rr][cc]: continue  
if m[rr][cc] == '#': continue
```

```
rq.enqueue(rr)  
cq.enqueue(cc)  
visited[rr][cc] = true  
nodes_in_next_layer++
```



```
function explore_neighbours(r, c):  
    for(i = 0; i < 4; i++):  
        rr = r + dr[i]  
        cc = c + dc[i]  
  
        # Skip out of bounds locations  
        if rr < 0 or cc < 0: continue  
        if rr >= R or cc >= C: continue  
  
        # Skip visited locations or blocked cells  
        if visited[rr][cc]: continue  
        if m[rr][cc] == '#': continue  
  
        rq.enqueue(rr)  
        cq.enqueue(cc)  
        visited[rr][cc] = true  
        nodes_in_next_layer++
```

```
function explore_neighbours(r, c):  
    for(i = 0; i < 4; i++):  
        rr = r + dr[i]  
        cc = c + dc[i]  
  
        # Skip out of bounds locations  
        if rr < 0 or cc < 0: continue  
        if rr >= R or cc >= C: continue  
  
        # Skip visited locations or blocked cells  
        if visited[rr][cc]: continue  
        if m[rr][cc] == '#': continue  
  
        rq.enqueue(rr)  
        cq.enqueue(cc)  
        visited[rr][cc] = true  
        nodes_in_next_layer++
```

```
function solve():
    rq.enqueue(sr)
    cq.enqueue(sc)
    visited[sr][sc] = true
    while rq.size() > 0: # or cq.size() > 0
        r = rq.dequeue()
        c = cq.dequeue()
        if m[r][c] == 'E':
            reached_end = true
            break
        explore_neighbours(r, c)
        nodes_left_in_layer--
        if nodes_left_in_layer == 0:
            nodes_left_in_layer = nodes_in_next_layer
            nodes_in_next_layer = 0
            move_count++
    if reached_end:
        return move_count
    return -1
```

```
function solve():
    rq.enqueue(sr)
    cq.enqueue(sc)
    visited[sr][sc] = true
    while rq.size() > 0: # or cq.size() > 0
        r = rq.dequeue()
        c = cq.dequeue()
        if m[r][c] == 'E':
            reached_end = true
            break
        explore_neighbours(r, c)
        nodes_left_in_layer--
        if nodes_left_in_layer == 0:
            nodes_left_in_layer = nodes_in_next_layer
            nodes_in_next_layer = 0
            move_count++
    if reached_end:
        return move_count
    return -1
```

```
function solve():  
    rq.enqueue(sr)  
    cq.enqueue(sc)  
    visited[sr][sc] = true  
    while rq.size() > 0: # or cq.size() > 0  
        r = rq.dequeue()  
        c = cq.dequeue()  
        if m[r][c] == 'E':  
            reached_end = true  
            break  
        explore_neighbours(r, c)  
        nodes_left_in_layer--  
        if nodes_left_in_layer == 0:  
            nodes_left_in_layer = nodes_in_next_layer  
            nodes_in_next_layer = 0  
            move_count++  
    if reached_end:  
        return move_count  
    return -1
```

# Summary

Representing a grid as an adjacency list and adjacency matrix.

Empty Grid

0	1
2	3
4	5

Adjacency List:

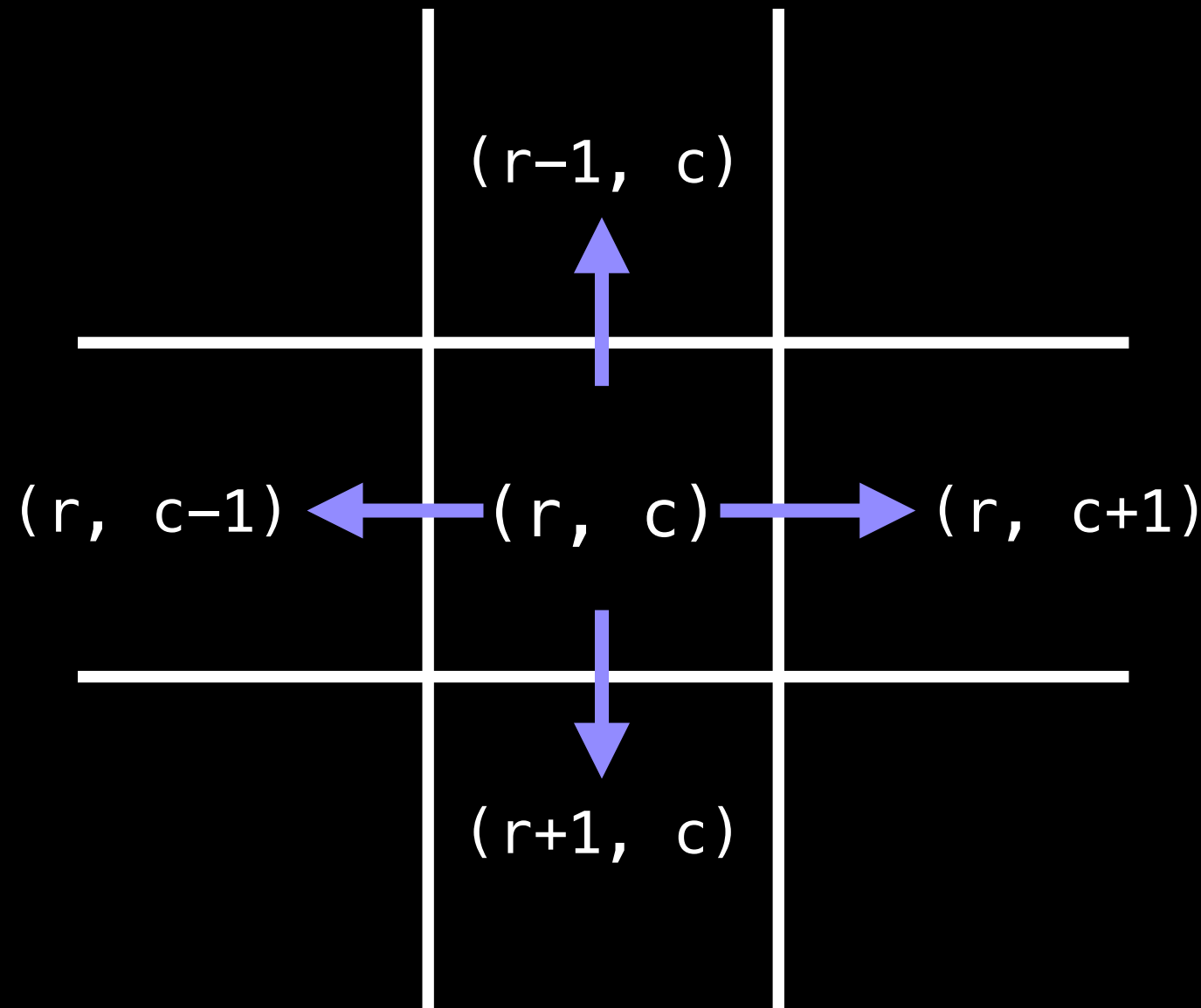
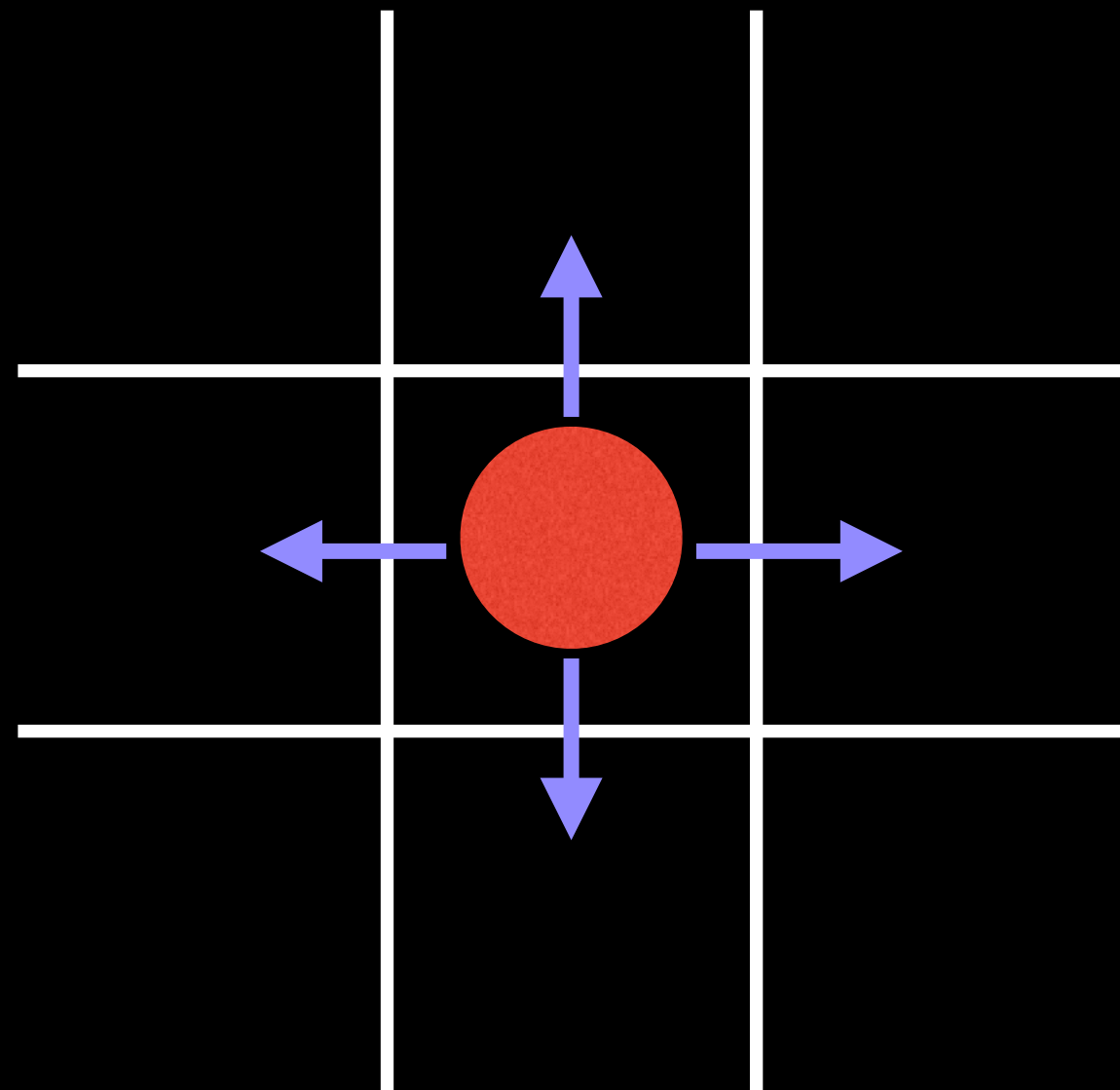
0 → [1, 2]  
1 → [0, 3]  
2 → [0, 3, 4]  
3 → [1, 2, 5]  
4 → [2, 5]  
5 → [3, 4]

Adjacency Matrix:

	0	1	2	3	4	5
0	0	1	1	0	0	0
1	1	0	0	1	0	0
2	1	0	0	1	1	0
3	0	1	1	0	0	1
4	0	0	1	0	0	1
5	0	0	0	1	1	0

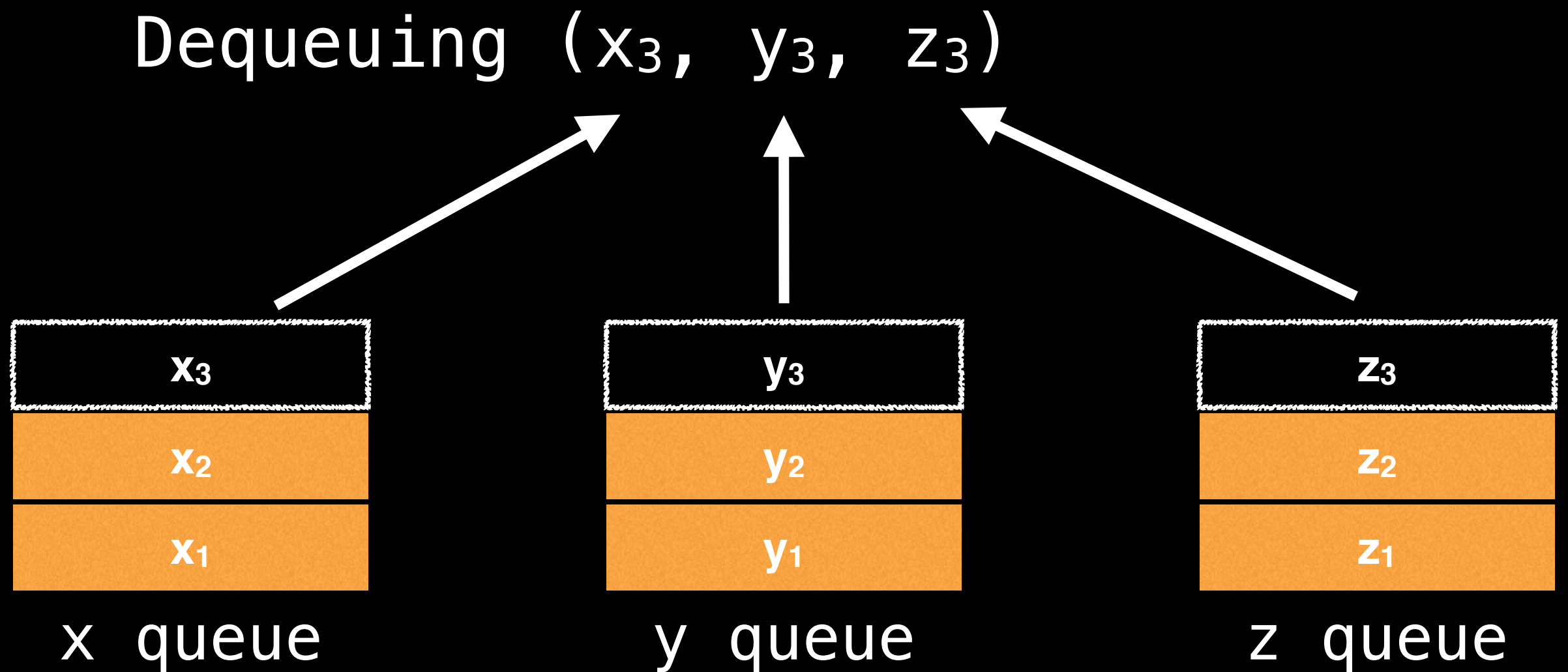
# Summary

Using direction vectors to visit neighbouring cells.



# Summary

Explored an alternative way to represent multi dimensional coordinates using multiple queues.





# Summary

How to use BFS on a grid to find the shortest path between two cells.

