Graph Theory: Breadth First Search

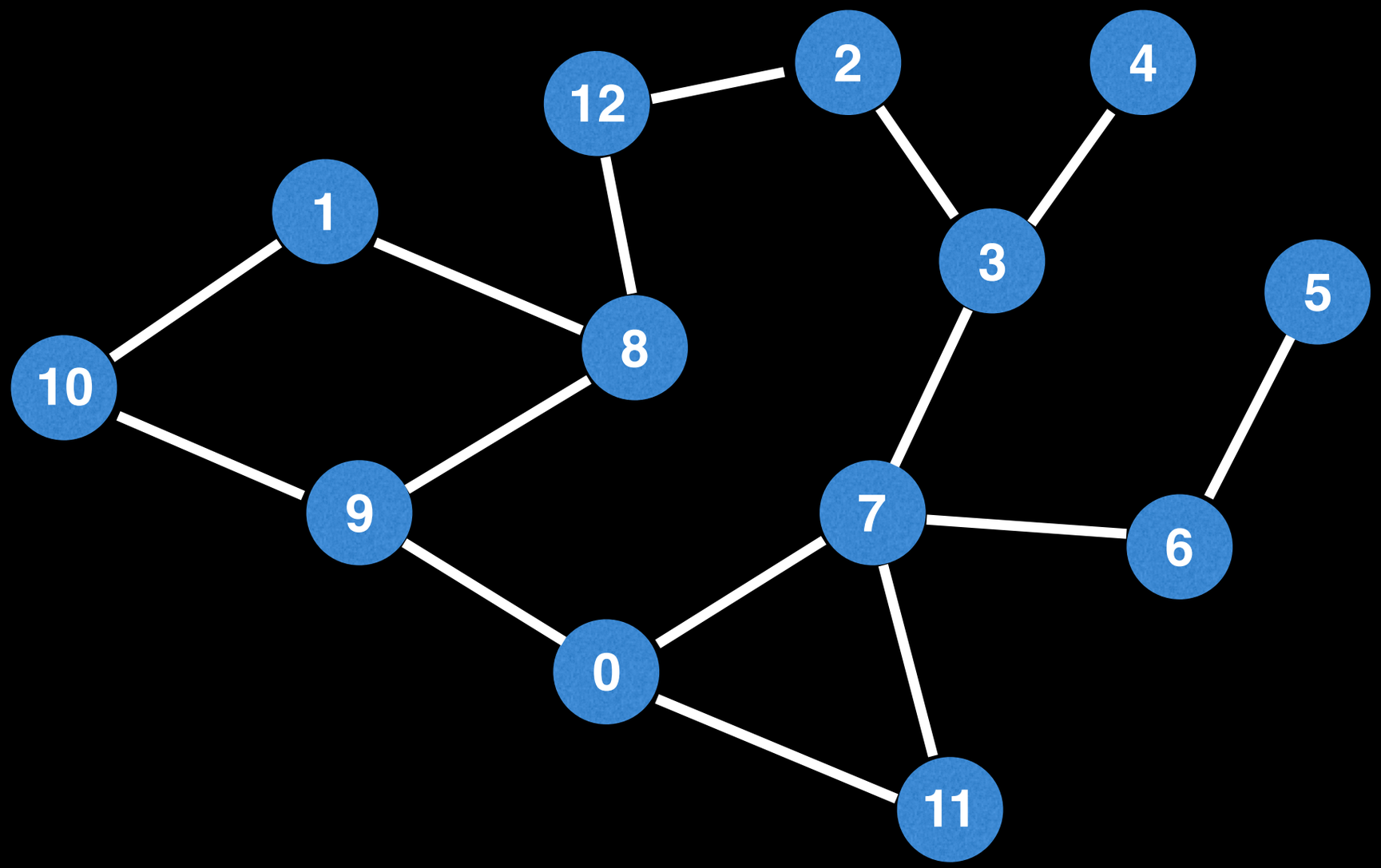
William Fiset

BFS overview

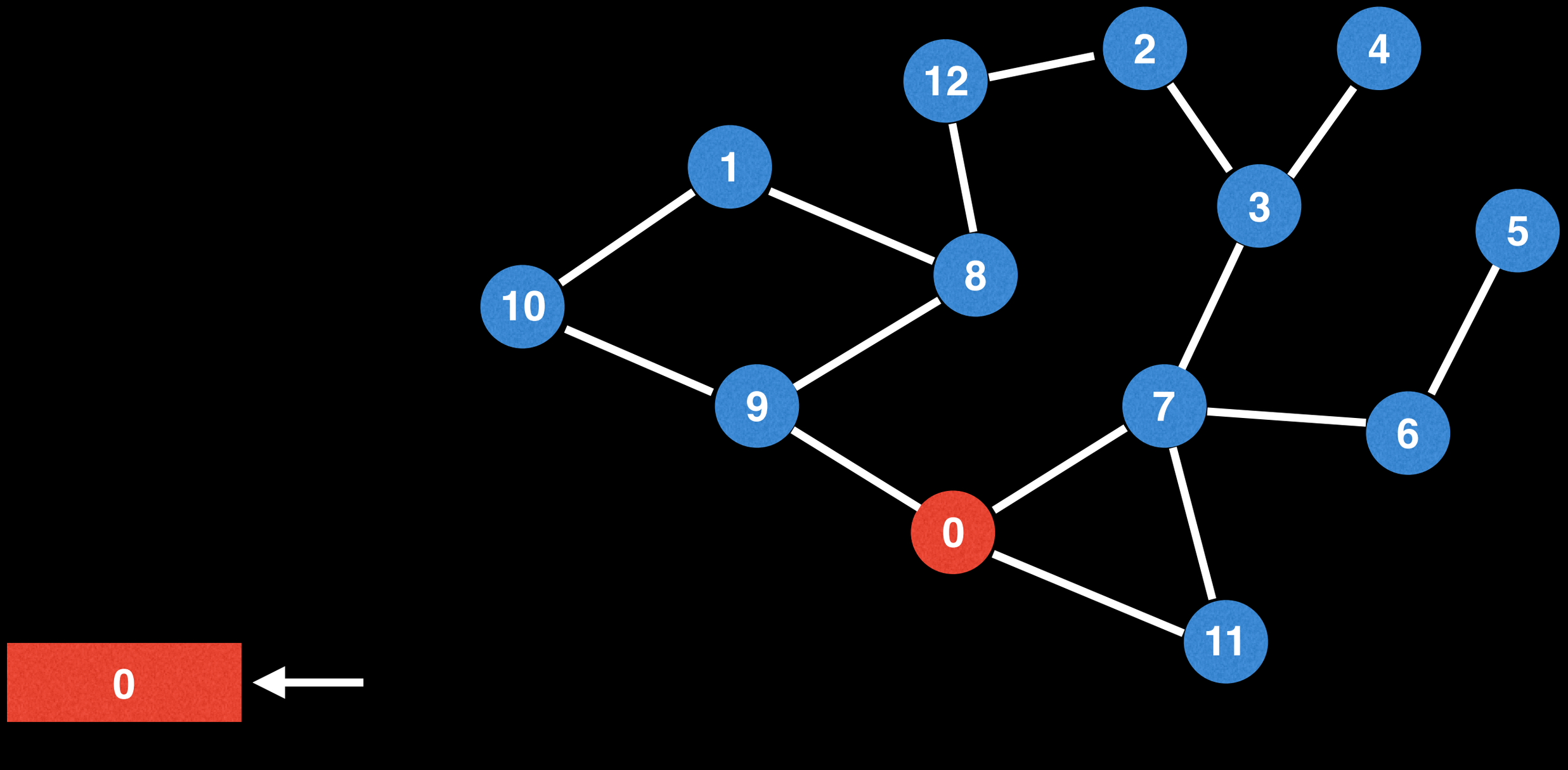
The **Breadth First Search (BFS)** is another fundamental search algorithm used to explore nodes and edges of a graph. It runs with a time complexity of **O(V+E)** and is often used as a building block in other algorithms.

The BFS algorithm is particularly useful for finding the **shortest path on unweighted graphs** and for testing connectivity.

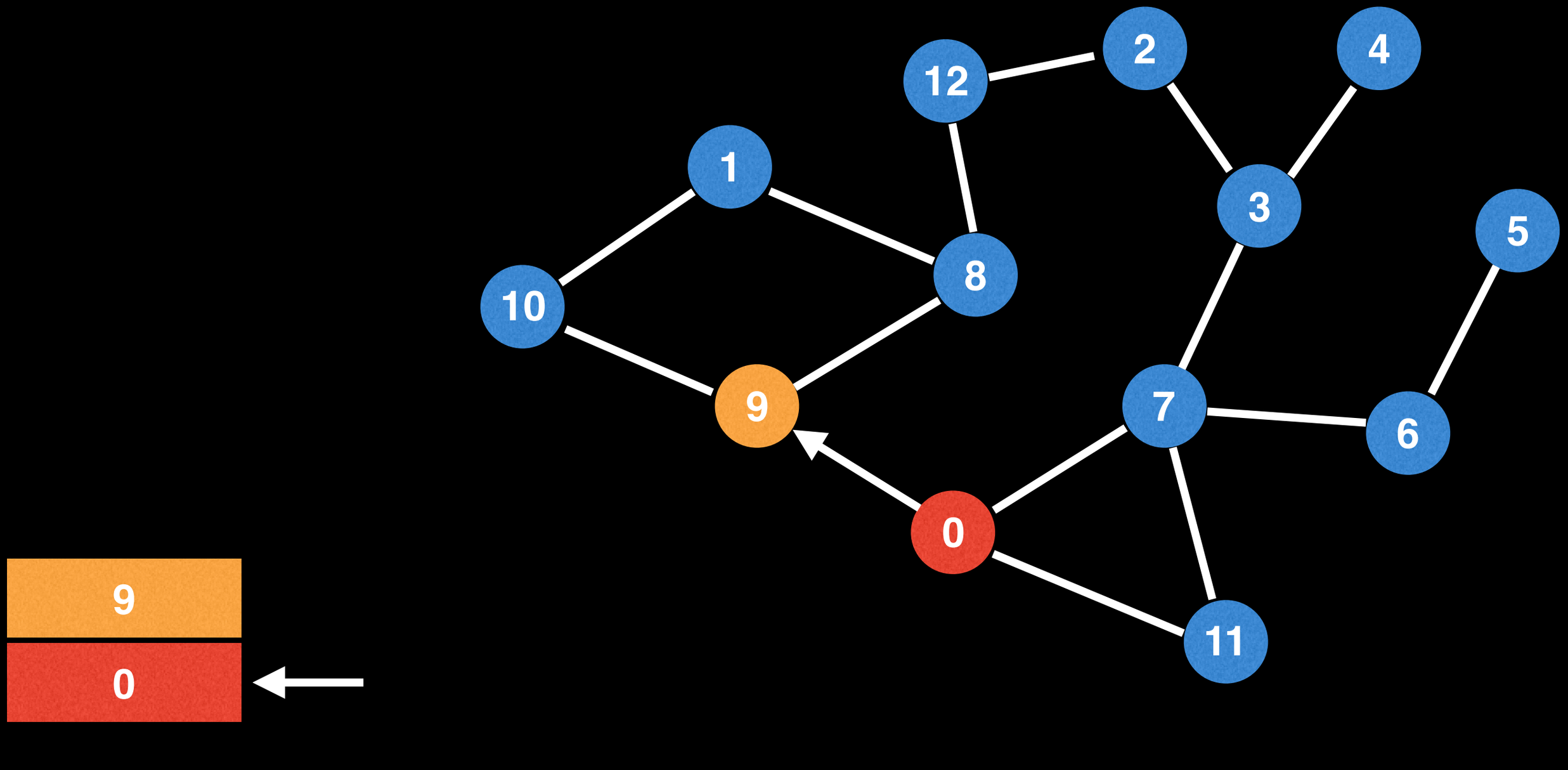
A BFS starts at some arbitrary node of a graph and explores the neighbour nodes first, before moving to the next level neighbours.



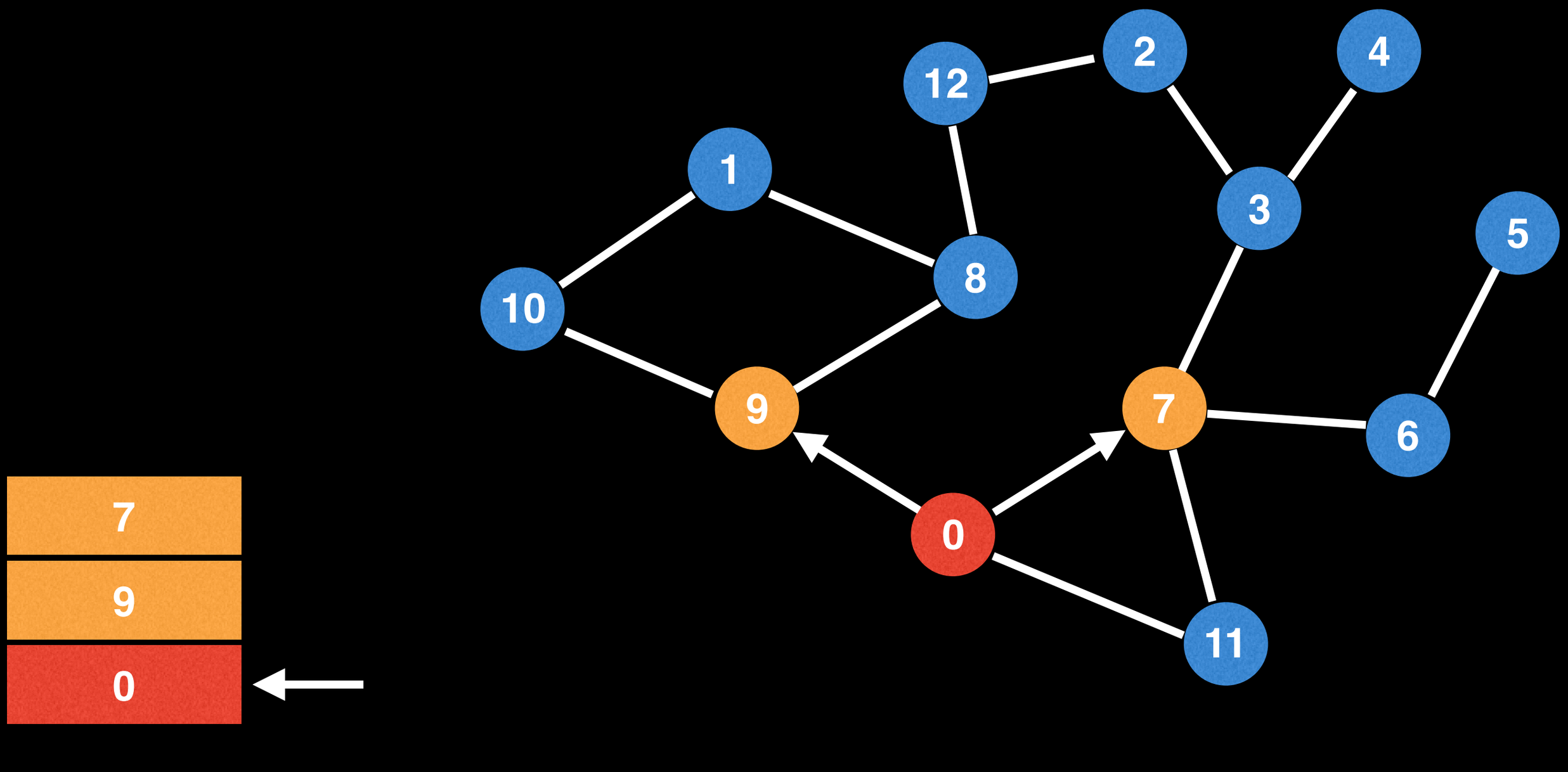
A BFS starts at some arbitrary node of a graph and explores the neighbour nodes first, before moving to the next level neighbours.



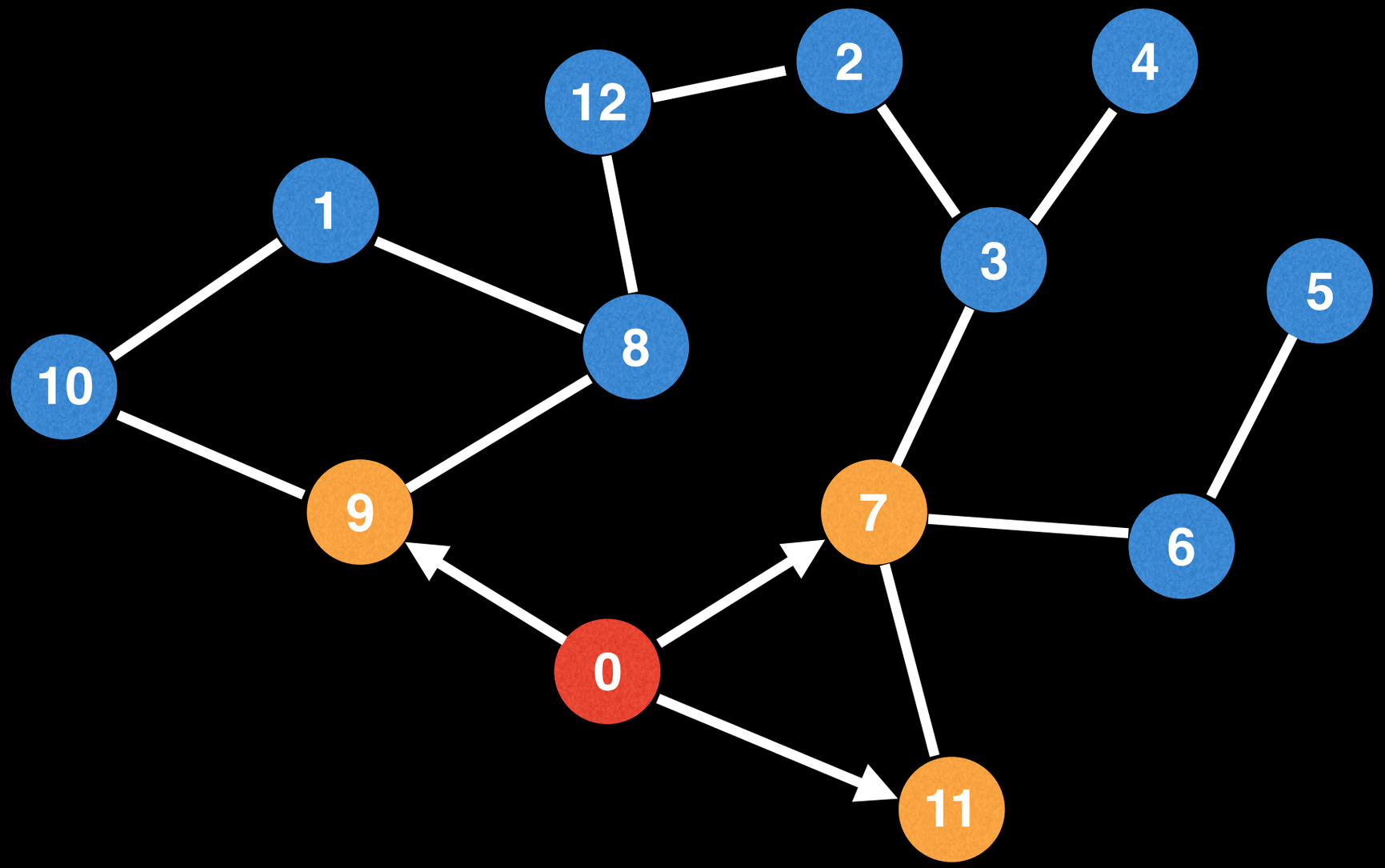
A BFS starts at some arbitrary node of a graph and explores the neighbour nodes first, before moving to the next level neighbours.

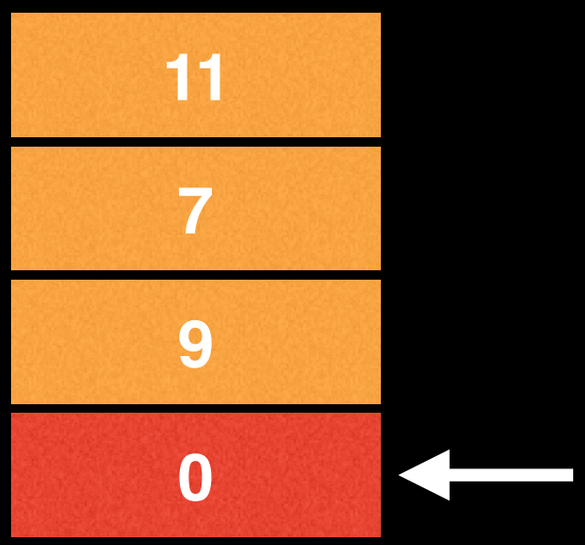


A BFS starts at some arbitrary node of a graph and explores the neighbour nodes first, before moving to the next level neighbours.

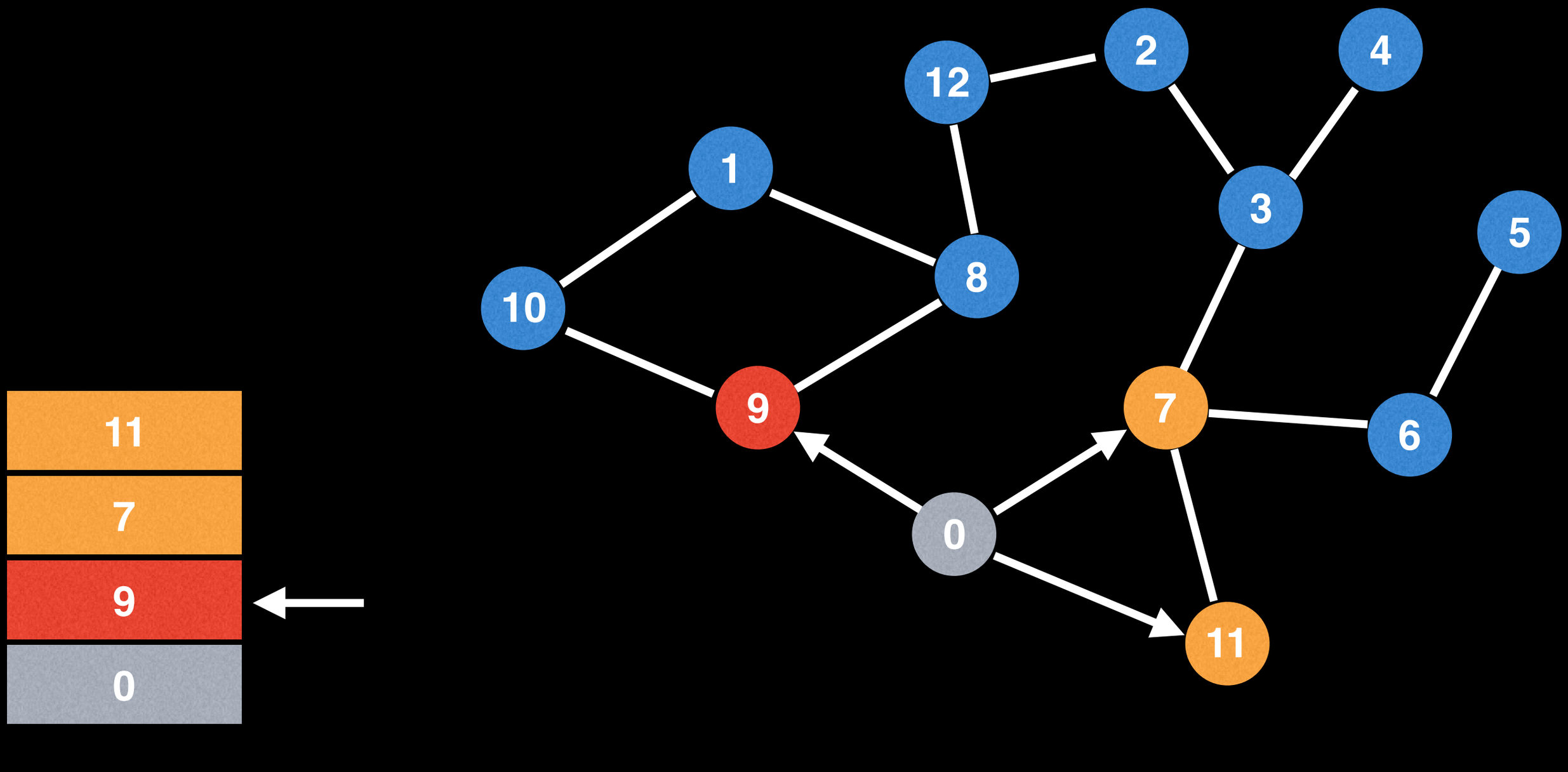


A BFS starts at some arbitrary node of a graph and explores the neighbour nodes first, before moving to the next level neighbours.

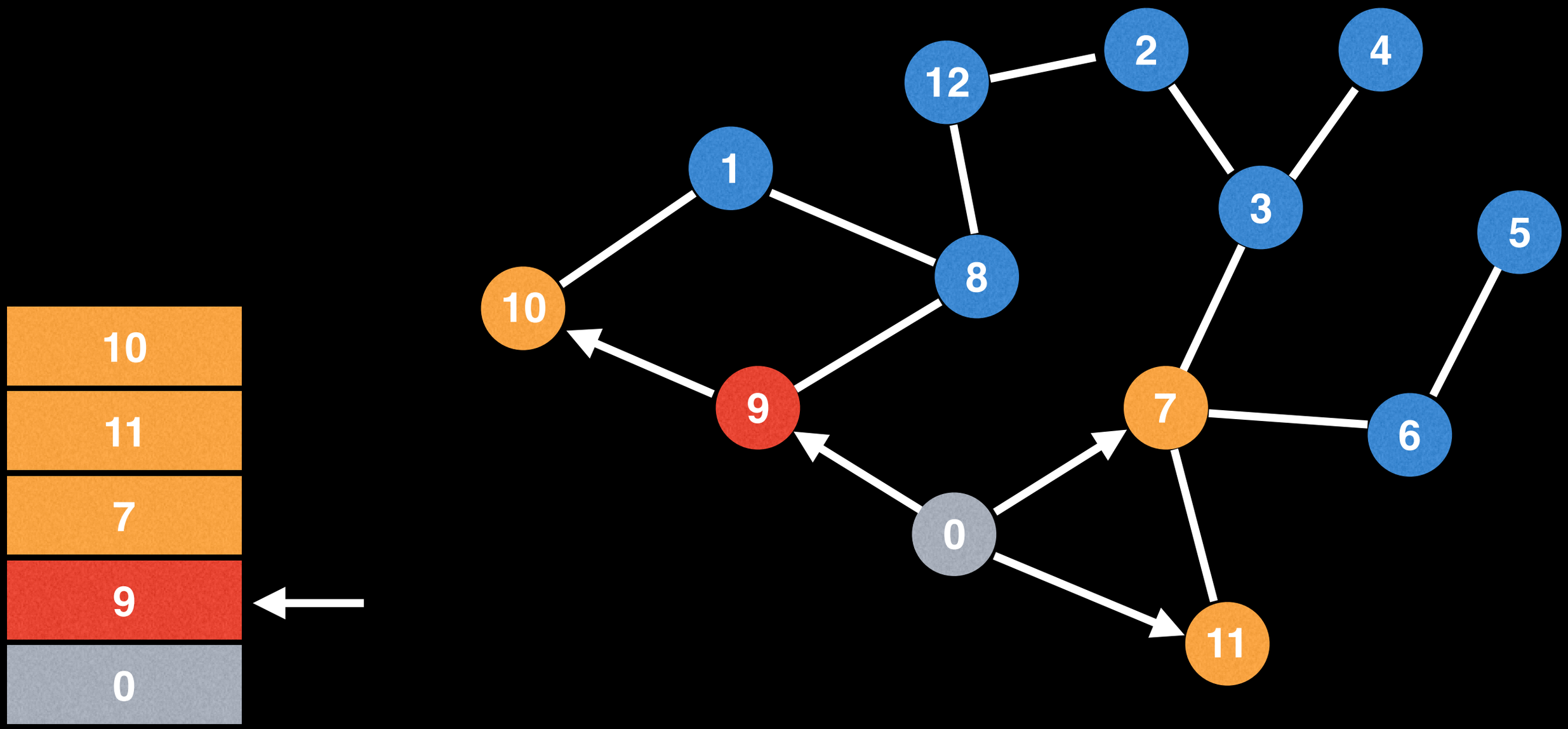




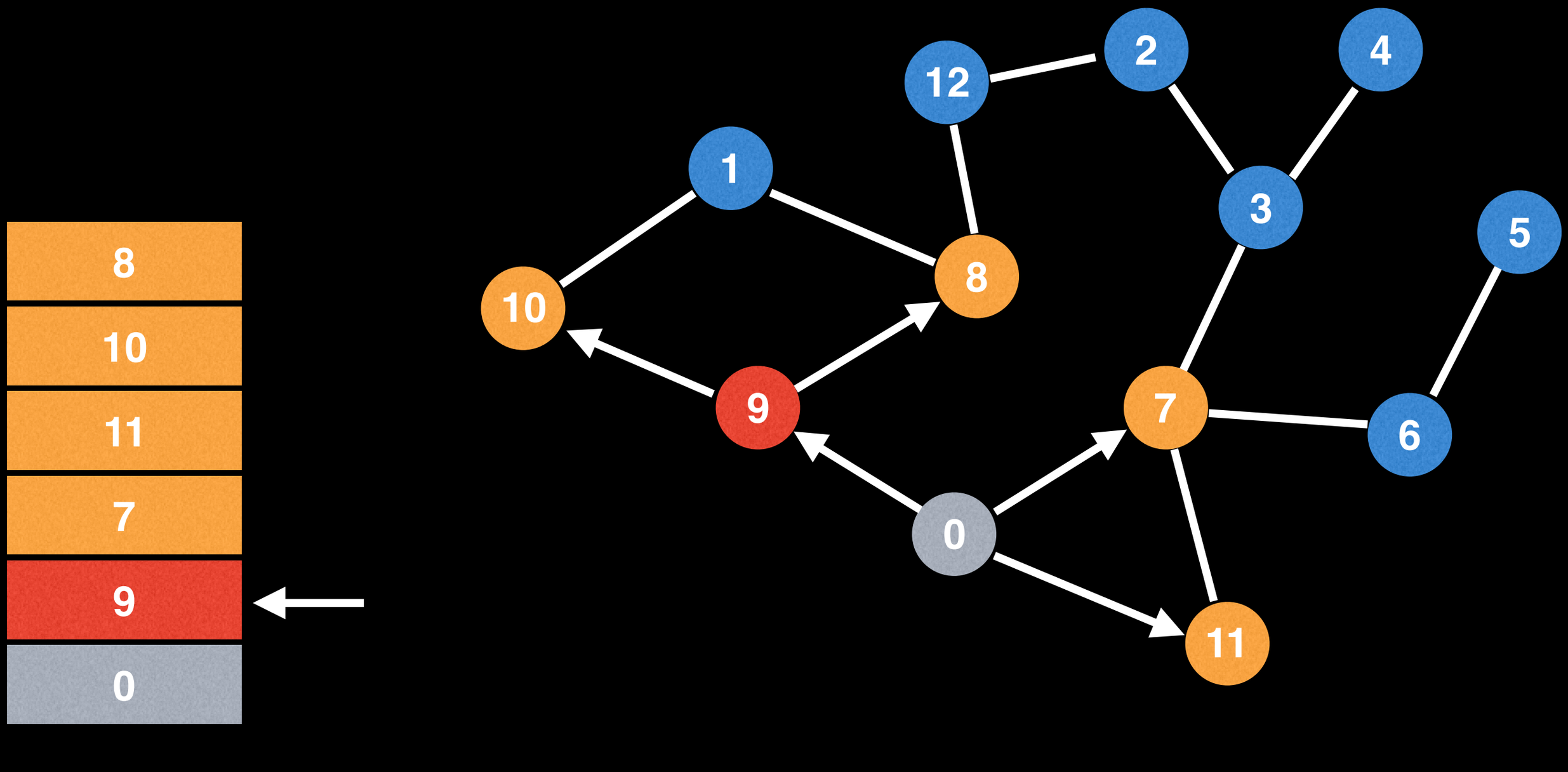
A BFS starts at some arbitrary node of a graph and explores the neighbour nodes first, before moving to the next level neighbours.



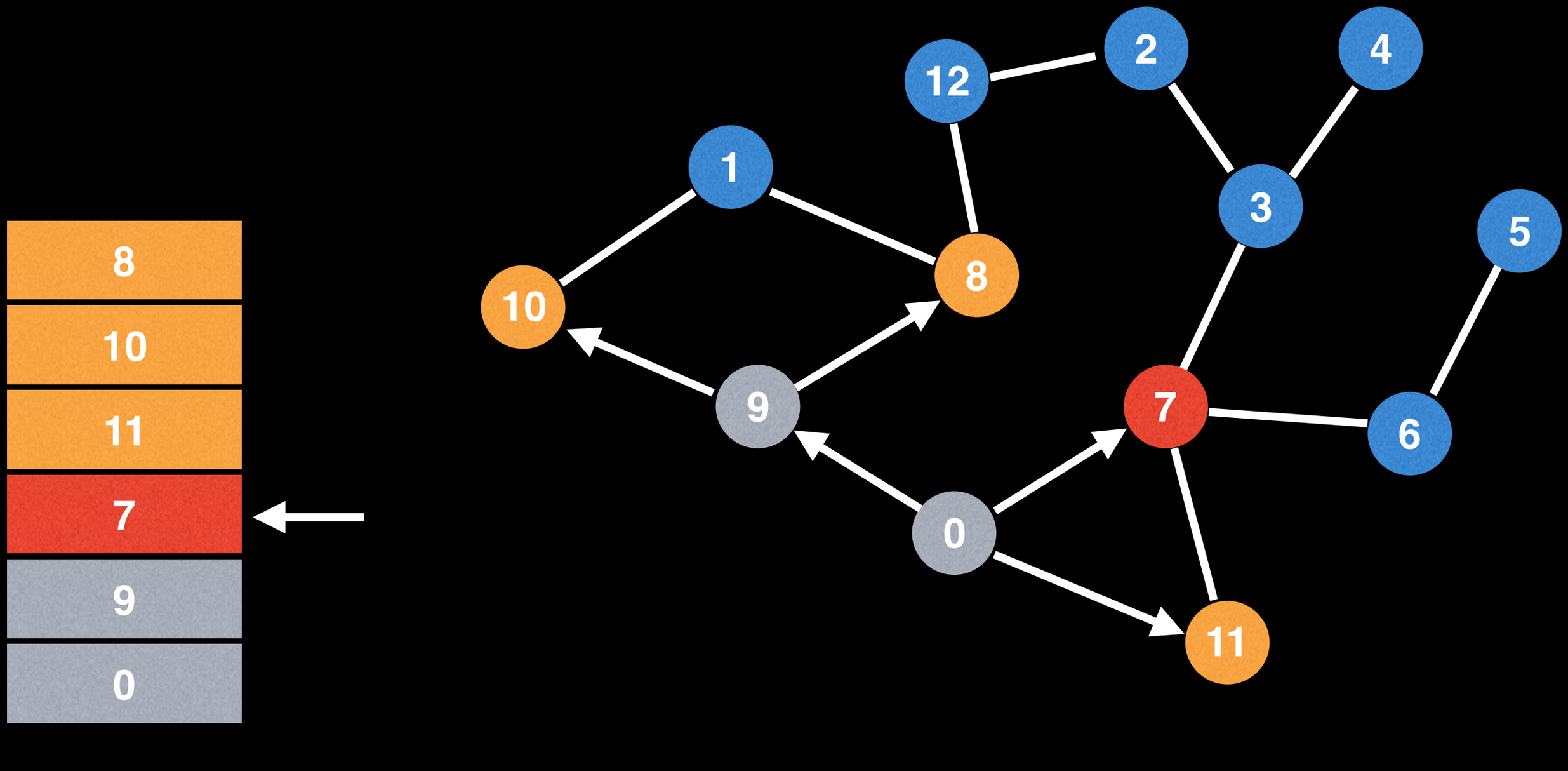
A BFS starts at some arbitrary node of a graph and explores the neighbour nodes first, before moving to the next level neighbours.



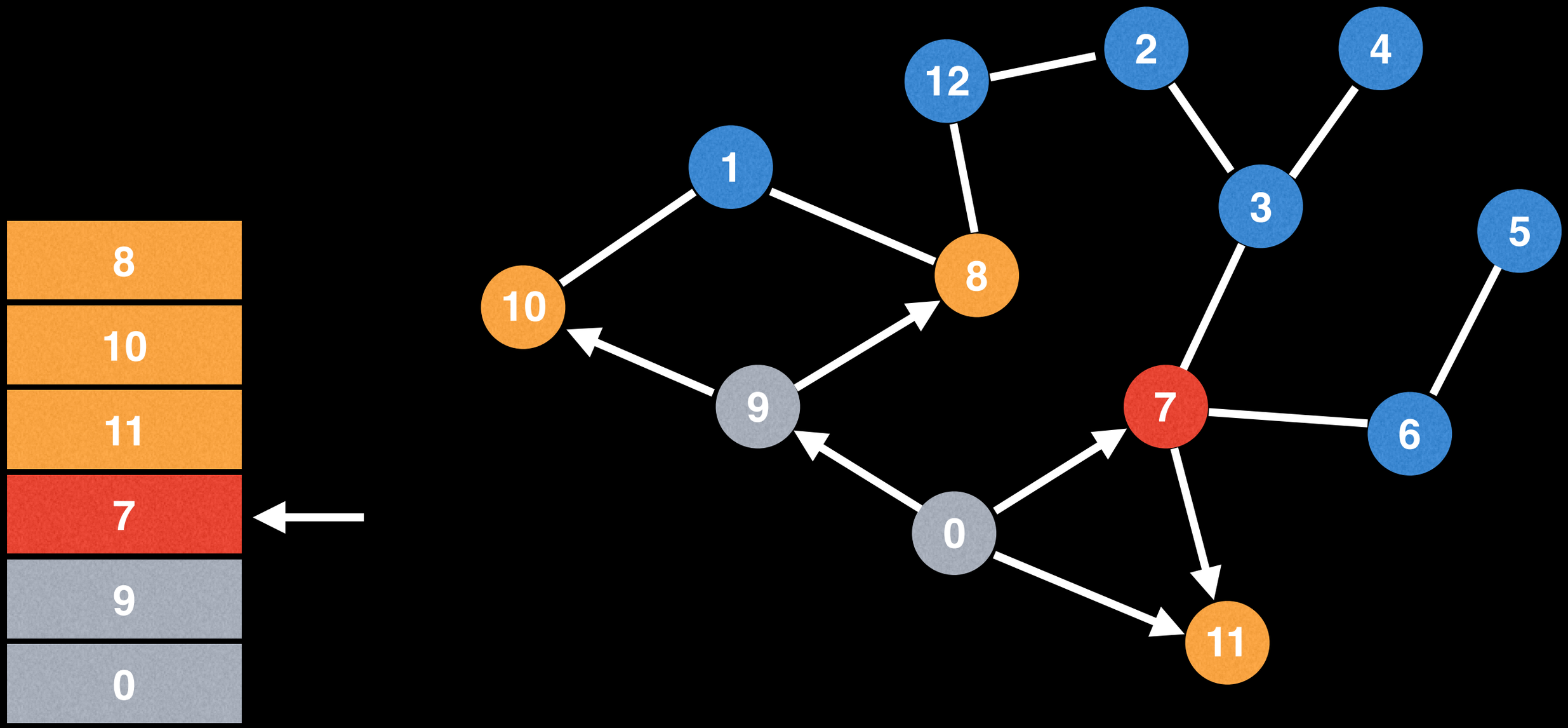
A BFS starts at some arbitrary node of a graph and explores the neighbour nodes first, before moving to the next level neighbours.



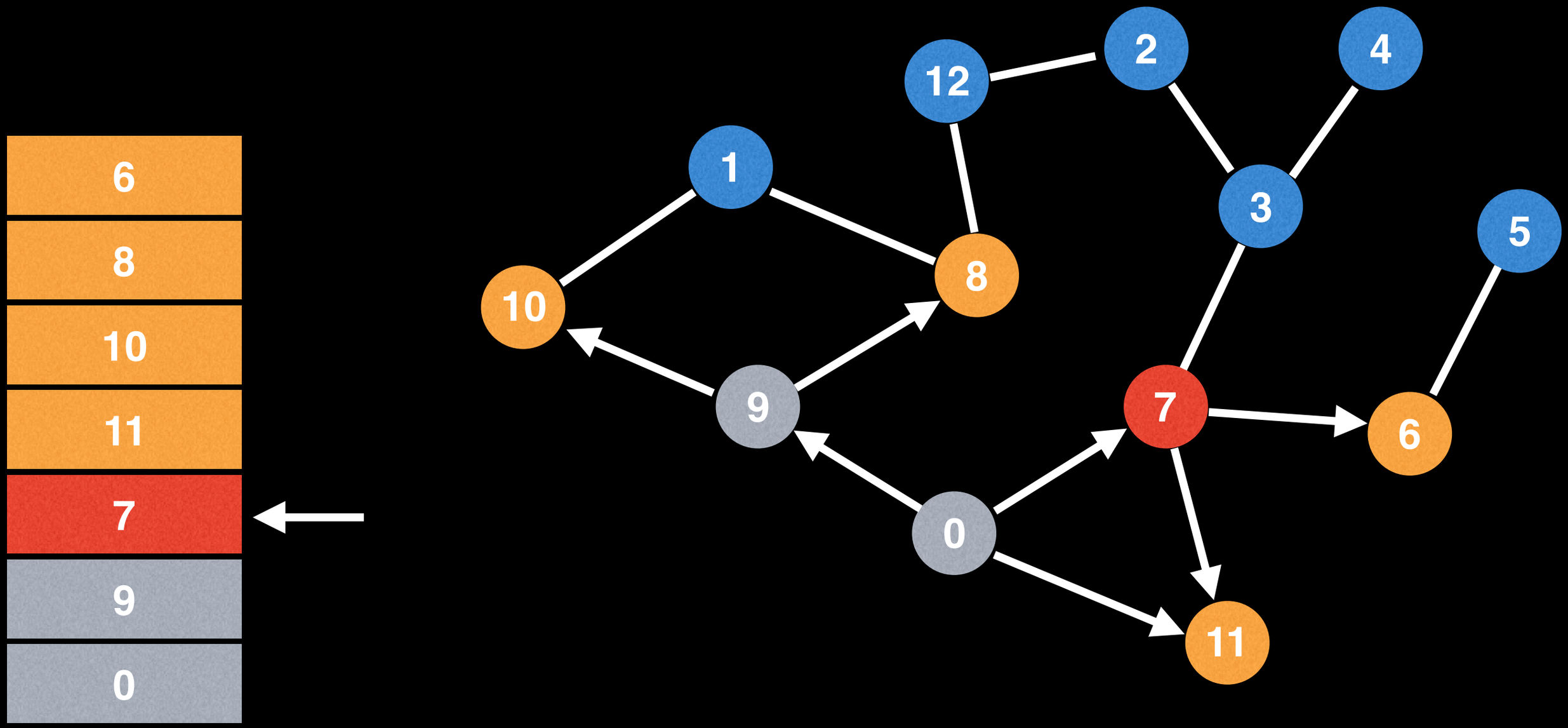
A BFS starts at some arbitrary node of a graph and explores the neighbour nodes first, before moving to the next level neighbours.



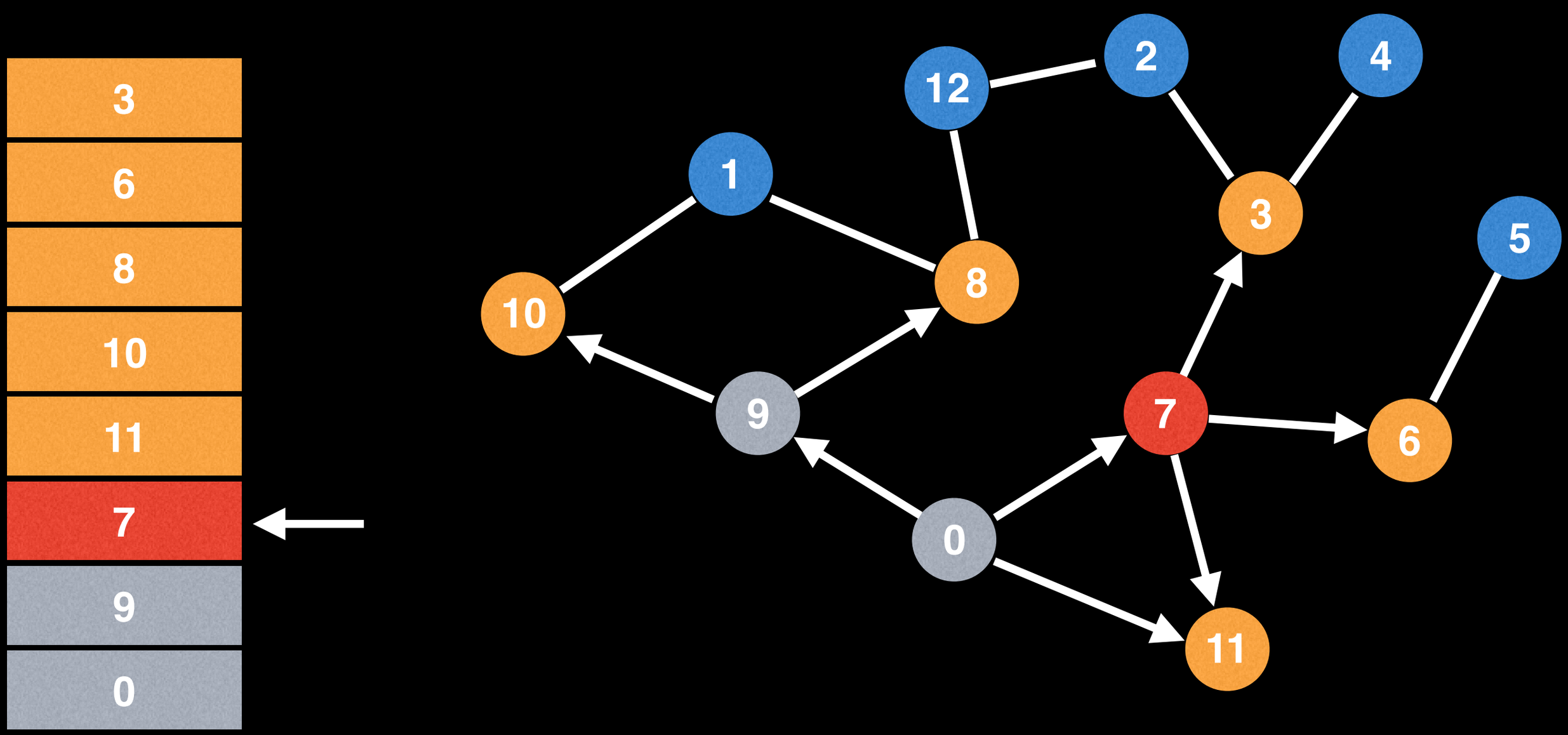
A BFS starts at some arbitrary node of a graph and explores the neighbour nodes first, before moving to the next level neighbours.



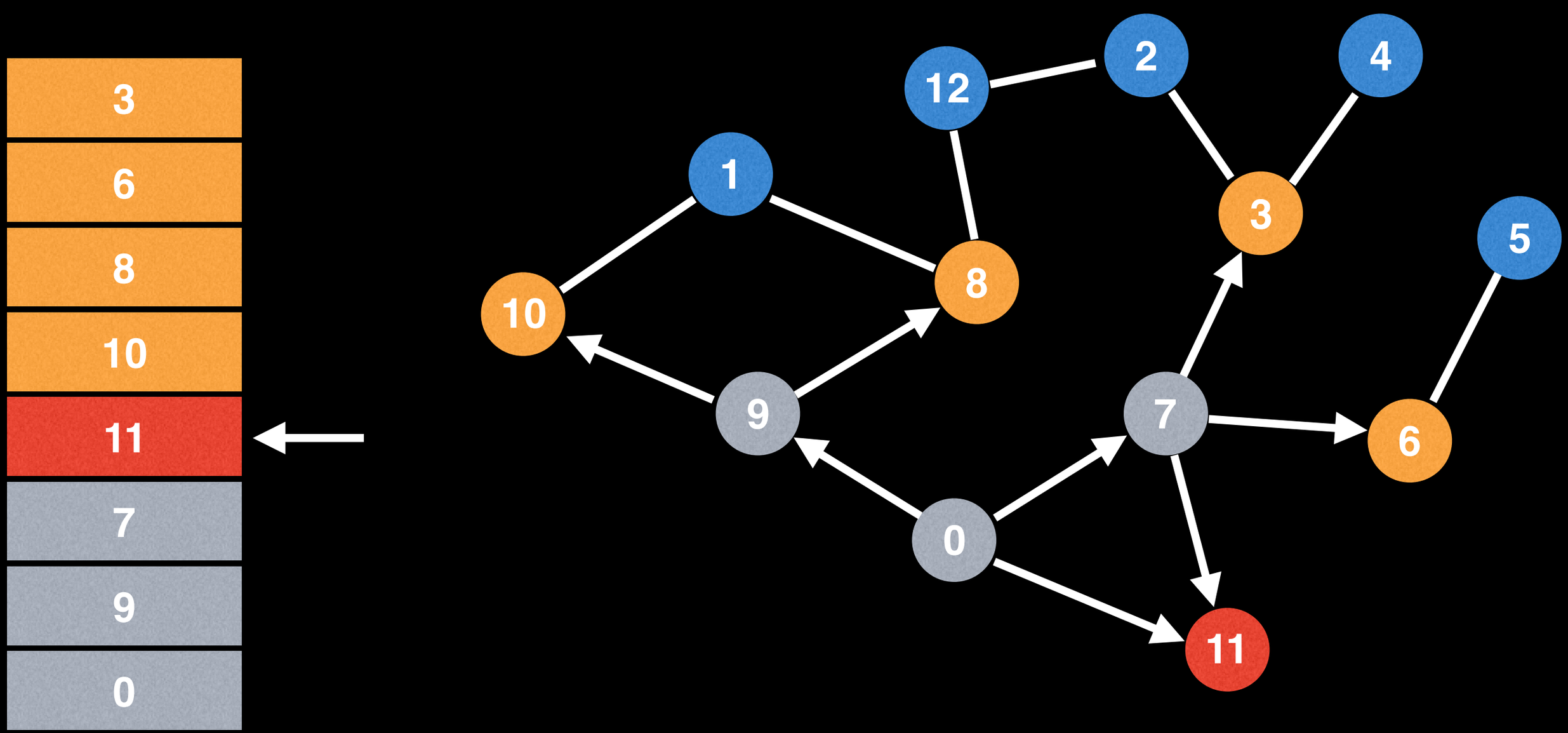
A BFS starts at some arbitrary node of a graph and explores the neighbour nodes first, before moving to the next level neighbours.



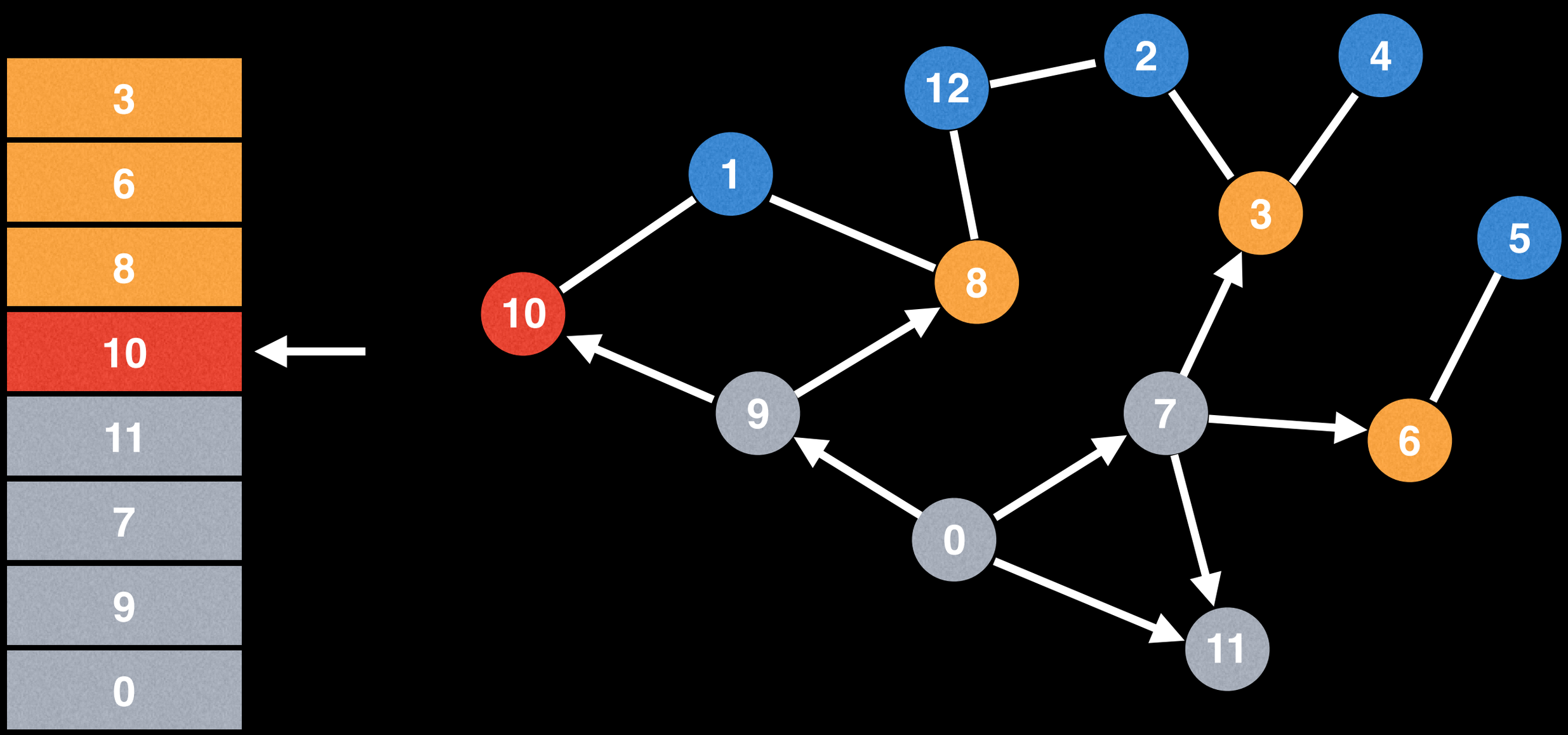
A BFS starts at some arbitrary node of a graph and explores the neighbour nodes first, before moving to the next level neighbours.



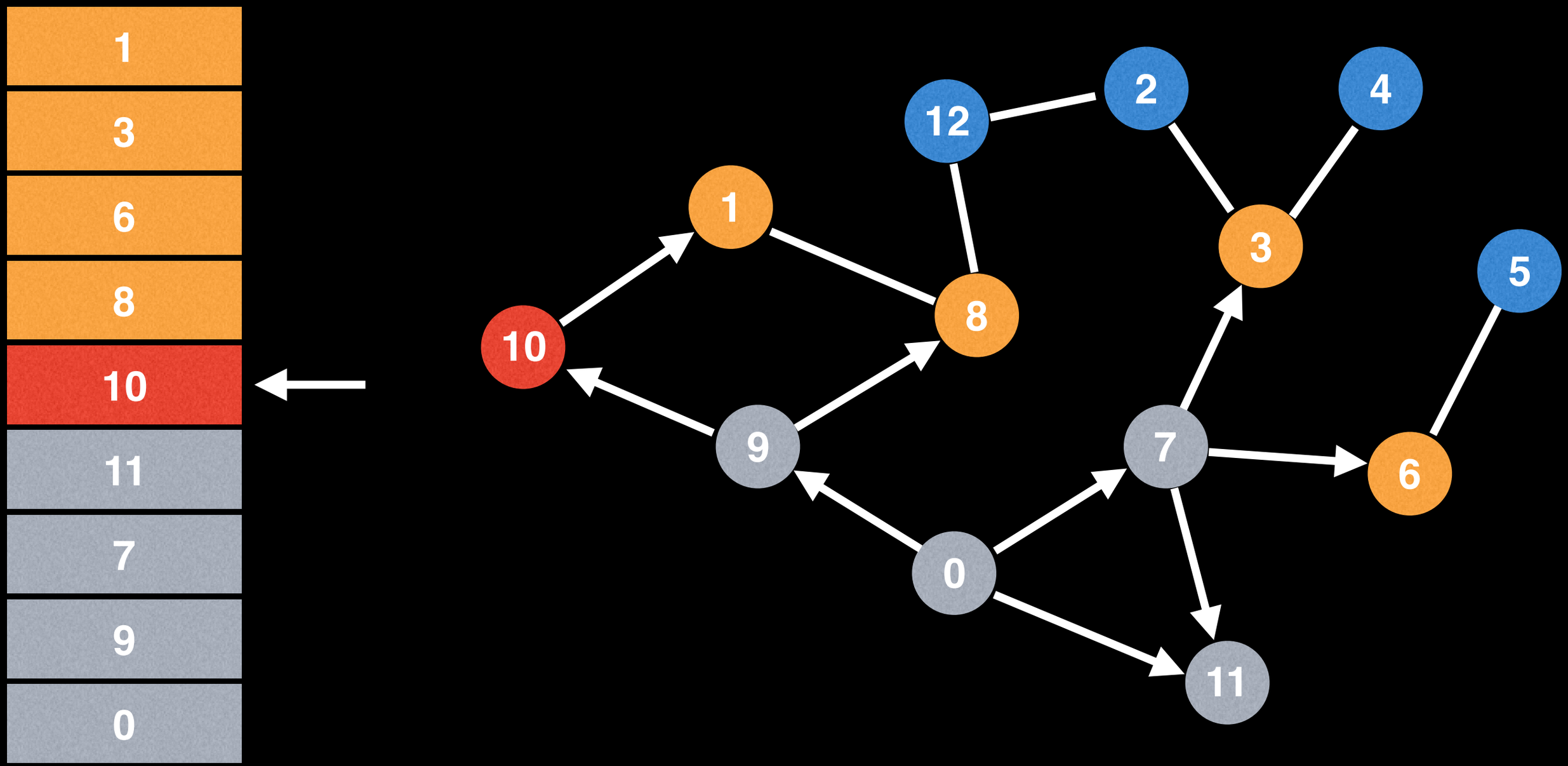
A BFS starts at some arbitrary node of a graph and explores the neighbour nodes first, before moving to the next level neighbours.



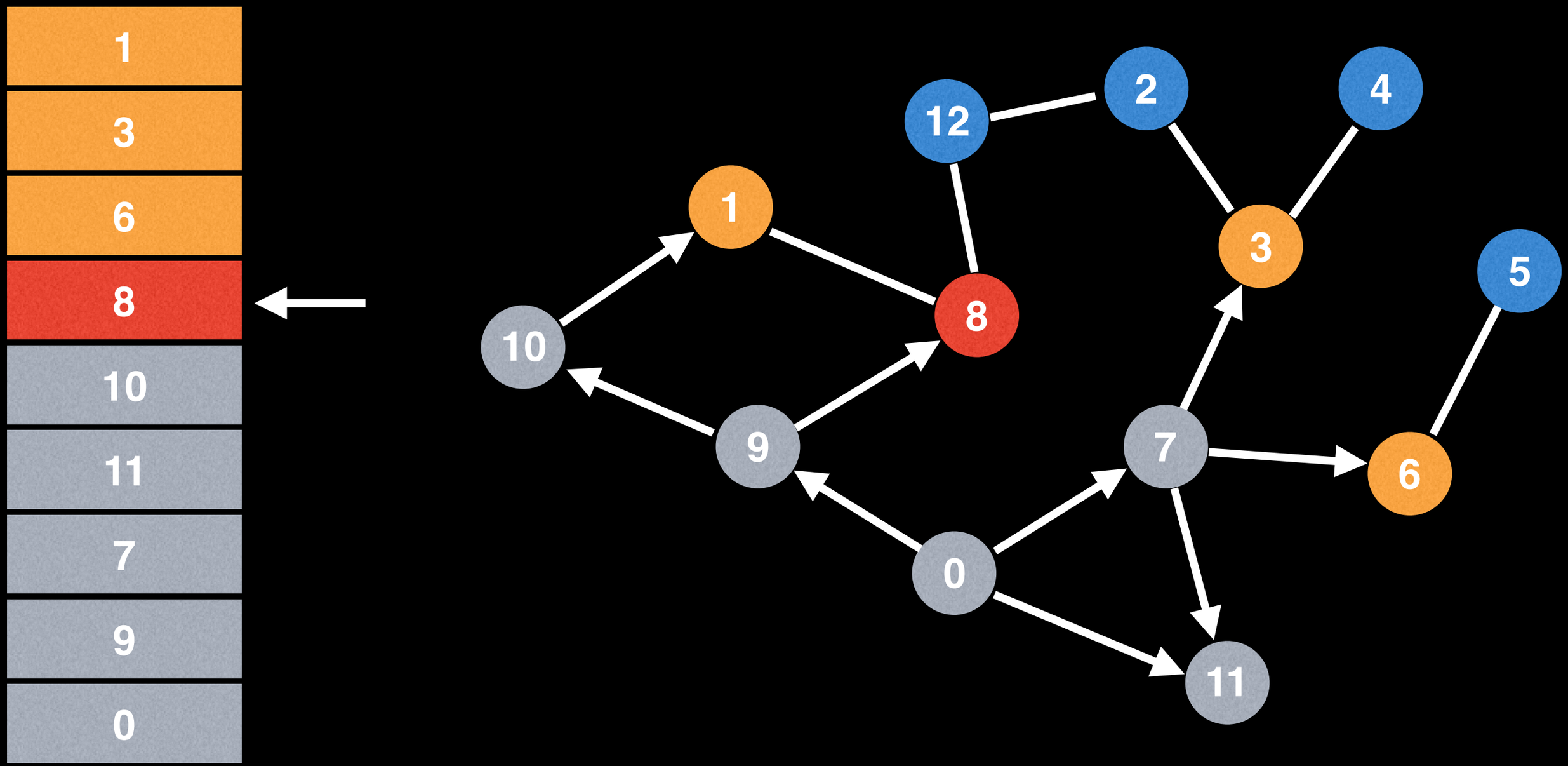
A BFS starts at some arbitrary node of a graph and explores the neighbour nodes first, before moving to the next level neighbours.



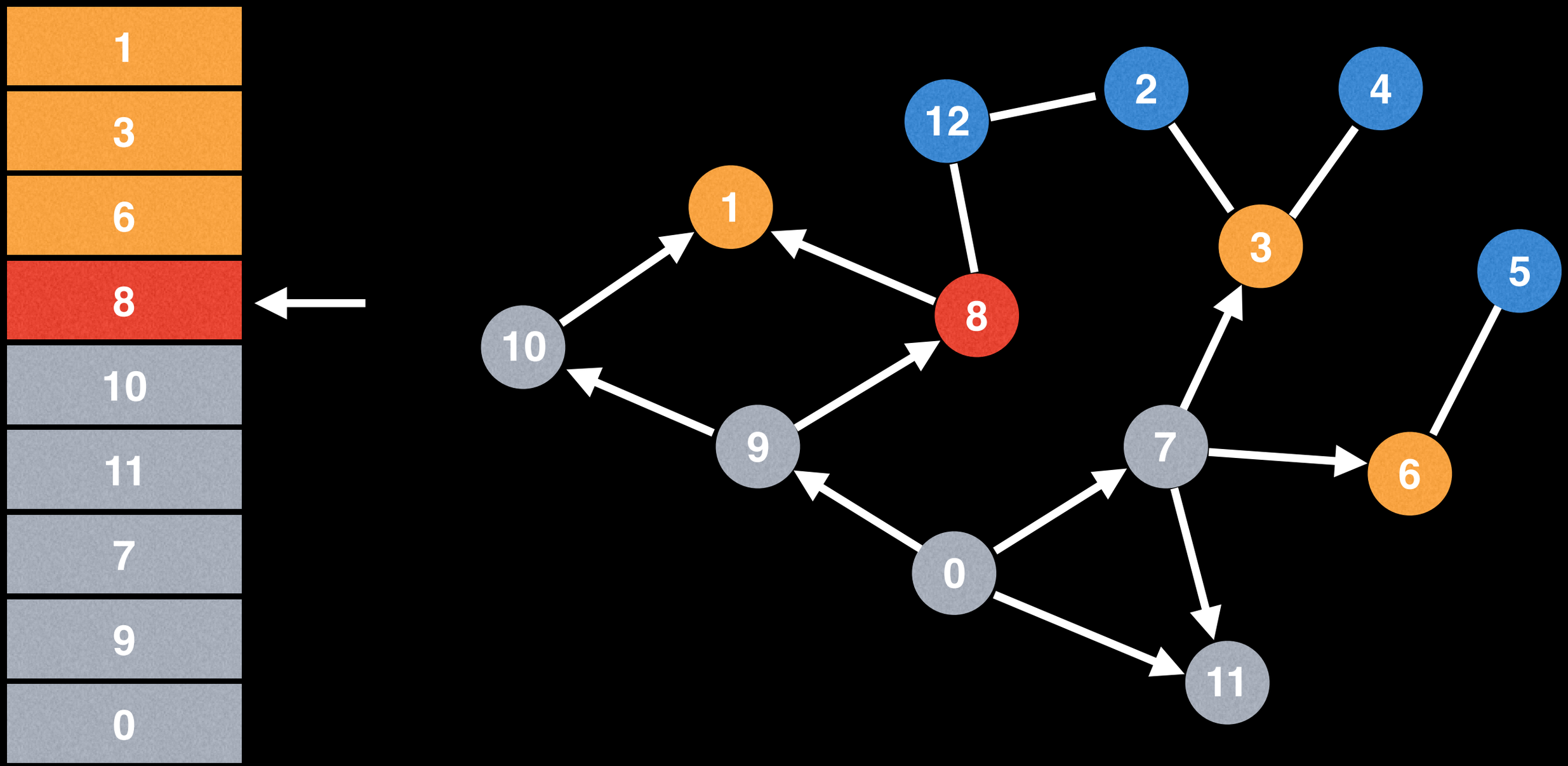
A BFS starts at some arbitrary node of a graph and explores the neighbour nodes first, before moving to the next level neighbours.



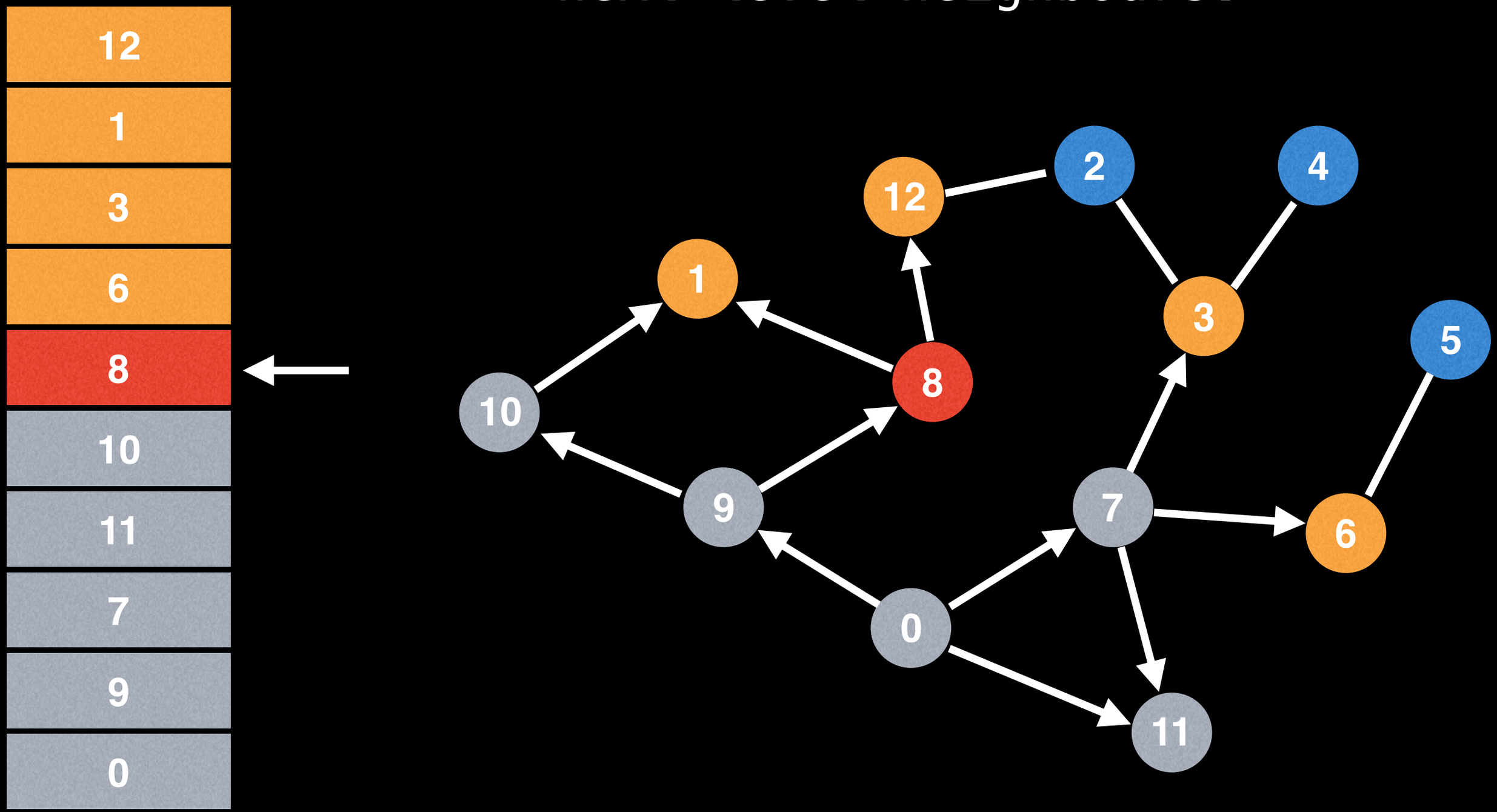
A BFS starts at some arbitrary node of a graph and explores the neighbour nodes first, before moving to the next level neighbours.



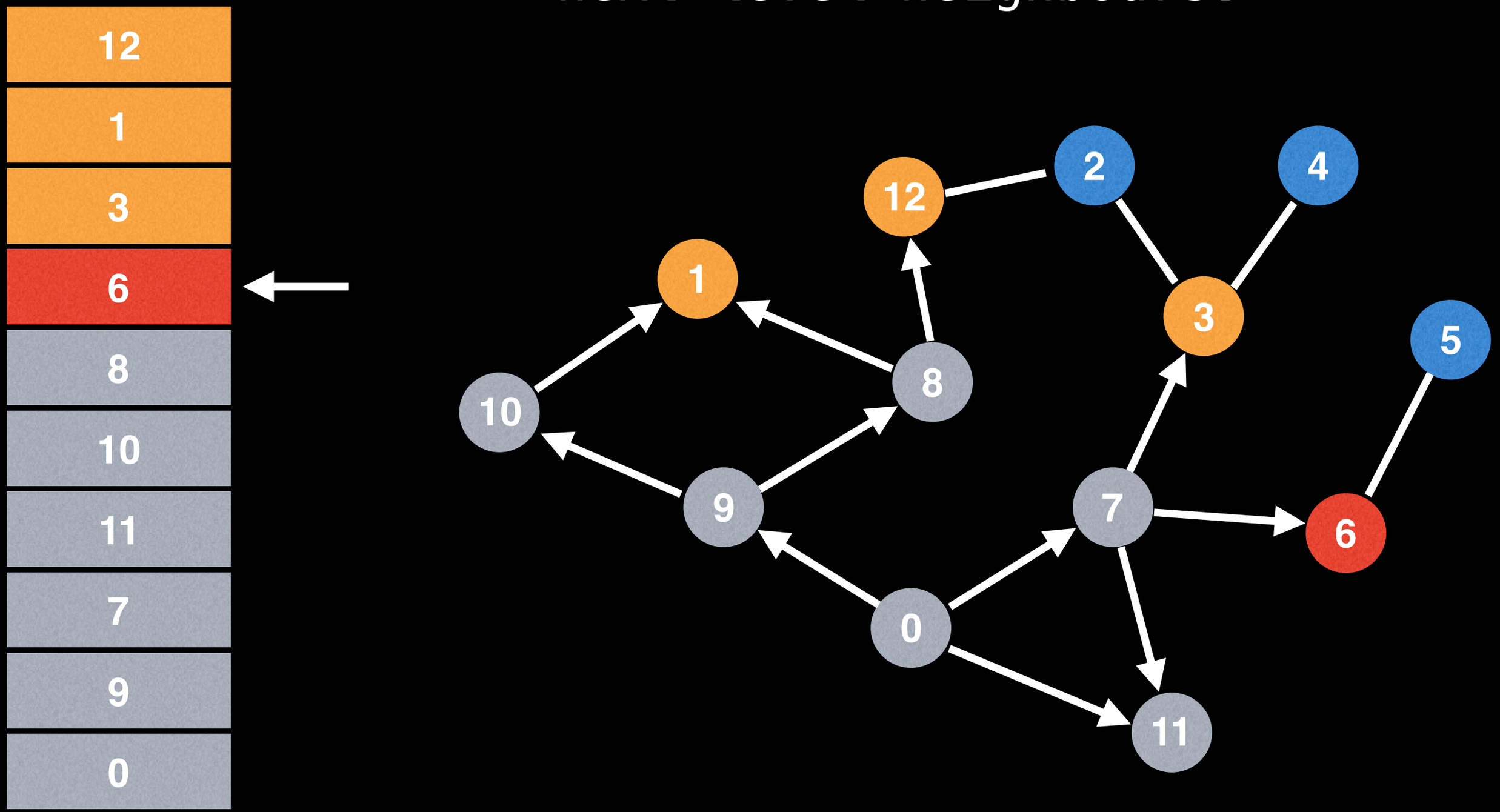
A BFS starts at some arbitrary node of a graph and explores the neighbour nodes first, before moving to the next level neighbours.



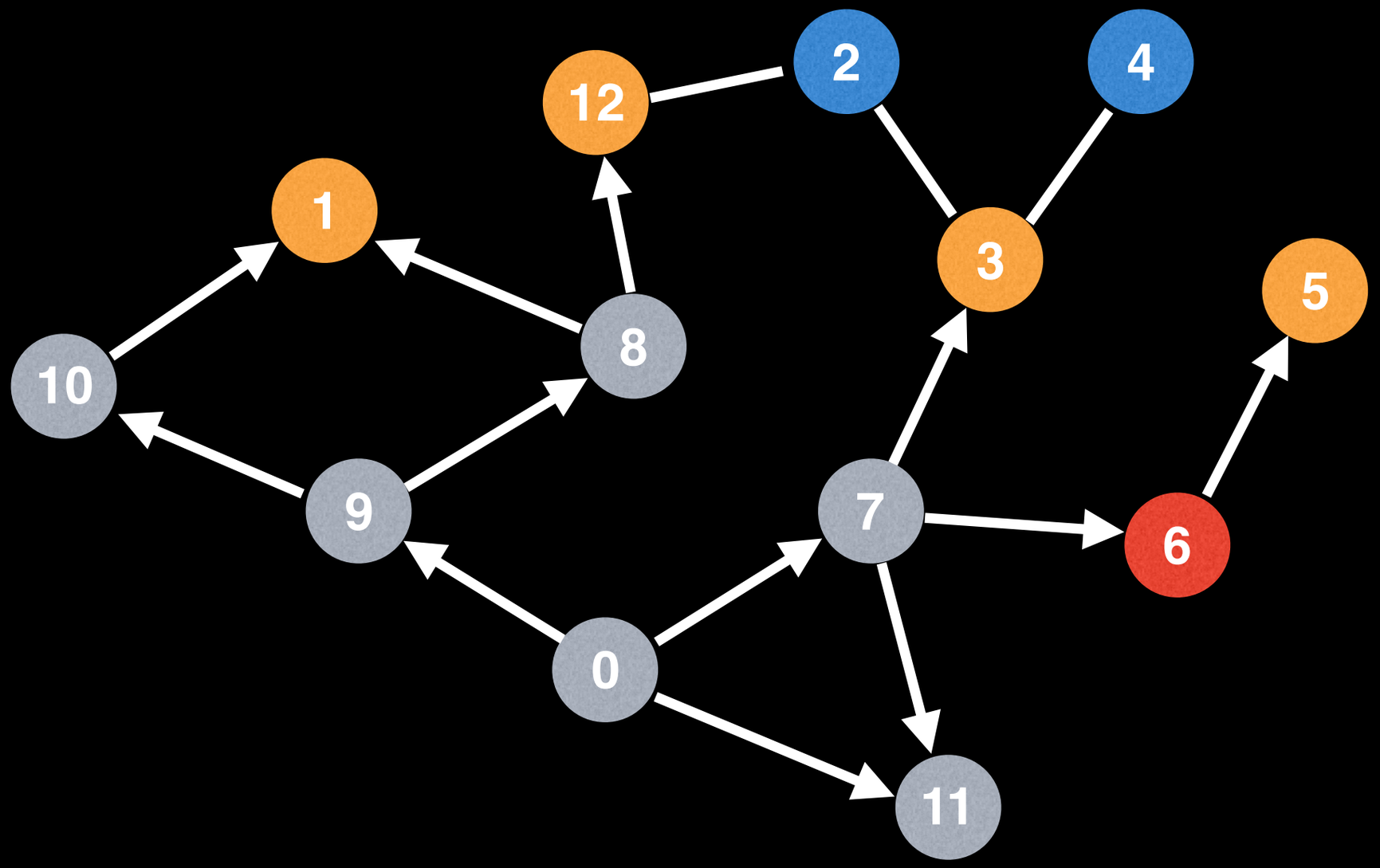
A BFS starts at some arbitrary node of a graph and explores the neighbour nodes first, before moving to the next level neighbours.



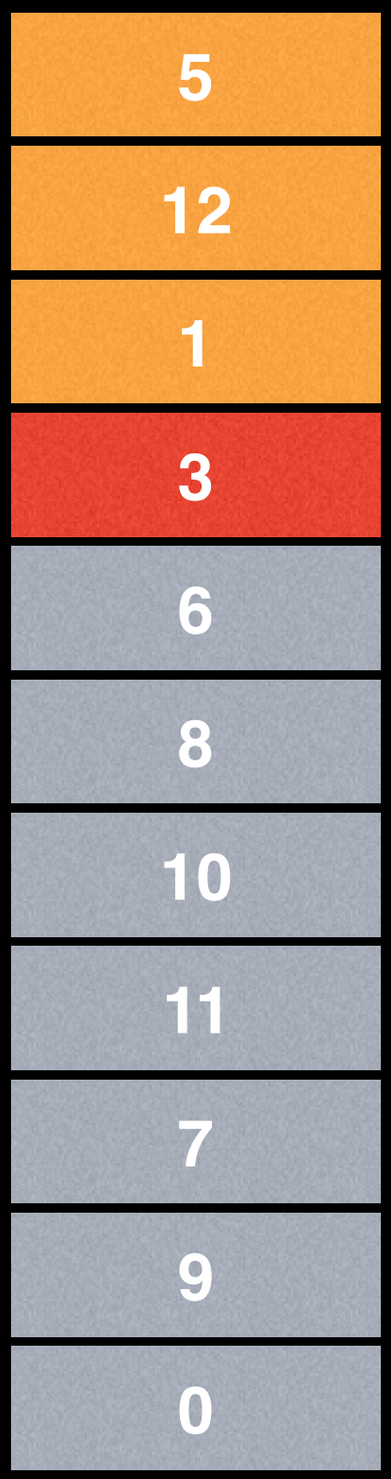
A BFS starts at some arbitrary node of a graph and explores the neighbour nodes first, before moving to the next level neighbours.

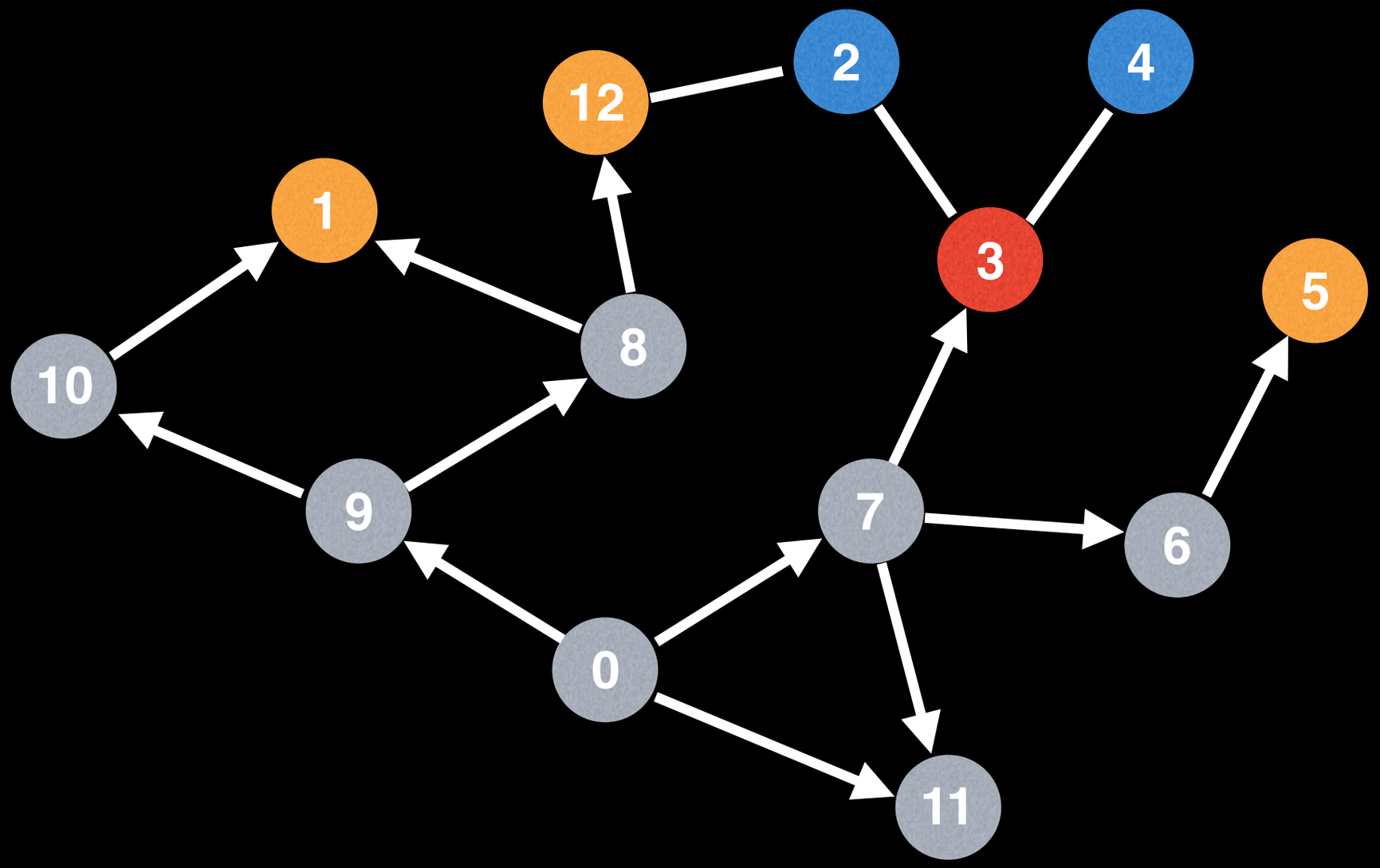


A BFS starts at some arbitrary node of a graph and explores the neighbour nodes first, before moving to the next level neighbours.

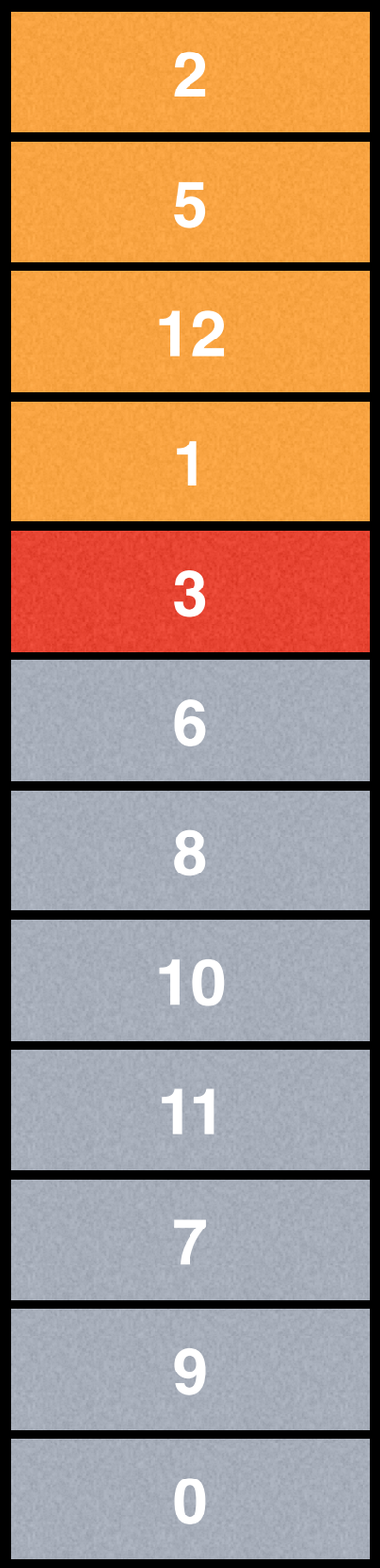


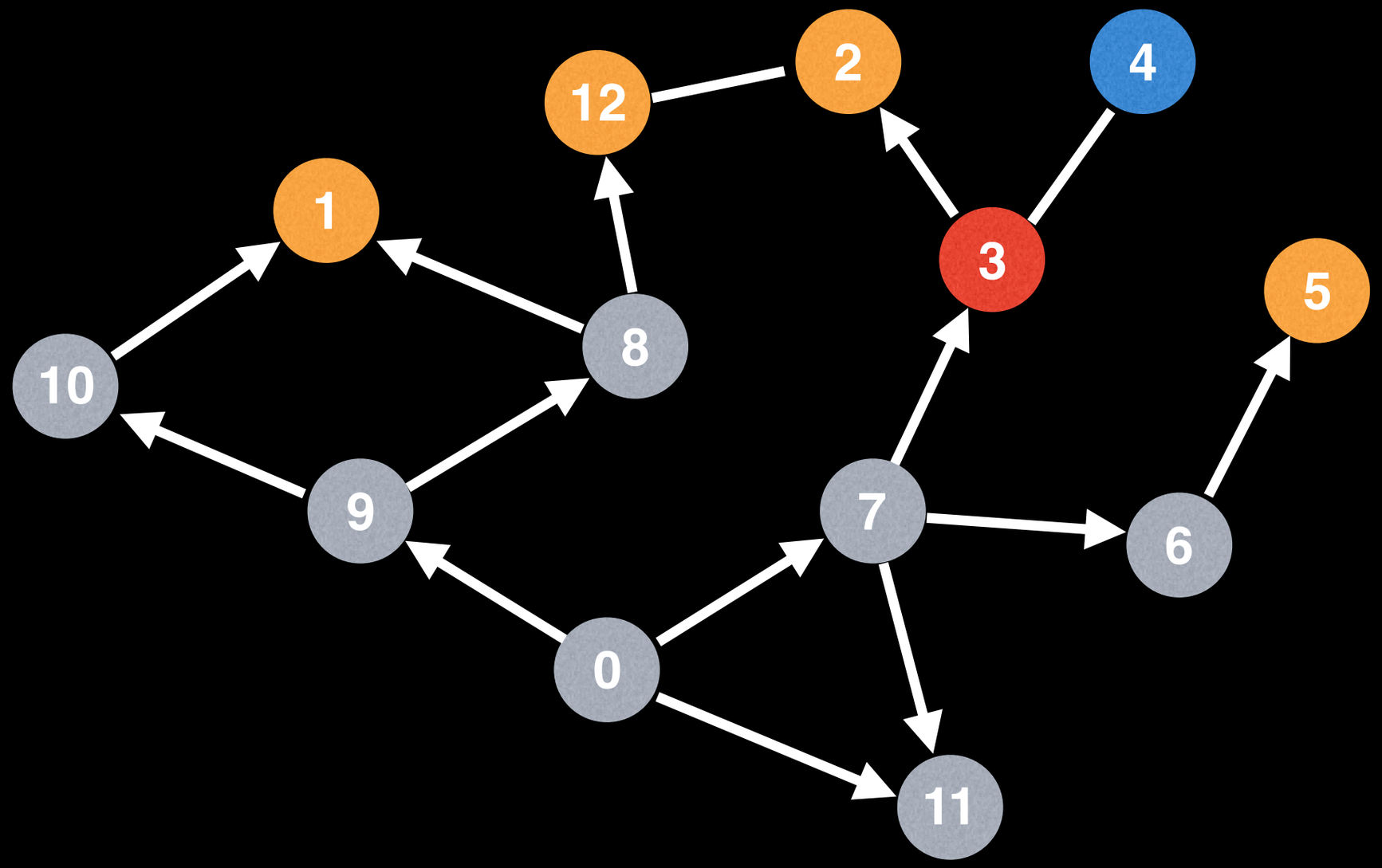
A BFS starts at some arbitrary node of a graph and explores the neighbour nodes first, before moving to the next level neighbours.



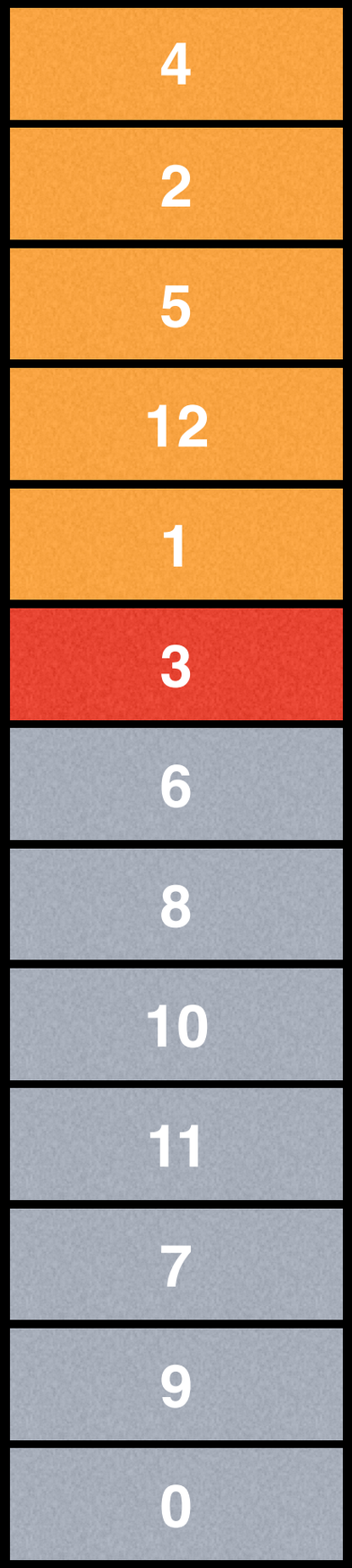


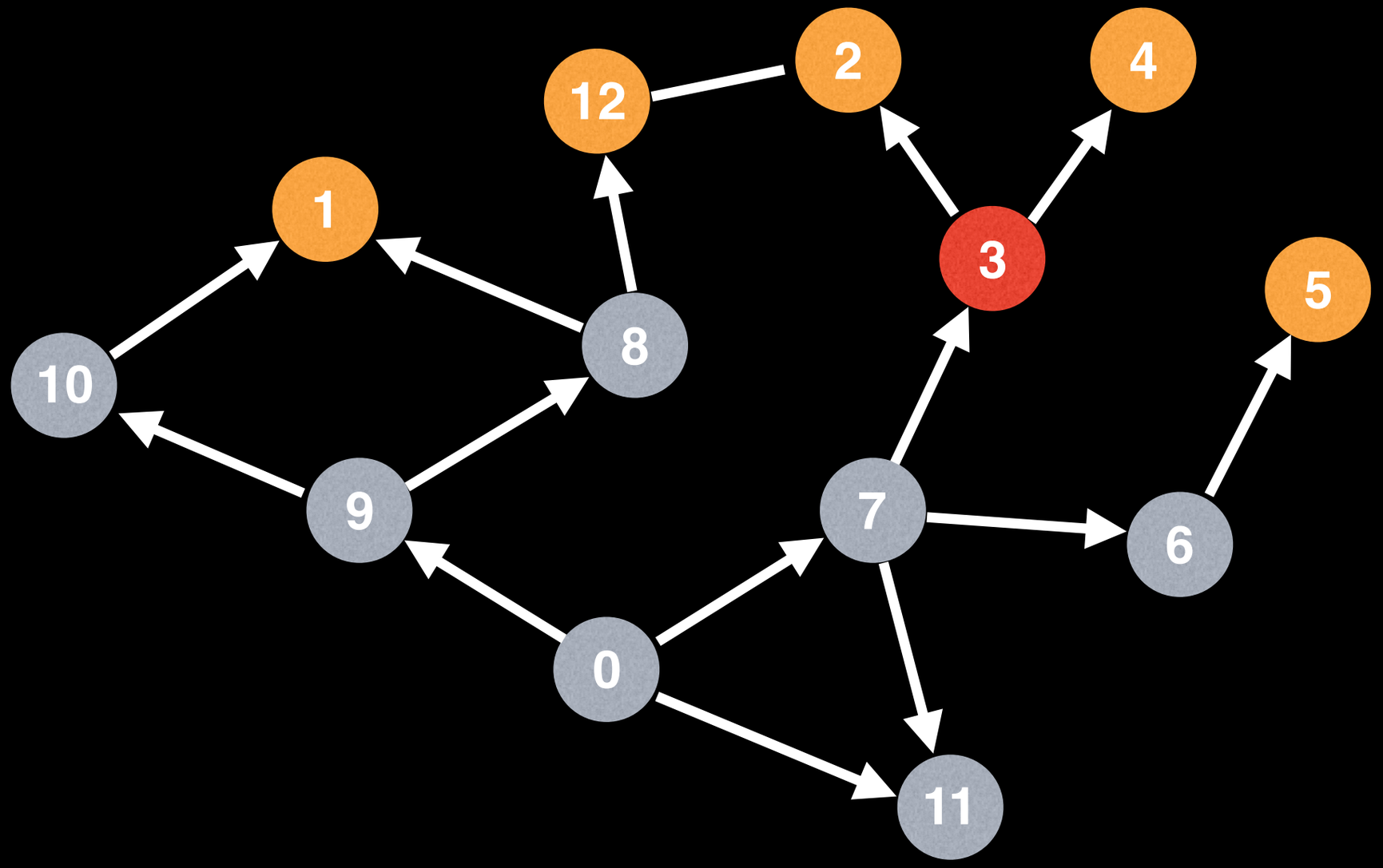
A BFS starts at some arbitrary node of a graph and explores the neighbour nodes first, before moving to the next level neighbours.



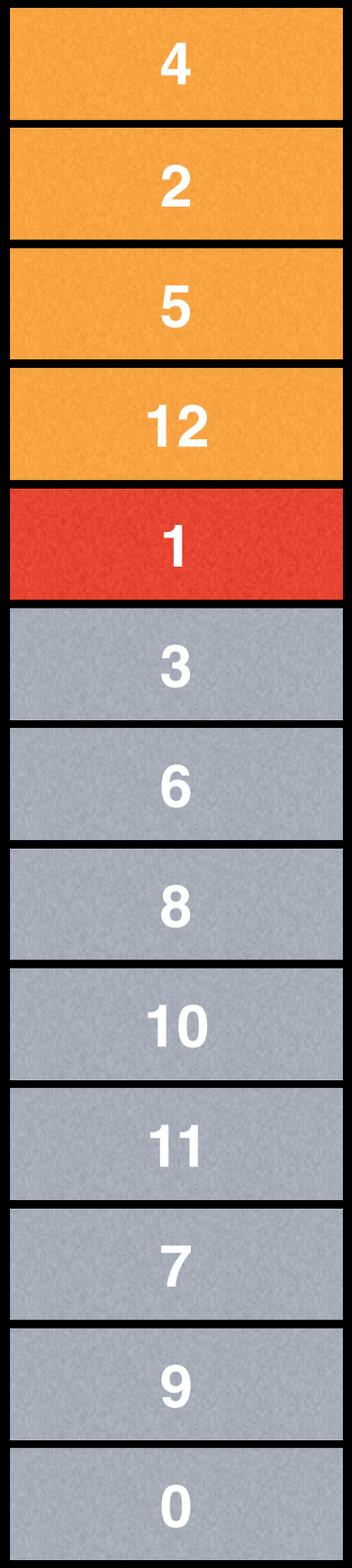


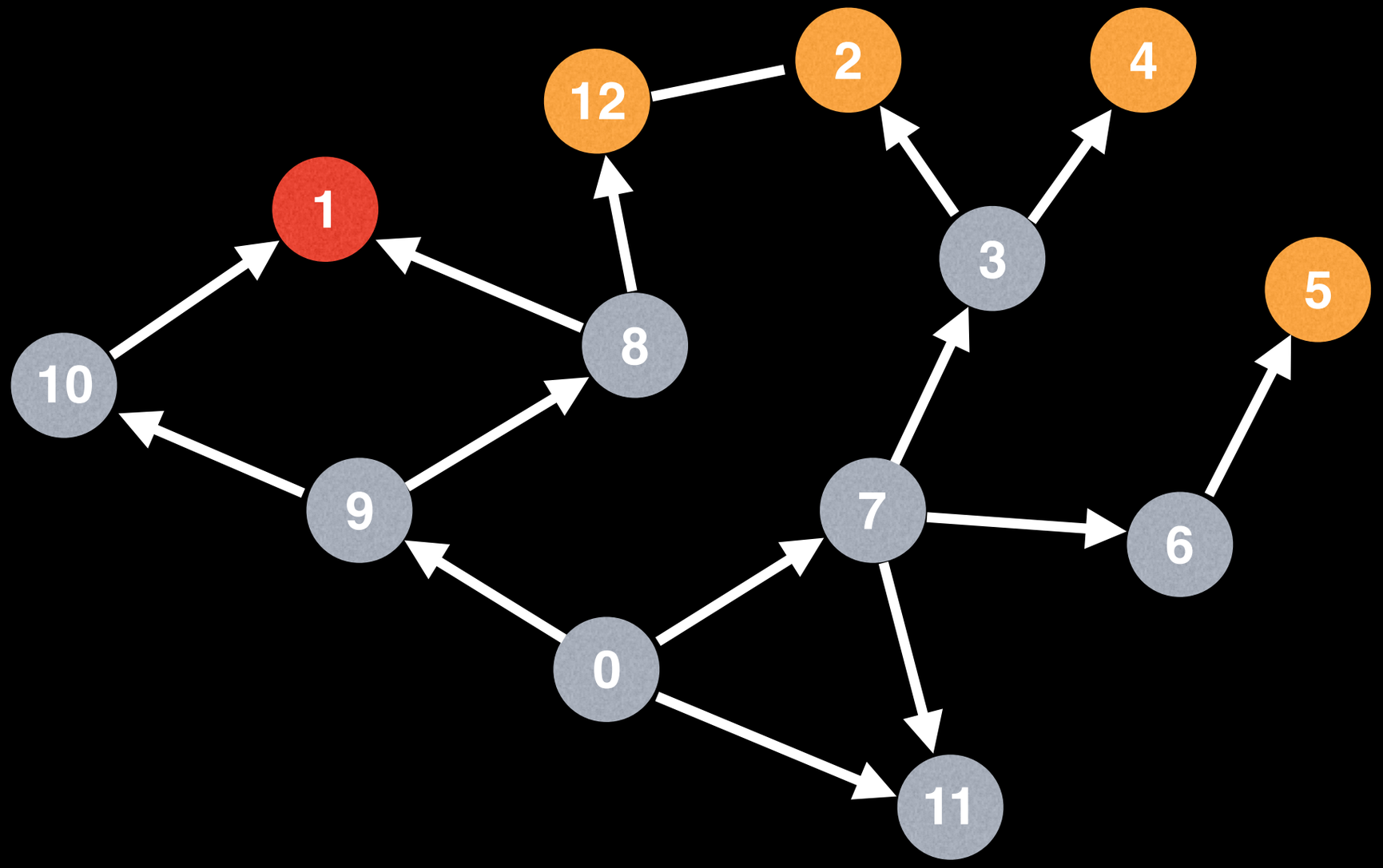
A BFS starts at some arbitrary node of a graph and explores the neighbour nodes first, before moving to the next level neighbours.



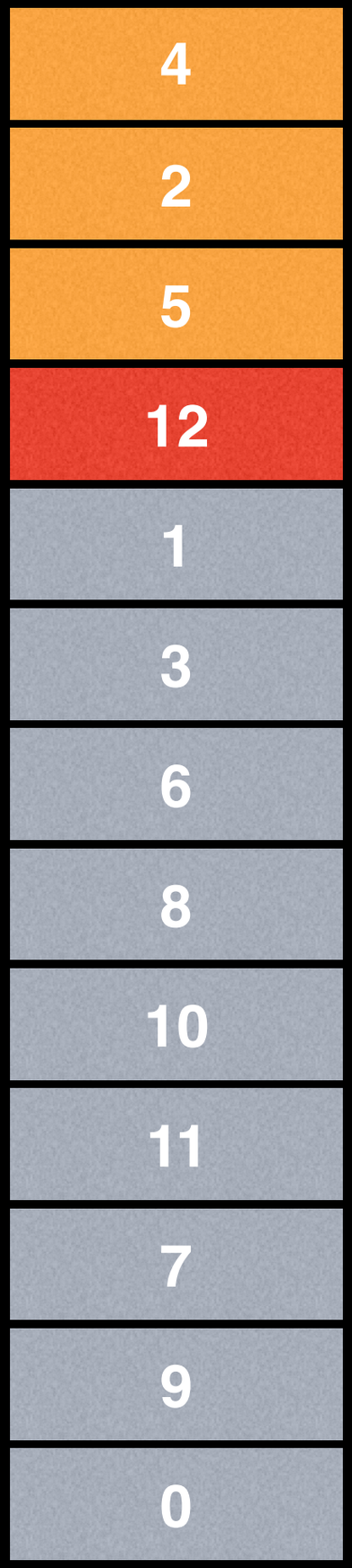


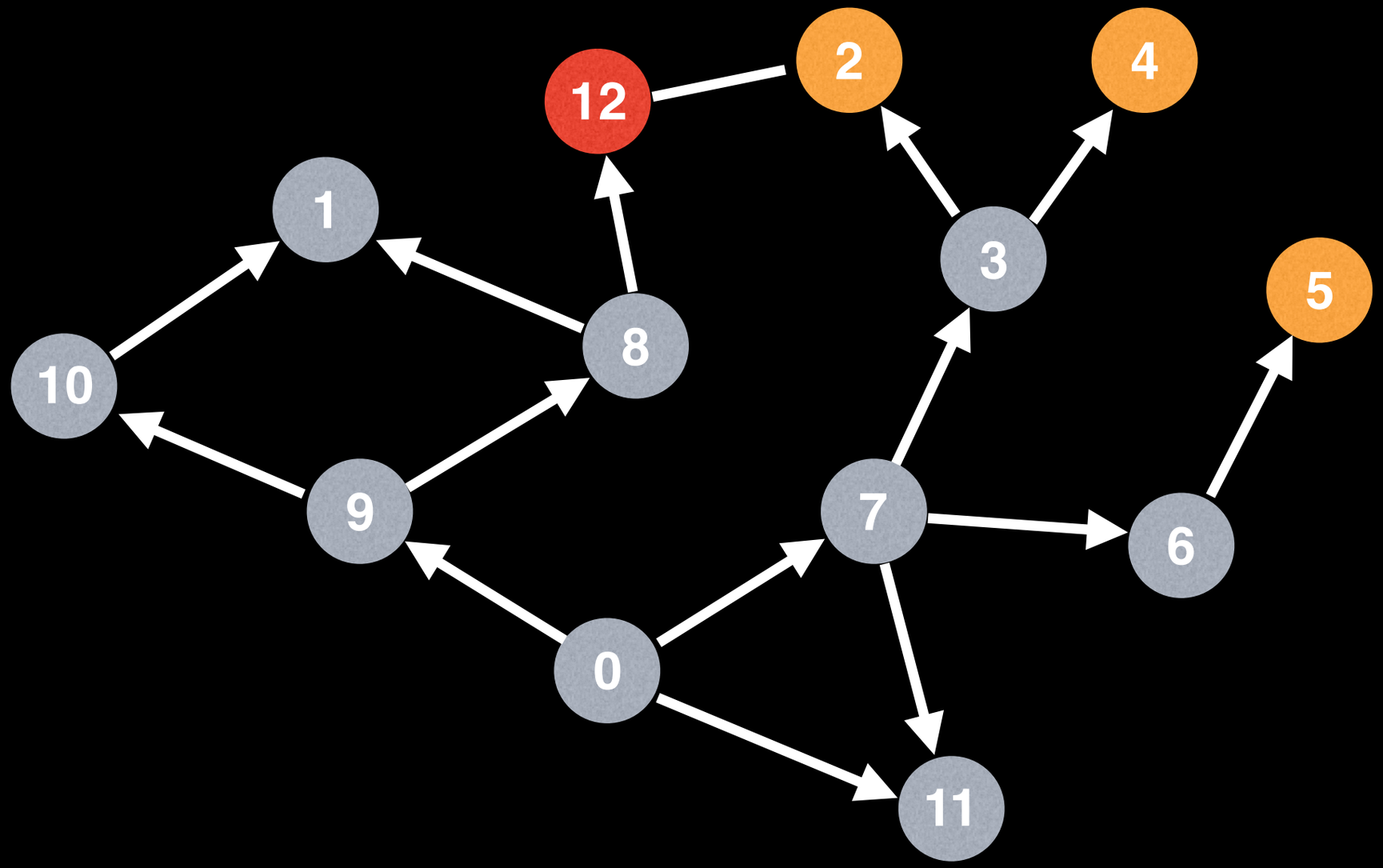
A BFS starts at some arbitrary node of a graph and explores the neighbour nodes first, before moving to the next level neighbours.



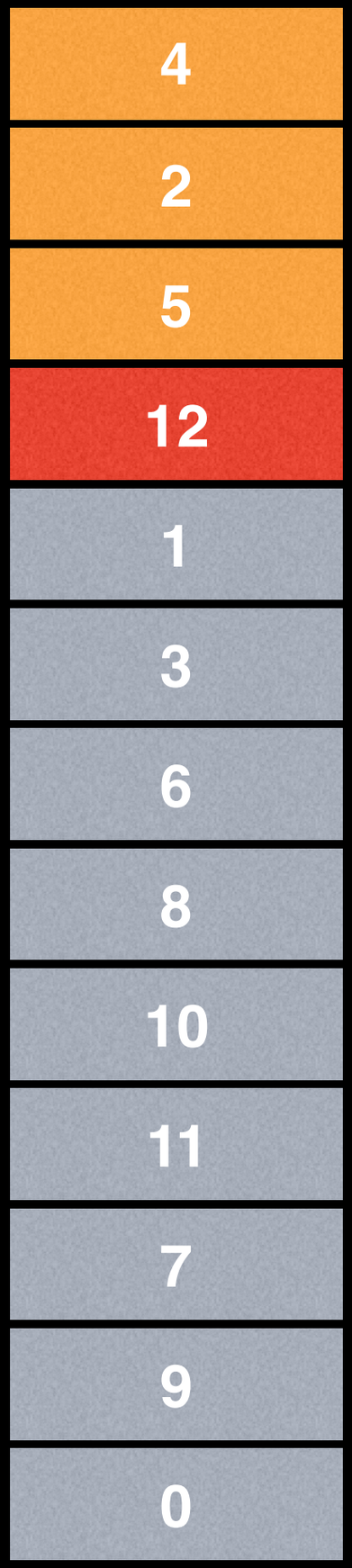


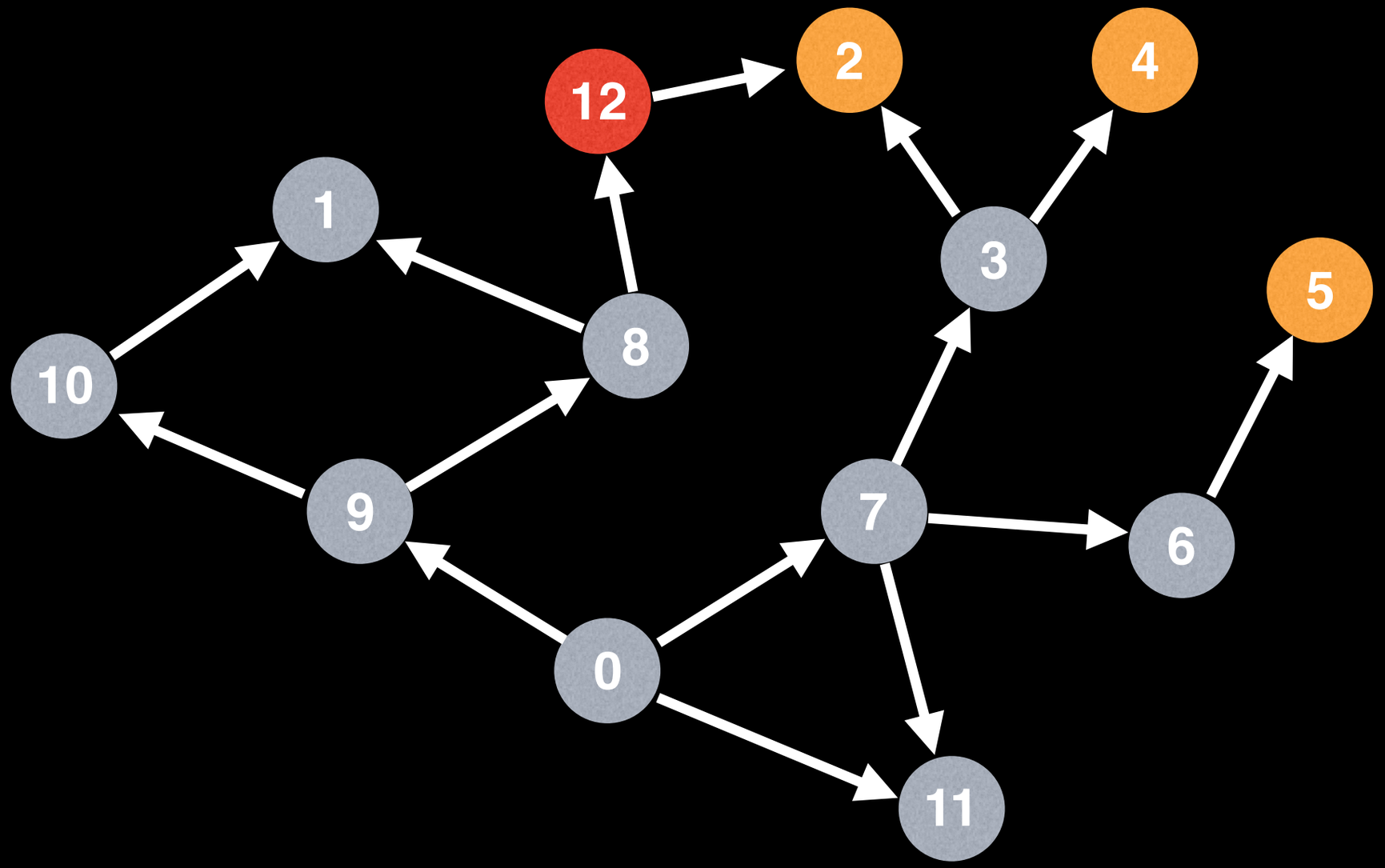
A BFS starts at some arbitrary node of a graph and explores the neighbour nodes first, before moving to the next level neighbours.



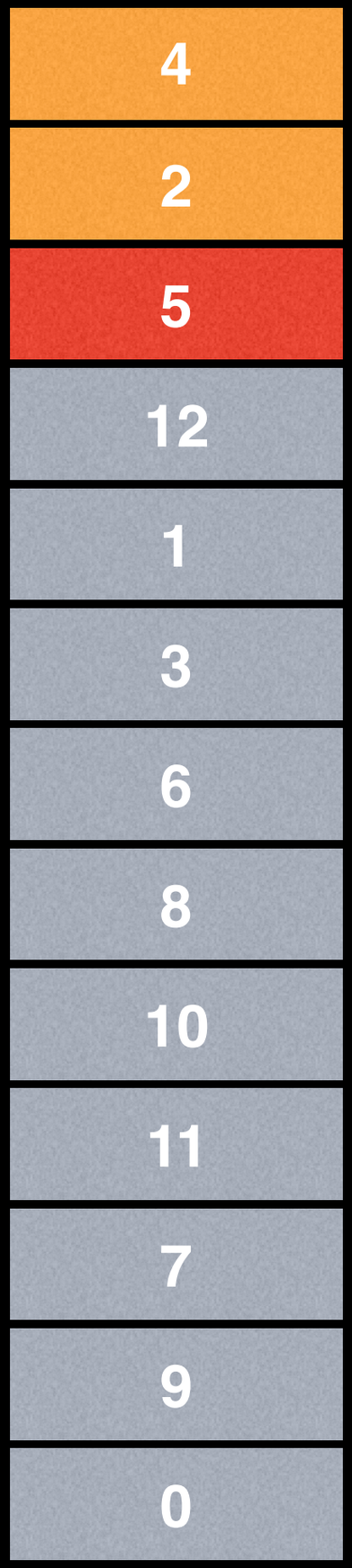


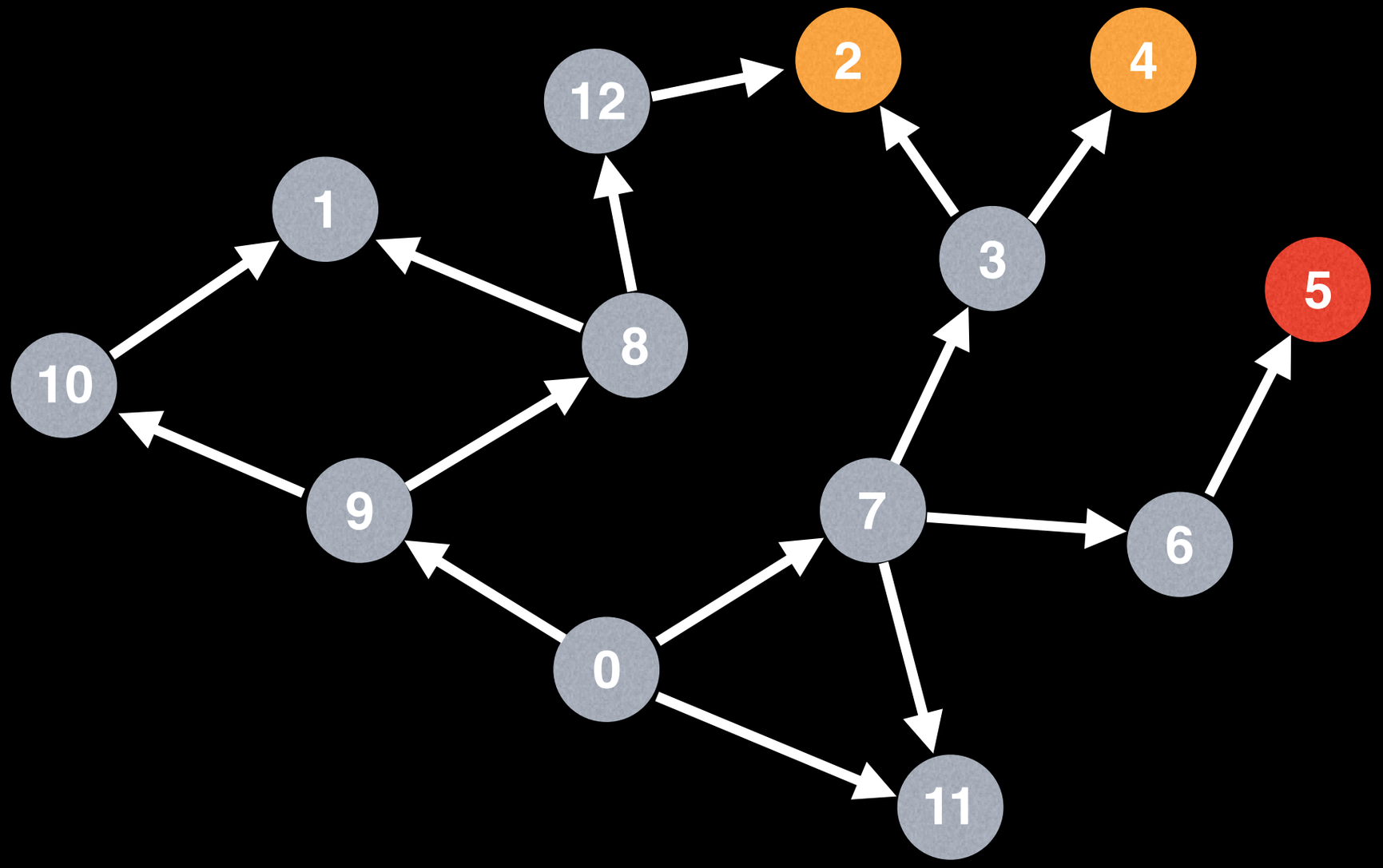
A BFS starts at some arbitrary node of a graph and explores the neighbour nodes first, before moving to the next level neighbours.



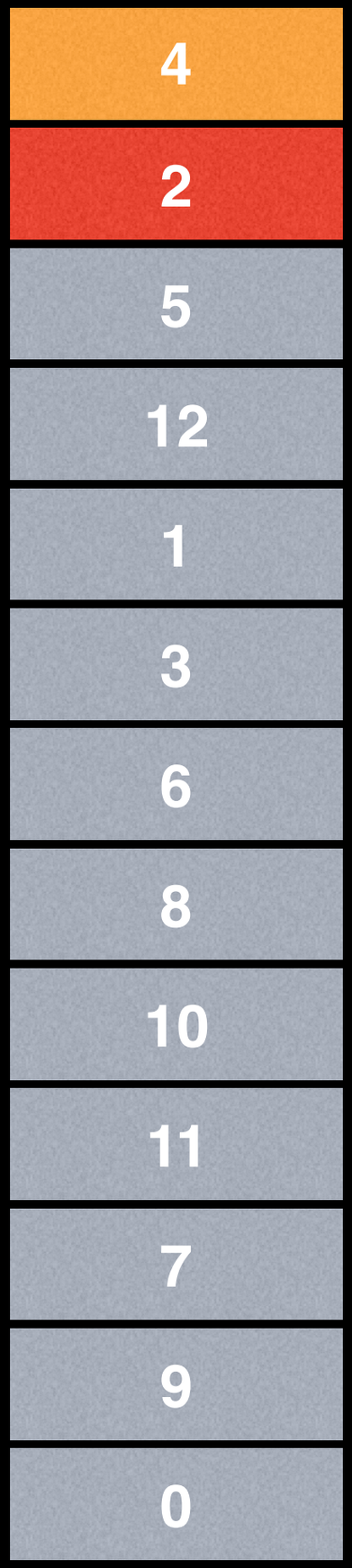


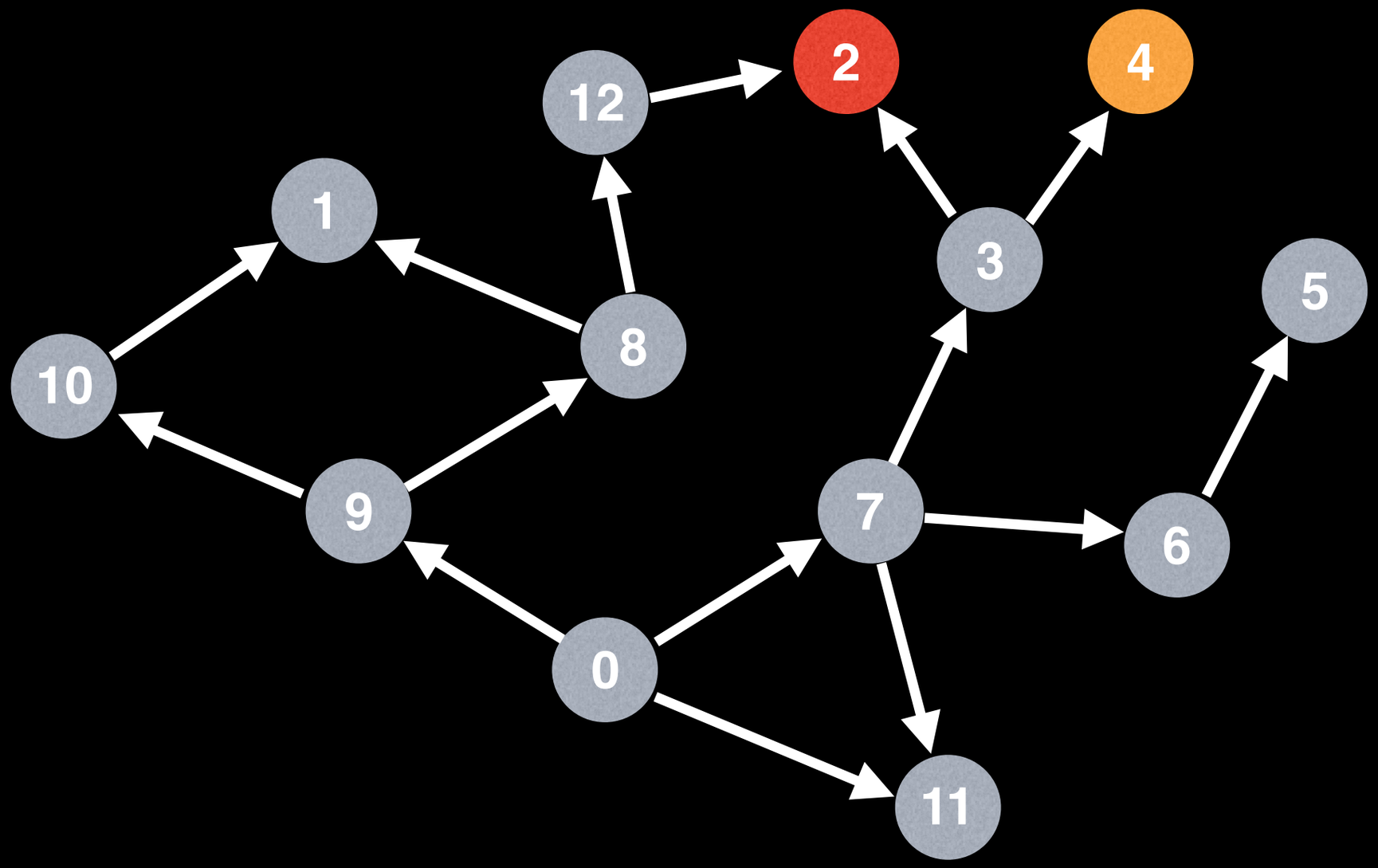
A BFS starts at some arbitrary node of a graph and explores the neighbour nodes first, before moving to the next level neighbours.





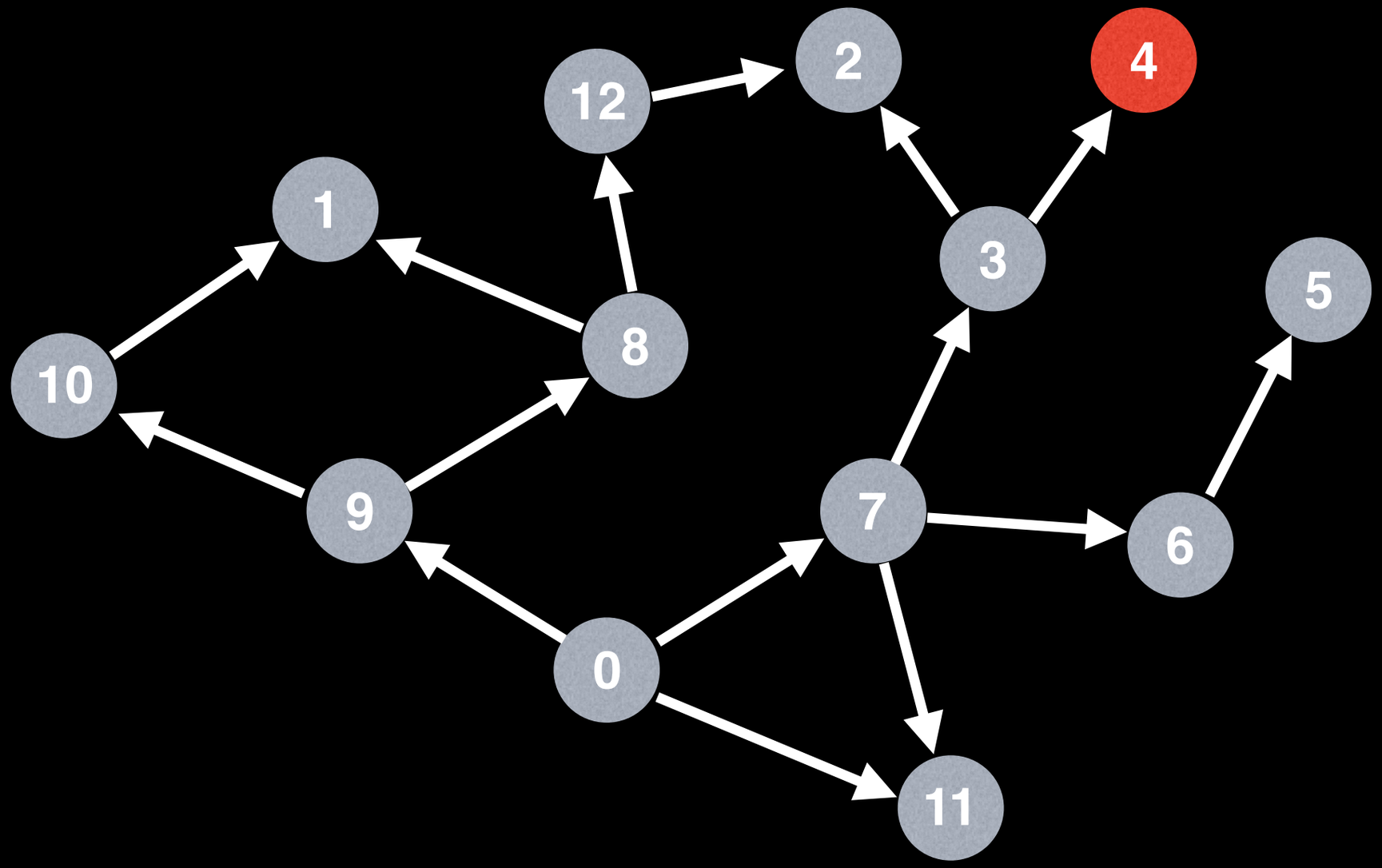
A BFS starts at some arbitrary node of a graph and explores the neighbour nodes first, before moving to the next level neighbours.





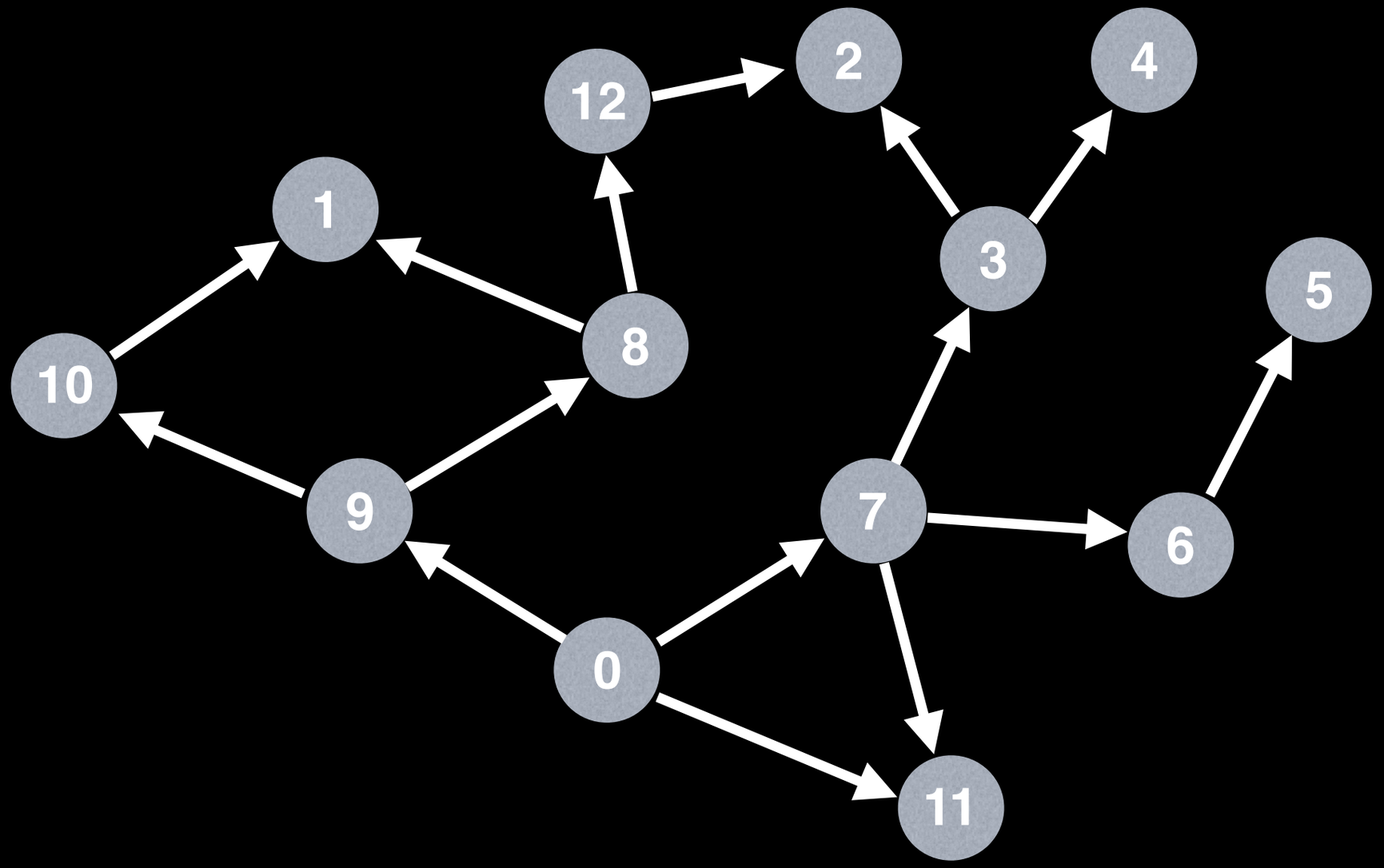
4
2
5
12
1
3
6
8
10
11
7
9
0

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



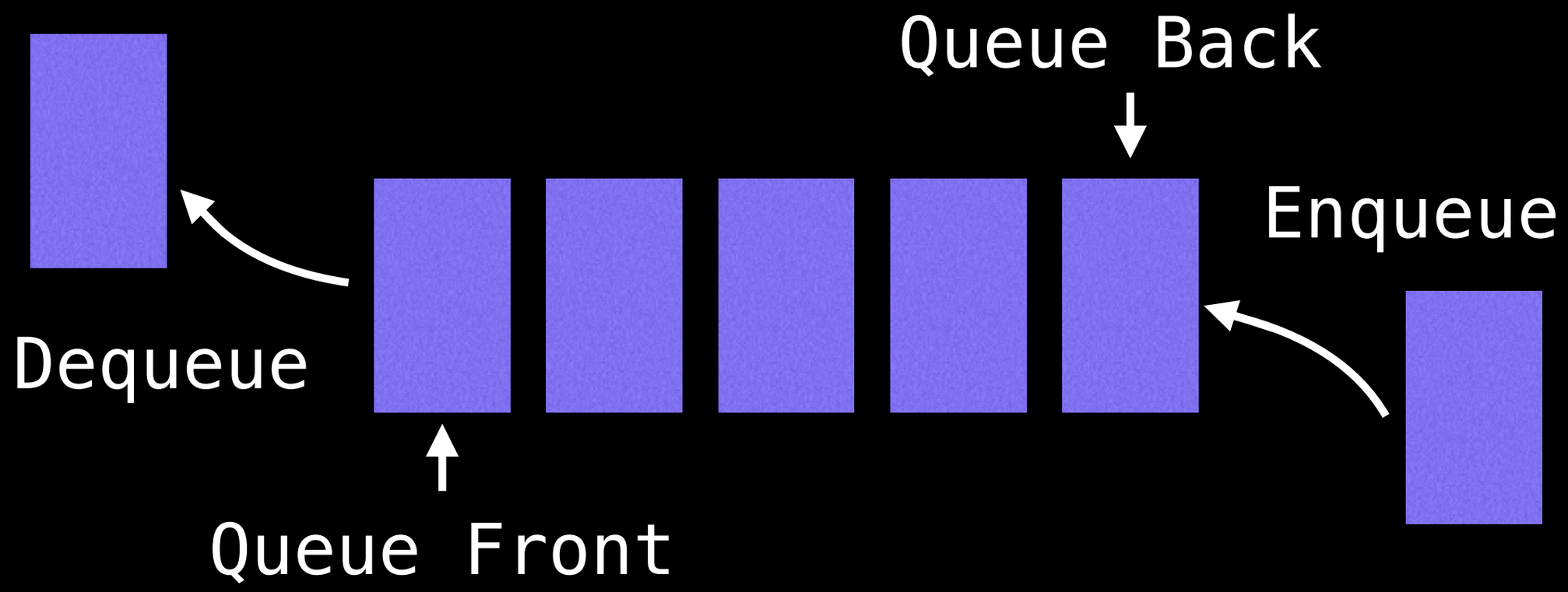
4
2
5
12
1
3
6
8
10
11
7
9
0

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



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 ≤ e,s < n
function bfs(s, e):

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

  # Return reconstructed path from s -> 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 ≤ e,s < n
function bfs(s, e):

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

  # Return reconstructed path from s -> 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 ≤ e,s < n
function bfs(s, e):

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

  # Return reconstructed path from s -> 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 ≤ e,s < n
function bfs(s, e):

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

    # Return reconstructed path from s -> 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 ≤ e,s < n
function bfs(s, e):

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

  # Return reconstructed path from s -> 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 ≤ e,s < n
function bfs(s, e):

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

  # Return reconstructed path from s -> e
  return reconstructPath(s, e, prev)
```

```
function reconstructPath(s, e):

  # 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):

    # 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):

  # 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):

  # 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 []
```