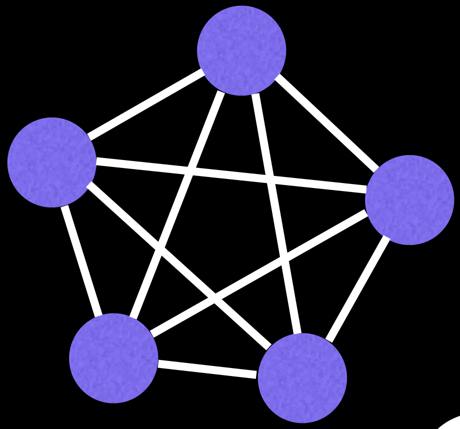
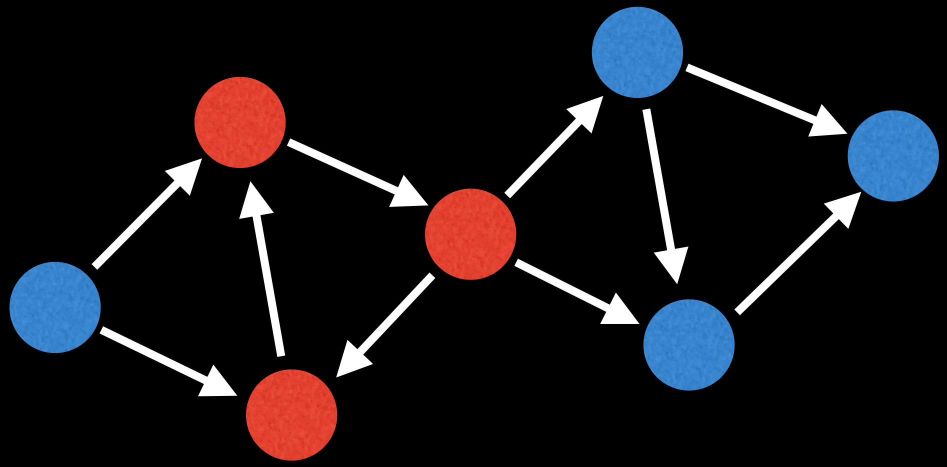
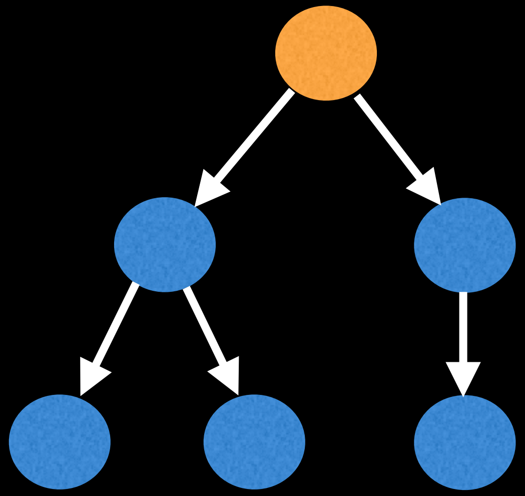
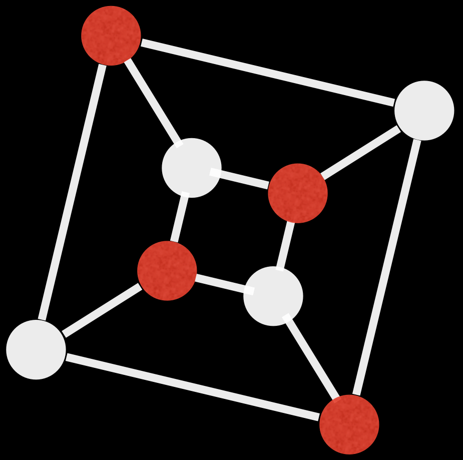
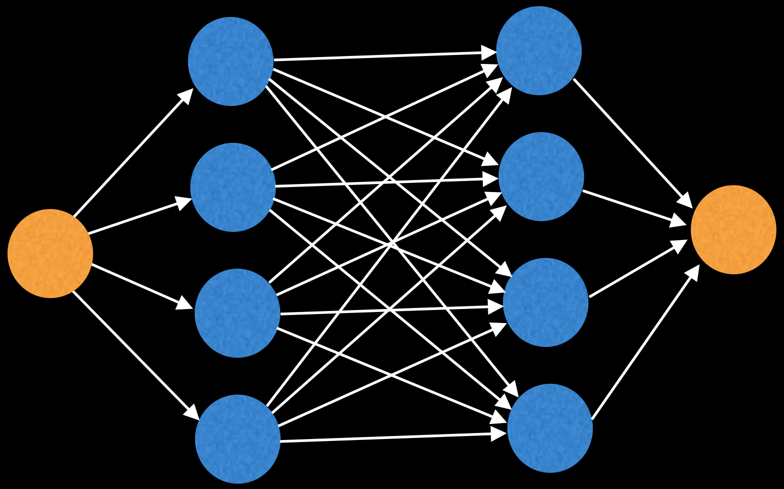
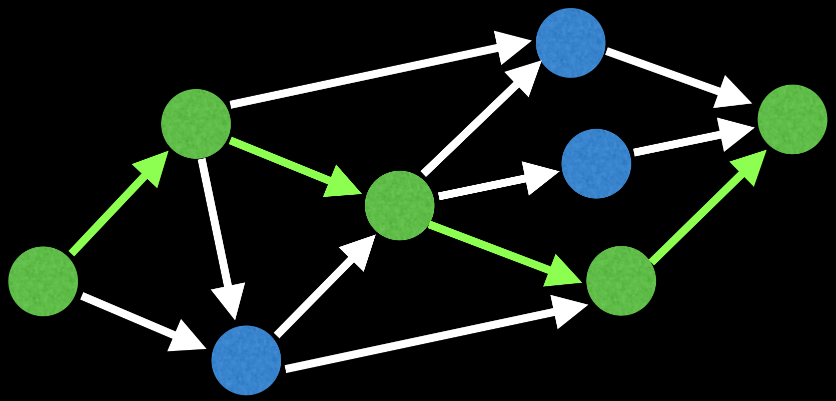




Graph Theory Video Series

Dijkstra’s Shortest Path Algorithm

William Fiset

What is Dijkstra’s algorithm?

Dijkstra’s algorithm is a Single Source Shortest Path (SSSP) algorithm for graphs with **non-negative edge weights**.

Depending on how the algorithm is implemented and what data structures are used the time complexity is typically **O(E*log(V))** which is competitive against other shortest path algorithms.

Algorithm prerequisites

One constraint for Dijkstra’s algorithm is that the graph must only contain **non-negative edge weights**. This constraint is imposed to ensure that once a node has been visited its optimal distance cannot be improved.

This is property is especially important because it enables Dijkstra’s algorithm to act in a greedy manner by always selecting the next most promising node.

Outline

The goal of this slide deck is for you to understand how to implement Dijkstra’s algorithm and implement it efficiently.

- Lazy Dijkstra’s animation
- Lazy Dijkstra’s pseudo-code
- Finding SP + stopping early optimization
- Using indexed priority queue + decreaseKey to reduce space and increase performance.
- Eager Dijkstra’s animation
- Eager Dijkstra’s pseudo-code
- Heap optimization with D-ary heap

Quick Algorithm Overview

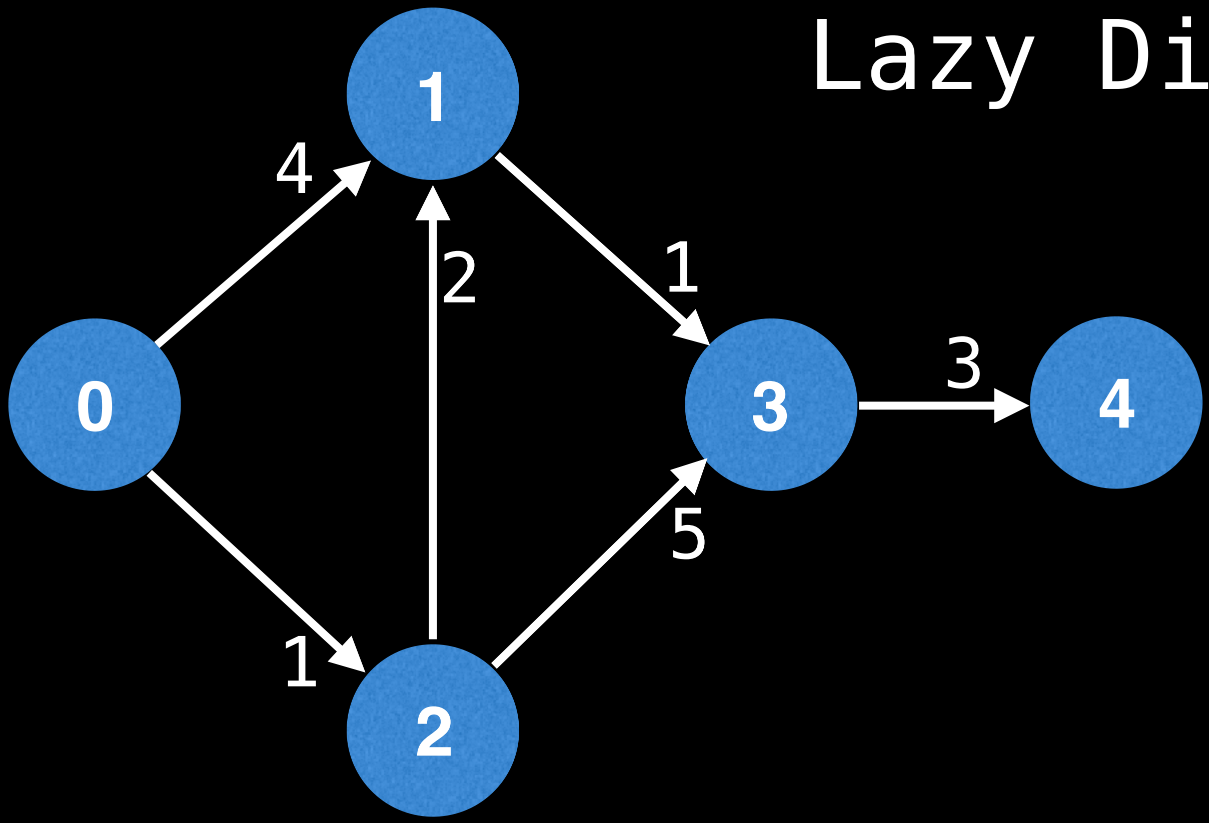
Maintain a ‘dist’ array where the distance to every node is positive infinity. Mark the distance to the start node ’s' to be 0.

Maintain a PQ of key-value pairs of (node index, distance) pairs which tell you which node to visit next based on sorted min value.

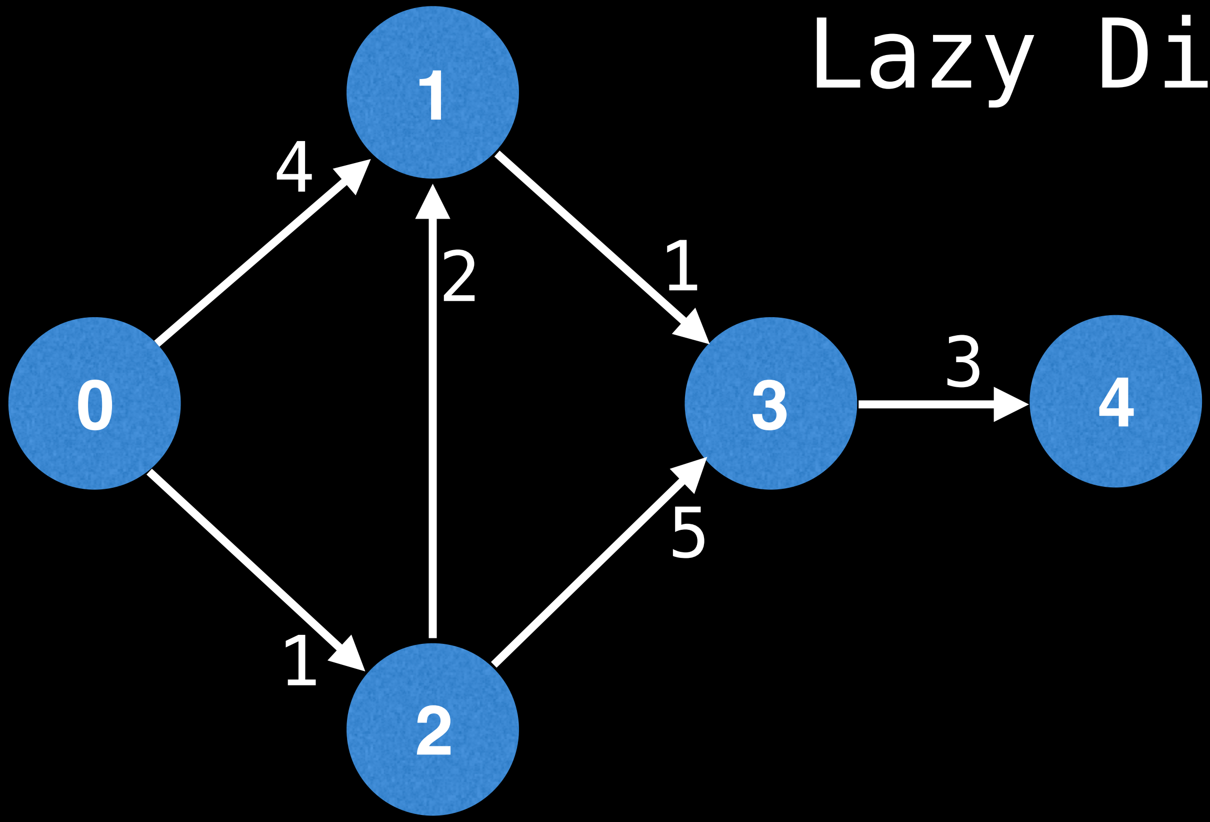
Insert (s, 0) into the PQ and loop while PQ is not empty pulling out the next most promising (node index, distance) pair.

Iterate over all edges outwards from the current node and relax each edge appending a new (node index, distance) key-value pair to the PQ for every relaxation.

Lazy Dijkstra's



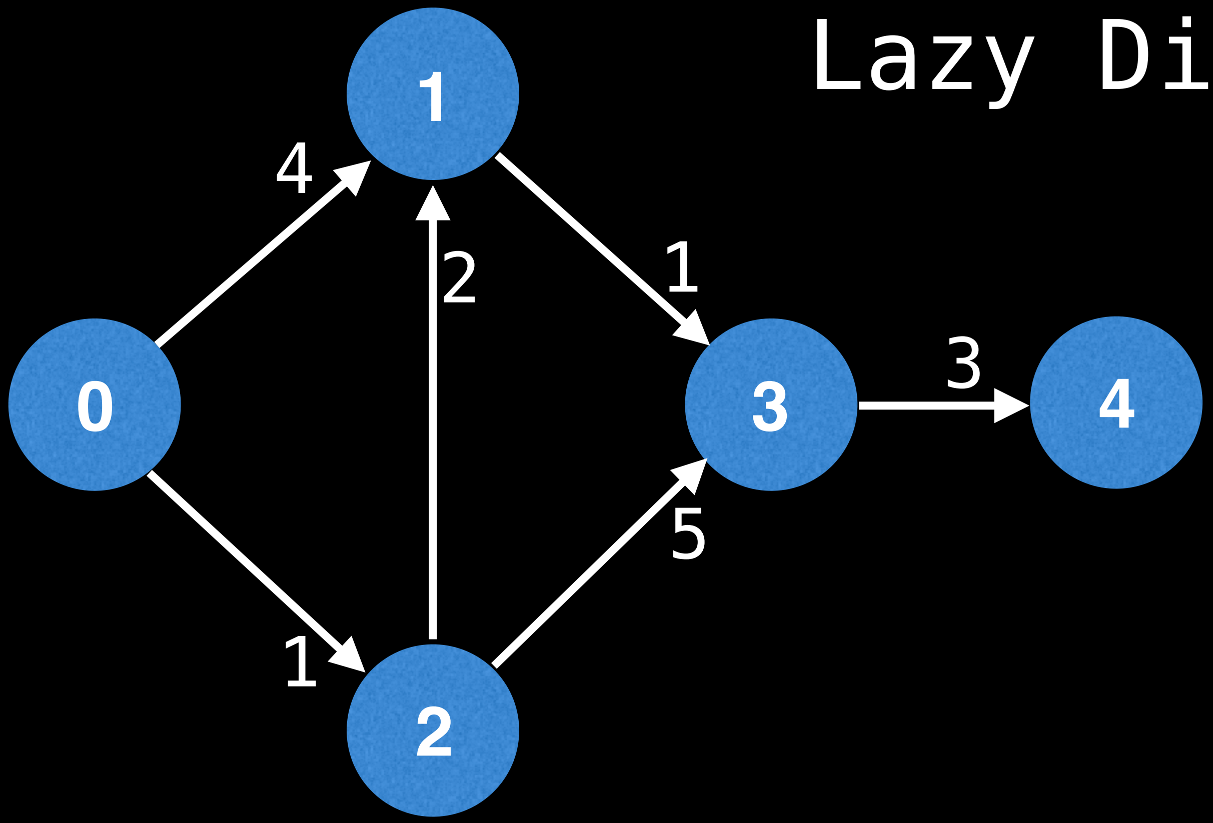
Lazy Dijkstra's



dist

0	1	2	3	4
∞	∞	∞	∞	∞

Lazy Dijkstra's

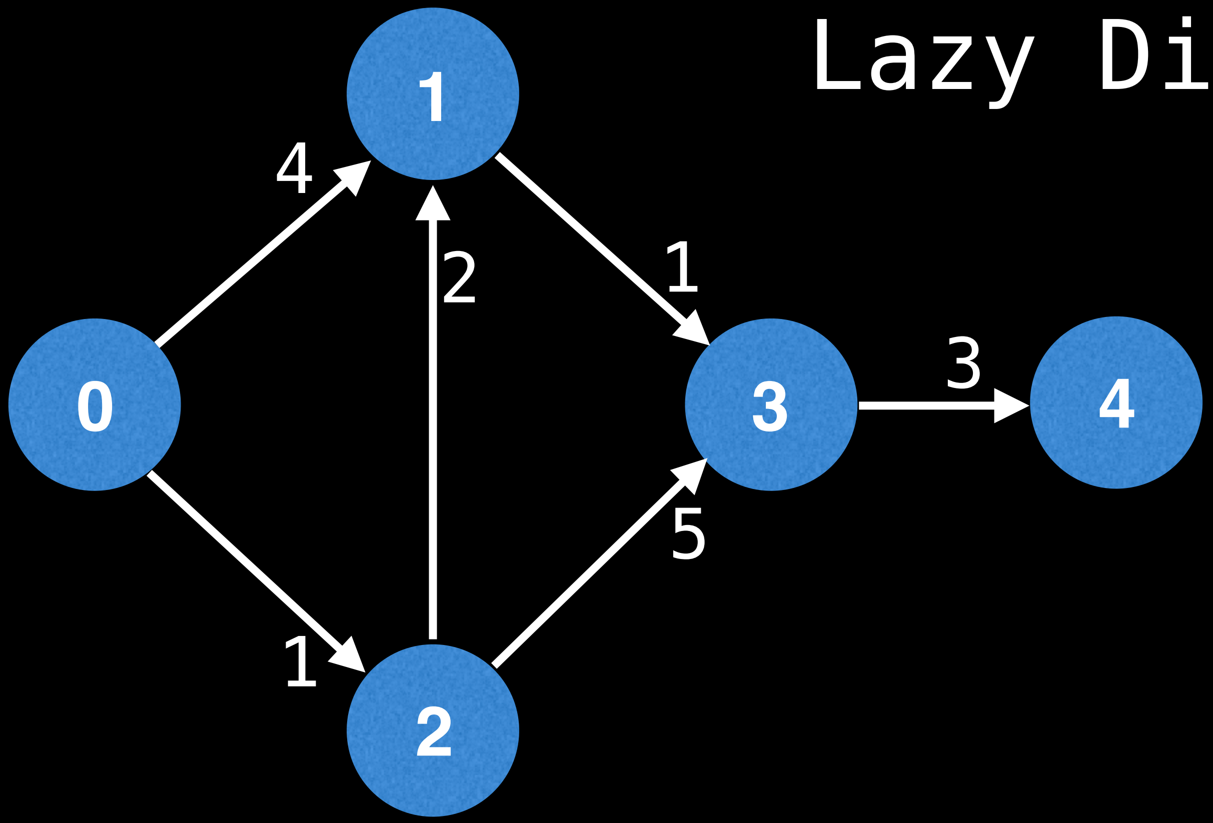


(index, dist)
key-value pairs

dist

0	1	2	3	4
∞	∞	∞	∞	∞

Lazy Dijkstra's

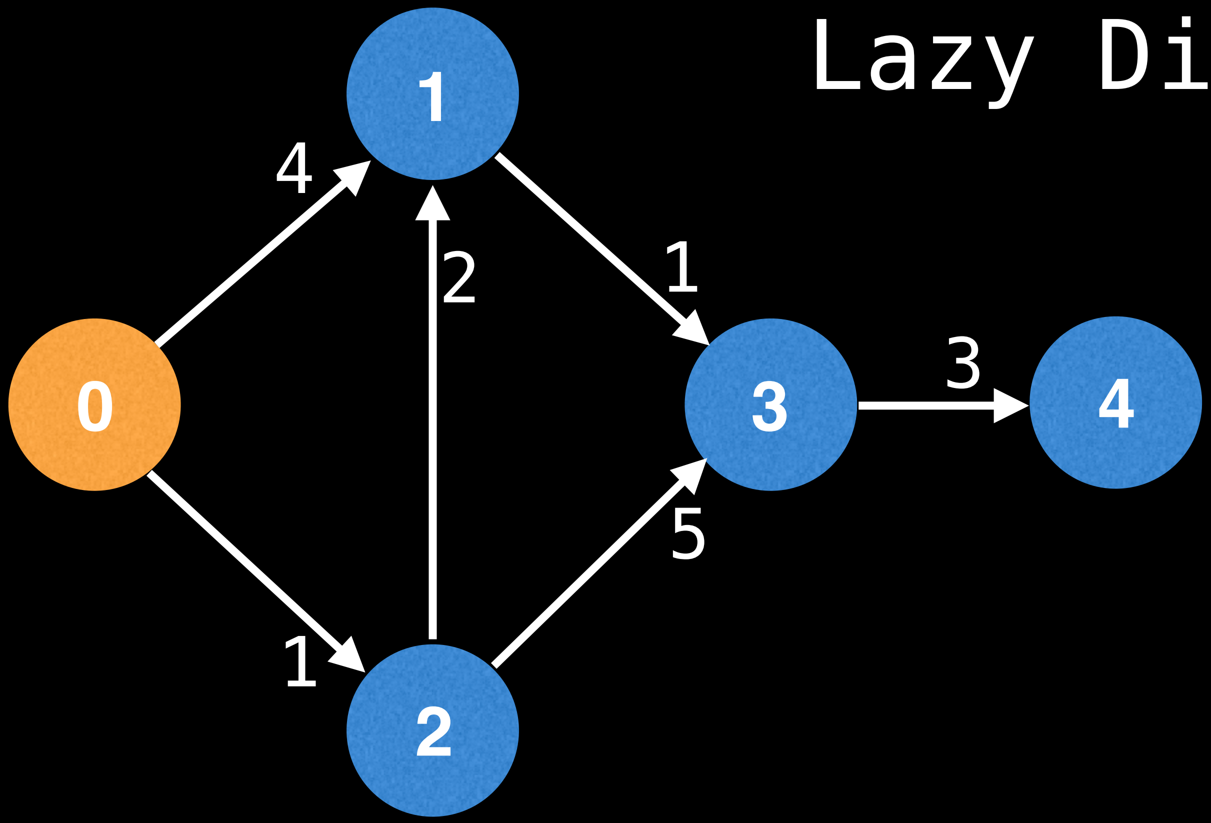


(index, dist) key-value pairs
(0, 0)

dist

0	1	2	3	4
0	∞	∞	∞	∞

Lazy Dijkstra's

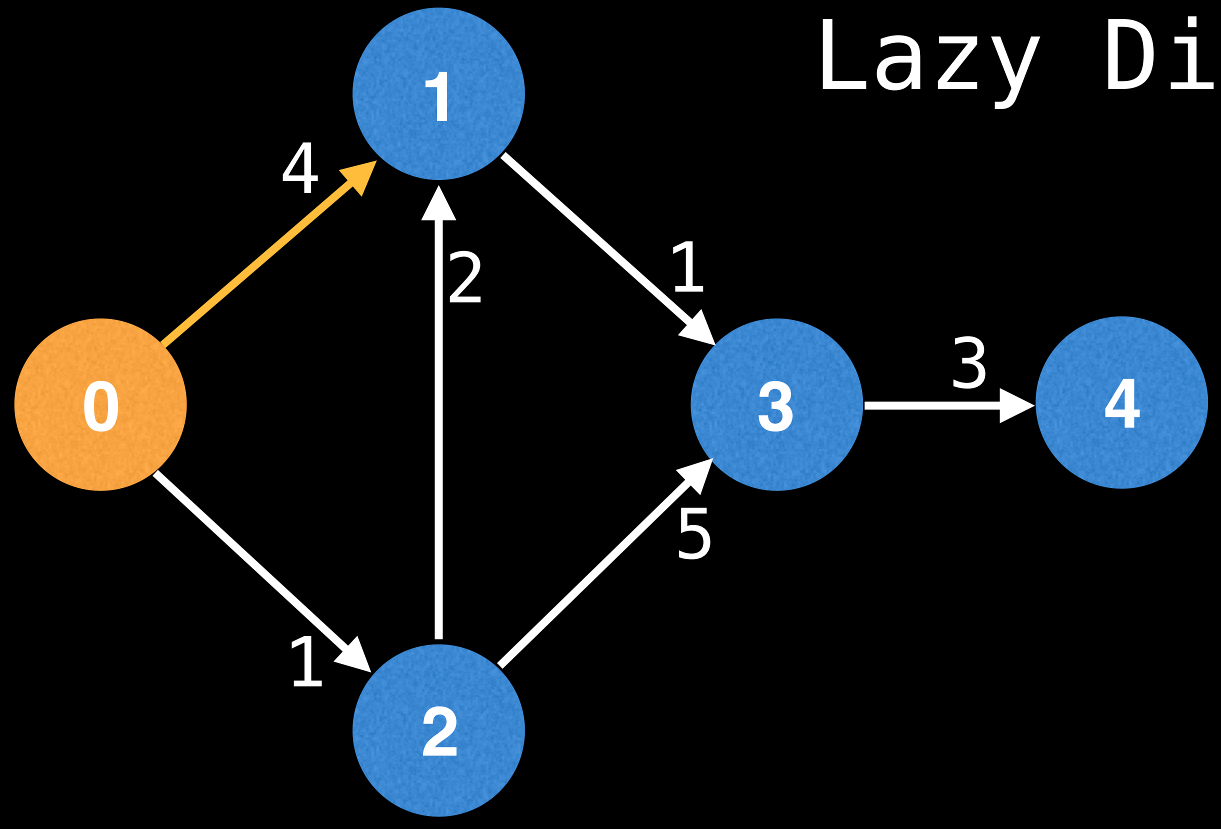


(index, dist) key-value pairs
→ (0, 0)

dist

0	1	2	3	4
0	∞	∞	∞	∞

Lazy Dijkstra's



(index, dist)
key-value pairs

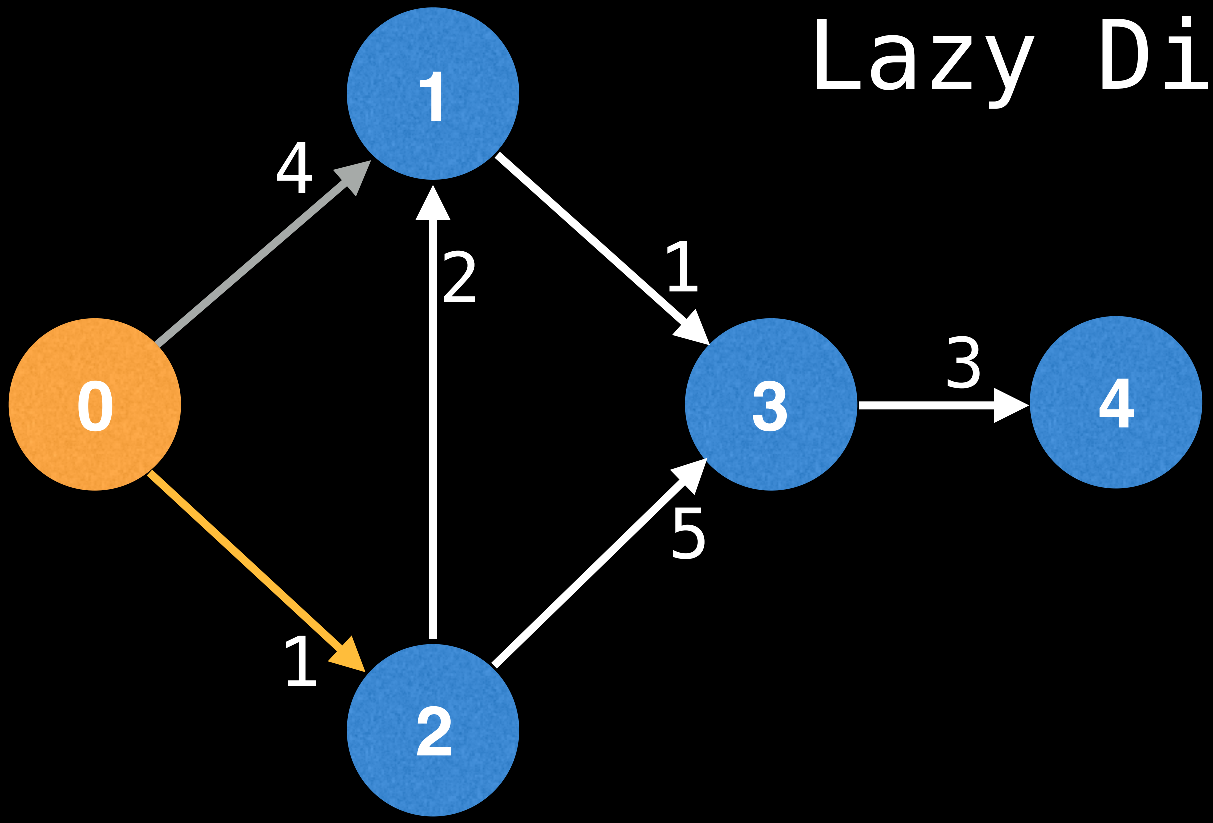
→ (0, 0)
(1, 4)

dist

0	1	2	3	4
0	4	∞	∞	∞

Best distance from node 0 to node 1 is:
dist[0] + edge.cost = 0 + 4 = 4

Lazy Dijkstra's



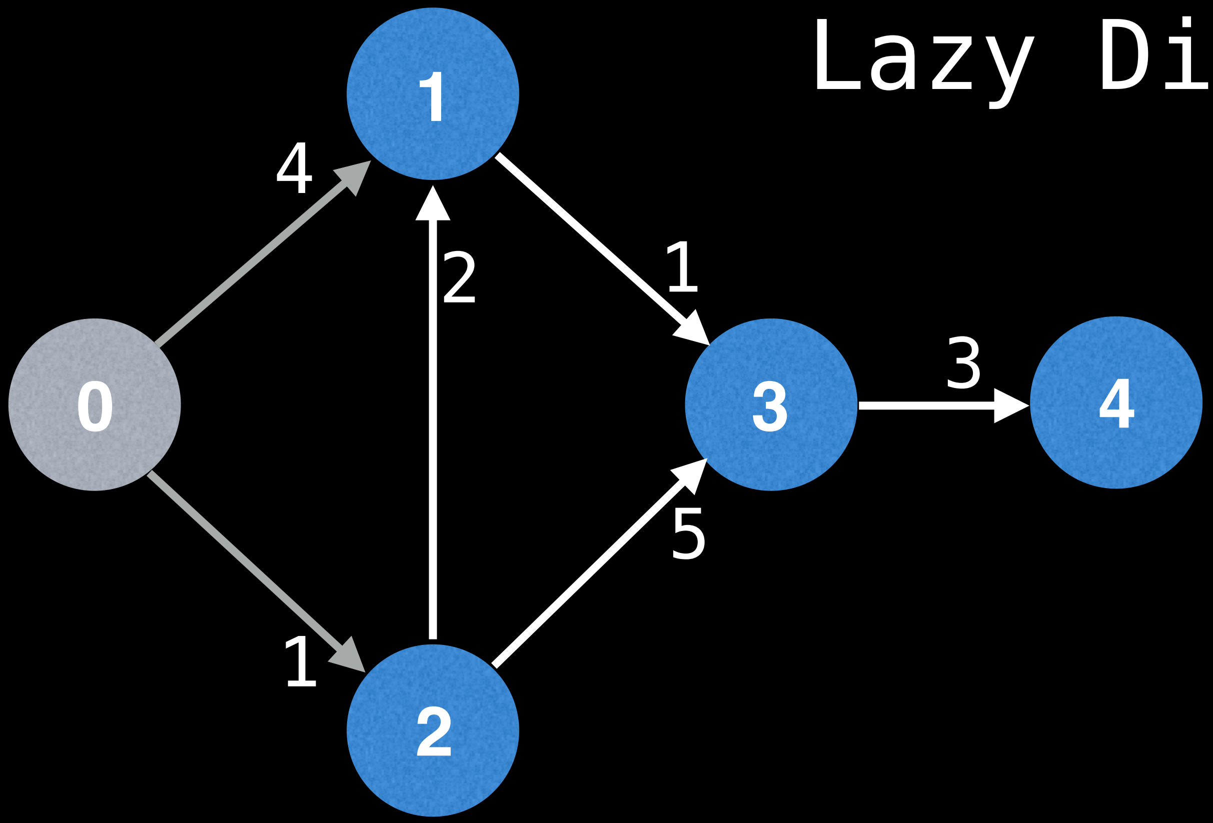
(index, dist) key-value pairs
→ (0, 0)
(1, 4)
(2, 1)

dist

0	1	2	3	4
0	4	1	∞	∞

Best distance from node 0 to node 2 is:
dist[0] + edge.cost = 0 + 1 = 1

Lazy Dijkstra's



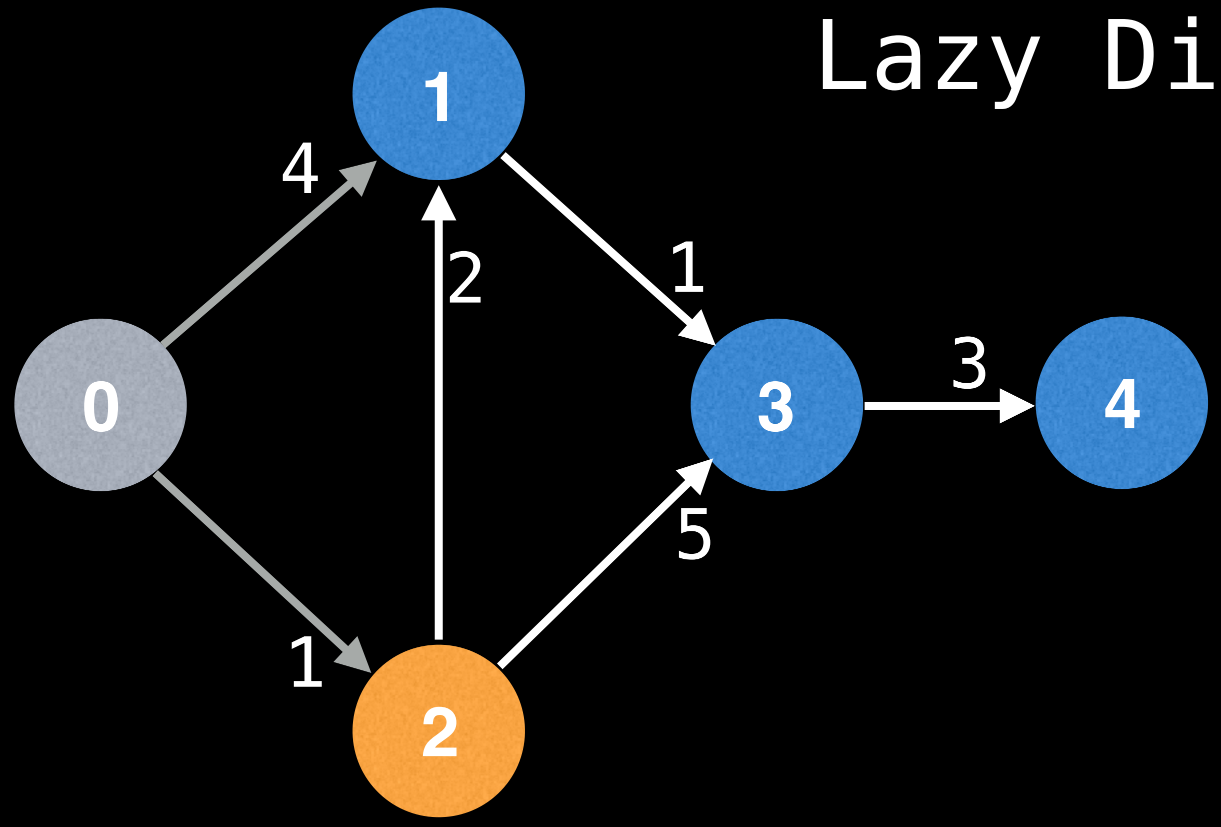
(index, dist)
key-value pairs

→ ~~(0, 0)~~
(1, 4)
(2, 1)

dist

0	1	2	3	4
0	4	1	∞	∞

Lazy Dijkstra's



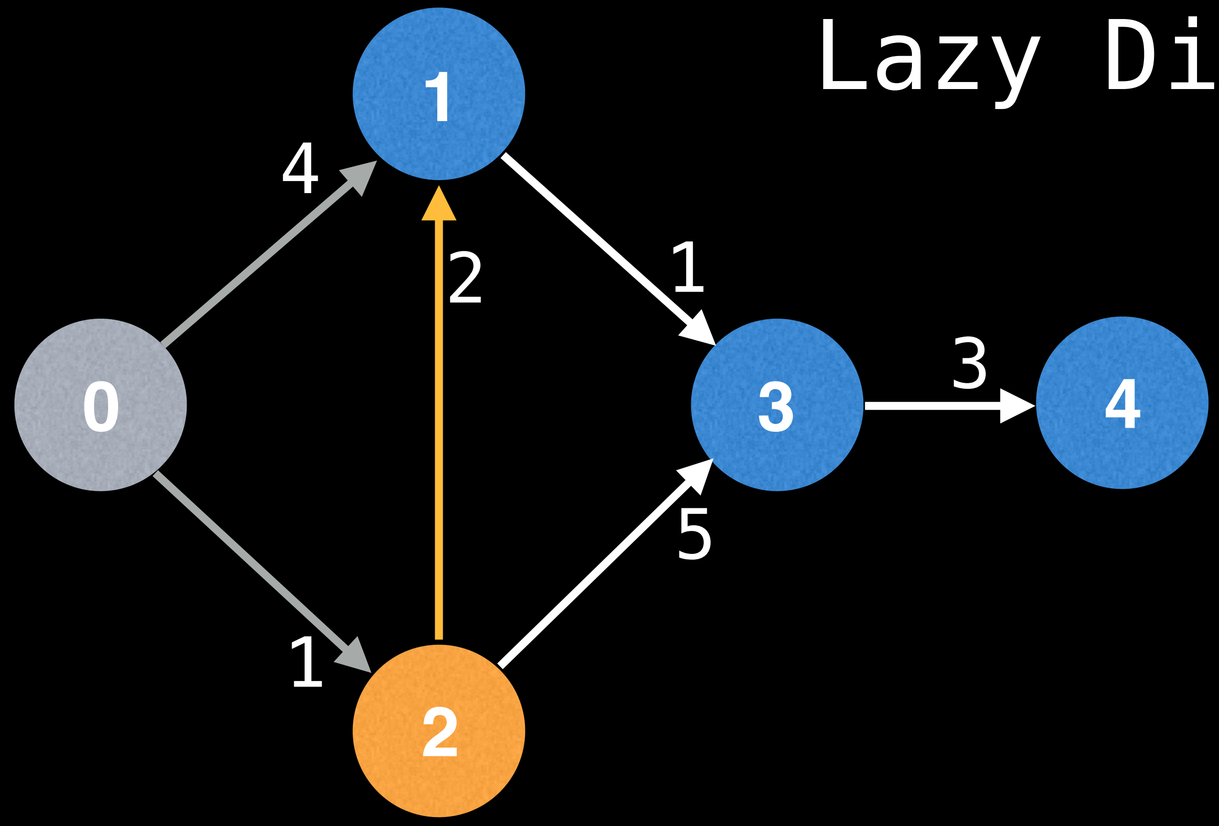
(index, dist)
key-value pairs

~~(0, 0)~~
(1, 4)
→ (2, 1)

dist

0	1	2	3	4
0	4	1	∞	∞

Lazy Dijkstra's



(index, dist)
key-value pairs

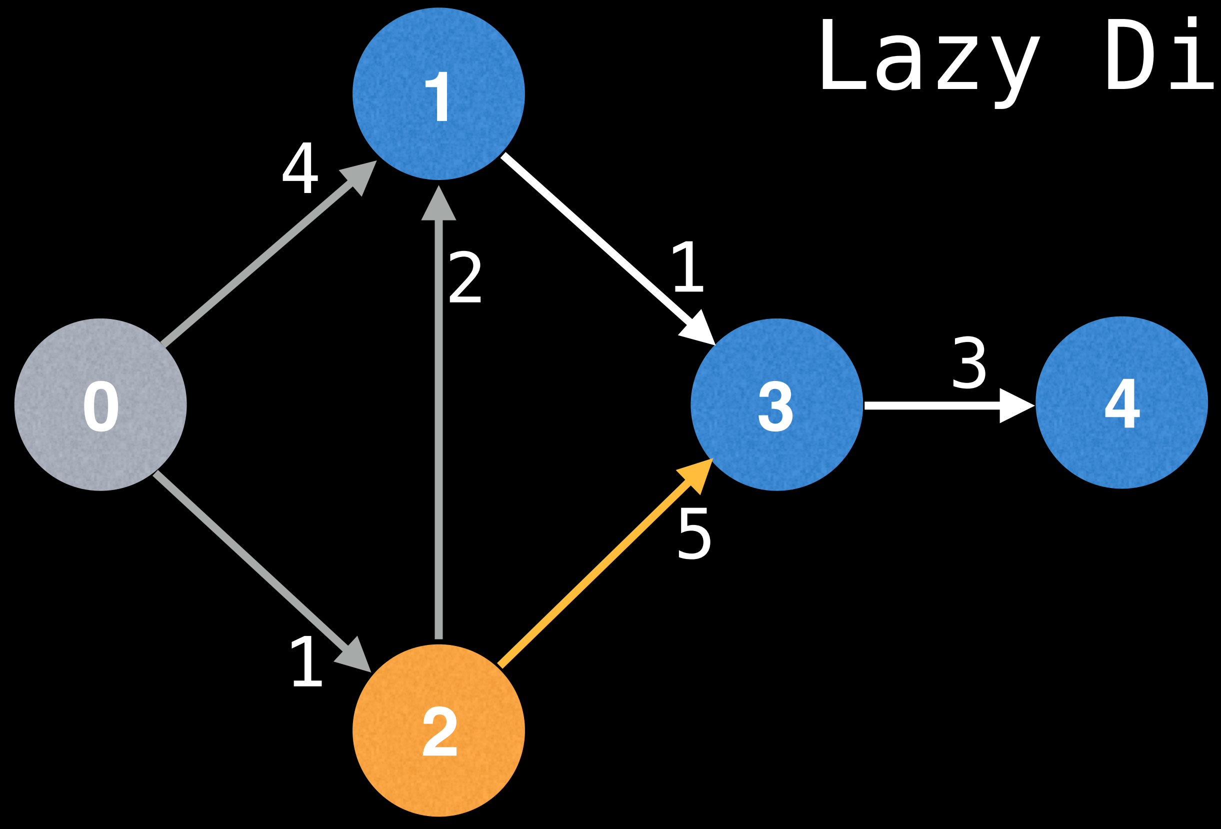
~~(0, 0)~~
(1, 4)
→ (2, 1)
(1, 3)

dist

0	1	2	3	4
0	3	1	∞	∞

Best distance from node 2 to node 1 is:
dist[2] + edge.cost = 1 + 2 = 3

Lazy Dijkstra's



(index, dist)
key-value pairs

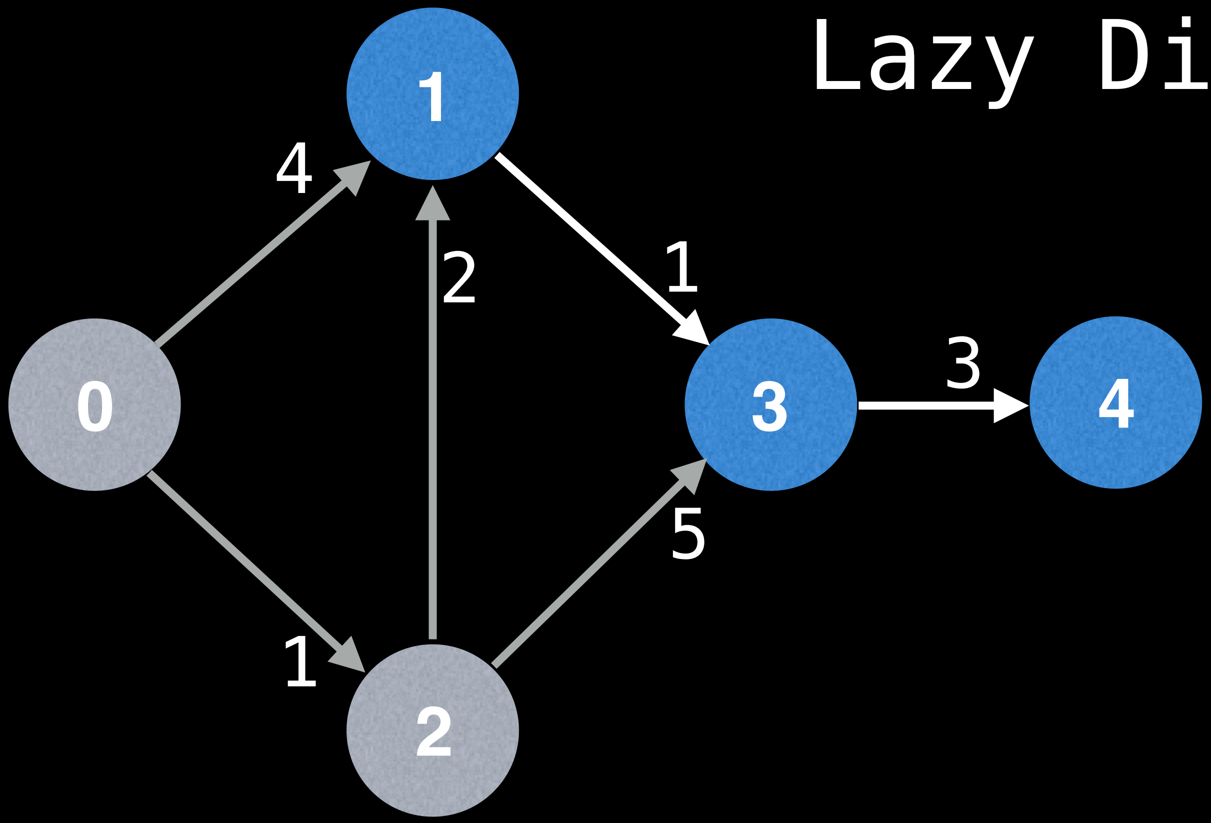
~~(0, 0)~~
(1, 4)
→ (2, 1)
(1, 3)
(3, 6)

dist

0	1	2	3	4
0	3	1	6	∞

Best distance from node 2 to node 3 is:
dist[2] + edge.cost = 1 + 5 = 6

Lazy Dijkstra's

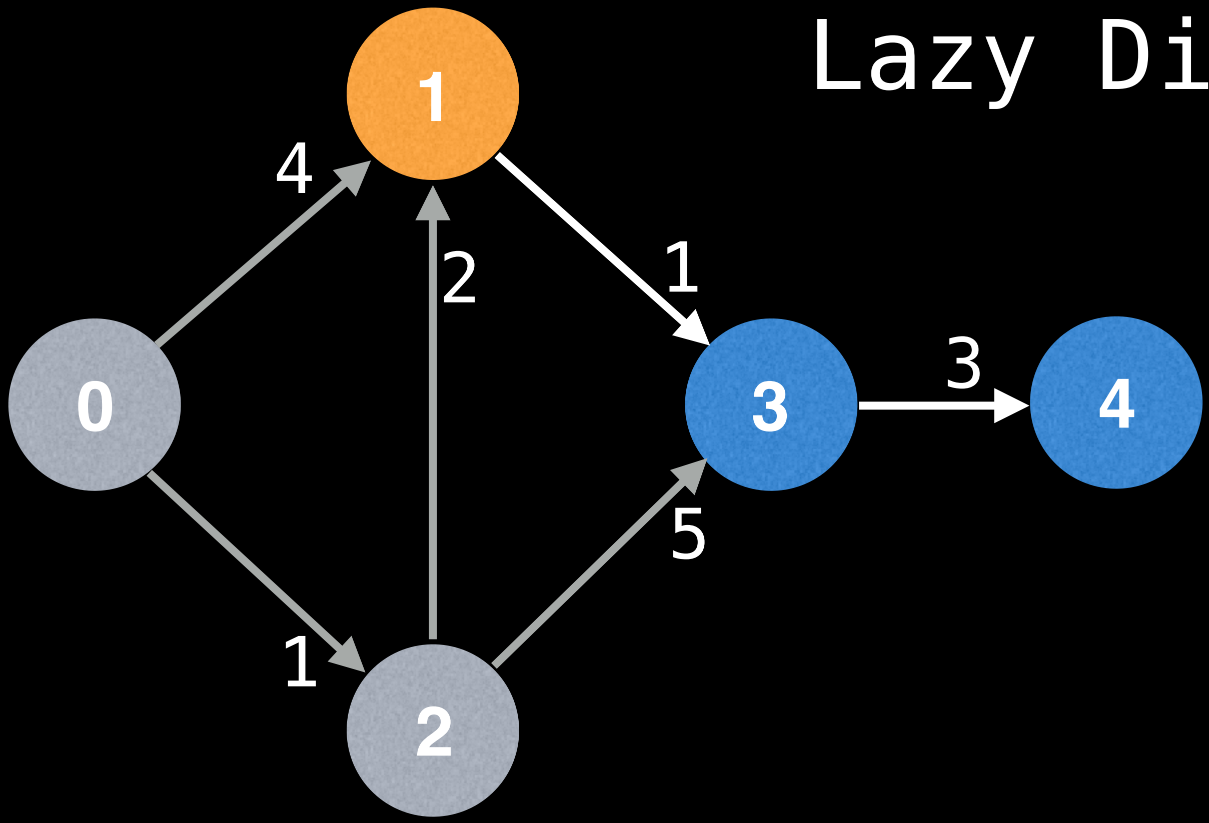


(index, dist) key-value pairs
~~(0, 0)~~
(1, 4)
→ ~~(2, 1)~~
(1, 3)
(3, 6)

dist

0	1	2	3	4
0	3	1	6	∞

Lazy Dijkstra's



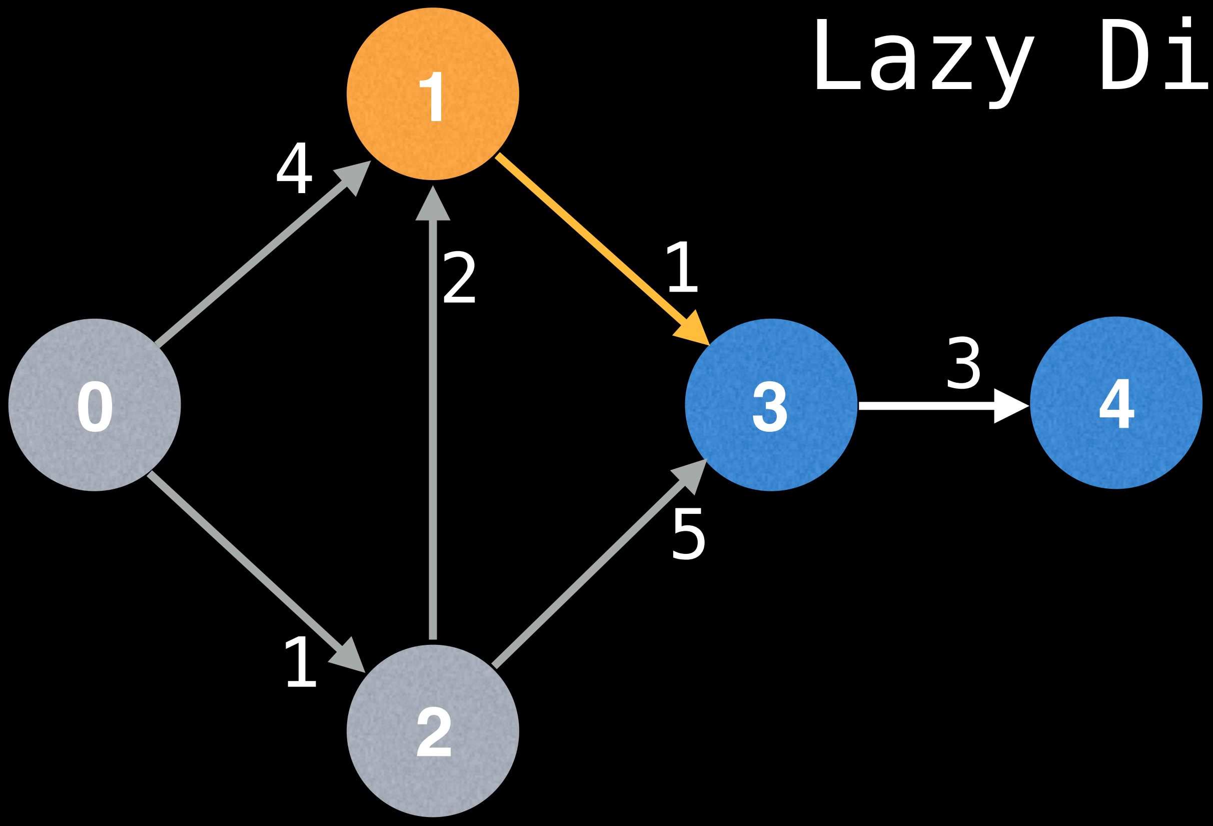
(index, dist)
key-value pairs

~~(0, 0)~~
(1, 4)
~~(2, 1)~~
→ (1, 3)
(3, 6)

dist

0	1	2	3	4
0	3	1	6	∞

Lazy Dijkstra's



(index, dist)
key-value pairs

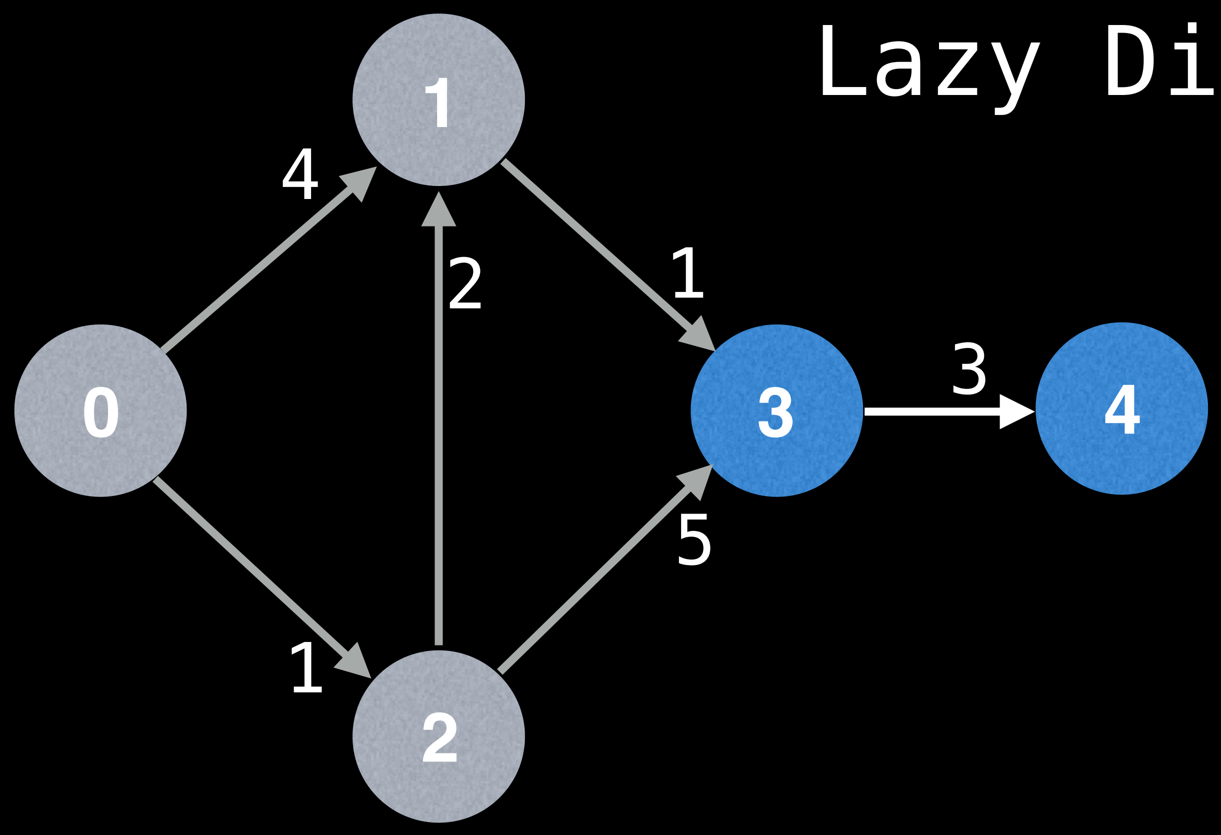
~~(0, 0)~~
(1, 4)
~~(2, 1)~~
→ (1, 3)
(3, 6)
(3, 4)

dist

0	1	2	3	4
0	3	1	4	∞

Best distance from node 1 to node 3 is:
dist[1] + edge.cost = 3 + 1 = 4

Lazy Dijkstra's

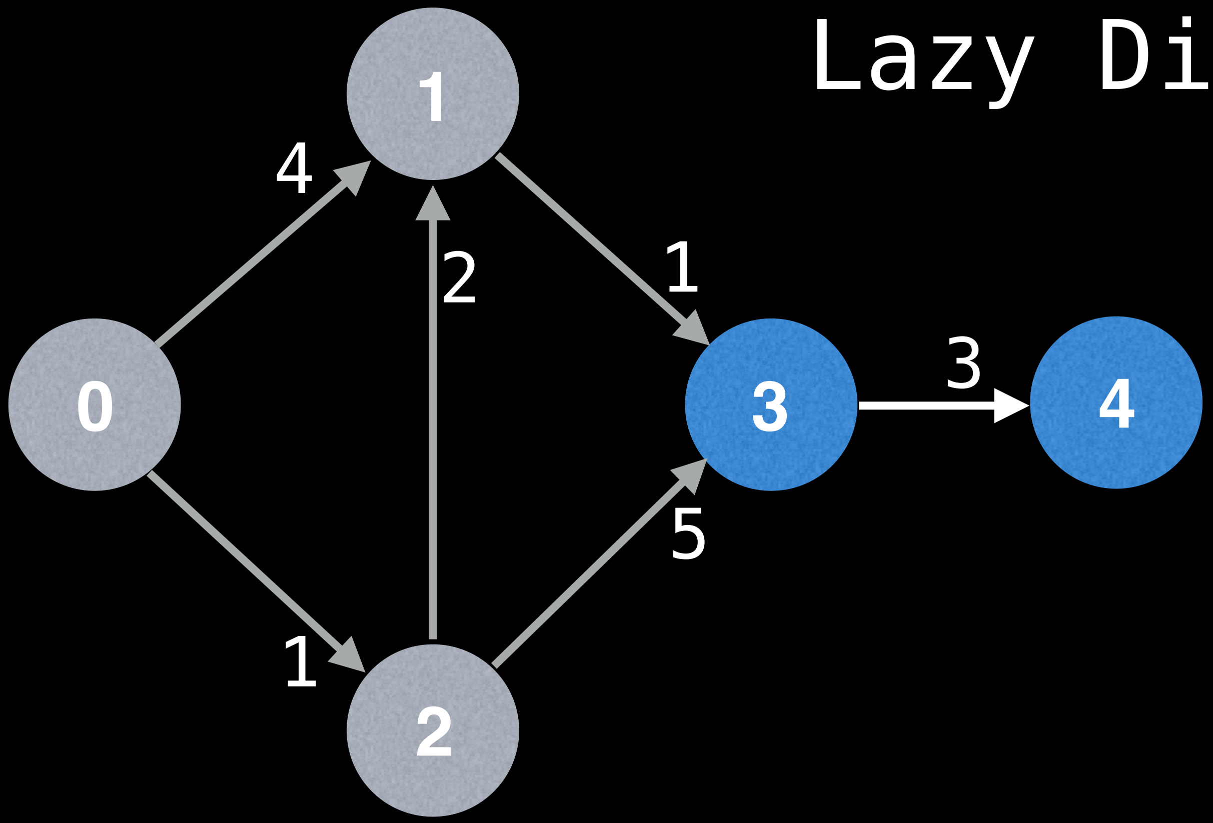


(index, dist) key-value pairs
~~(0, 0)~~
(1, 4)
~~(2, 1)~~
→ ~~(1, 3)~~
(3, 6)
(3, 4)

dist

0	1	2	3	4
0	3	1	4	∞

Lazy Dijkstra's



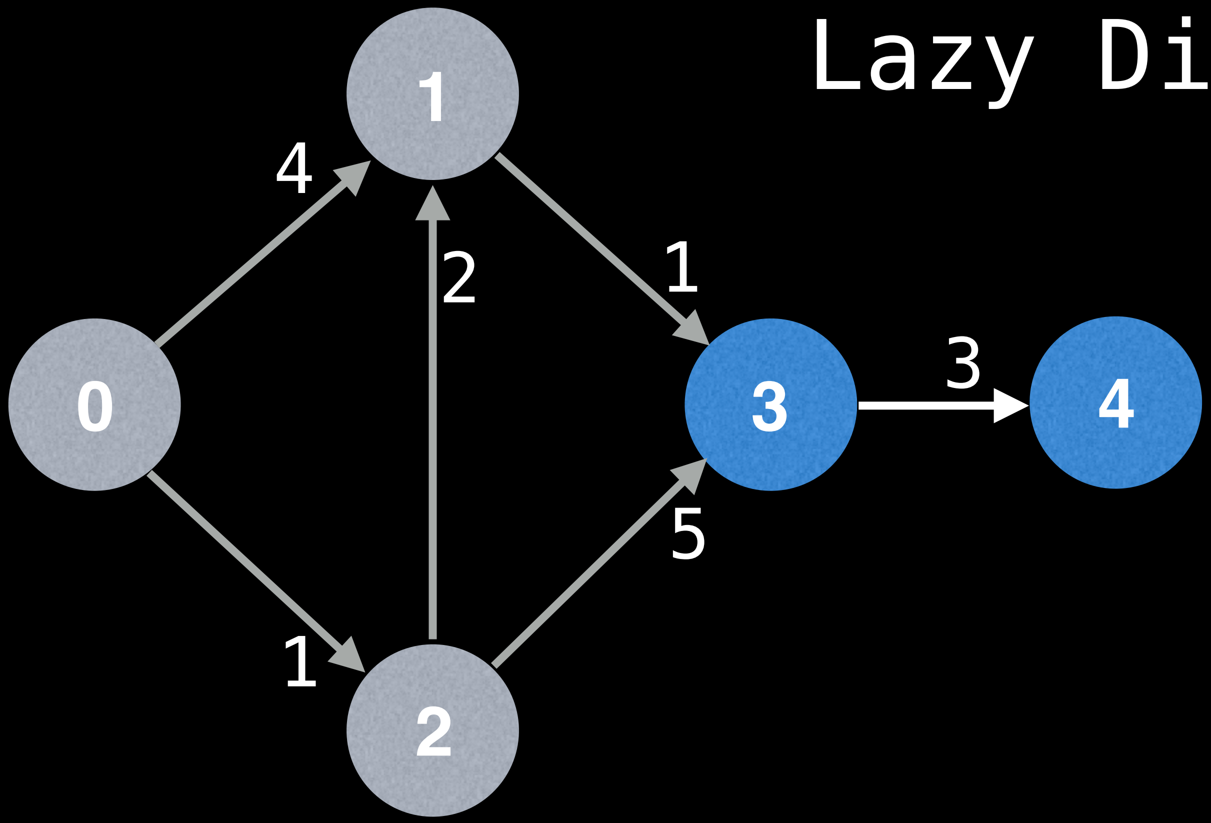
(index, dist)
key-value pairs

~~(0, 0)~~
→ (1, 4)
~~(2, 1)~~
~~(1, 3)~~
(3, 6)
(3, 4)

dist

0	1	2	3	4
0	3	1	4	∞

Lazy Dijkstra's



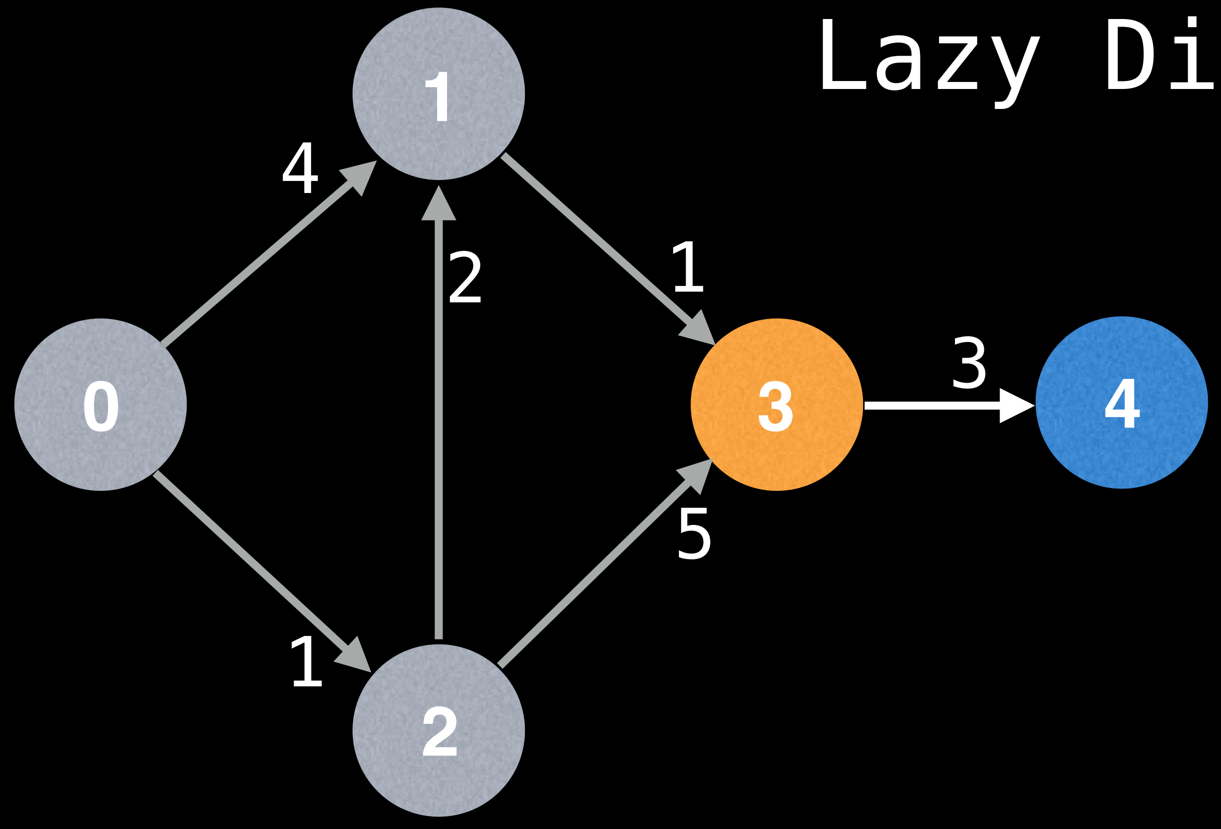
(index, dist) key-value pairs
~~(0, 0)~~
→ (1, $\boxed{4}$)
~~(2, 1)~~
~~(1, 3)~~
(3, 6)
(3, 4)

dist

0	1	2	3	4
0	$\boxed{3}$	1	4	∞

We have already found a better route to get to node 1 (since dist[1] has value 3) so we can ignore this entry in the PQ.

Lazy Dijkstra's



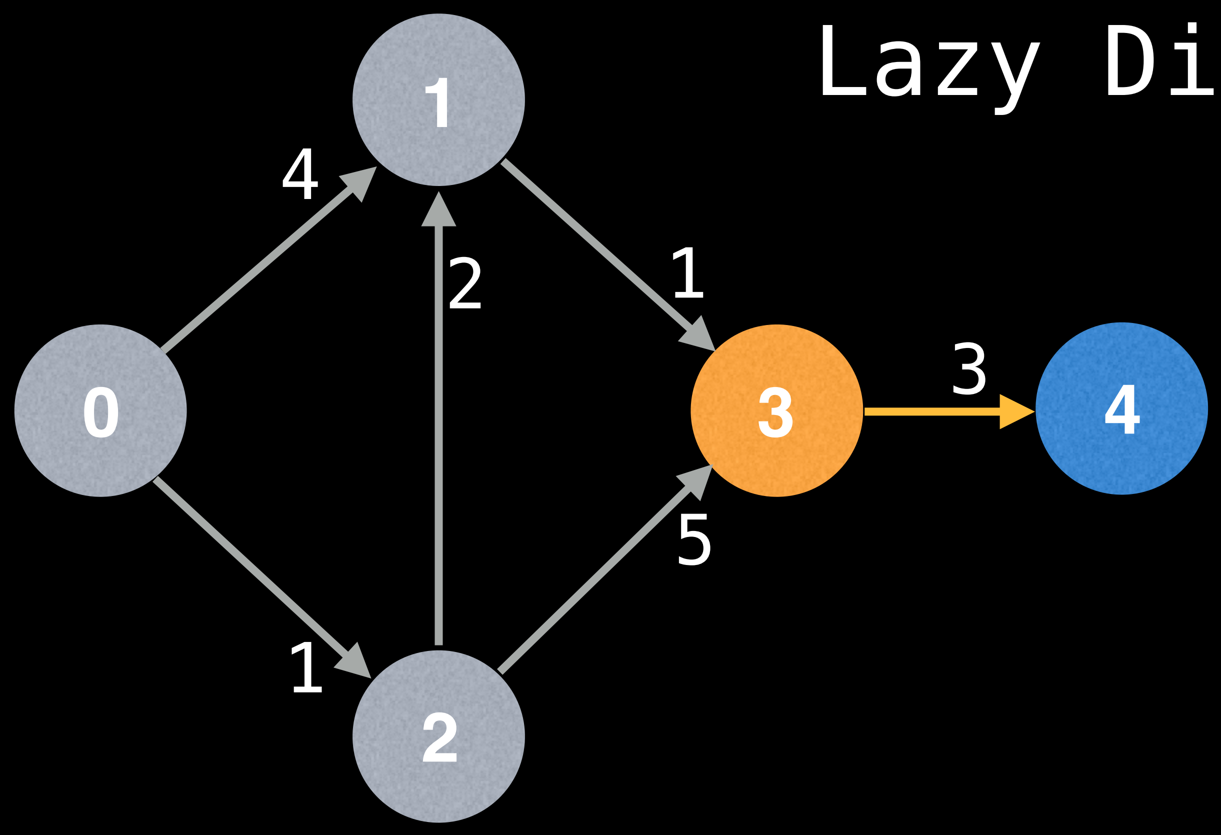
(index, dist)
key-value pairs

~~(0, 0)~~
~~(1, 4)~~
~~(2, 1)~~
~~(1, 3)~~
(3, 6)
→ (3, 4)

dist

0	1	2	3	4
0	3	1	4	∞

Lazy Dijkstra's



(index, dist)
key-value pairs

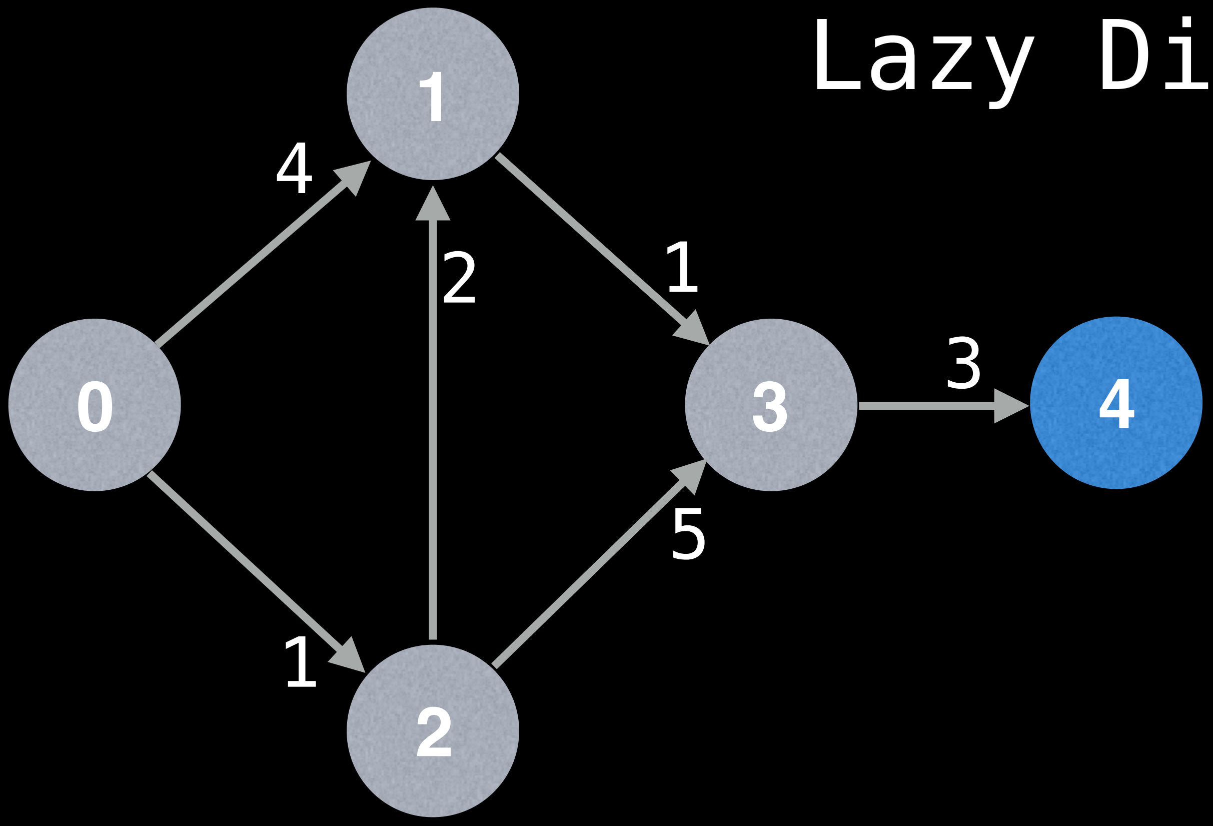
~~(0, 0)~~
~~(1, 4)~~
~~(2, 1)~~
~~(1, 3)~~
(3, 6)
→ (3, 4)
(4, 7)

dist

0	1	2	3	4
0	3	1	4	7

Best distance from node 3 to node 4 is:
dist[3] + edge.cost = 4 + 3 = 7

Lazy Dijkstra's

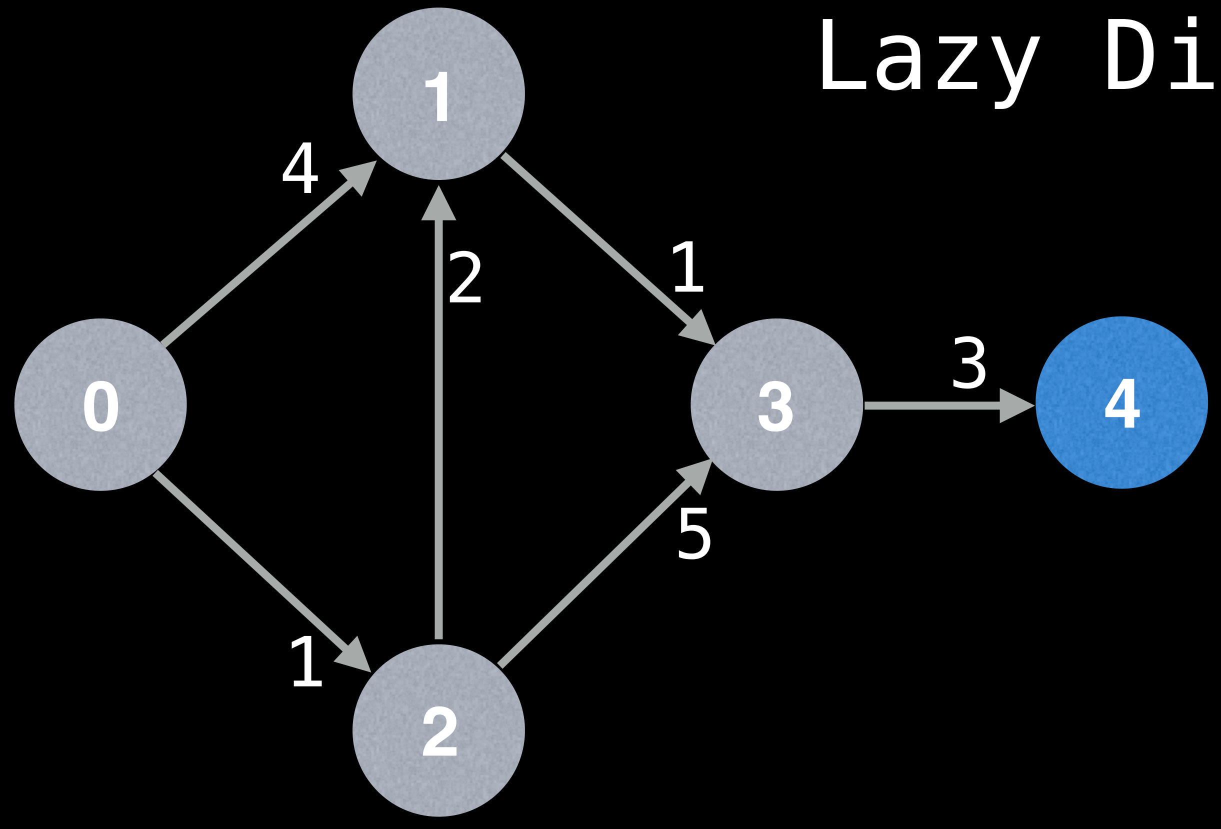


(index, dist) key-value pairs
~~(0, 0)~~
~~(1, 4)~~
~~(2, 1)~~
~~(1, 3)~~
(3, 6)
→ ~~(3, 4)~~
(4, 7)

dist

0	1	2	3	4
0	3	1	4	7

Lazy Dijkstra's

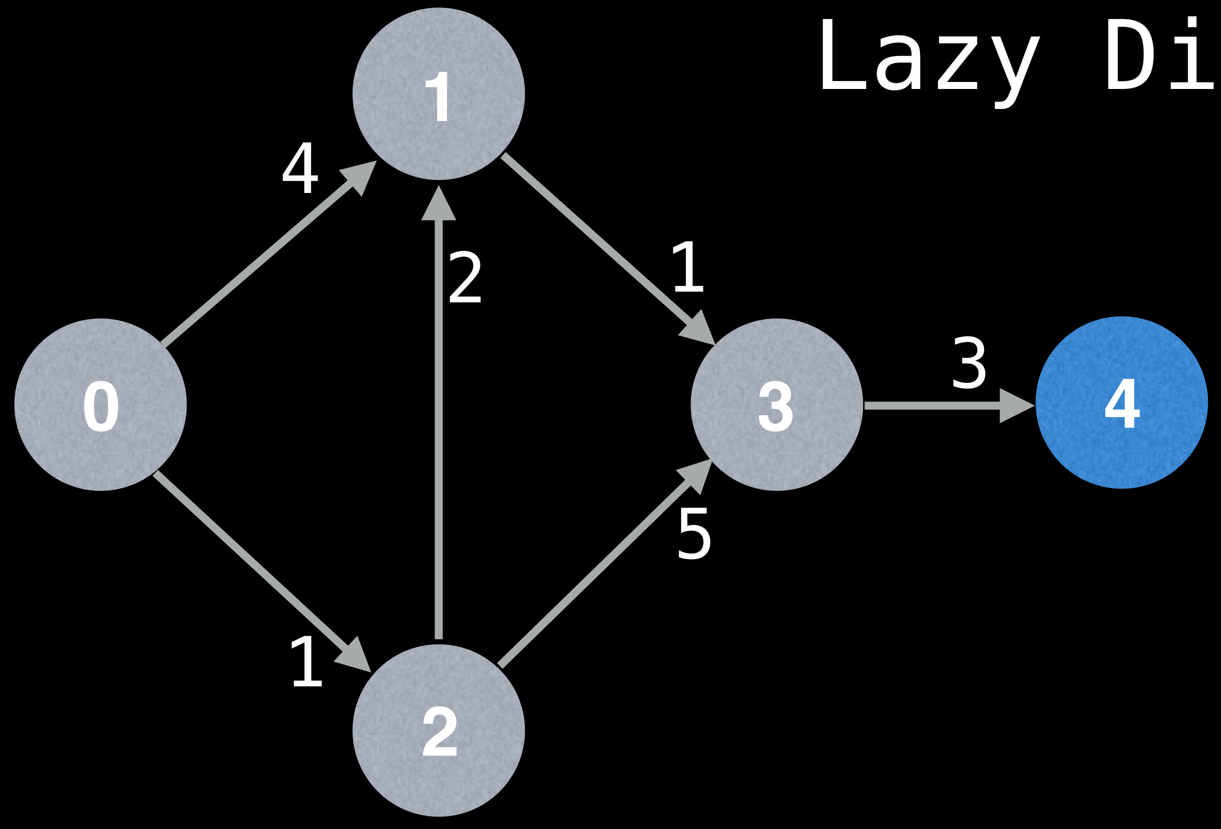


(index, dist) key-value pairs
~~(0, 0)~~
~~(1, 4)~~
~~(2, 1)~~
~~(1, 3)~~
→ (3, 6)
~~(3, 4)~~
(4, 7)

dist

0	1	2	3	4
0	3	1	4	7

Lazy Dijkstra's



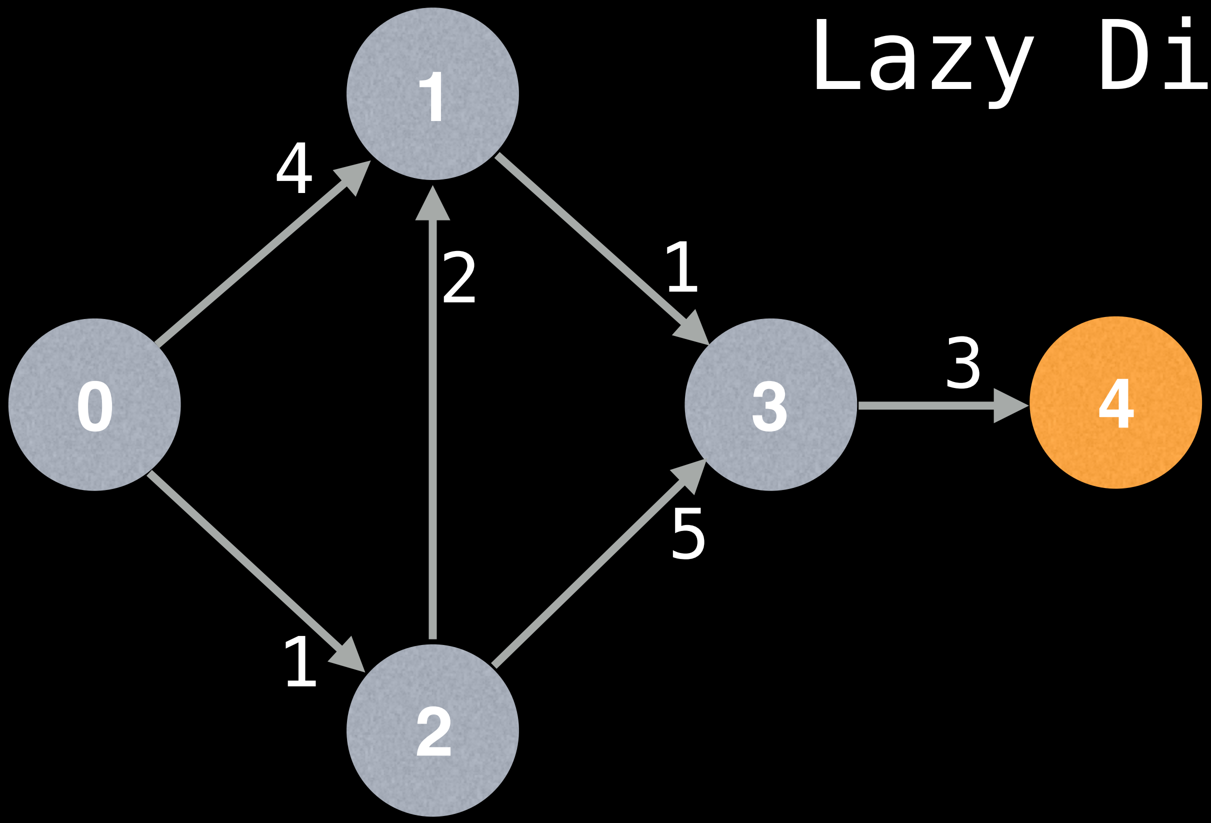
(index, dist) key-value pairs
~~(0, 0)~~
~~(1, 4)~~
~~(2, 1)~~
~~(1, 3)~~
→ (3, $\boxed{6}$)
~~(3, 4)~~
(4, 7)

dist

0	1	2	3	4
0	3	1	$\boxed{4}$	7

We have already found a better route to get to node 3 (since dist[3] has value 4) so we can ignore this entry in the PQ.

Lazy Dijkstra's



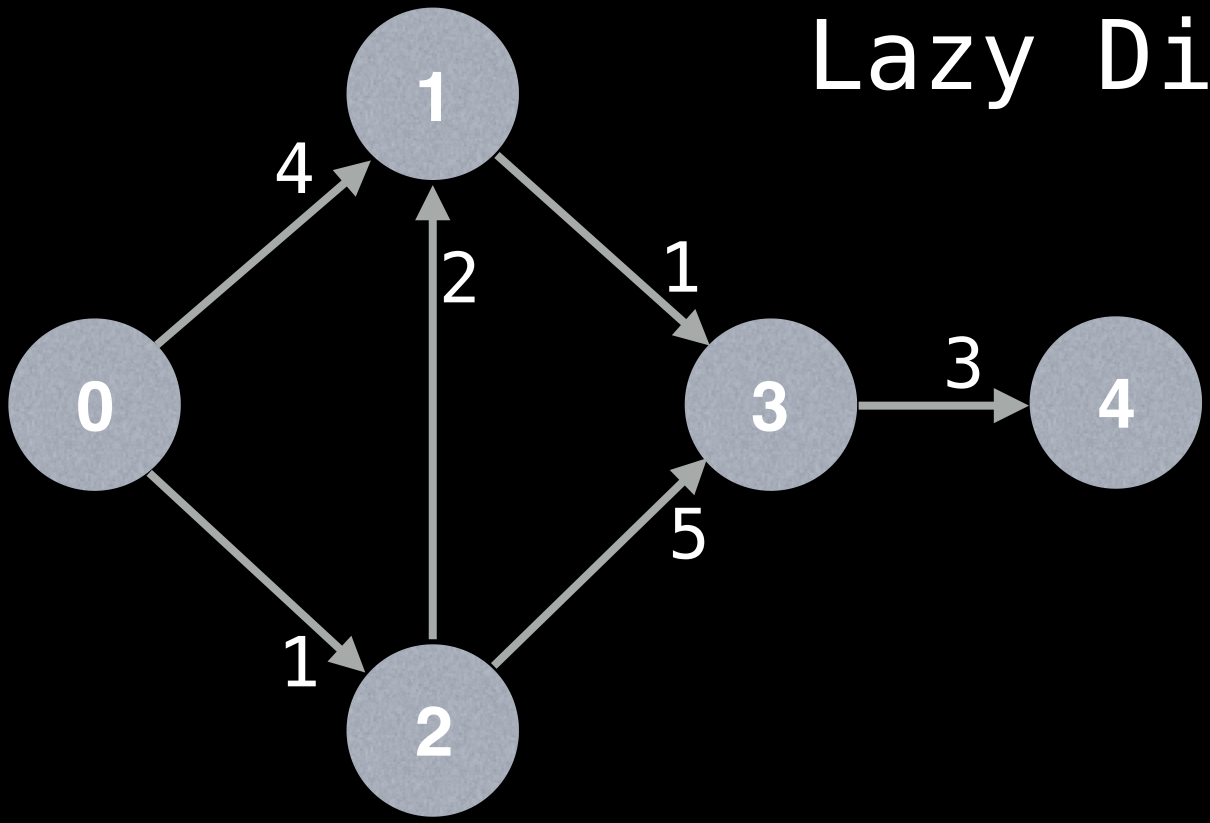
(index, dist)
key-value pairs

~~(0, 0)~~
~~(1, 4)~~
~~(2, 1)~~
~~(1, 3)~~
~~(3, 6)~~
~~(3, 4)~~
→ (4, 7)

dist

0	1	2	3	4
0	3	1	4	7

Lazy Dijkstra's



(index, dist)
key-value pairs

~~(0, 0)~~
~~(1, 4)~~
~~(2, 1)~~
~~(1, 3)~~
~~(3, 6)~~
~~(3, 4)~~
~~(4, 7)~~

dist

0	1	2	3	4
0	3	1	4	7

```
# Runs Dijkstra’s algorithm and returns an array that contains
# the shortest distance to every node from the start node s.
# g - adjacency list of weighted graph
# n - the number of nodes in the graph
# s - the index of the starting node (0 ≤ s < n)
function dijkstra(g, n, s):
  vis = [false, false, … , false] # size n
  dist = [∞, ∞, … ∞, ∞] # size n
  dist[s] = 0
  pq = empty priority queue
  pq.insert((s, 0))
  while pq.size() != 0:
    index, minValue = pq.poll()
    vis[index] = true
    for (edge : g[index]):
      if vis[edge.to]: continue
      newDist = dist[index] + edge.cost
      if newDist < dist[edge.to]:
        dist[edge.to] = newDist
        pq.insert((edge.to, newDist))
  return dist
```

```
# Runs Dijkstra's algorithm and returns an array that contains
# the shortest distance to every node from the start node s.
# g - adjacency list of weighted graph
# n - the number of nodes in the graph
# s - the index of the starting node (0 ≤ s < n)
function dijkstra(g, n, s):
  vis = [false, false, … , false] # size n
  dist = [∞, ∞, … ∞, ∞] # size n
  dist[s] = 0
  pq = empty priority queue
  pq.insert((s, 0))
  while pq.size() != 0:
    index, minValue = pq.poll()
    vis[index] = true
    for (edge : g[index]):
      if vis[edge.to]: continue
      newDist = dist[index] + edge.cost
      if newDist < dist[edge.to]:
        dist[edge.to] = newDist
        pq.insert((edge.to, newDist))
  return dist
```

```
# Runs Dijkstra's algorithm and returns an array that contains
# the shortest distance to every node from the start node s.
# g - adjacency list of weighted graph
# n - the number of nodes in the graph
# s - the index of the starting node (0 ≤ s < n)
function dijkstra(g, n, s):
  vis = [false, false, … , false] # size n
  dist = [∞, ∞, … ∞, ∞] # size n
  dist[s] = 0
  pq = empty priority queue
  pq.insert((s, 0))
  while pq.size() != 0:
    index, minValue = pq.poll()
    vis[index] = true
    for (edge : g[index]):
      if vis[edge.to]: continue
      newDist = dist[index] + edge.cost
      if newDist < dist[edge.to]:
        dist[edge.to] = newDist
        pq.insert((edge.to, newDist))
  return dist
```

```
# Runs Dijkstra’s algorithm and returns an array that contains
# the shortest distance to every node from the start node s.
# g - adjacency list of weighted graph
# n - the number of nodes in the graph
# s - the index of the starting node (0 ≤ s < n)
function dijkstra(g, n, s):
  vis = [false, false, … , false] # size n
  dist = [∞, ∞, … ∞, ∞] # size n
  dist[s] = 0
  pq = empty priority queue
  pq.insert((s, 0))
  while pq.size() != 0:
    index, minValue = pq.poll()
    vis[index] = true
    for (edge : g[index]):
      if vis[edge.to]: continue
      newDist = dist[index] + edge.cost
      if newDist < dist[edge.to]:
        dist[edge.to] = newDist
        pq.insert((edge.to, newDist))
  return dist
```

Assume PQ stores (node index, best distance) pairs sorted by minimum distance.

```
# Runs Dijkstra’s algorithm and returns an array that contains
# the shortest distance to every node from the start node s.
# g - adjacency list of weighted graph
# n - the number of nodes in the graph
# s - the index of the starting node (0 ≤ s < n)
function dijkstra(g, n, s):
  vis = [false, false, … , false] # size n
  dist = [∞, ∞, … ∞, ∞] # size n
  dist[s] = 0
  pq = empty priority queue
  pq.insert((s, 0))
  while pq.size() != 0:
    index, minValue = pq.poll()
    vis[index] = true
    for (edge : g[index]):
      if vis[edge.to]: continue
      newDist = dist[index] + edge.cost
      if newDist < dist[edge.to]:
        dist[edge.to] = newDist
        pq.insert((edge.to, newDist))
  return dist
```

```
# Runs Dijkstra's algorithm and returns an array that contains
# the shortest distance to every node from the start node s.
# g - adjacency list of weighted graph
# n - the number of nodes in the graph
# s - the index of the starting node (0 ≤ s < n)
function dijkstra(g, n, s):
  vis = [false, false, … , false] # size n
  dist = [∞, ∞, … ∞, ∞] # size n
  dist[s] = 0
  pq = empty priority queue
  pq.insert((s, 0))
  while pq.size() != 0:
    index, minValue = pq.poll()
    vis[index] = true
    for (edge : g[index]):
      if vis[edge.to]: continue
      newDist = dist[index] + edge.cost
      if newDist < dist[edge.to]:
        dist[edge.to] = newDist
        pq.insert((edge.to, newDist))
  return dist
```

```
# Runs Dijkstra’s algorithm and returns an array that contains
# the shortest distance to every node from the start node s.
# g - adjacency list of weighted graph
# n - the number of nodes in the graph
# s - the index of the starting node (0 ≤ s < n)
function dijkstra(g, n, s):
  vis = [false, false, … , false] # size n
  dist = [∞, ∞, … ∞, ∞] # size n
  dist[s] = 0
  pq = empty priority queue
  pq.insert((s, 0))
  while pq.size() != 0:
    index, minValue = pq.poll()
    vis[index] = true
    for (edge : g[index]):
      if vis[edge.to]: continue
      newDist = dist[index] + edge.cost
      if newDist < dist[edge.to]:
        dist[edge.to] = newDist
        pq.insert((edge.to, newDist))
  return dist
```

```
# Runs Dijkstra's algorithm and returns an array that contains
# the shortest distance to every node from the start node s.
# g - adjacency list of weighted graph
# n - the number of nodes in the graph
# s - the index of the starting node (0 ≤ s < n)
function dijkstra(g, n, s):
  vis = [false, false, … , false] # size n
  dist = [∞, ∞, … ∞, ∞] # size n
  dist[s] = 0
  pq = empty priority queue
  pq.insert((s, 0))
  while pq.size() != 0:
    index, minValue = pq.poll()
    vis[index] = true
    for (edge : g[index]):
      if vis[edge.to]: continue
      newDist = dist[index] + edge.cost
      if newDist < dist[edge.to]:
        dist[edge.to] = newDist
        pq.insert((edge.to, newDist))
  return dist
```

```
# Runs Dijkstra’s algorithm and returns an array that contains
# the shortest distance to every node from the start node s.
# g - adjacency list of weighted graph
# n - the number of nodes in the graph
# s - the index of the starting node (0 ≤ s < n)
function dijkstra(g, n, s):
  vis = [false, false, … , false] # size n
  dist = [∞, ∞, … ∞, ∞] # size n
  dist[s] = 0
  pq = empty priority queue
  pq.insert((s, 0))
  while pq.size() != 0:
    index, minValue = pq.poll()
    vis[index] = true
    for (edge : g[index]):
      if vis[edge.to]: continue
      newDist = dist[index] + edge.cost
      if newDist < dist[edge.to]:
        dist[edge.to] = newDist
        pq.insert((edge.to, newDist))
  return dist
```

In practice most standard libraries do not support the decrease key operation for PQs. A way to get around this is to add a new (node index, best distance) pair every time we update the distance to a node.

```
# Runs Dijkstra's algorithm and returns an array that contains
# the shortest distance to every node from the start node s.
# g - adjacency list of weighted graph
# n - the number of nodes in the graph
# s - the index of the starting node (0 ≤ s < n)
function dijkstra(g, n, s):
  vis = [false, false, … , false] # size n
  dist = [∞, ∞, … ∞, ∞] # size n
  dist[s] = 0
  pq = empty priority queue
  pq.insert((s, 0))
  while pq.size() != 0:
    index, minValue = pq.poll()
    vis[index] = true
    for (edge : g[index]):
      if vis[edge.to]: continue
      newDist = dist[index] + edge.cost
      if newDist < dist[edge.to]:
        dist[edge.to] = newDist
        pq.insert((edge.to, newDist))
  return dist
```

As a result, it is possible to have duplicate node indices in the PQ. This is not ideal, but inserting a new key-value pair in **O(log(n))** is much faster than searching for the key in the PQ which takes **O(n)**

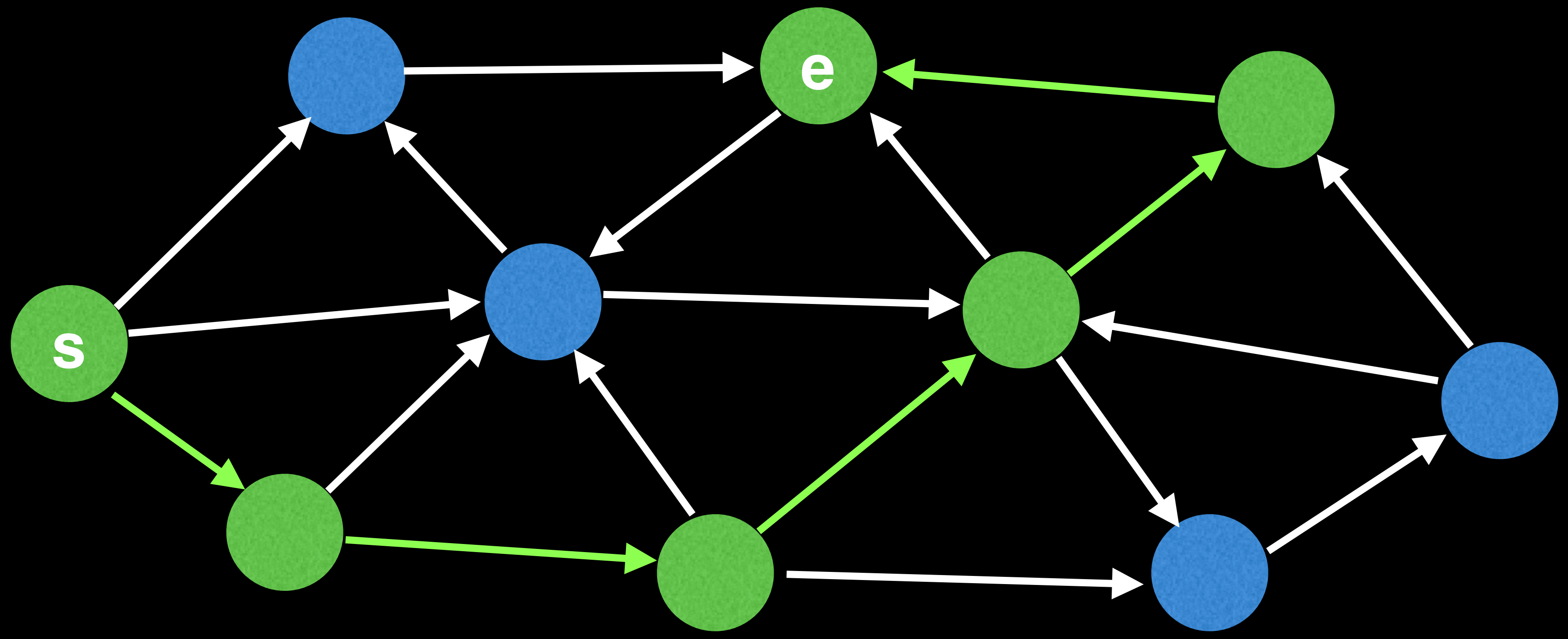
```
# Runs Dijkstra's algorithm and returns an array that contains
# the shortest distance to every node from the start node s.
# g - adjacency list of weighted graph
# n - the number of nodes in the graph
# s - the index of the starting node (0 ≤ s < n)
function dijkstra(g, n, s):
  vis = [false, false, … , false] # size n
  dist = [∞, ∞, … ∞, ∞] # size n
  dist[s] = 0
  pq = empty priority queue
  pq.insert((s, 0))
  while pq.size() != 0:
    index, minValue = pq.poll()
    vis[index] = true

    for (edge : g[index]):
      if vis[edge.to]: continue
      newDist = dist[index] + edge.cost
      if newDist < dist[edge.to]:
        dist[edge.to] = newDist
        pq.insert((edge.to, newDist))
  return dist
```

```
# Runs Dijkstra’s algorithm and returns an array that contains
# the shortest distance to every node from the start node s.
# g - adjacency list of weighted graph
# n - the number of nodes in the graph
# s - the index of the starting node (0 ≤ s < n)
function dijkstra(g, n, s):
  vis = [false, false, … , false] # size n
  dist = [∞, ∞, … ∞, ∞] # size n
  dist[s] = 0
  pq = empty priority queue
  pq.insert((s, 0))
  while pq.size() != 0:
    index, minValue = pq.poll()
    vis[index] = true
    if dist[index] < minValue: continue
    for (edge : g[index]):
      if vis[edge.to]: continue
      newDist = dist[index] + edge.cost
      if newDist < dist[edge.to]:
        dist[edge.to] = newDist
        pq.insert((edge.to, newDist))
  return dist
```

A neat optimization we can do which ignores stale (index, dist) pairs in our PQ is to skip nodes where we already found a better path routing through others nodes before we got to processing this node.

Finding the optimal path

If you wish to not only find the optimal distance to a particular node but also **what sequence of nodes were taken** to get there you need to track some additional information.

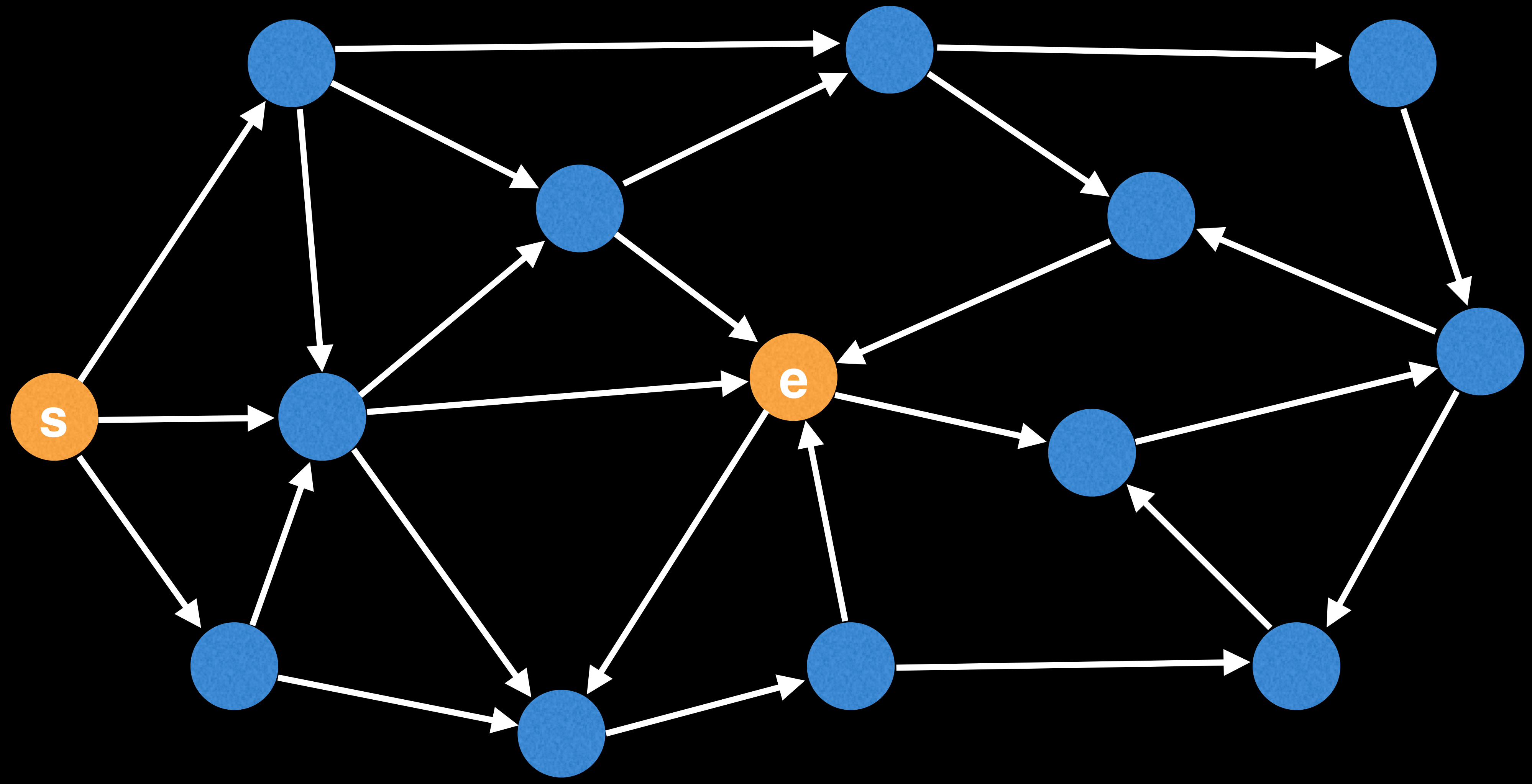


```
# Runs Dijkstra's algorithm and returns an array that contains
# the shortest distance to every node from the start node s and
# the prev array to reconstruct the shortest path itself
# g - adjacency list of weighted graph
# n - the number of nodes in the graph
# s - the index of the starting node (0 ≤ s < n)
function dijkstra(g, n, s):
  vis = [false, false, … , false] # size n
  prev = [null, null, …, null] # size n
  dist = [∞, ∞, … ∞, ∞] # size n
  dist[s] = 0
  pq = empty priority queue
  pq.insert((s, 0))
  while pq.size() != 0:
    index, minValue = pq.poll()
    vis[index] = true
    if dist[index] < minValue: continue
    for (edge : g[index]):
      if vis[edge.to]: continue
      newDist = dist[index] + edge.cost
      if newDist < dist[edge.to]:
        prev[edge.to] = index
        dist[edge.to] = newDist
        pq.insert((edge.to, newDist))
  return (dist, prev)
```

```
# Finds the shortest path between two nodes.
# g - adjacency list of weighted graph
# n - the number of nodes in the graph
# s - the index of the starting node (0 ≤ s < n)
# e - the index of the end node (0 ≤ e < n)
function findShortestPath(g, n, s, e):
  dist, prev = dijkstra(g, n, s)
  path = []
  if (dist[e] == ∞) return path
  for (at = e; at != null; at = prev[at])
    path.add(at)
  path.reverse()
  return path
```

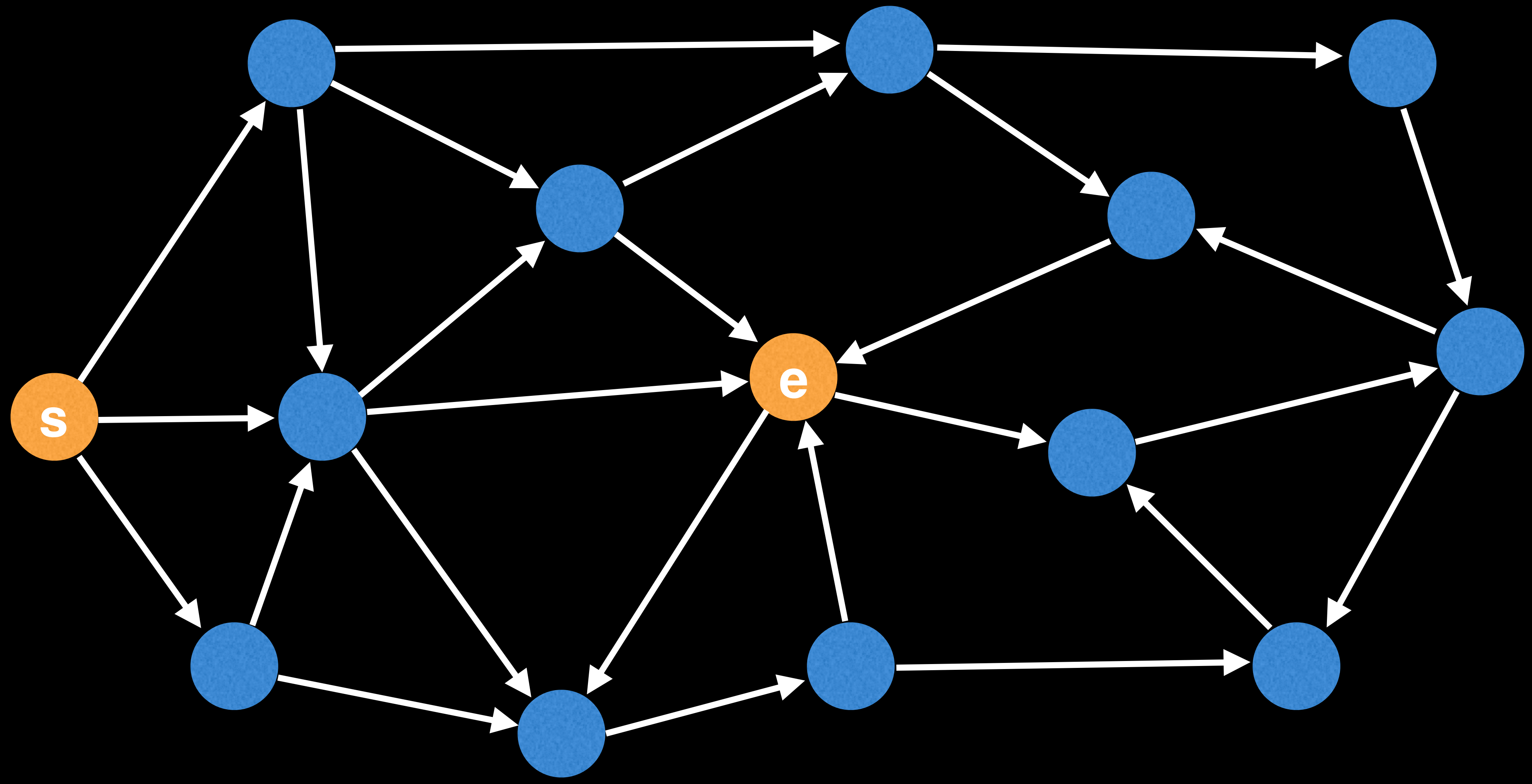
Stopping Early

Q: Suppose you know the destination node you're trying to reach is 'e' and you start at node 's' do you still have to visit every node in the graph?



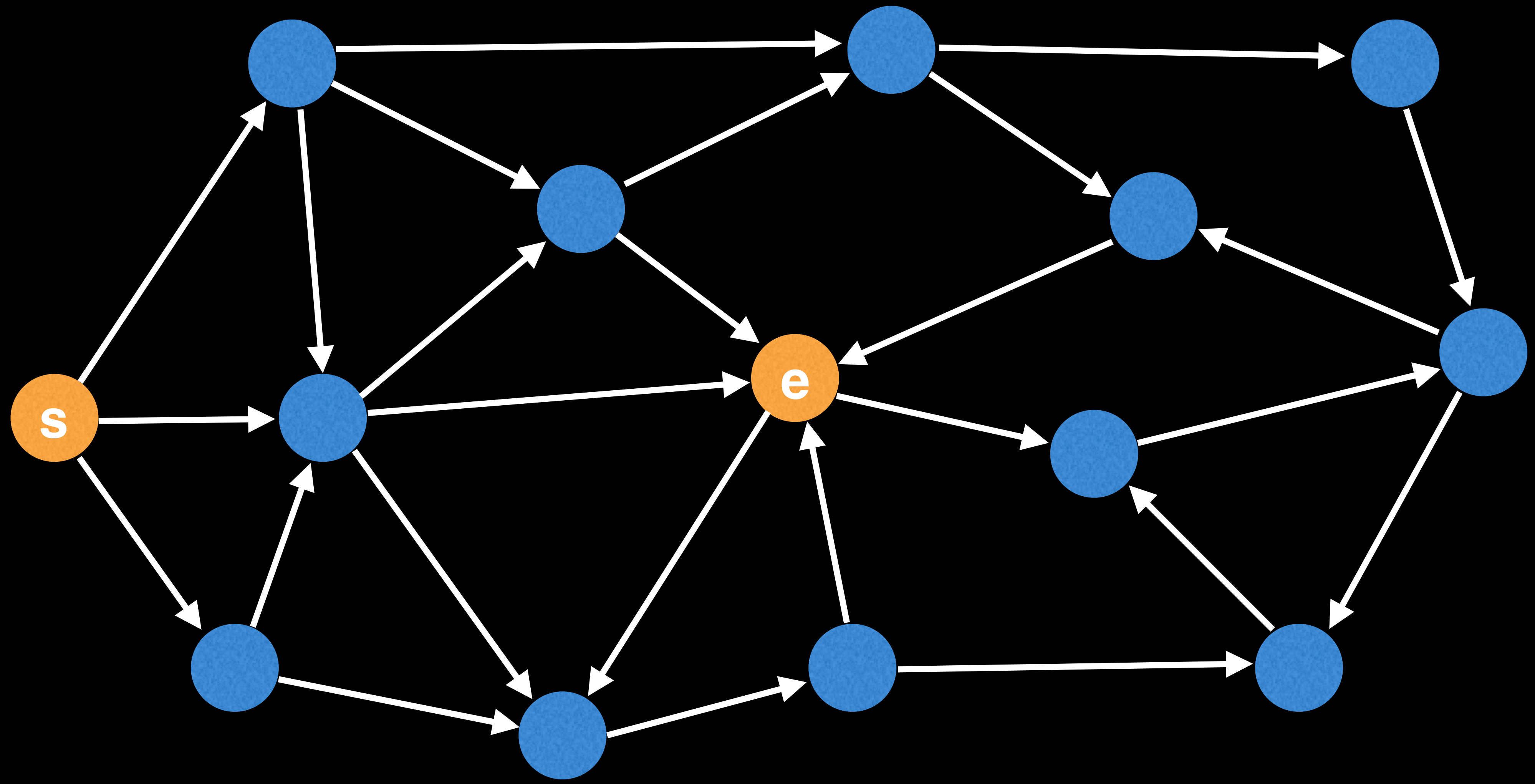
Stopping Early

A: Yes, in the worst case. However, it is possible to stop early once you have finished visiting the destination node.



Stopping Early

The main idea for stopping early is that Dijkstra’s algorithm processes each next most promising node in order. So if the destination node has been visited, its shortest distance will not change as more future nodes are visited.



```
# Runs Dijkstra's algorithm and returns the shortest distance
# between nodes 's' and 'e'. If there is no path, ∞ is returned.
# g - adjacency list of weighted graph
# n - the number of nodes in the graph
# s - the index of the starting node (0 ≤ s < n)
# e - the index of the end node (0 ≤ e < n)
function dijkstra(g, n, s, e):
  vis = [false, false, … , false] # size n
  dist = [∞, ∞, … ∞, ∞] # size n
  dist[s] = 0
  pq = empty priority queue
  pq.insert((s, 0))
  while pq.size() != 0:
    index, minValue = pq.poll()
    vis[index] = true
    if dist[index] < minValue: continue
    for (edge : g[index]):
      if vis[edge.to]: continue
      newDist = dist[index] + edge.cost
      if newDist < dist[edge.to]:
        dist[edge.to] = newDist
        pq.insert((edge.to, newDist))
    if index == e:
      return dist[e]
  return ∞
```

Eager Dijkstra’s using an Indexed Priority Queue

Our current lazy implementation of Dijkstra’s inserts **duplicate key-value pairs** (keys being the node index and the value being the shortest distance to get to that node) in our PQ because it’s more efficient to insert a new key-value pair in **O(log(n))** than it is to update an existing key’s value in **O(n)**.

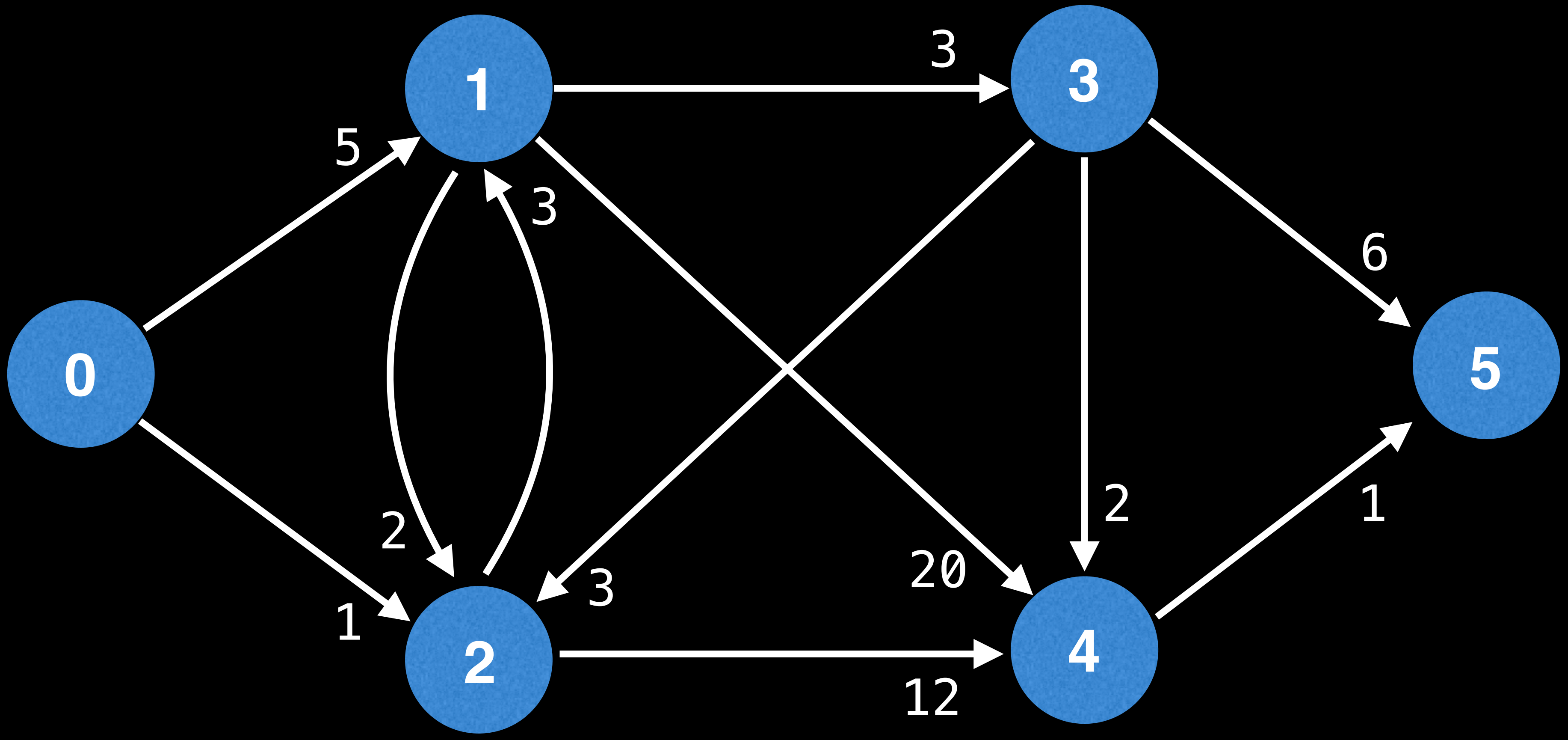
This approach is inefficient for dense graphs because we end up with **several stale outdated key-value pairs** in our PQ. The eager version of Dijkstra’s avoids duplicate key-value pairs and supports efficient value updates in **O(log(n))** by using an **Indexed Priority Queue (IPQ)**

Indexed Priority Queue DS Video

<insert video clip>

github.com/williamfiset/data-structures

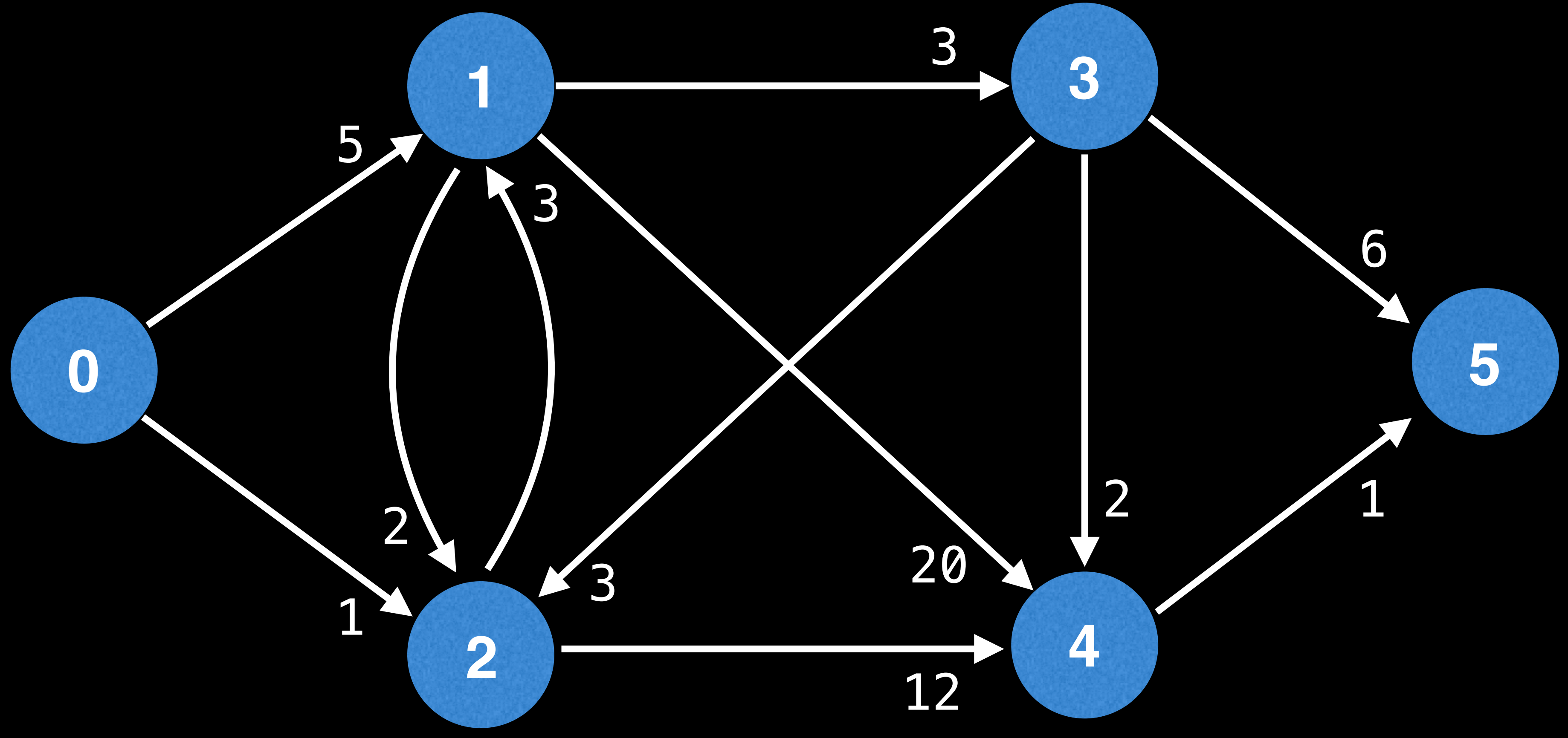
Eager Dijkstra's



(index, dist)
key-value pairs

	0	1	2	3	4	5
dist =	∞	∞	∞	∞	∞	∞

Eager Dijkstra's

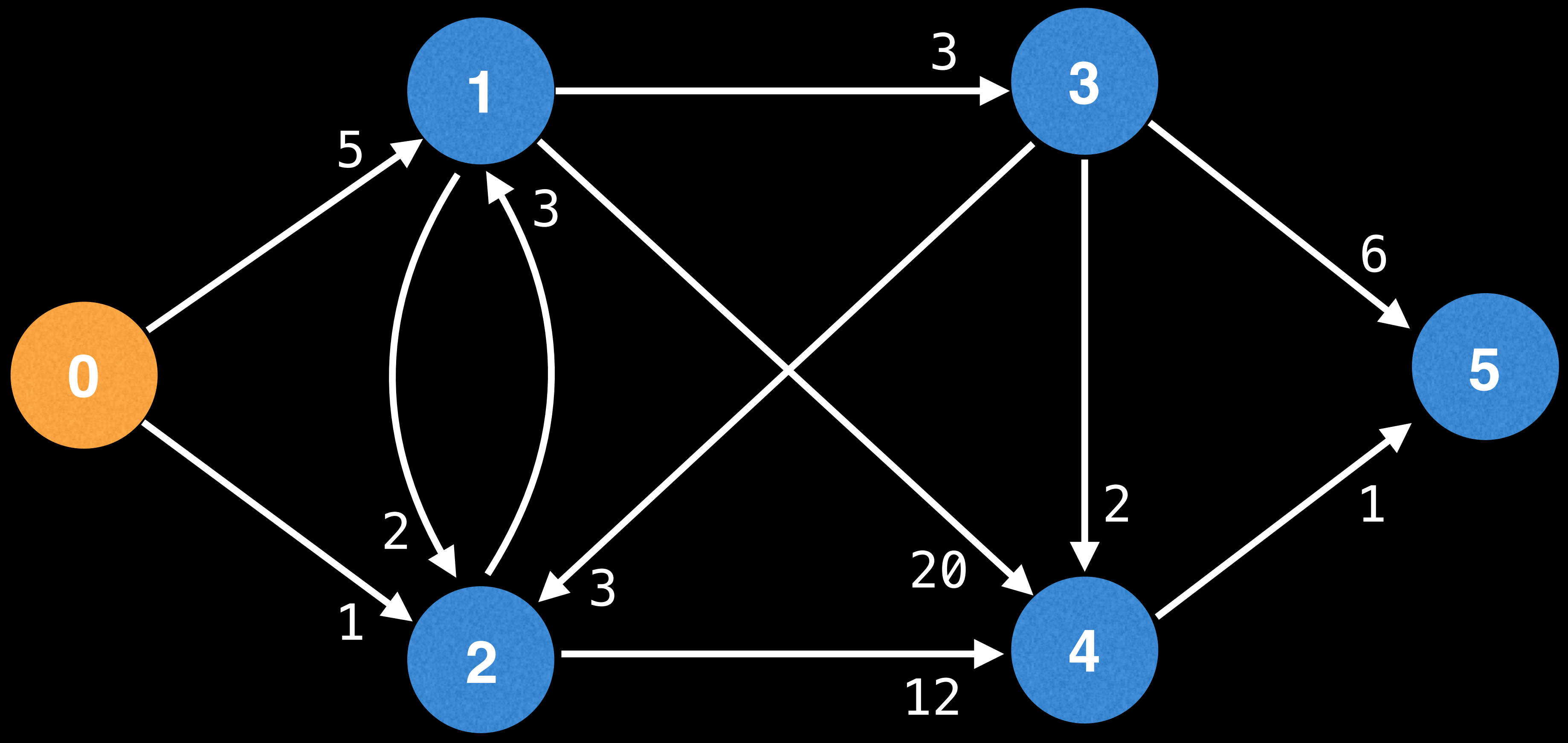


(index, dist)
key-value pairs

(0, 0)

	0	1	2	3	4	5
dist =	0	∞	∞	∞	∞	∞

Eager Dijkstra's

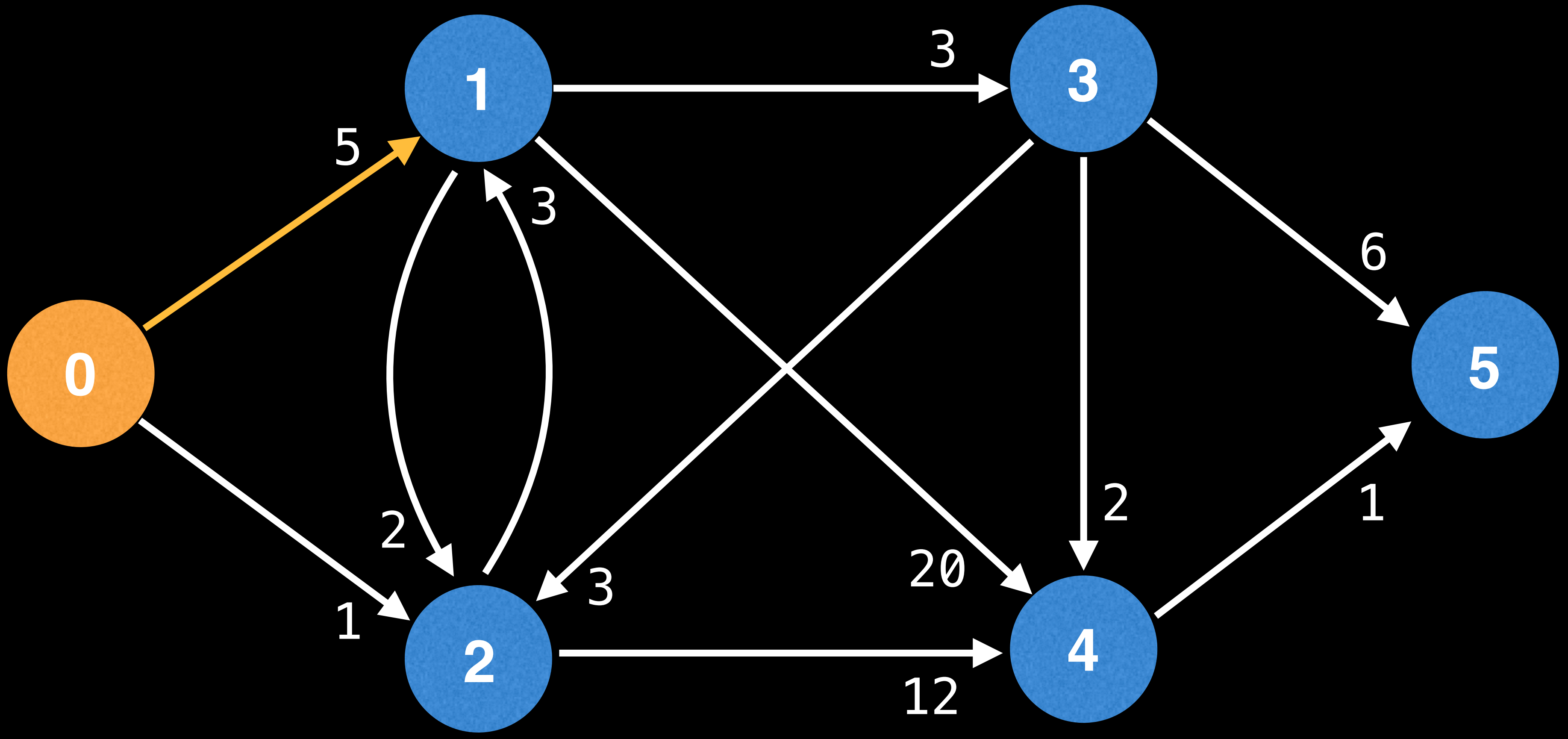


(index, dist)
key-value pairs

→ (0, 0)

	0	1	2	3	4	5
dist =	0	∞	∞	∞	∞	∞

Eager Dijkstra's



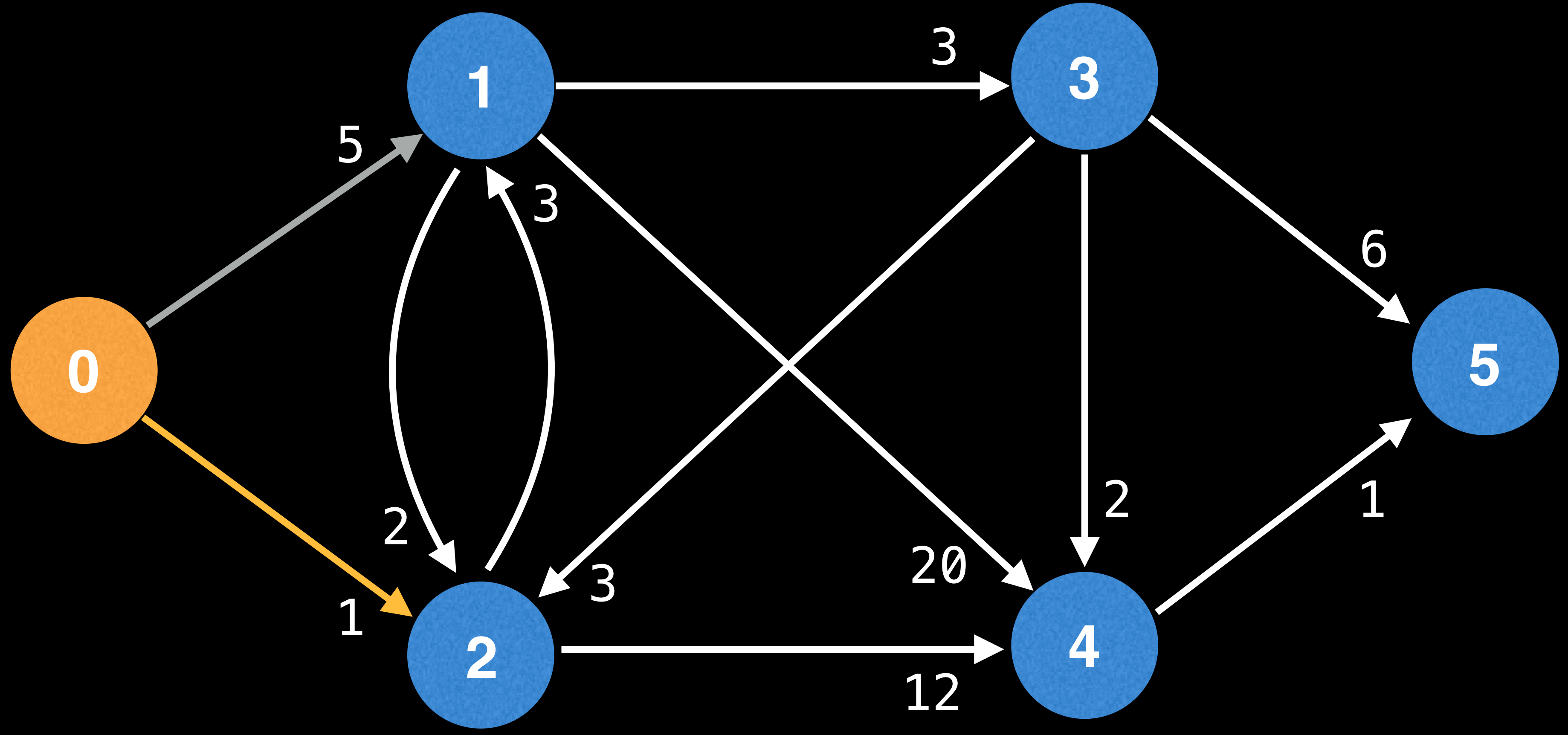
(index, dist)
key-value pairs

→ (0, 0)
(1, 5)

	0	1	2	3	4	5
dist =	0	5	∞	∞	∞	∞

Best distance from node 0 to node 1:
dist[0] + edge.cost = 0 + 5 = 5

Eager Dijkstra's



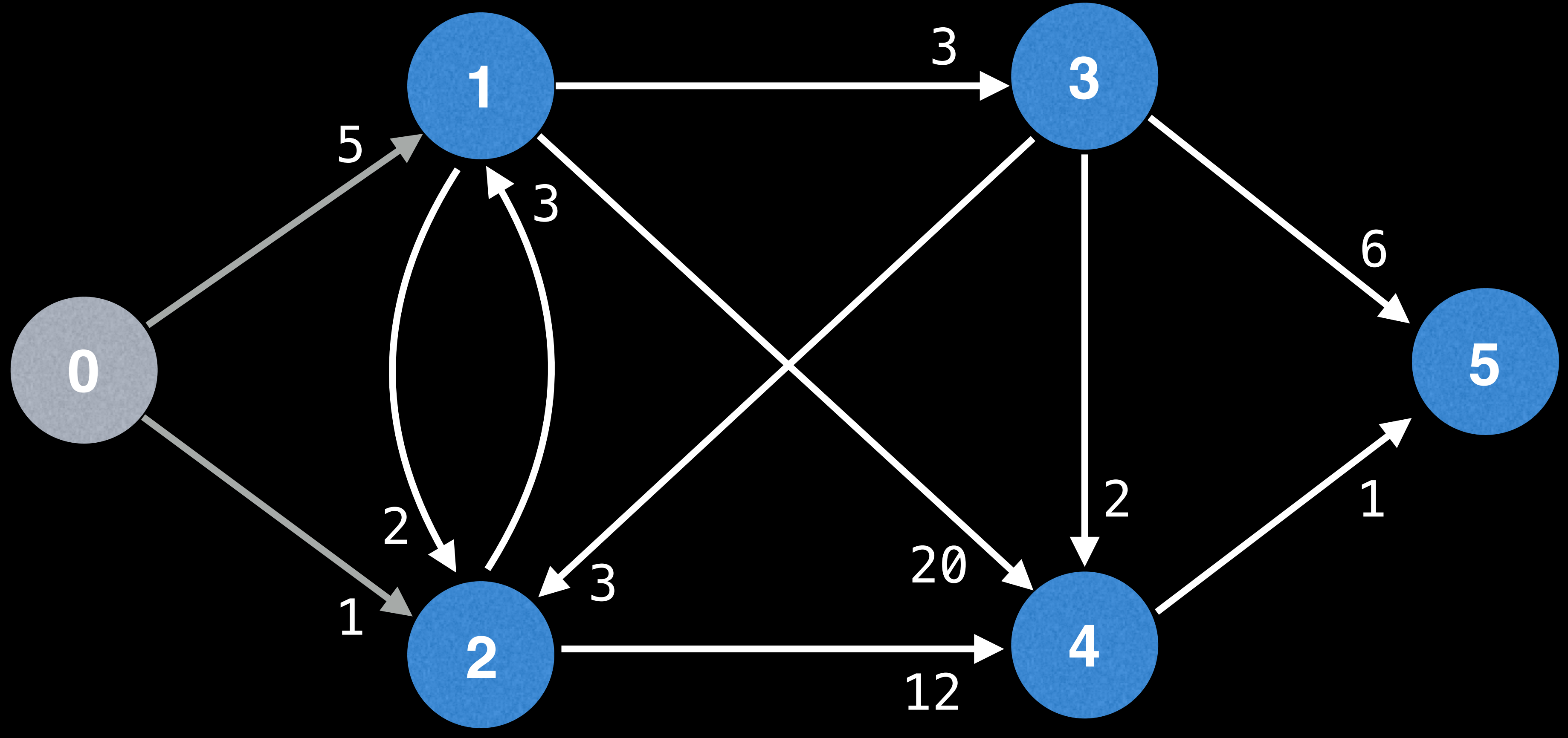
(index, dist)
key-value pairs

→ (0, 0)
(1, 5)
(2, 1)

	0	1	2	3	4	5
dist =	0	5	1	∞	∞	∞

Best distance from node 0 to node 2:
dist[0] + edge.cost = 0 + 1 = 1

Eager Dijkstra's

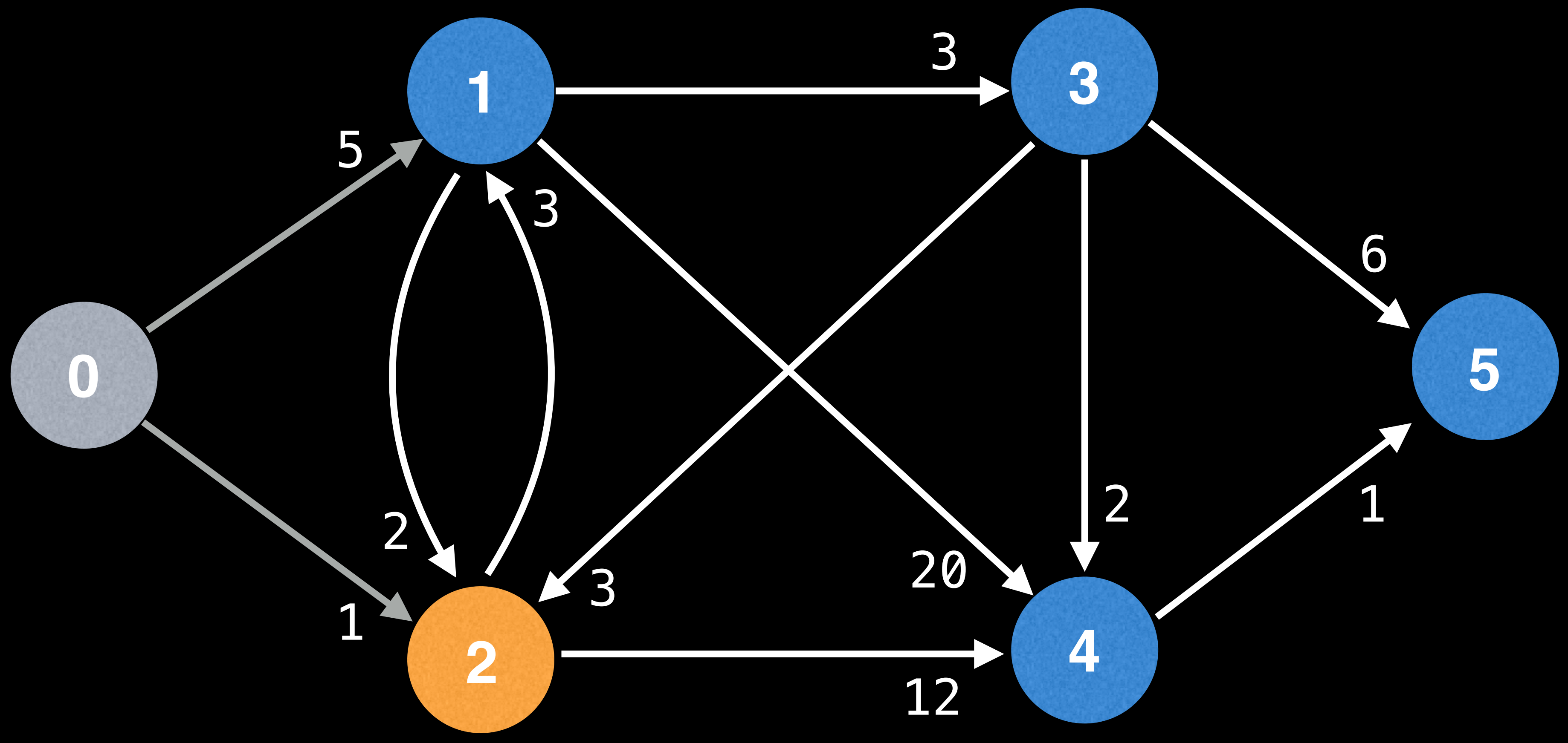


(index, dist)
key-value pairs

→ ~~(0, 0)~~
(1, 5)
(2, 1)

	0	1	2	3	4	5
dist =	0	5	1	∞	∞	∞

Eager Dijkstra's

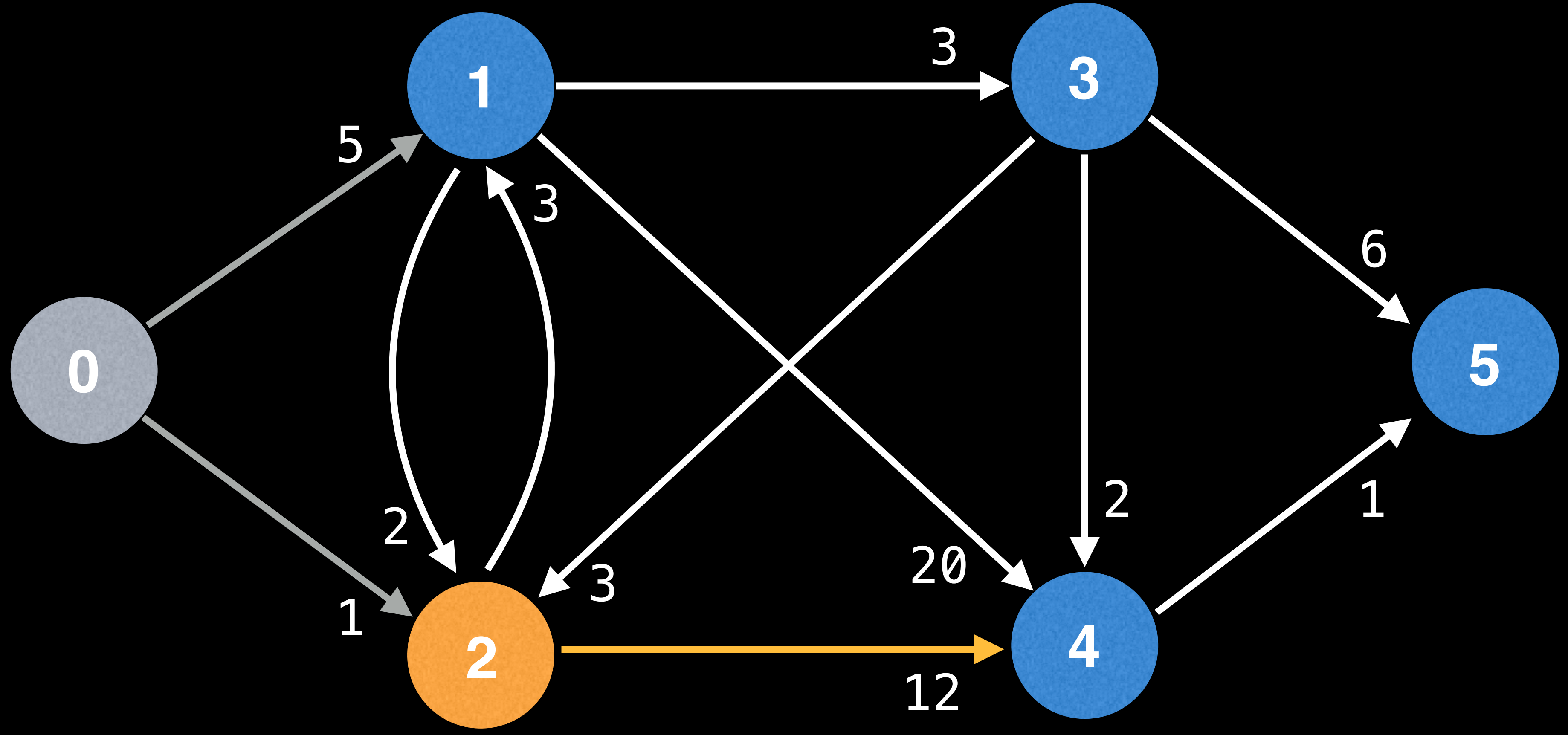


(index, dist)
key-value pairs

~~(0, 0)~~
(1, 5)
→ (2, 1)

	0	1	2	3	4	5
dist =	0	5	1	∞	∞	∞

Eager Dijkstra's



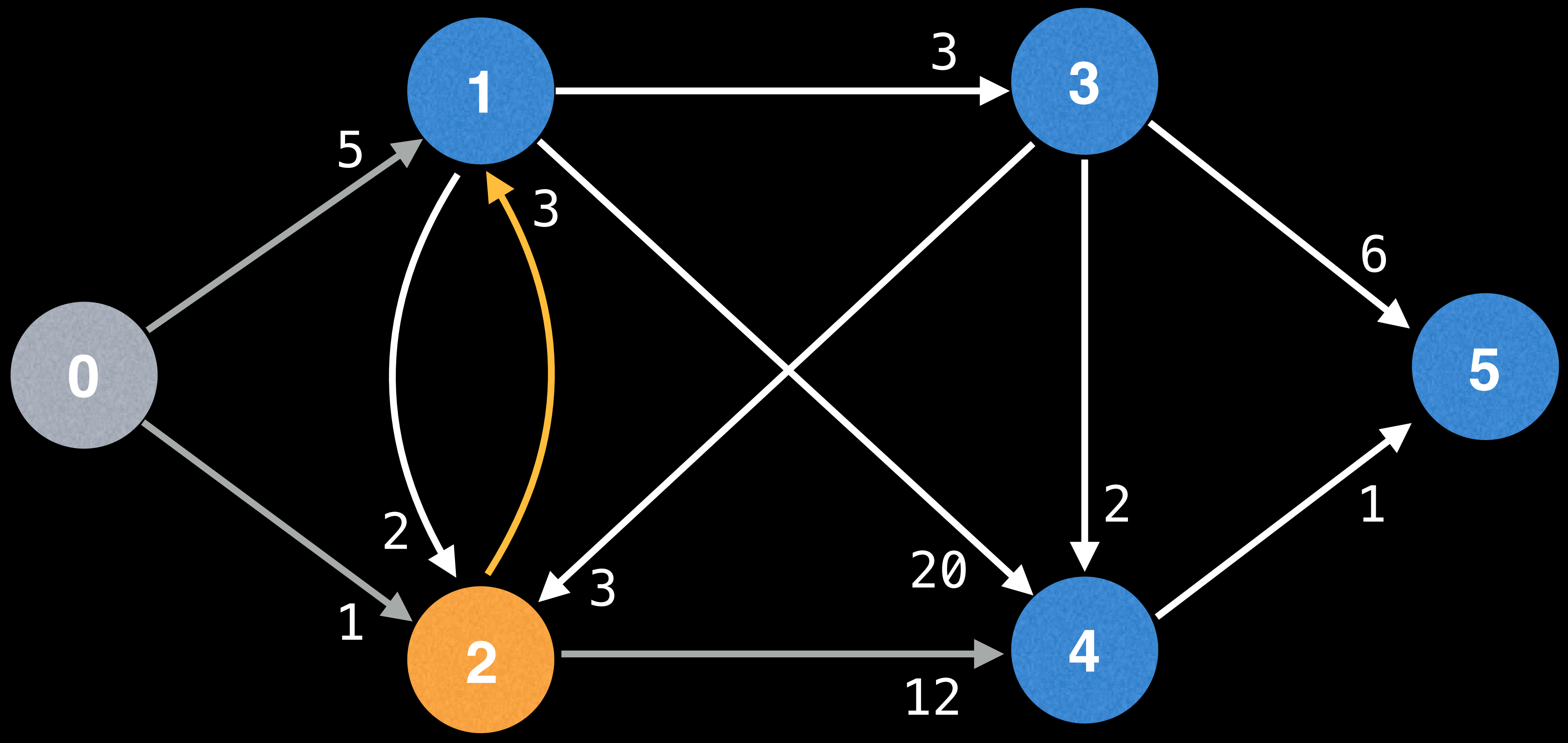
(index, dist)
key-value pairs

~~(0, 0)~~
(1, 5)
→ (2, 1)
(4, 13)

	0	1	2	3	4	5
dist =	0	5	1	∞	13	∞

Best distance from node 2 to node 4:
dist[2] + edge.cost = 1 + 12 = 13

Eager Dijkstra's



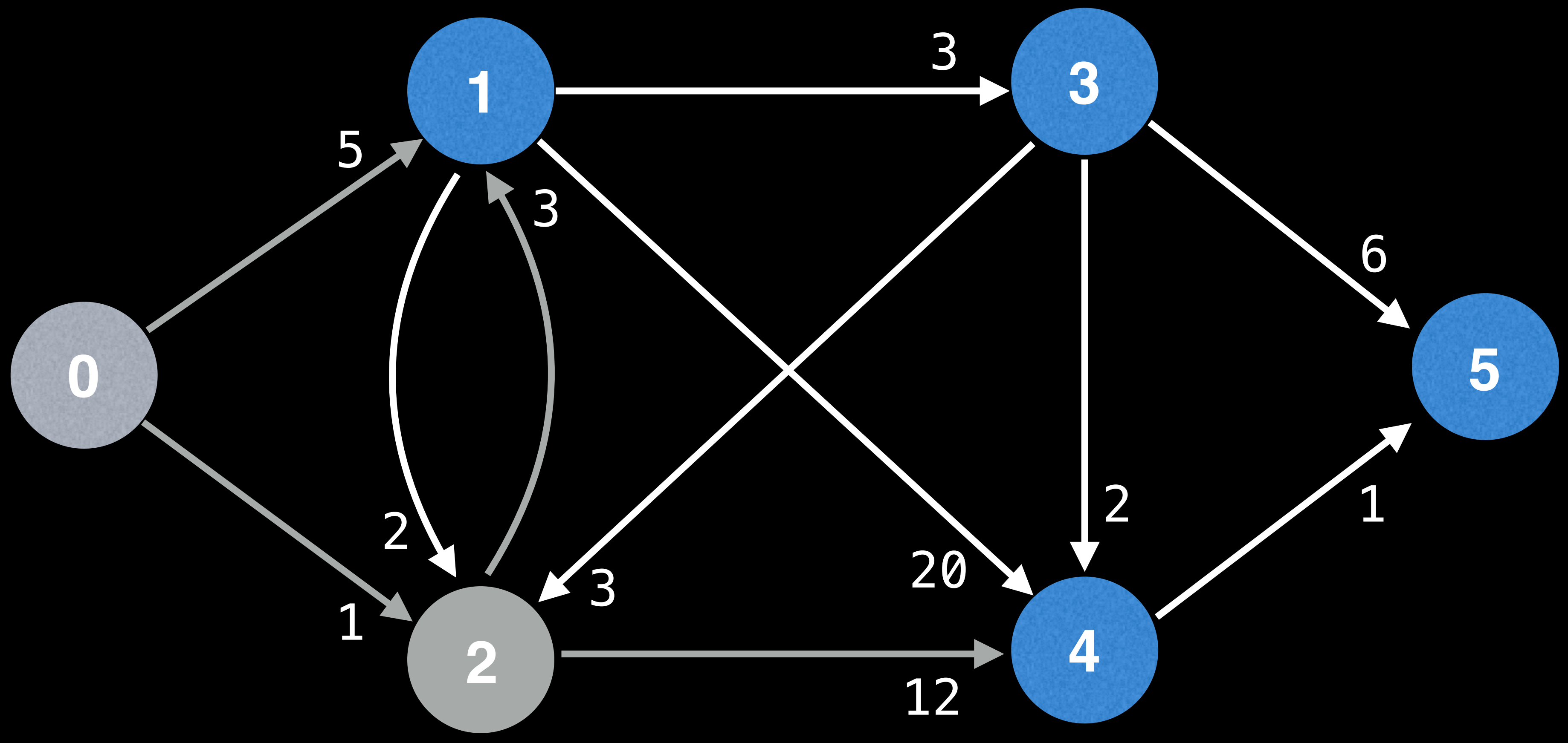
(index, dist) key-value pairs

~~(0, 0)~~
(1, 4)
→ (2, 1)
(4, 13)

	0	1	2	3	4	5
dist =	0	4	1	∞	13	∞

Best distance from node 2 to node 1:
dist[2] + edge.cost = 1 + 3 = 4

Eager Dijkstra's

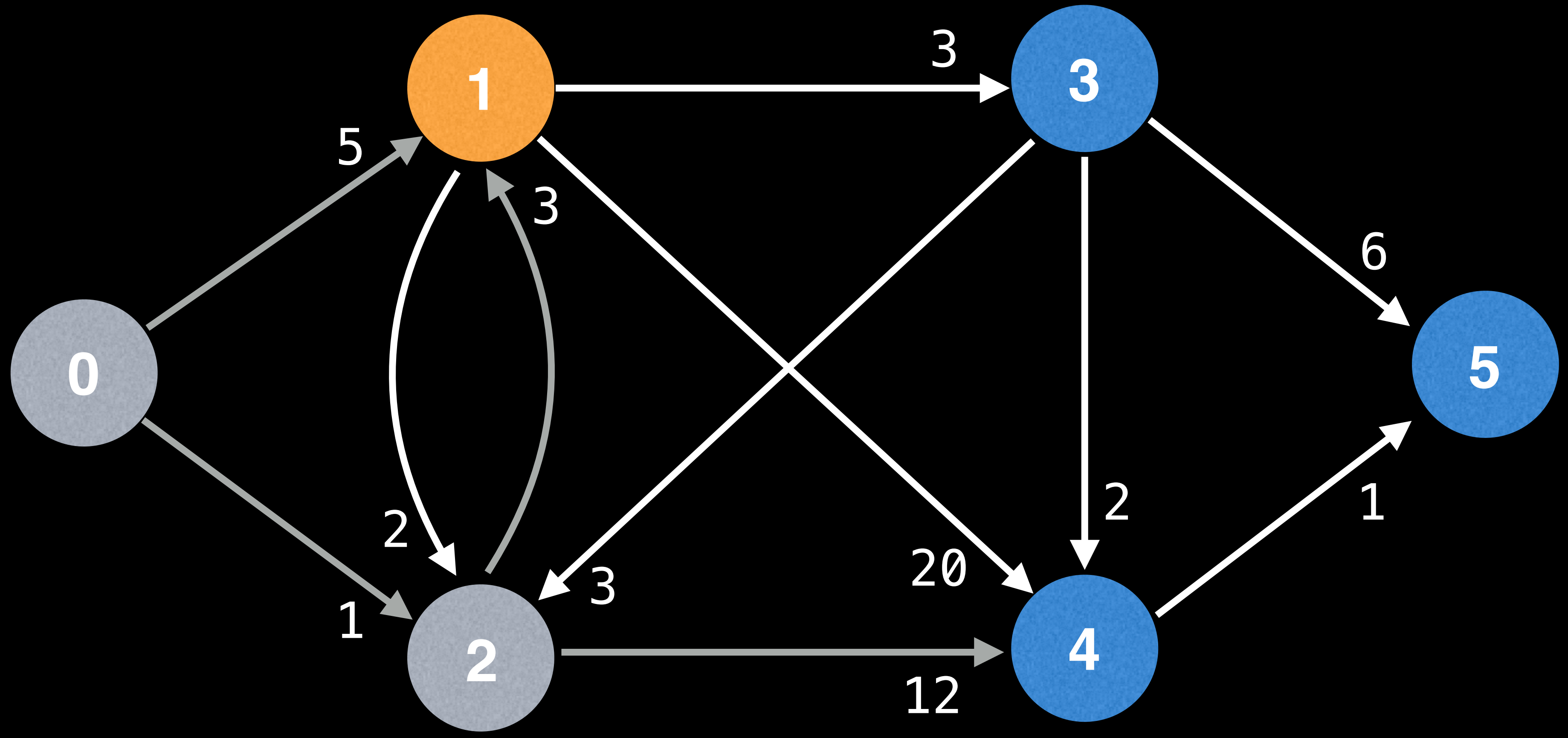


(index, dist) key-value pairs

~~(0, 0)~~
(1, 4)
→ ~~(2, 1)~~
(4, 13)

	0	1	2	3	4	5
dist =	0	4	1	∞	13	∞

Eager Dijkstra's

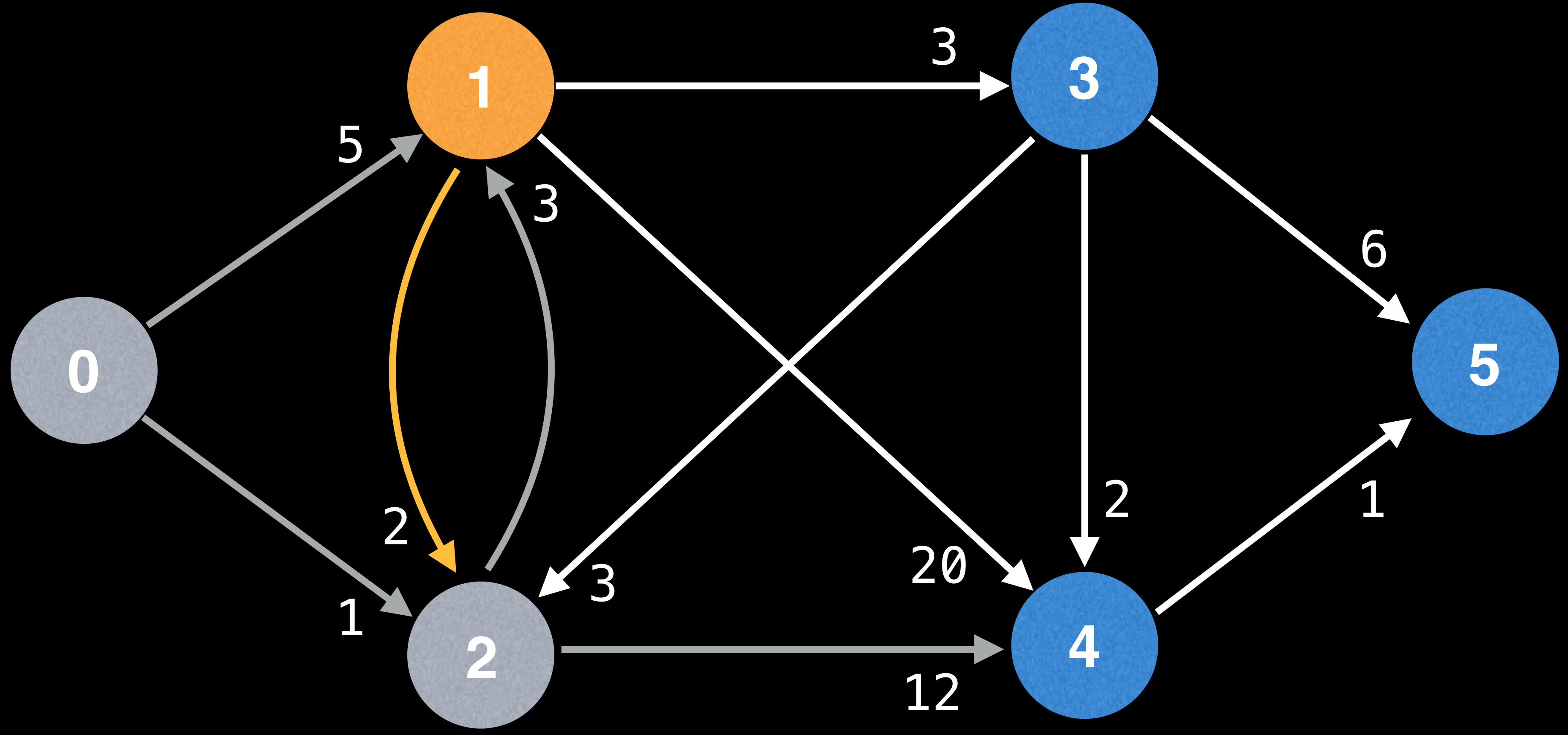


(index, dist)
key-value pairs

~~(0, 0)~~
→ (1, 4)
~~(2, 1)~~
(4, 13)

	0	1	2	3	4	5
dist =	0	4	1	∞	13	∞

Eager Dijkstra's



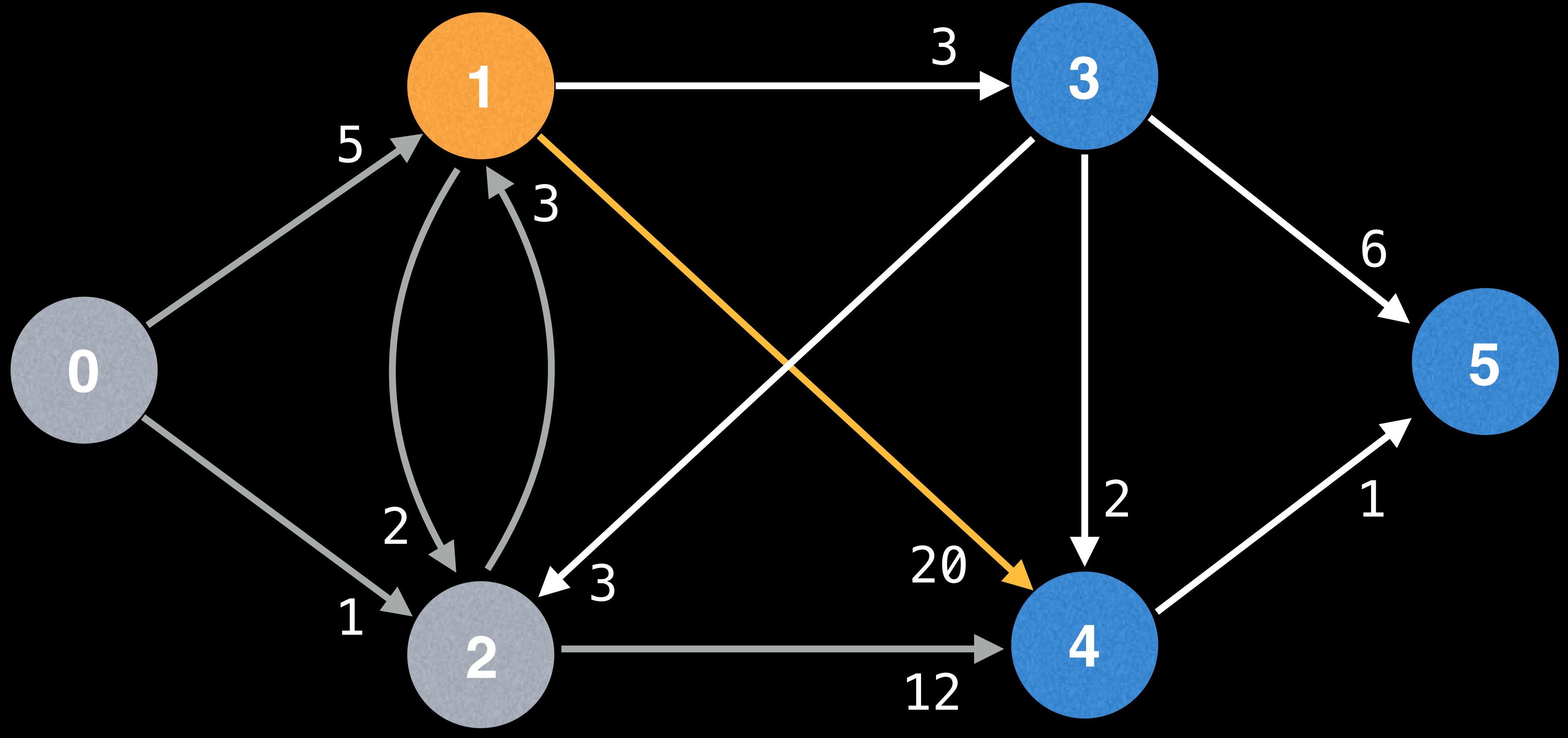
(index, dist)
key-value pairs

~~(0, 0)~~
→ (1, 4)
~~(2, 1)~~
(4, 13)

	0	1	2	3	4	5
dist =	0	4	1	∞	13	∞

Node 2 has already been visited so we cannot improve it's already best distance

Eager Dijkstra's



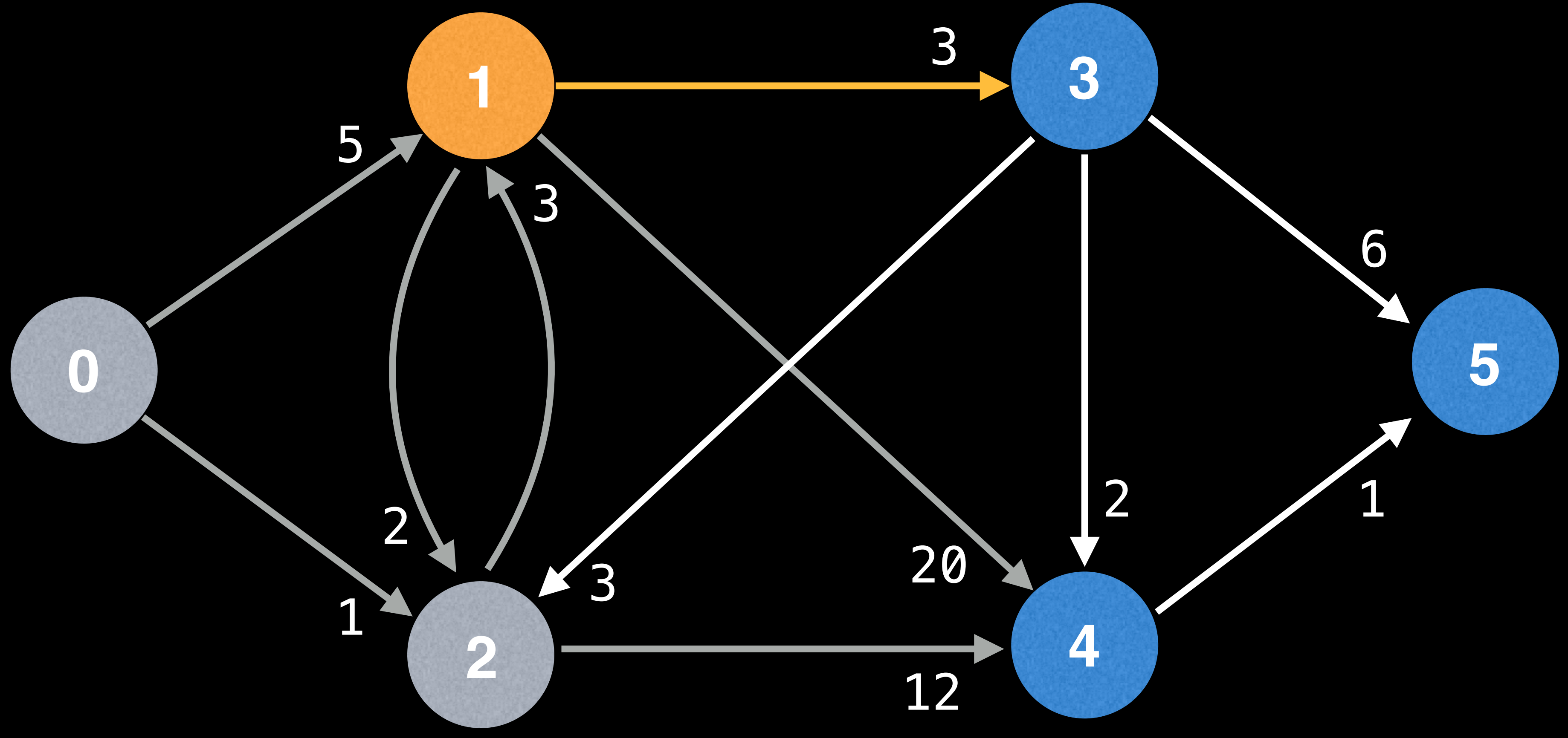
(index, dist)
key-value pairs

~~(0, 0)~~
→ (1, 4)
~~(2, 1)~~
(4, 13)

	0	1	2	3	4	5
dist =	0	4	1	∞	13	∞

dist[1] + edge.cost = 4 + 20 = 24 > dist[4] = 13
so we cannot update best distance.

Eager Dijkstra's



(index, dist)
key-value pairs

~~(0, 0)~~

→ (1, 4)

~~(2, 1)~~

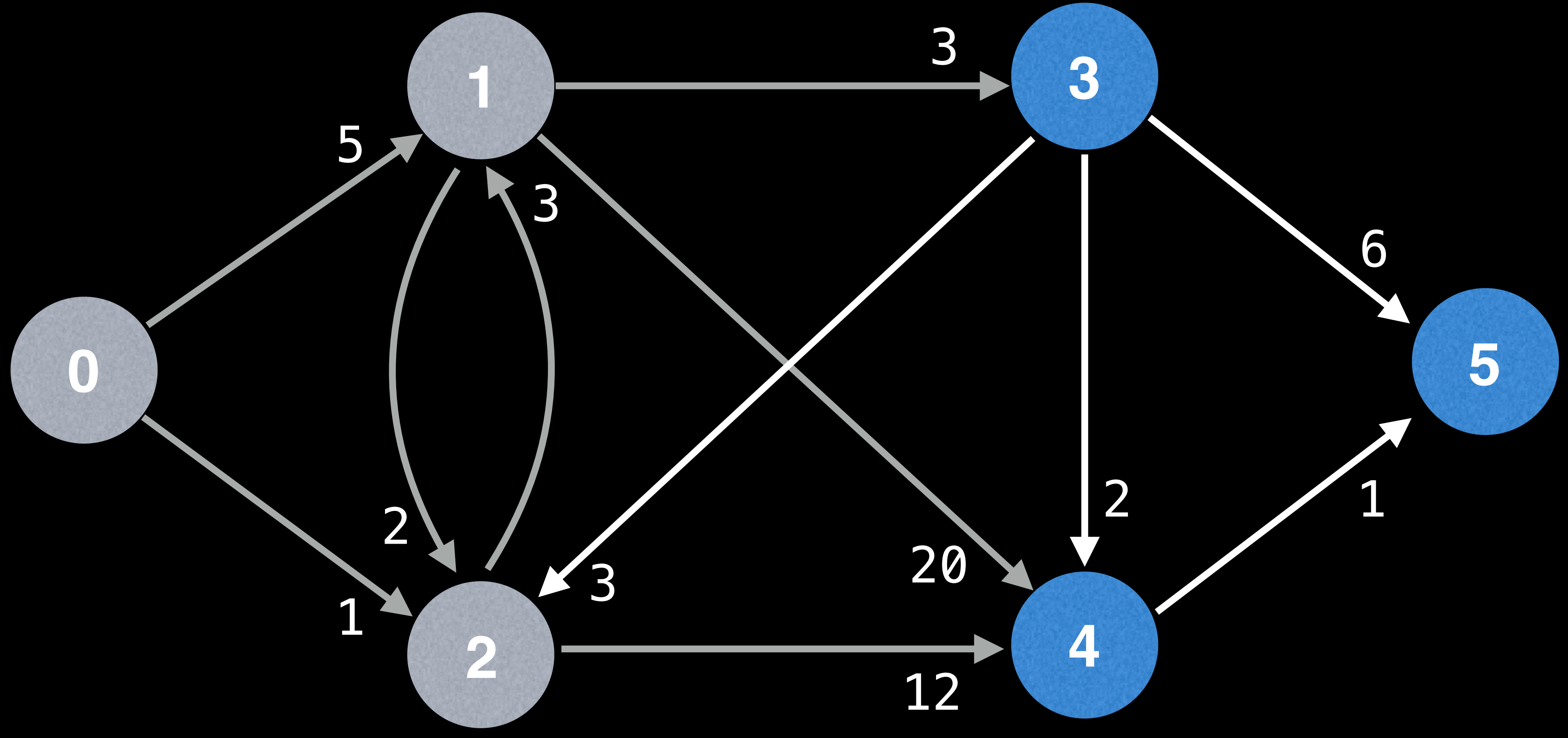
(4, 13)

(3, 7)

	0	1	2	3	4	5
dist =	0	4	1	7	13	∞

Best distance from node 1 to node 3:
dist[1] + edge.cost = 4 + 3 = 7

Eager Dijkstra's

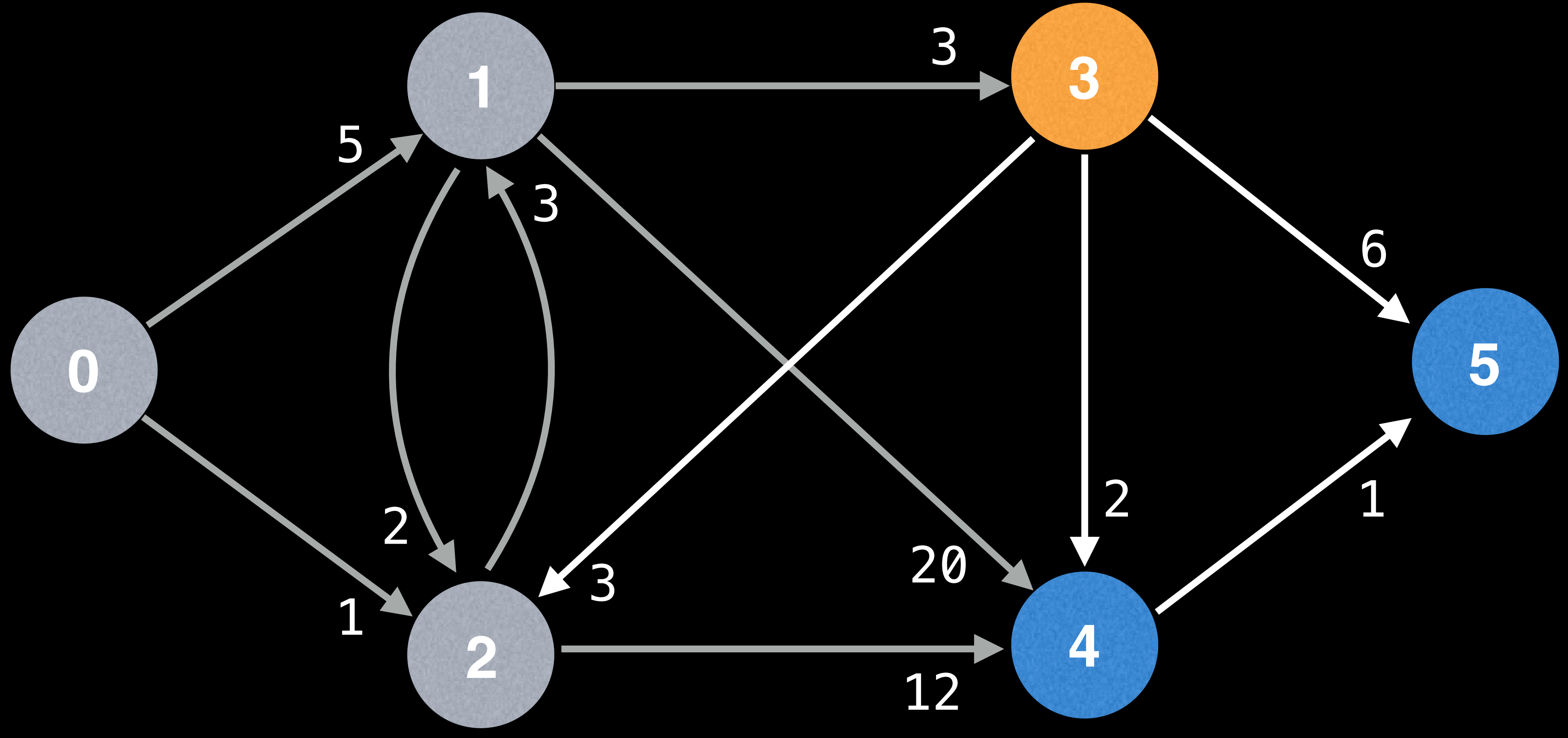


(index, dist)
key-value pairs

~~(0, 0)~~
→ ~~(1, 4)~~
~~(2, 1)~~
(4, 13)
(3, 7)

	0	1	2	3	4	5
dist =	0	4	1	7	13	∞

Eager Dijkstra's



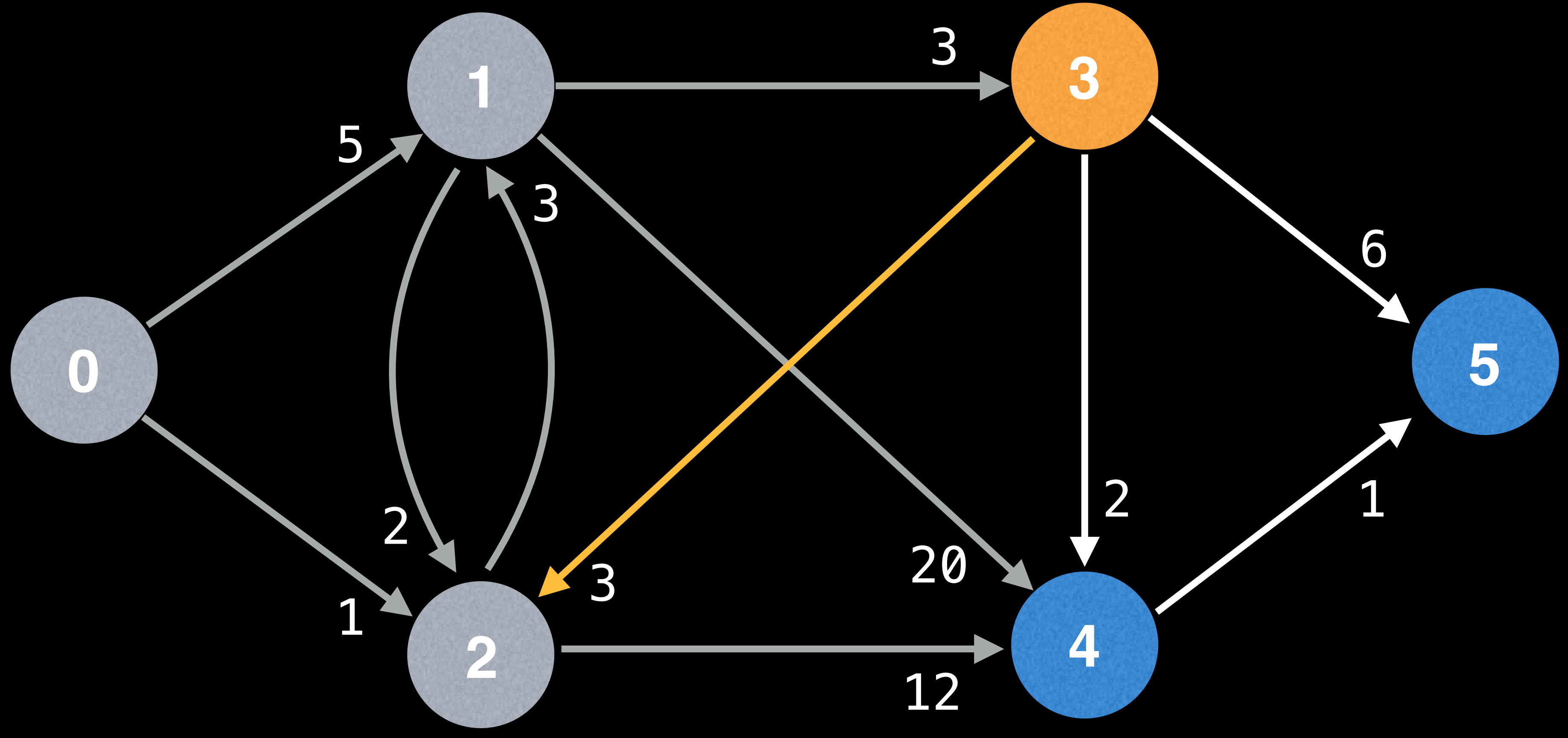
(index, dist)
key-value pairs

~~(0, 0)~~
~~(1, 4)~~
~~(2, 1)~~
(4, 13)
→ (3, 7)

dist =

0	1	2	3	4	5
0	4	1	7	13	∞

Eager Dijkstra's



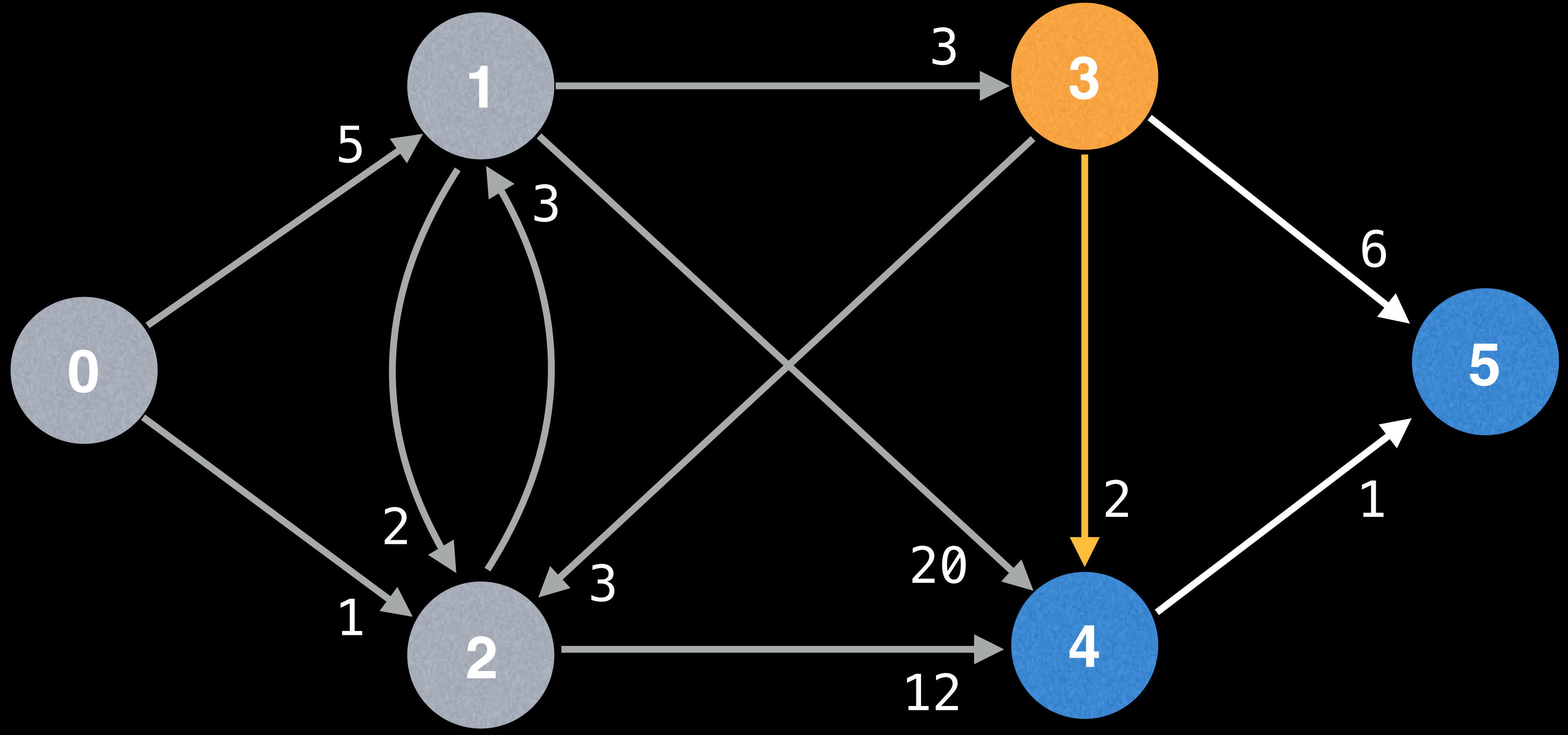
(index, dist)
key-value pairs

~~(0, 0)~~
~~(1, 4)~~
~~(2, 1)~~
(4, 13)
→ (3, 7)

	0	1	2	3	4	5
dist =	0	4	1	7	13	∞

Node 2 is already visited, therefore we cannot improve on its best distance

Eager Dijkstra's



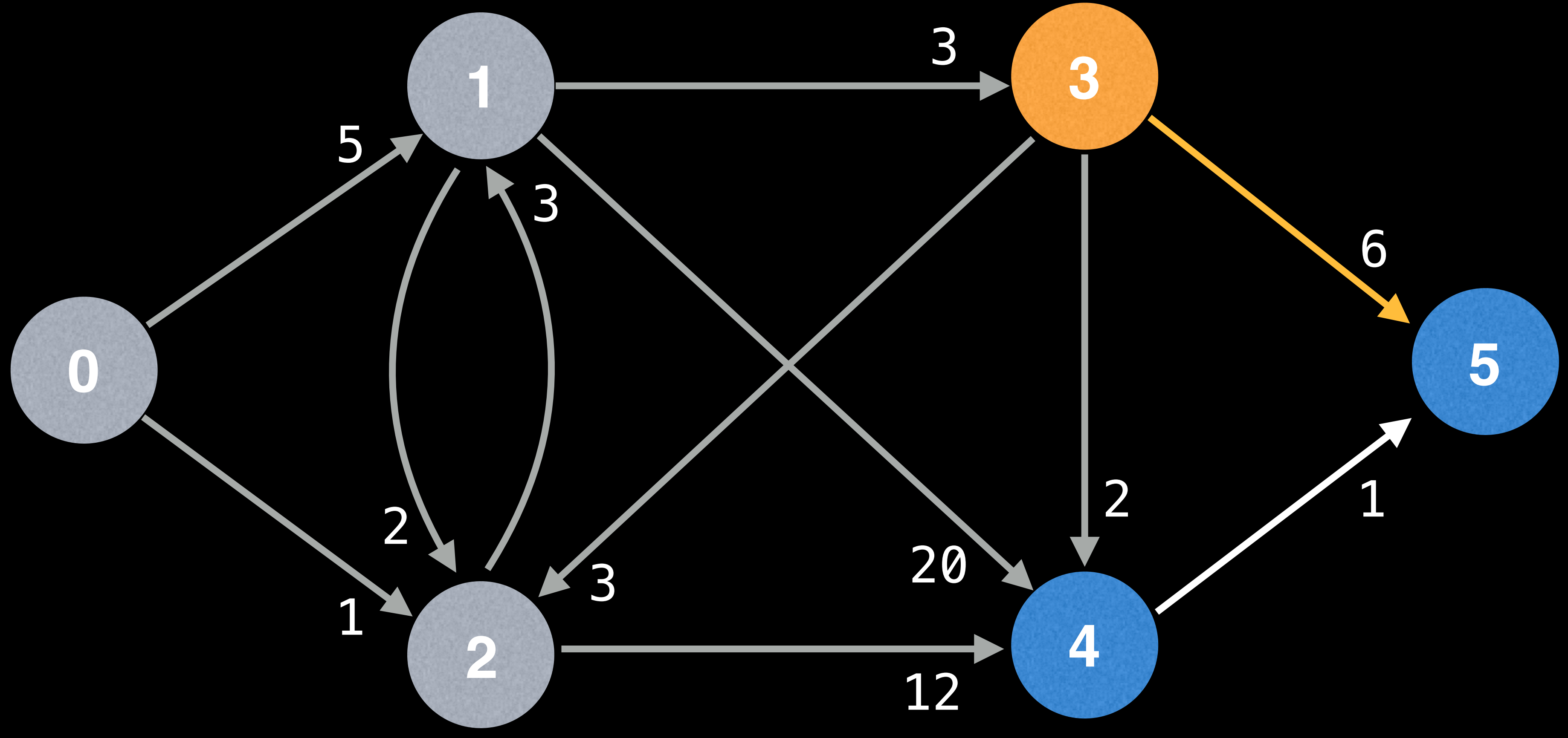
(index, dist)
key-value pairs

~~(0, 0)~~
~~(1, 4)~~
~~(2, 1)~~
(4, 9)
→ (3, 7)

	0	1	2	3	4	5
dist =	0	4	1	7	9	∞

Best distance from node 3 to node 4:
dist[3] + edge.cost = 7 + 2 = 9

Eager Dijkstra's



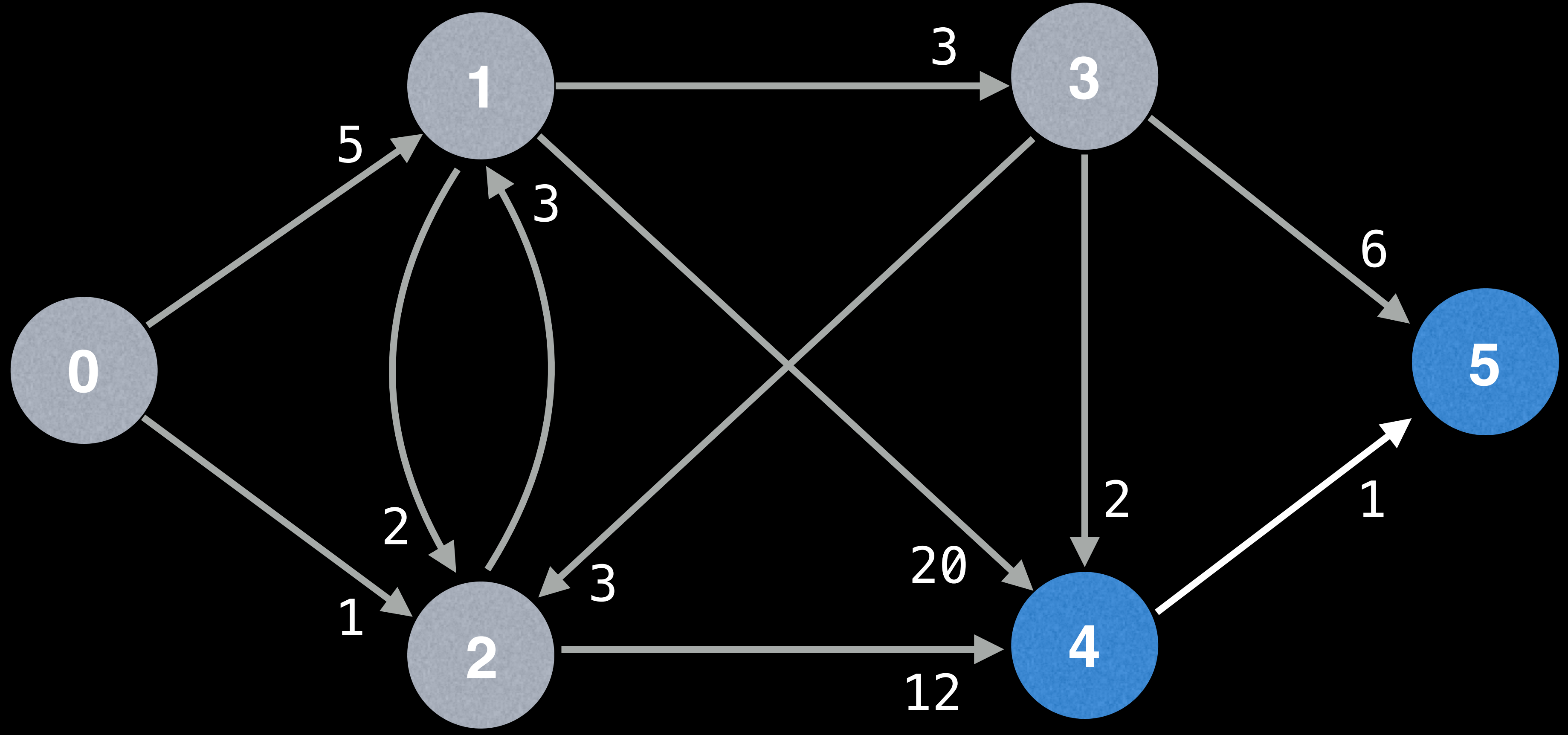
(index, dist)
key-value pairs

~~(0, 0)~~
~~(1, 4)~~
~~(2, 1)~~
(4, 9)
→ (3, 7)
(5, 13)

	0	1	2	3	4	5
dist =	0	4	1	7	9	13

Best distance from node 3 to node 5:
dist[3] + edge.cost = 7 + 6 = 13

Eager Dijkstra's

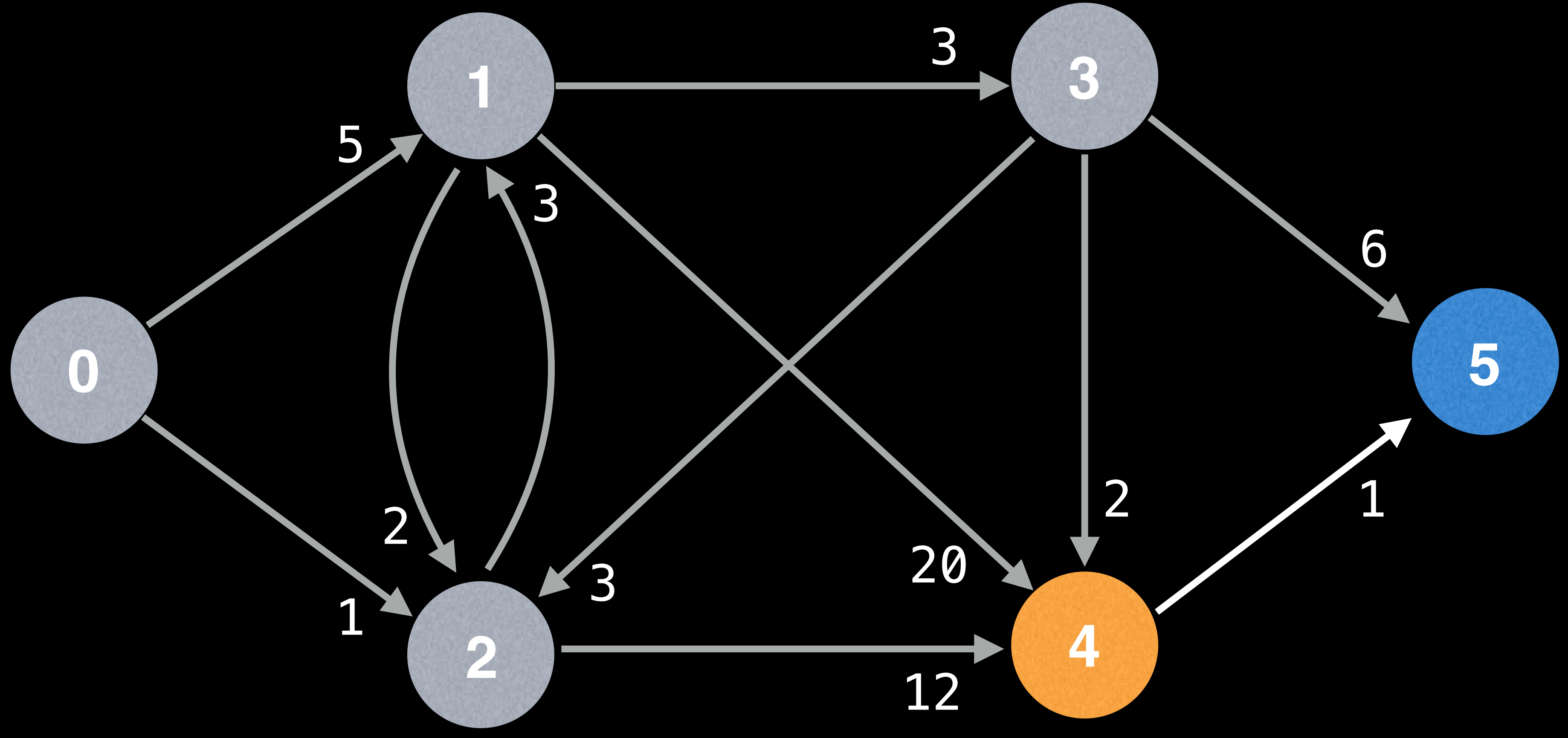


(index, dist)
key-value pairs

~~(0, 0)~~
~~(1, 4)~~
~~(2, 1)~~
(4, 9)
→ ~~(3, 7)~~
(5, 13)

	0	1	2	3	4	5
dist =	0	4	1	7	9	13

Eager Dijkstra's

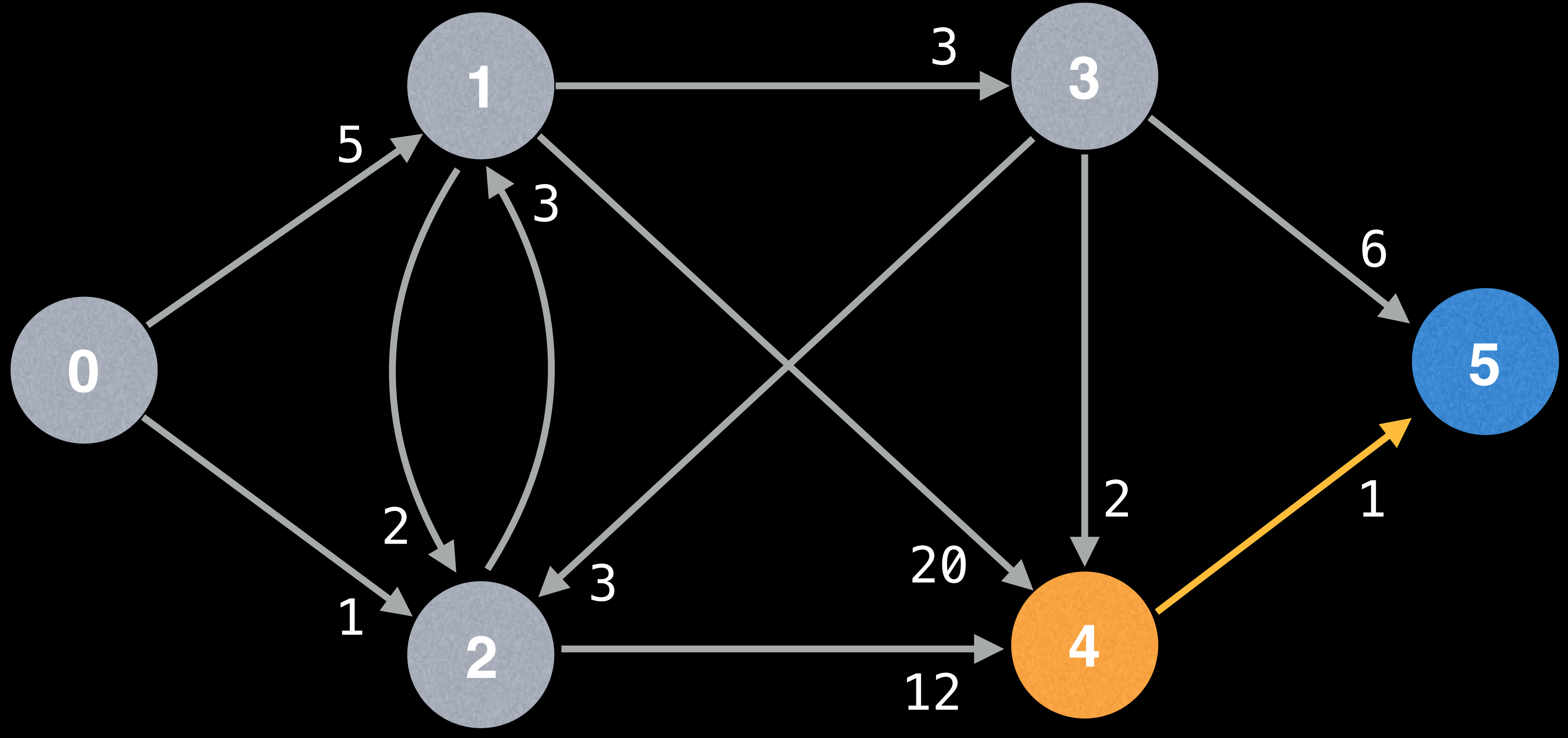


(index, dist)
key-value pairs

~~(0, 0)~~
~~(1, 4)~~
~~(2, 1)~~
→ (4, 9)
~~(3, 7)~~
(5, 13)

	0	1	2	3	4	5
dist =	0	4	1	7	9	13

Eager Dijkstra's



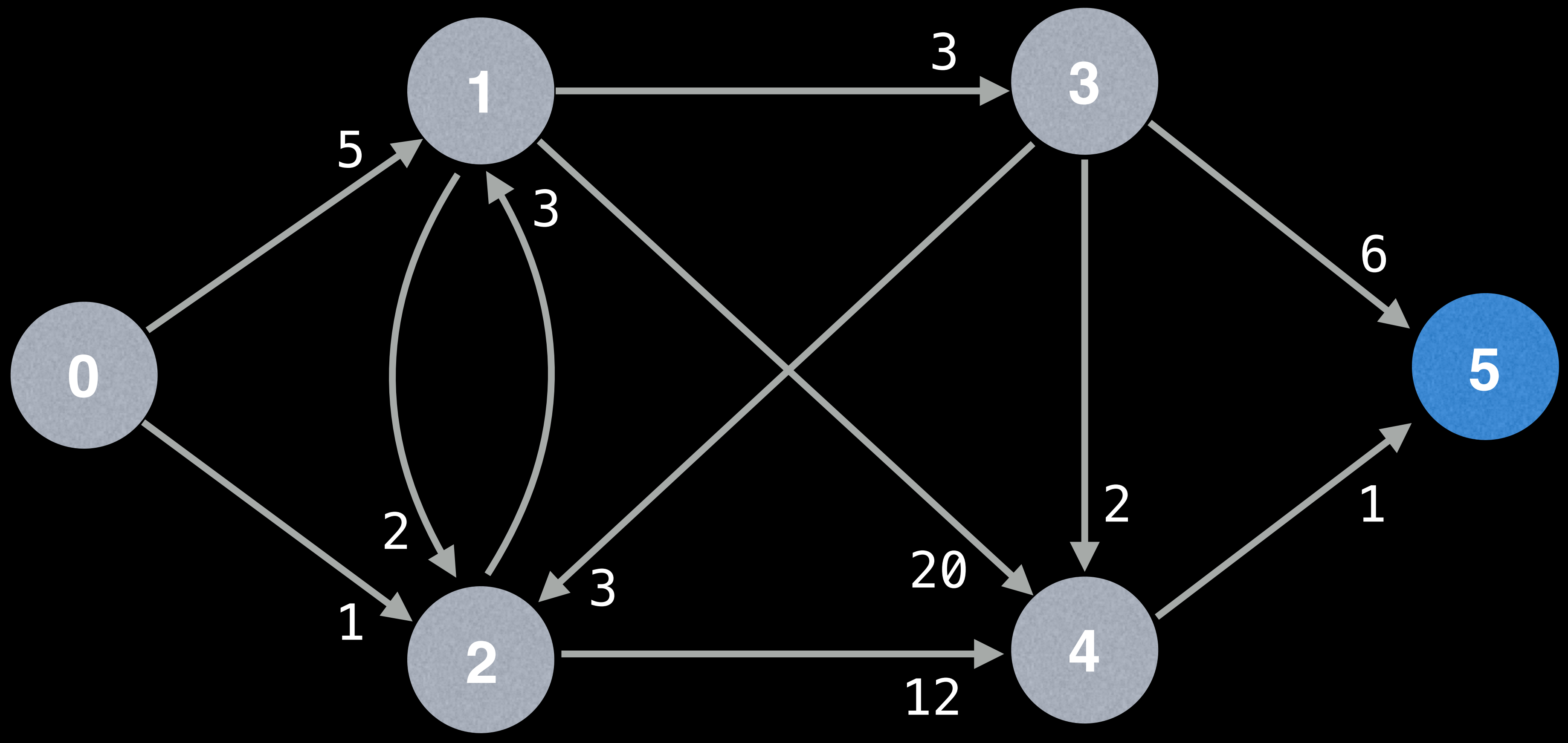
(index, dist)
key-value pairs

~~(0, 0)~~
~~(1, 4)~~
~~(2, 1)~~
→ (4, 9)
~~(3, 7)~~
(5, **10**)

	0	1	2	3	4	5
dist =	0	4	1	7	9	**10**

Best distance from node 4 to node 5:
dist[3] + edge.cost = 9 + 1 = 10

Eager Dijkstra's

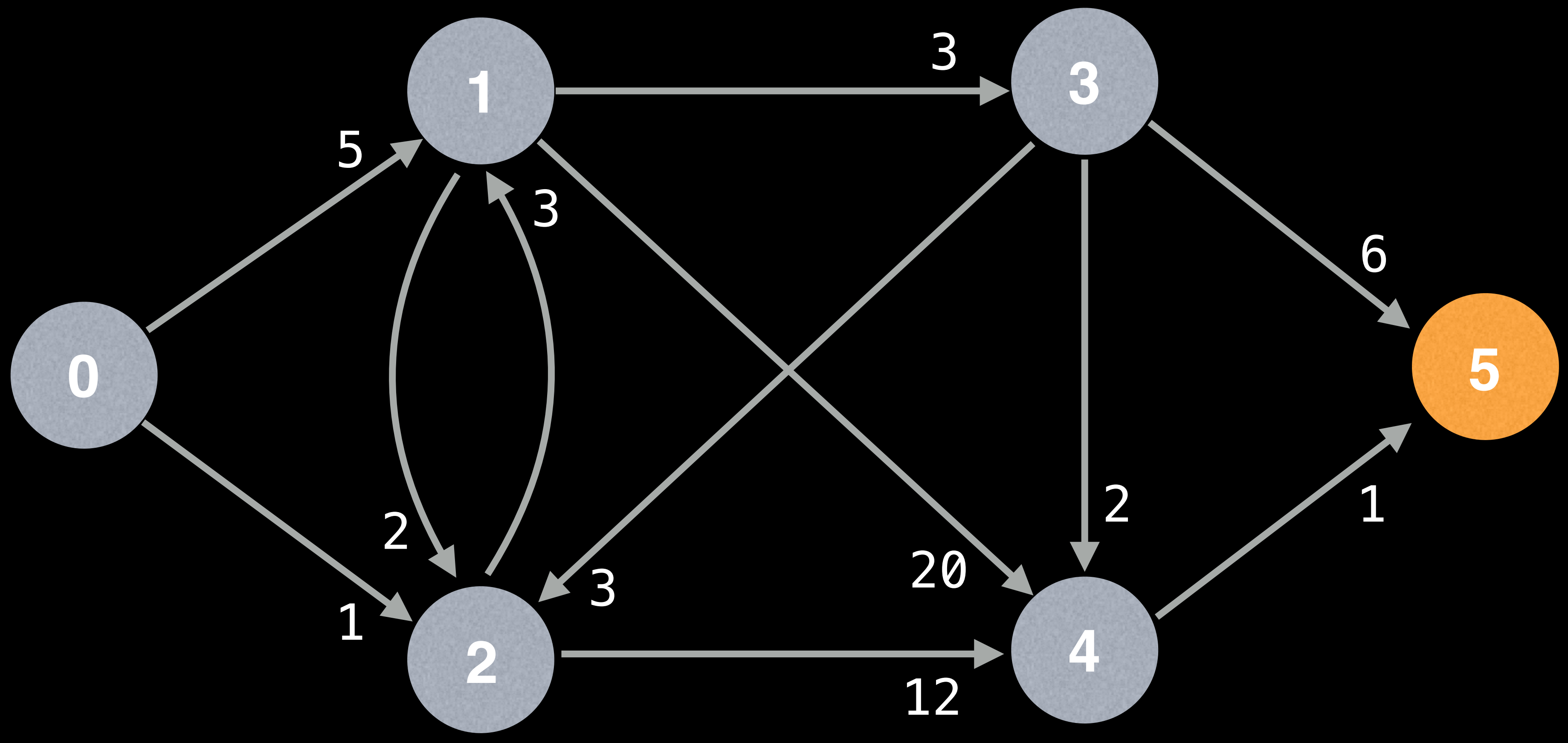


(index, dist) key-value pairs

~~(0, 0)~~
~~(1, 4)~~
~~(2, 1)~~
→ ~~(4, 9)~~
~~(3, 7)~~
(5, 10)

	0	1	2	3	4	5
dist =	0	4	1	7	9	10

Eager Dijkstra's

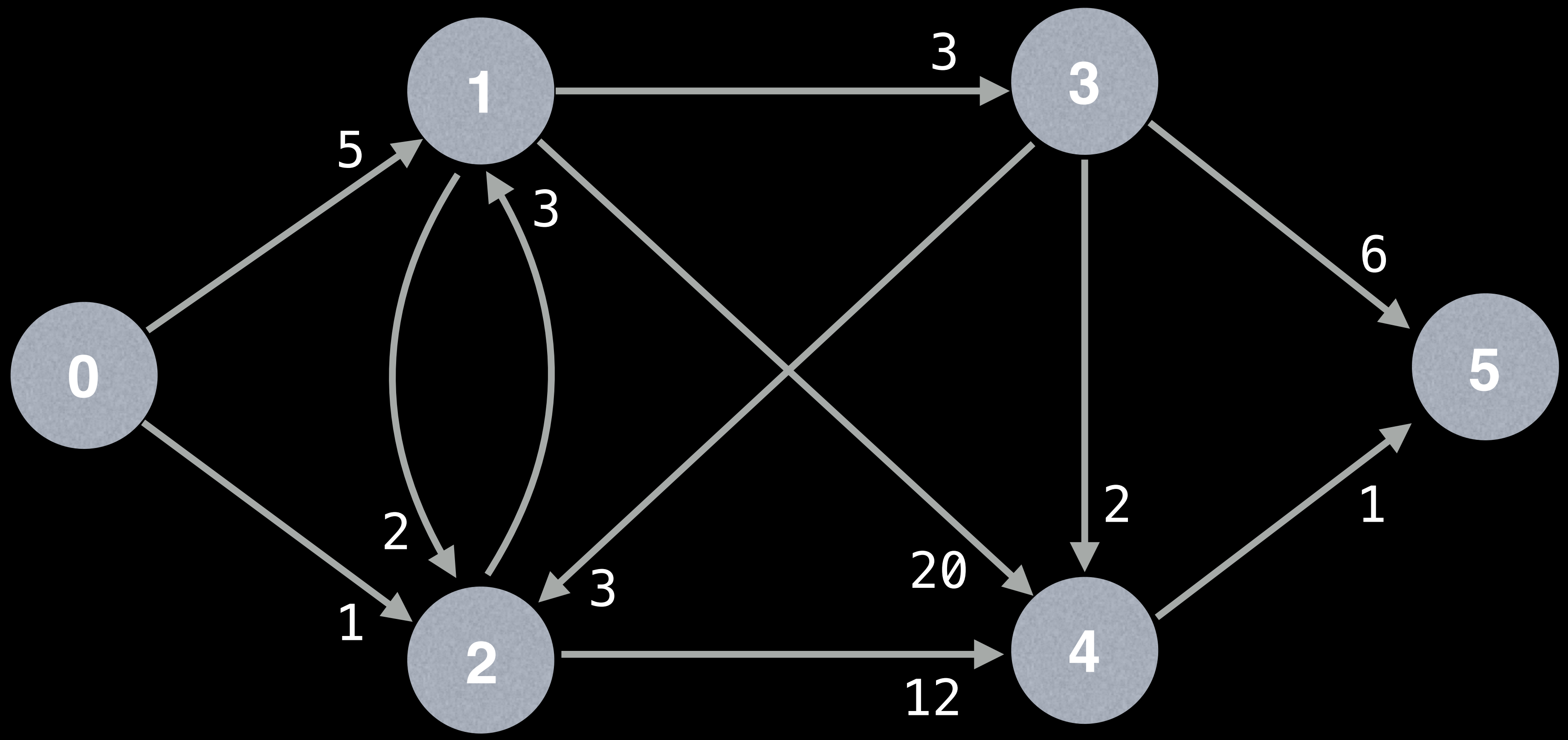


(index, dist) key-value pairs

~~(0, 0)~~
~~(1, 4)~~
~~(2, 1)~~
~~(4, 9)~~
~~(3, 7)~~
→ (5, 10)

	0	1	2	3	4	5
dist =	0	4	1	7	9	10

Eager Dijkstra's



(index, dist)
key-value pairs

~~(0, 0)~~
~~(1, 4)~~
~~(2, 1)~~
~~(4, 9)~~
~~(3, 7)~~
~~(5, 10)~~

dist =

0	1	2	3	4	5
0	4	1	7	9	10

```
# Runs Dijkstra's algorithm and returns an array that contains
# the shortest distance to every node from the start node s.
# g - adjacency list of weighted graph
# n - the number of nodes in the graph
# s - the index of the starting node (0 ≤ s < n)
function dijkstra(g, n, s):
  vis = [false, false, … , false] # size n
  dist = [∞, ∞, … ∞, ∞] # size n
  dist[s] = 0
  ipq = empty indexed priority queue
  ipq.insert(s, 0)
  while ipq.size() != 0:
    index, minValue = ipq.poll()
    vis[index] = true
    if dist[index] < minValue: continue
    for (edge : g[index]):
      if vis[edge.to]: continue
      newDist = dist[index] + edge.cost
      if newDist < dist[edge.to]:
        dist[edge.to] = newDist
        if !ipq.contains(edge.to):
          ipq.insert(edge.to, newDist)
        else:
          ipq.decreaseKey(edge.to, newDist)
  return dist
```

```
# Runs Dijkstra’s algorithm and returns an array that contains
# the shortest distance to every node from the start node s.
# g - adjacency list of weighted graph
# n - the number of nodes in the graph
# s - the index of the starting node (0 ≤ s < n)
function dijkstra(g, n, s):
  vis = [false, false, … , false] # size n
  dist = [∞, ∞, … ∞, ∞] # size n
  dist[s] = 0
  ipq = empty indexed priority queue
  ipq.insert(s, 0)
  while ipq.size() != 0:
    index, minValue = ipq.poll()
    vis[index] = true
    if dist[index] < minValue: continue
    for (edge : g[index]):
      if vis[edge.to]: continue
      newDist = dist[index] + edge.cost
      if newDist < dist[edge.to]:
        dist[edge.to] = newDist
        if !ipq.contains(edge.to):
          ipq.insert(edge.to, newDist)
        else:
          ipq.decreaseKey(edge.to, newDist)
  return dist
```

```
# Runs Dijkstra's algorithm and returns an array that contains
# the shortest distance to every node from the start node s.
# g - adjacency list of weighted graph
# n - the number of nodes in the graph
# s - the index of the starting node (0 ≤ s < n)
function dijkstra(g, n, s):
  vis = [false, false, … , false] # size n
  dist = [∞, ∞, … ∞, ∞] # size n
  dist[s] = 0
  ipq = empty indexed priority queue
  ipq.insert(s, 0)
  while ipq.size() != 0:
    index, minValue = ipq.poll()
    vis[index] = true
    if dist[index] < minValue: continue
    for (edge : g[index]):
      if vis[edge.to]: continue
      newDist = dist[index] + edge.cost
      if newDist < dist[edge.to]:
        dist[edge.to] = newDist
        if !ipq.contains(edge.to):
          ipq.insert(edge.to, newDist)
        else:
          ipq.decreaseKey(edge.to, newDist)
  return dist
```

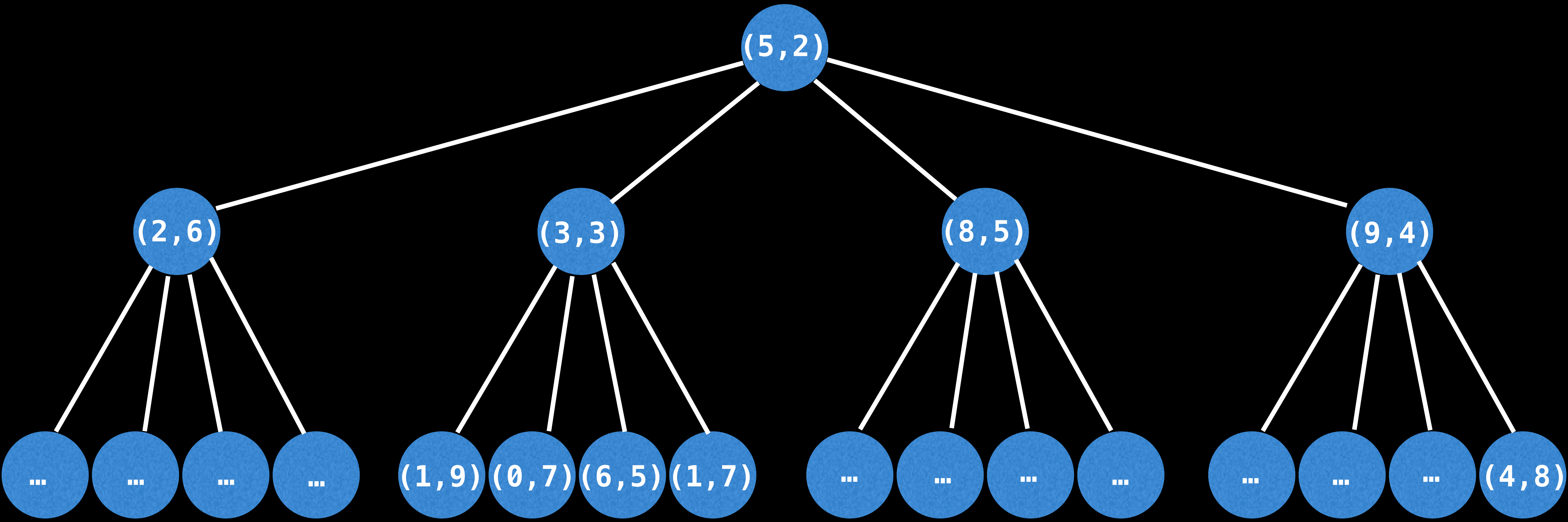
The main advantage to using decreaseKey is to prevent duplicate node indexes to be present in the PQ.

D-ary Heap optimization

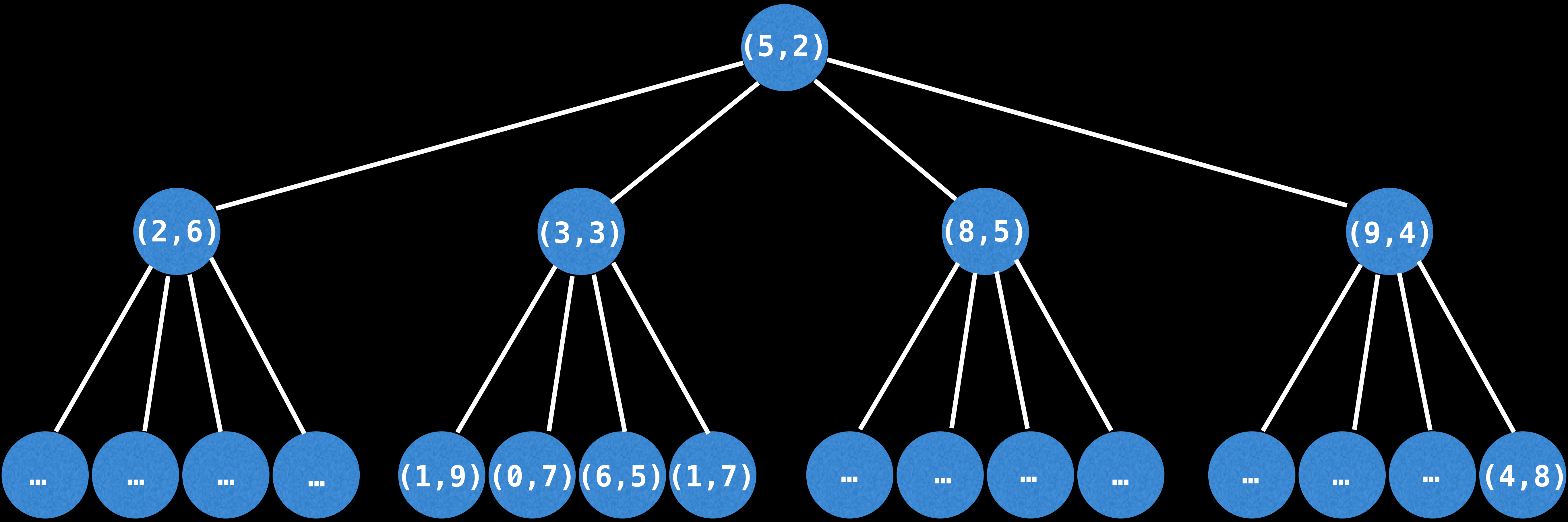
When executing Dijkstra’s algorithm, especially on dense graphs, there are a lot more updates (i.e decreaseKey operations) to key-value pairs than there are dequeue (poll) operations.

A **D-ary heap** is a heap variant in which each node has D children. This speeds up decrease key operations at the expense of more costly removals.

D-ary Heap (with D = 4)

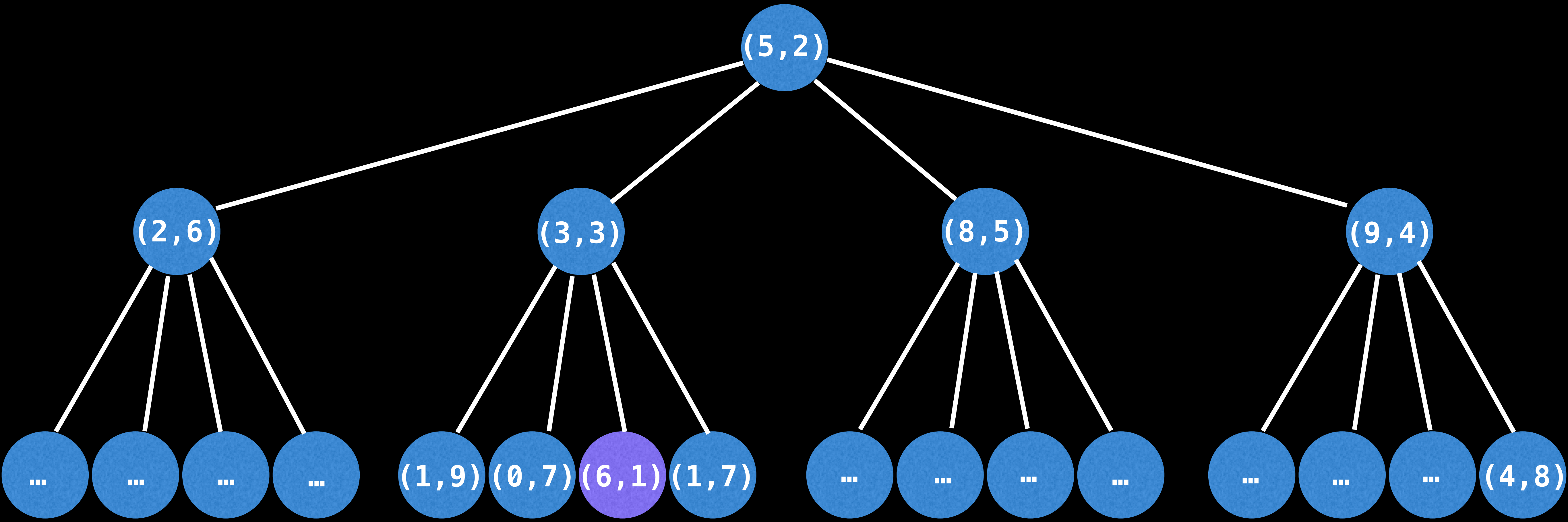


D-ary Heap (with D = 4)



Suppose we want to update the node with index 6 to have a new shortest distance of 1 (a.k.a **decreaseKey**(6, 1))

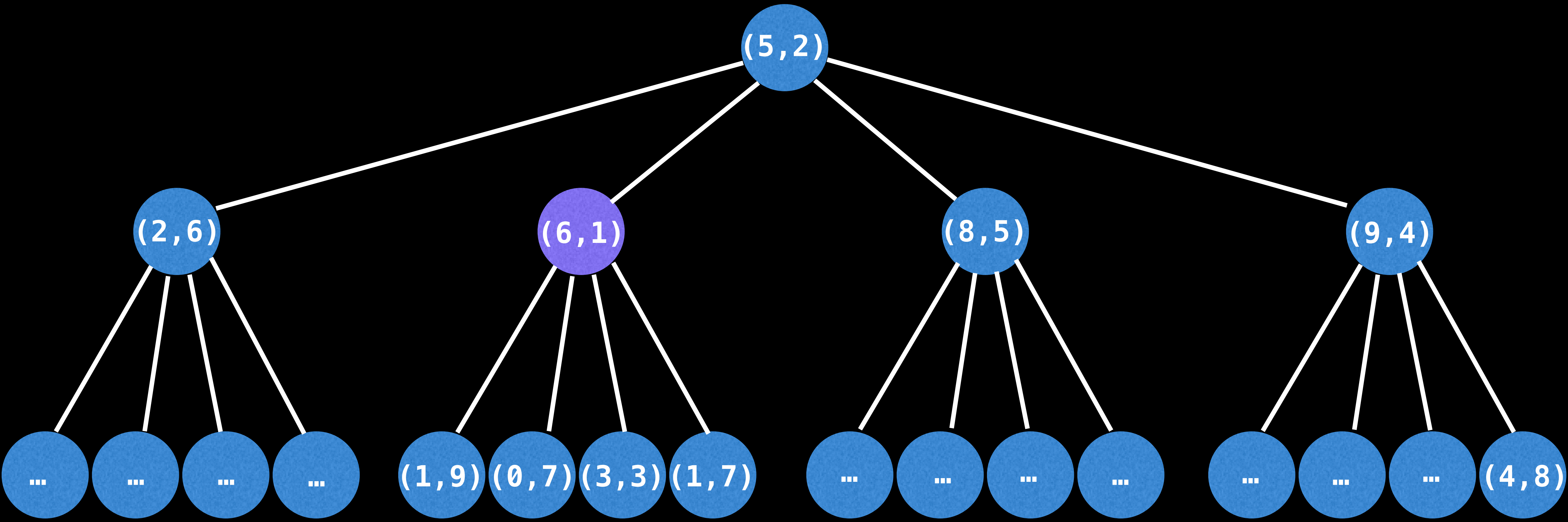
D-ary Heap (with D = 4)



Suppose we want to update the node with index 6 to have a new shortest distance of 1 (a.k.a **decreaseKey**(6, 1))

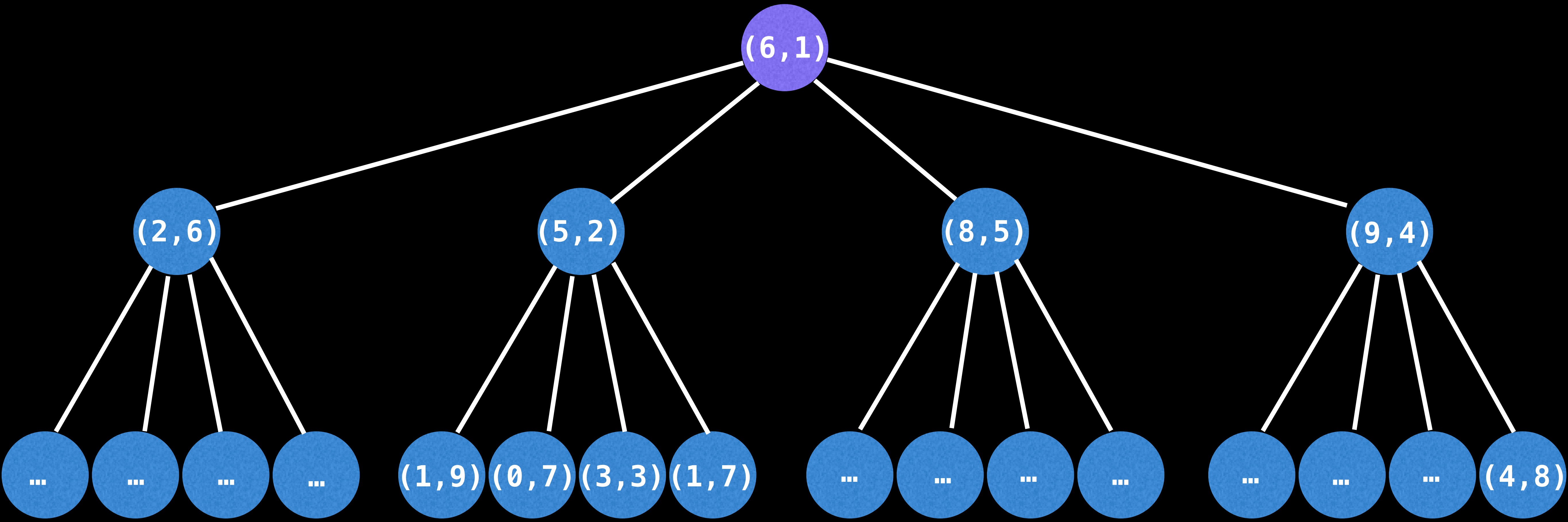
Assuming we have an Indexed D-ary Heap we can update the key's value in **O(1)** but we still need to adjust its position.

D-ary Heap (with D = 4)



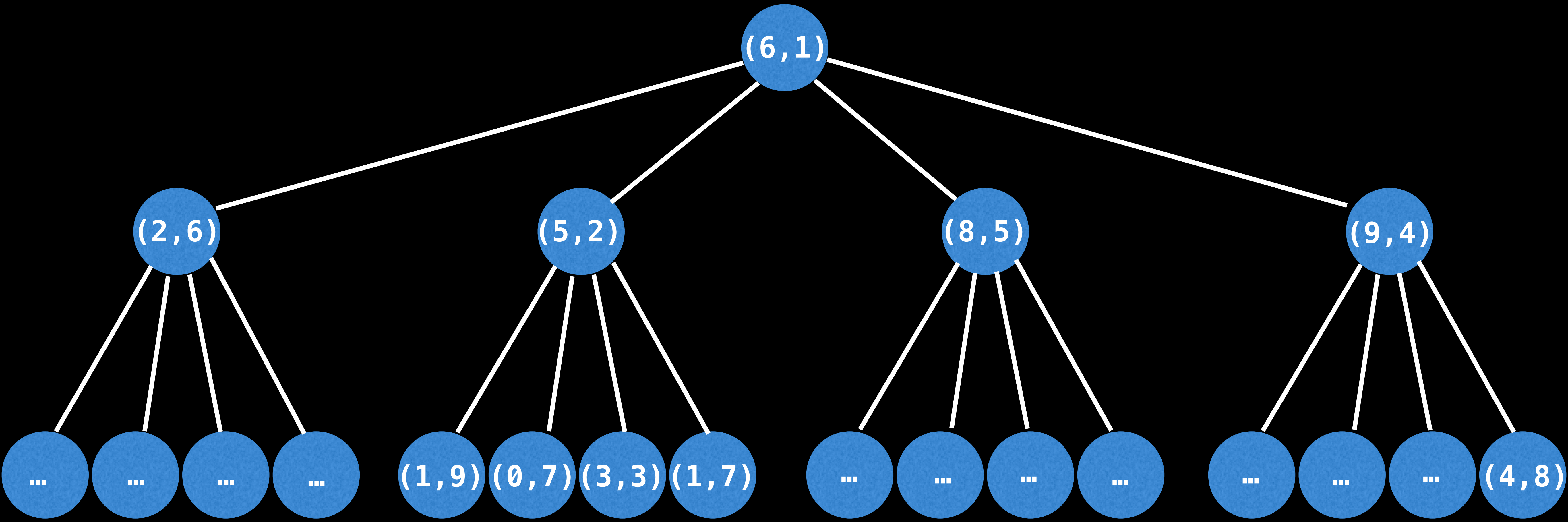
Suppose we want to update the node with index 6 to have a new shortest distance of 1 (a.k.a **decreaseKey**(6, 1))

D-ary Heap (with D = 4)



Suppose we want to update the node with index 6 to have a new shortest distance of 1 (a.k.a **decreaseKey**(6, 1))

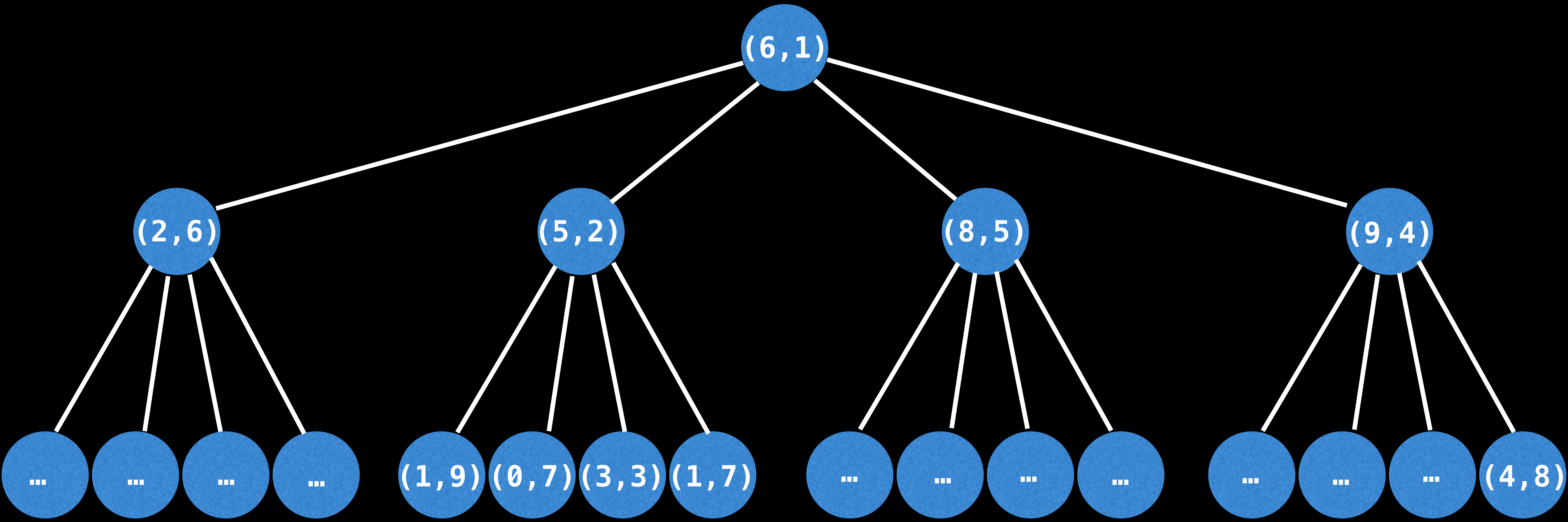
D-ary Heap (with D = 4)



Suppose we want to update the node with index 6 to have a new shortest distance of 1 (a.k.a **decreaseKey**(6, 1))

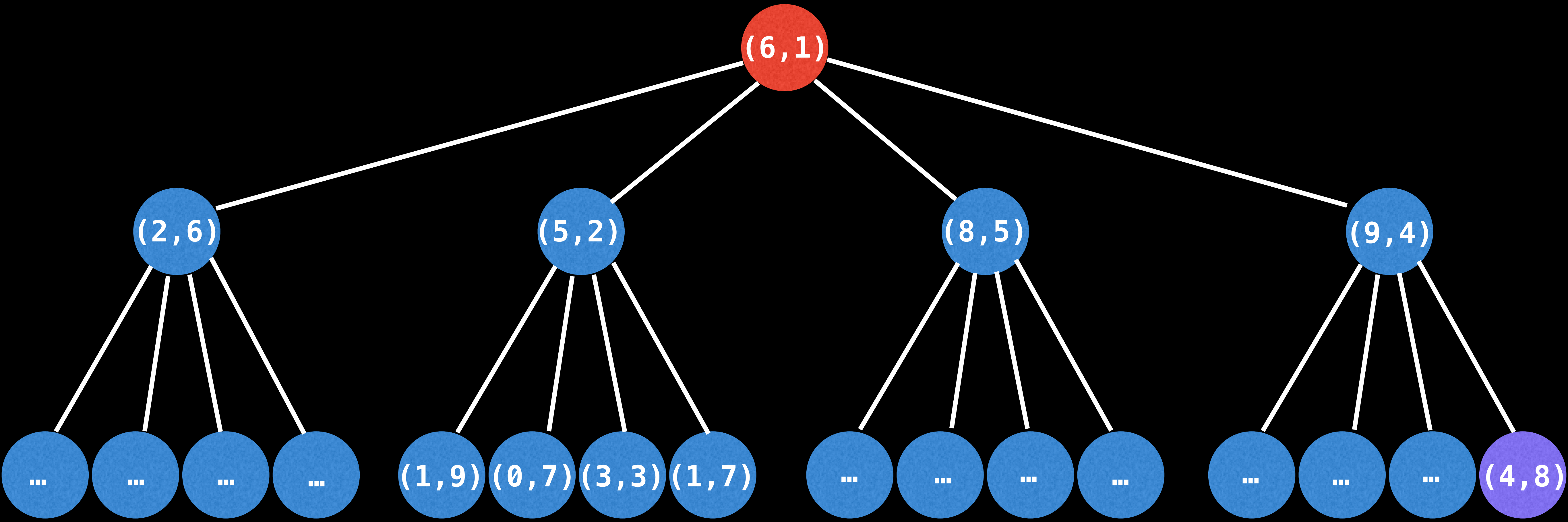
The whole update took only two operations because the heap was very flat.

D-ary Heap (with D = 4)



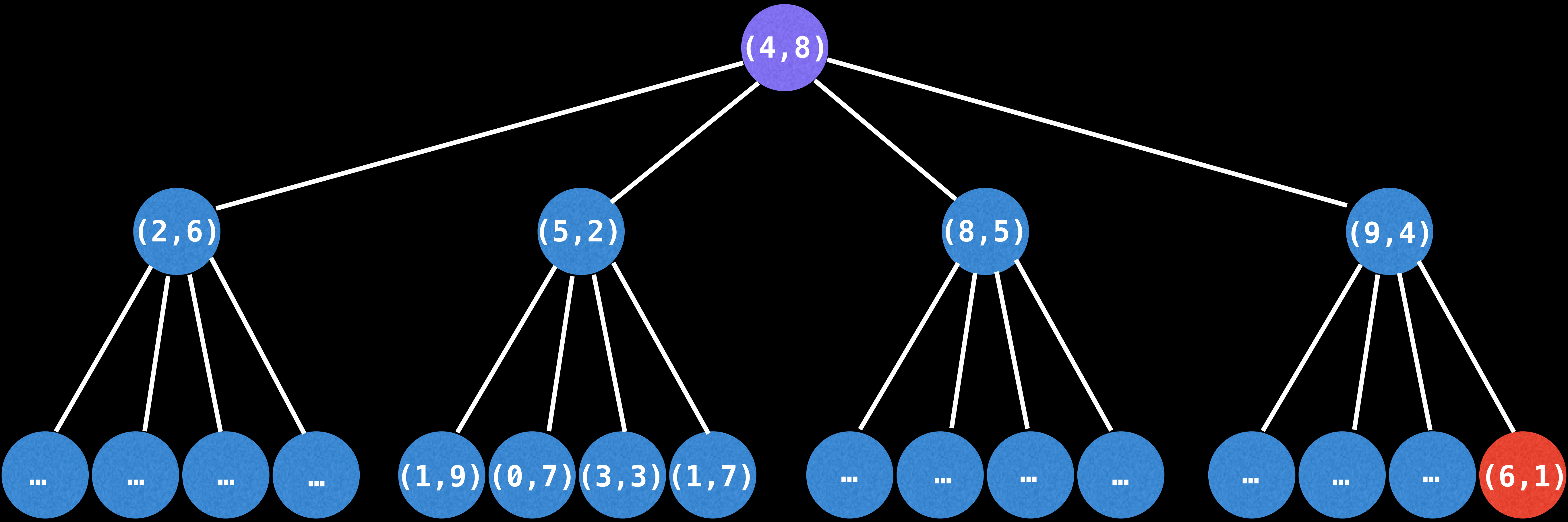
In contrast suppose we want to remove the root node.

D-ary Heap (with D = 4)



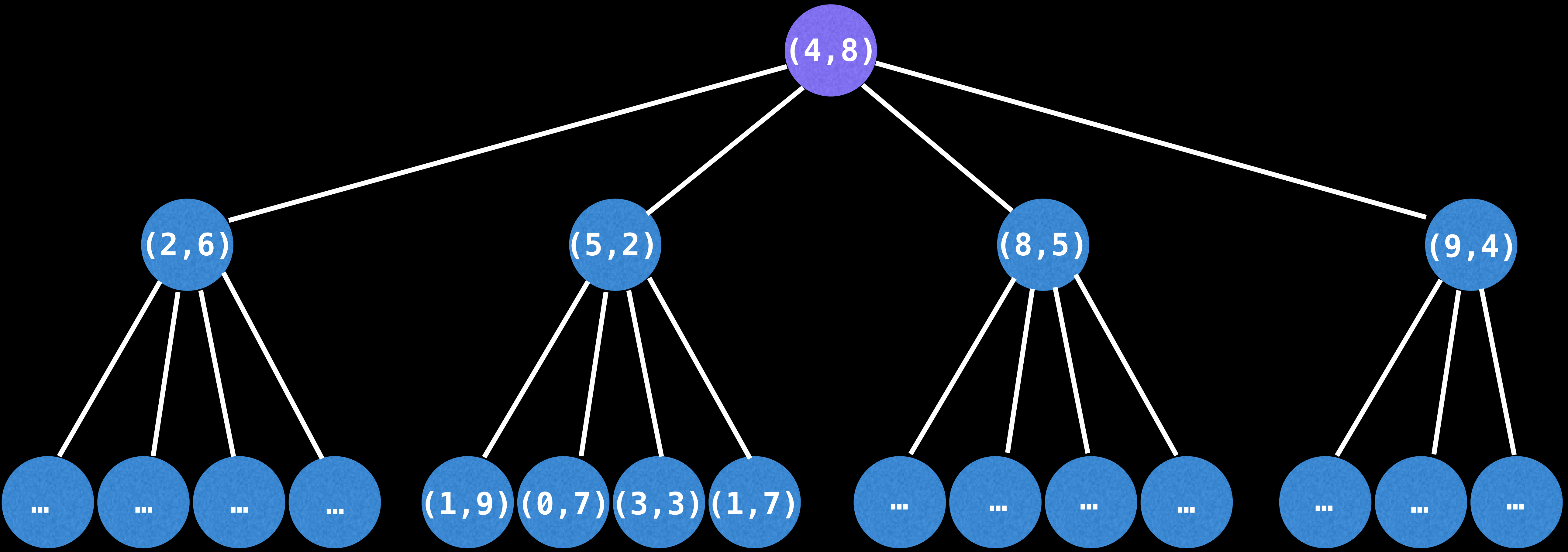
In contrast suppose we want to remove the root node.

D-ary Heap (with D = 4)



In contrast suppose we want to remove the root node.

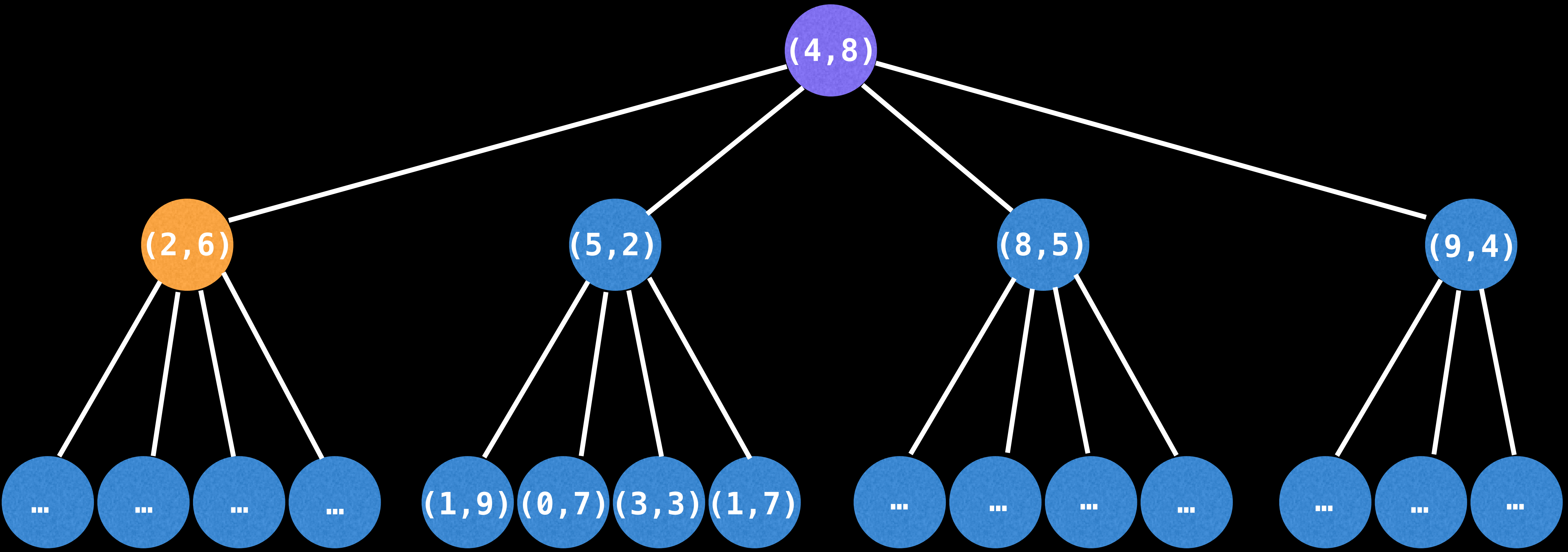
D-ary Heap (with D = 4)



In contrast suppose we want to remove the root node.

In a D-ary heap we have to search D children of the current node to find the minimum (k, v) pair to swap downwards.

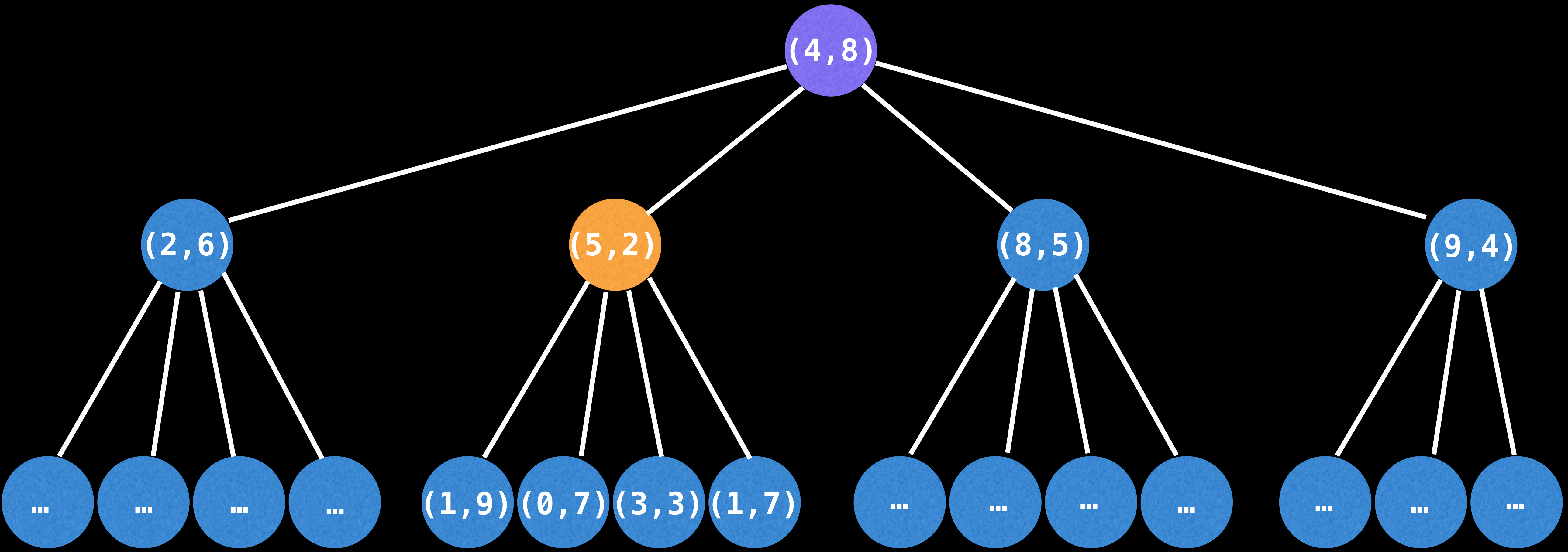
D-ary Heap (with D = 4)



In contrast suppose we want to remove the root node.

In a D-ary heap we have to search D children of the current node to find the minimum (k, v) pair to swap downwards.

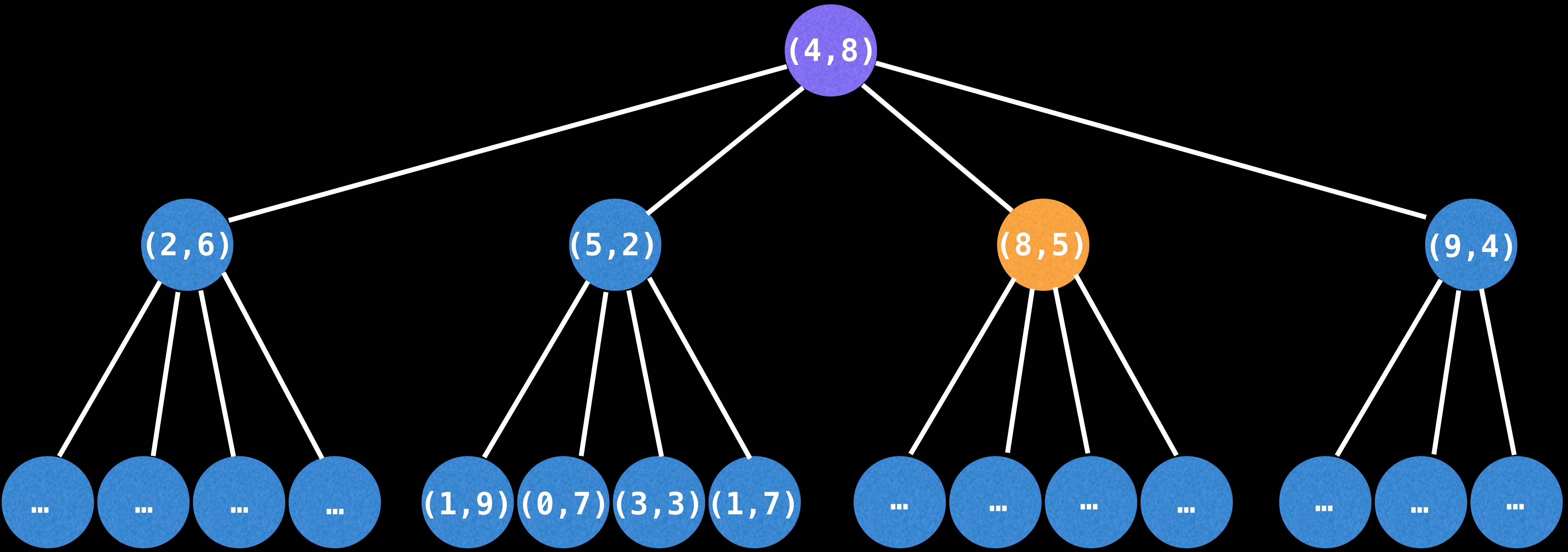
D-ary Heap (with D = 4)



In contrast suppose we want to remove the root node.

In a D-ary heap we have to search D children of the current node to find the minimum (k, v) pair to swap downwards.

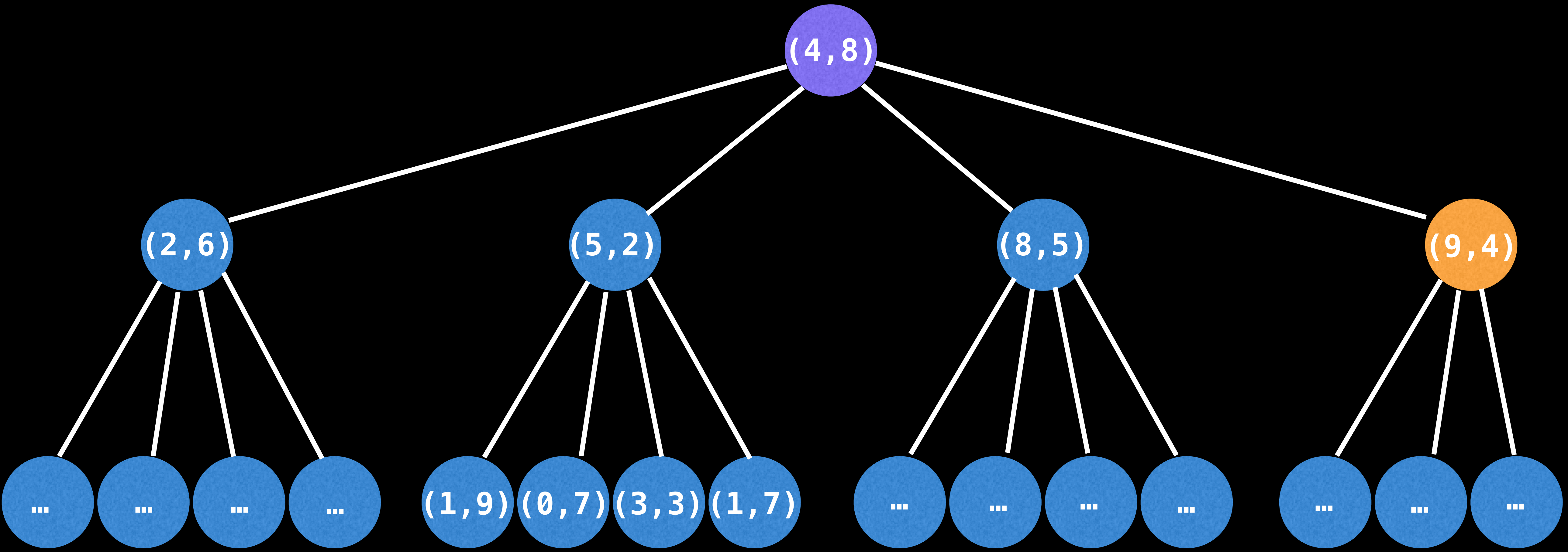
D-ary Heap (with D = 4)



In contrast suppose we want to remove the root node.

In a D-ary heap we have to search D children of the current node to find the minimum (k, v) pair to swap downwards.

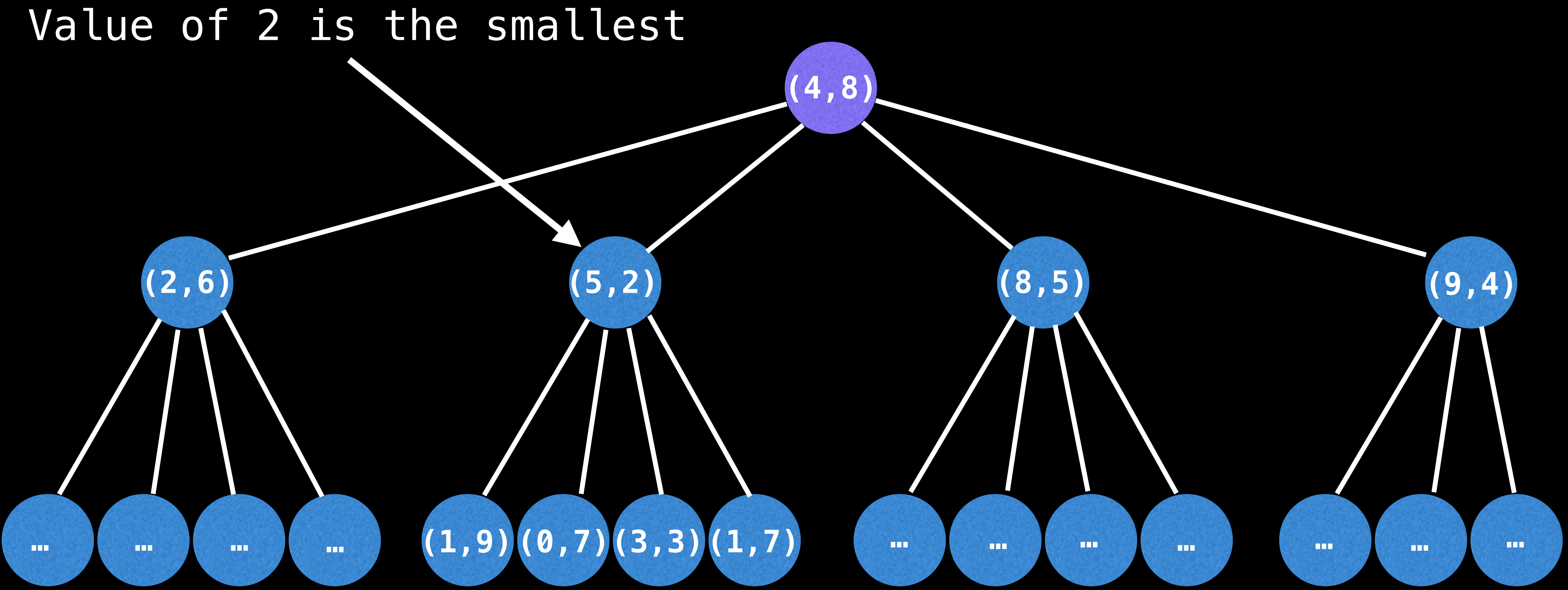
D-ary Heap (with D = 4)



In contrast suppose we want to remove the root node.

In a D-ary heap we have to search D children of the current node to find the minimum (k, v) pair to swap downwards.

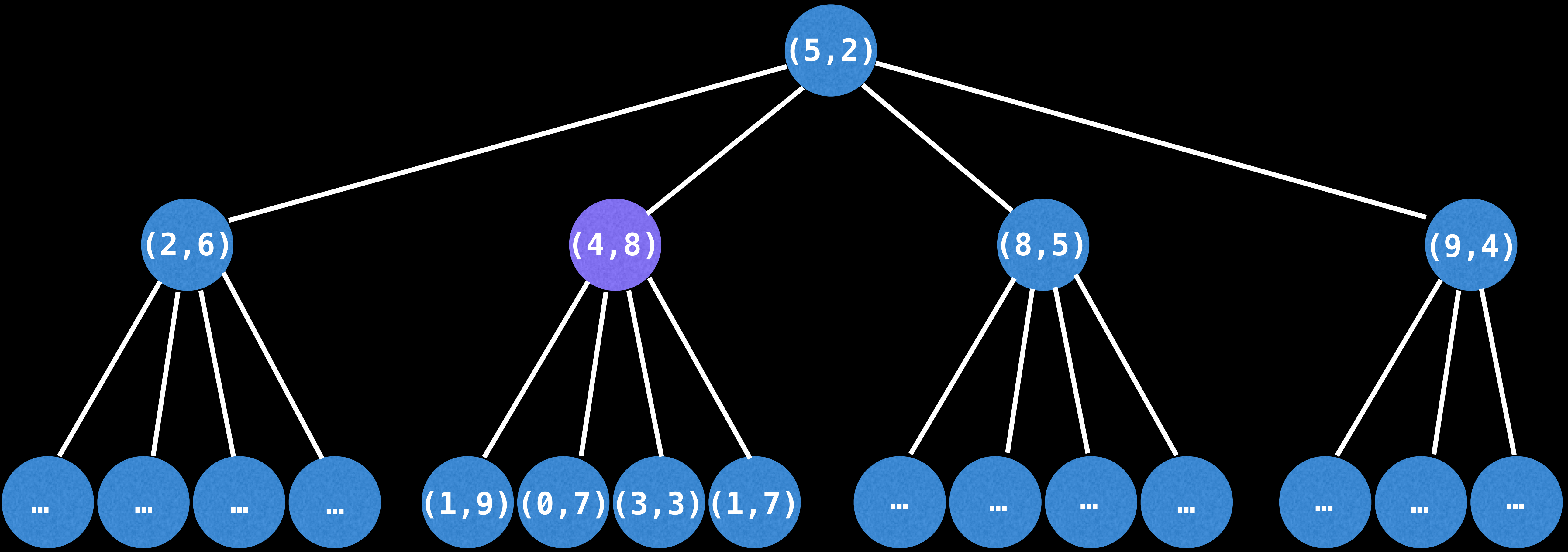
D-ary Heap (with D = 4)



In contrast suppose we want to remove the root node.

In a D-ary heap we have to search D children of the current node to find the minimum (k, v) pair to swap downwards.

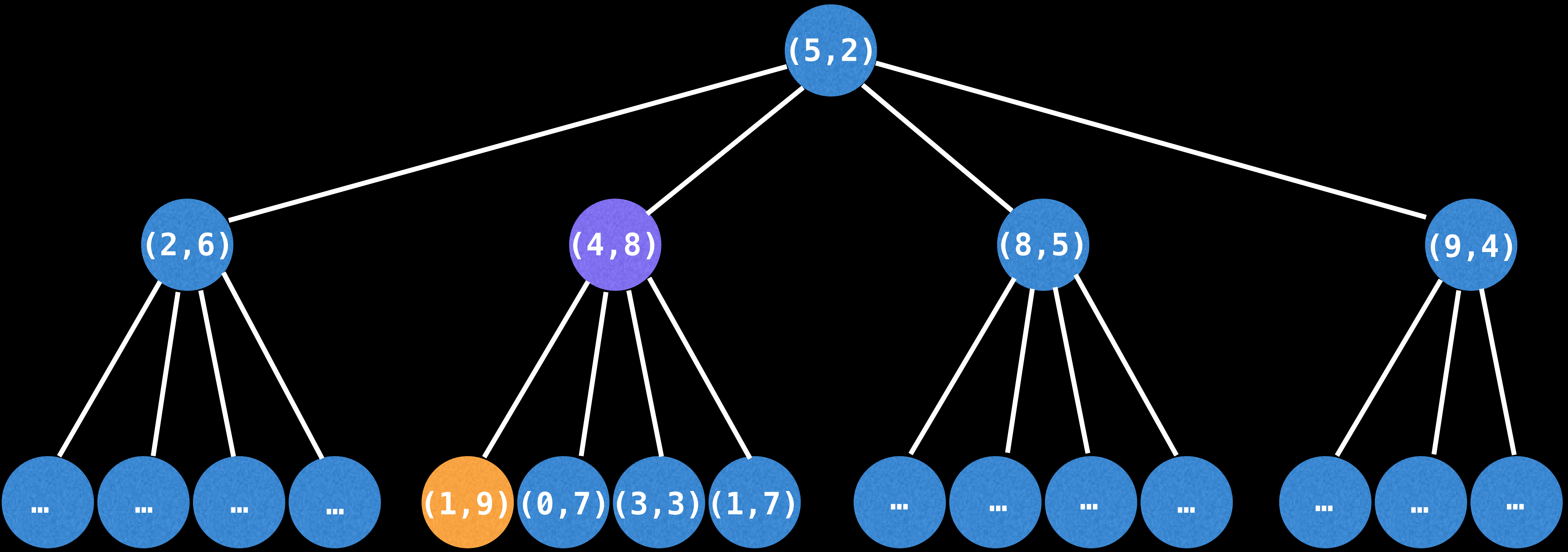
D-ary Heap (with D = 4)



In contrast suppose we want to remove the root node.

In a D-ary heap we have to search D children of the current node to find the minimum (k, v) pair to swap downwards.

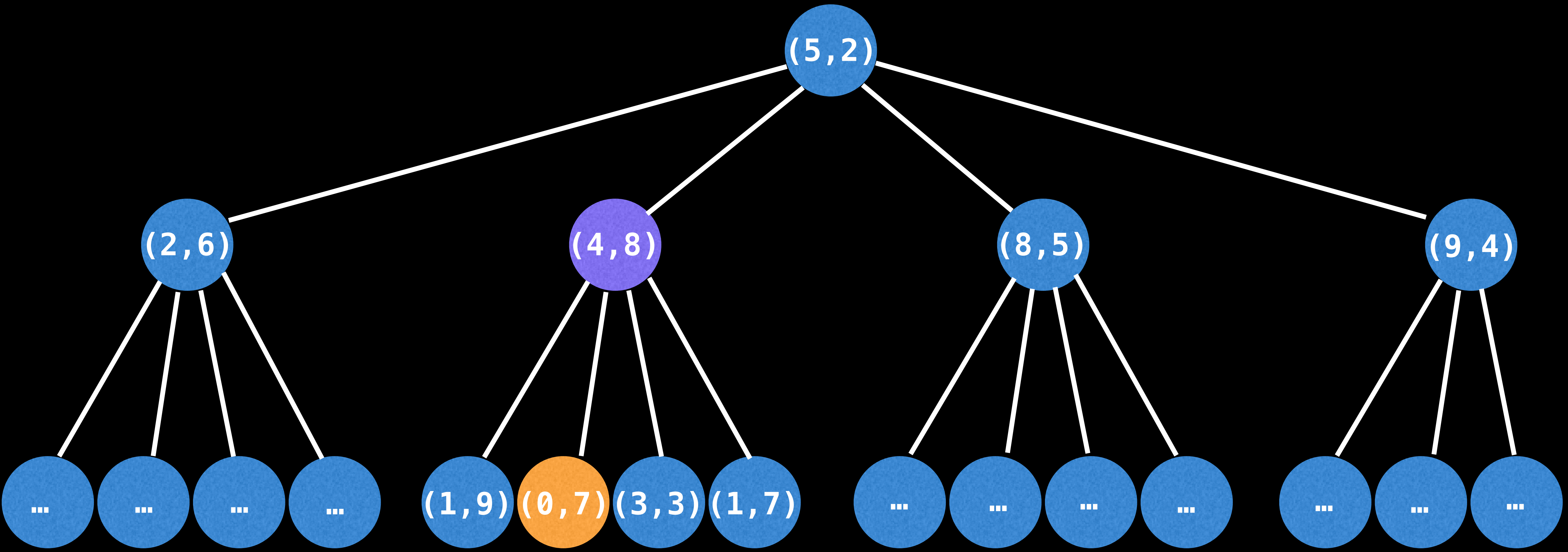
D-ary Heap (with D = 4)



In contrast suppose we want to remove the root node.

In a D-ary heap we have to search D children of the current node to find the minimum (k, v) pair to swap downwards.

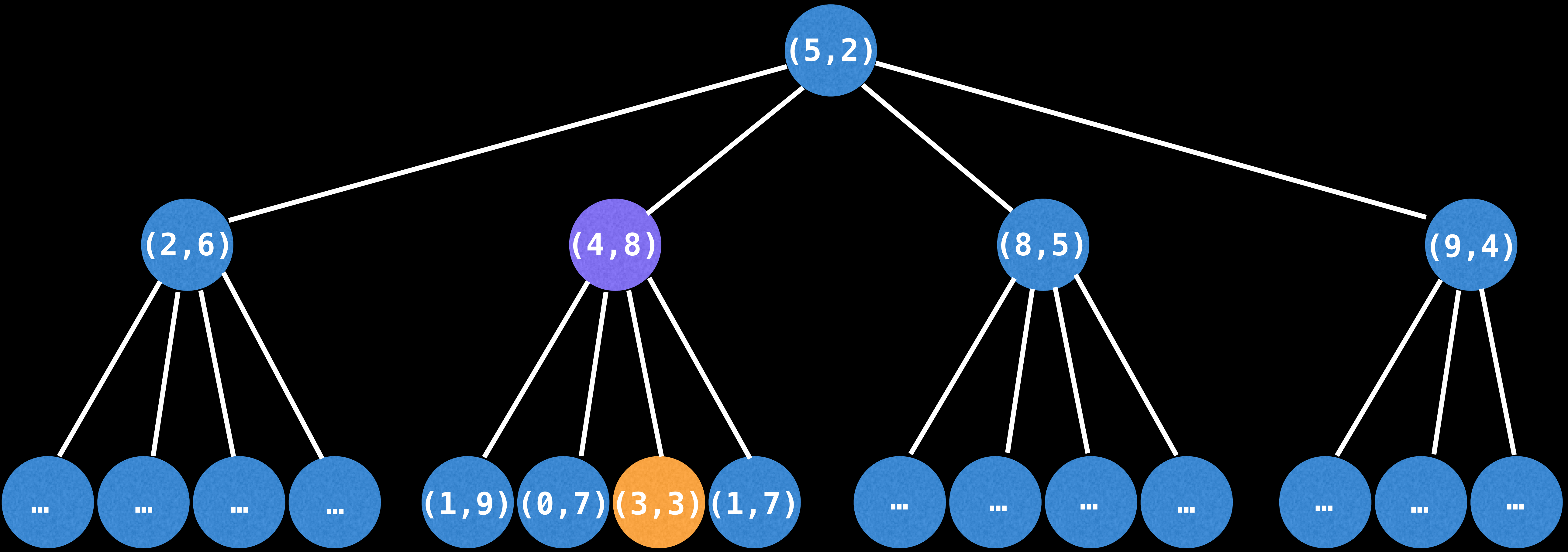
D-ary Heap (with D = 4)



In contrast suppose we want to remove the root node.

In a D-ary heap we have to search D children of the current node to find the minimum (k, v) pair to swap downwards.

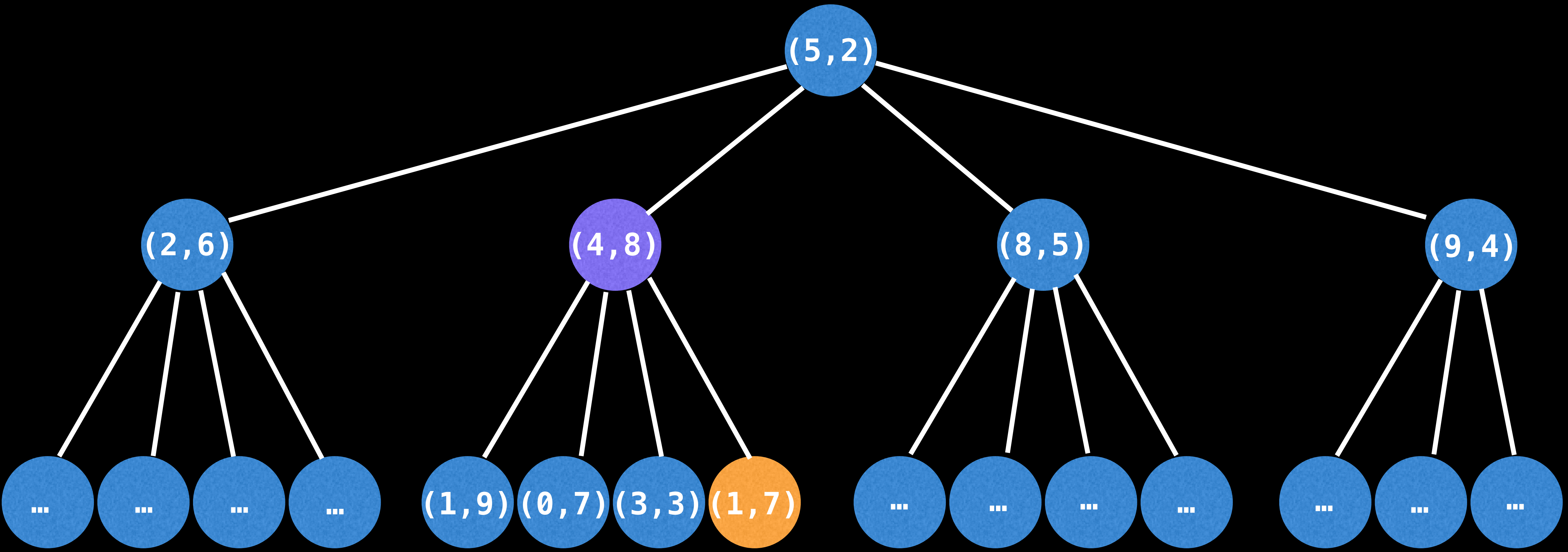
D-ary Heap (with D = 4)



In contrast suppose we want to remove the root node.

In a D-ary heap we have to search D children of the current node to find the minimum (k, v) pair to swap downwards.

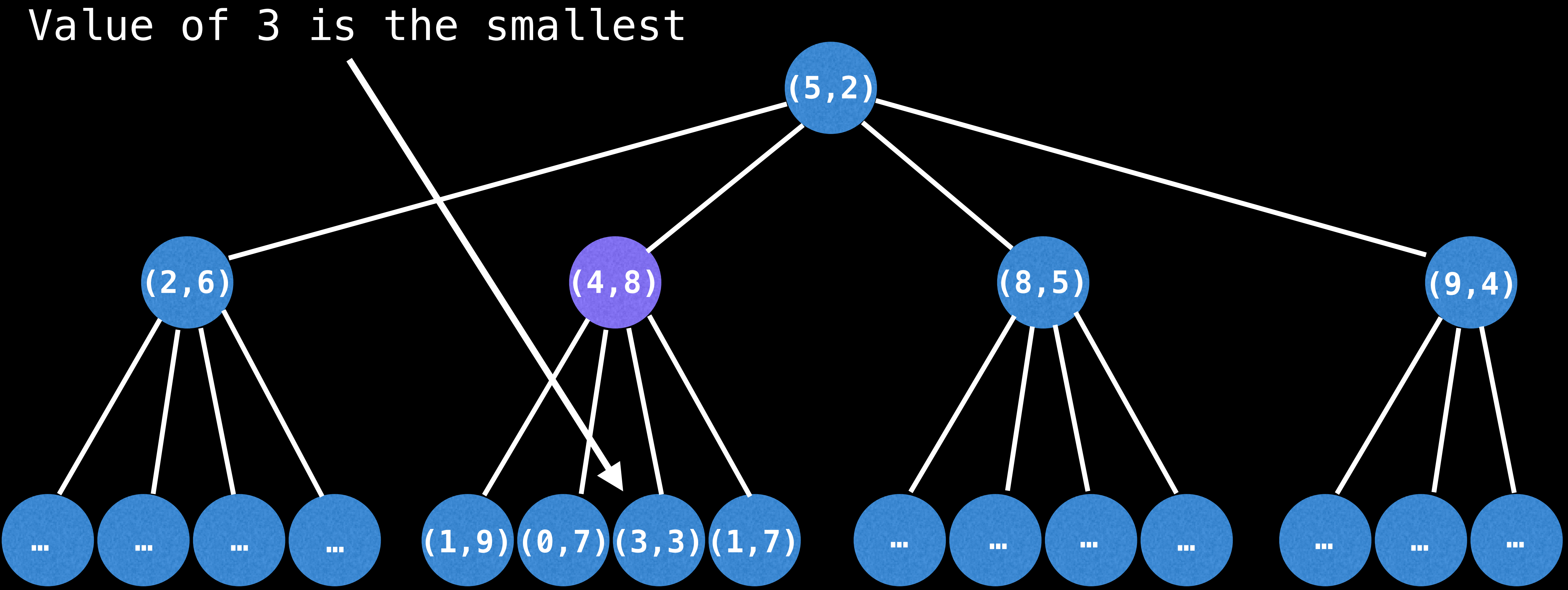
D-ary Heap (with D = 4)



In contrast suppose we want to remove the root node.

In a D-ary heap we have to search D children of the current node to find the minimum (k, v) pair to swap downwards.

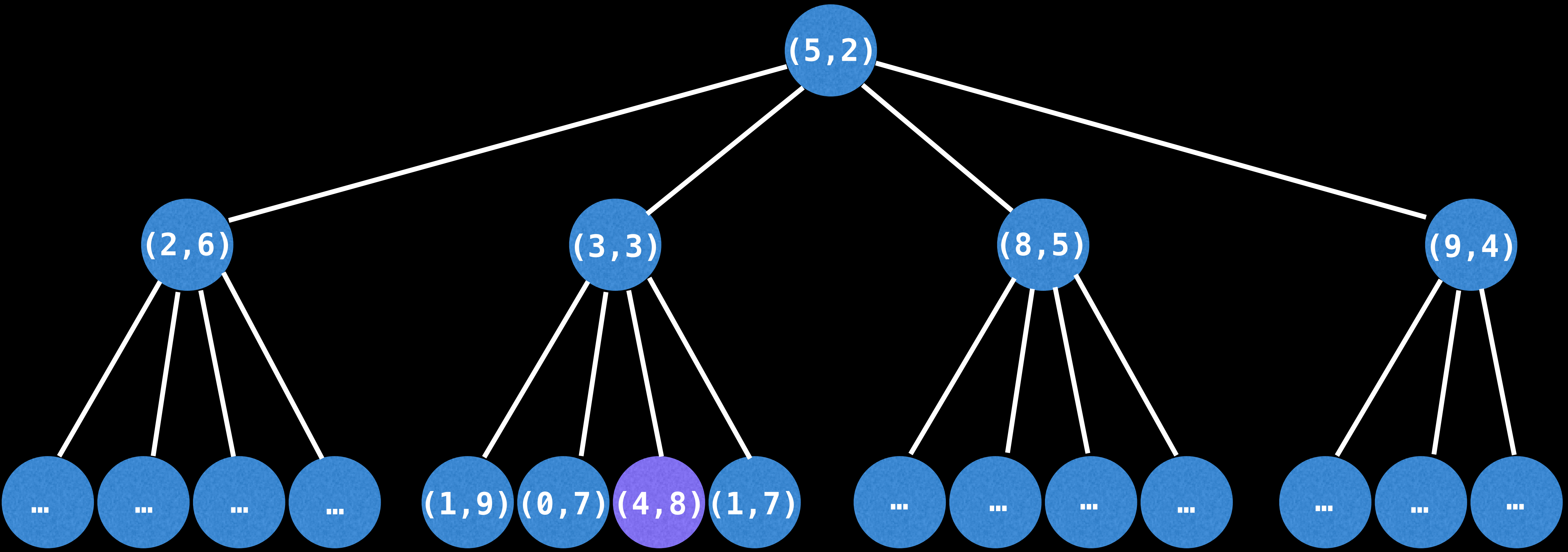
D-ary Heap (with D = 4)



In contrast suppose we want to remove the root node.

In a D-ary heap we have to search D children of the current node to find the minimum (k, v) pair to swap downwards.

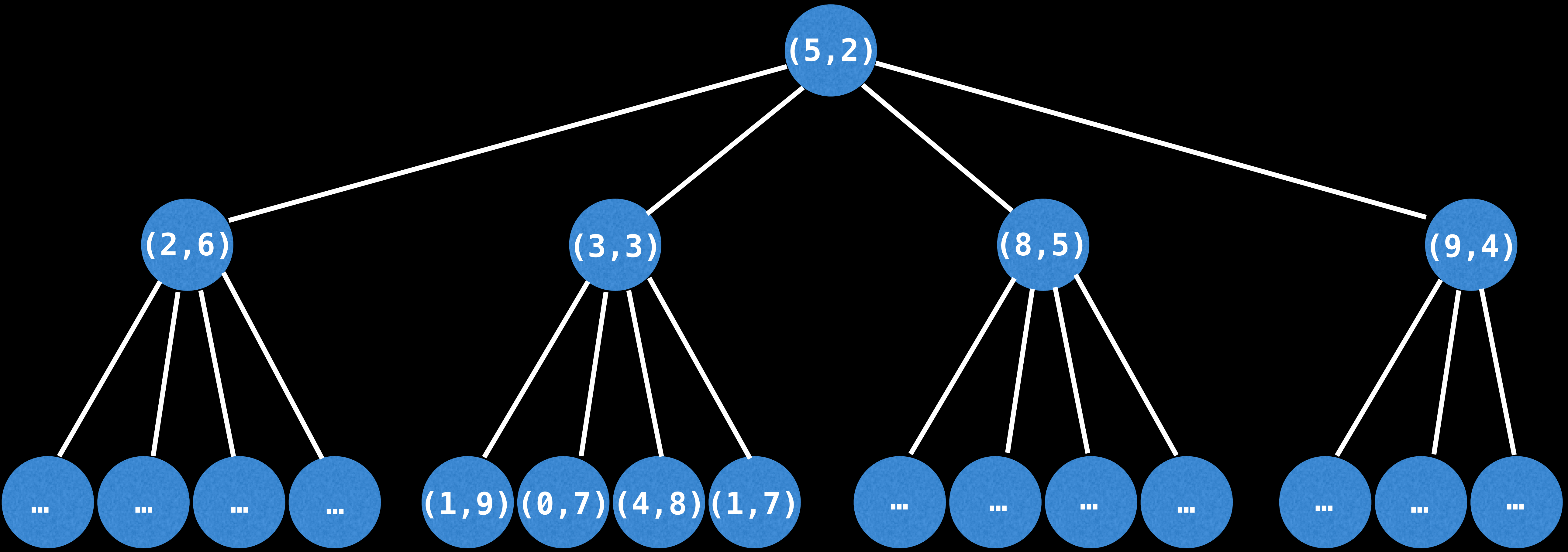
D-ary Heap (with D = 4)



In contrast suppose we want to remove the root node.

In a D-ary heap we have to search D children of the current node to find the minimum (k, v) pair to swap downwards.

D-ary Heap (with D = 4)



Removals are clearly much more expensive, but they are also a lot less common in Dijkstra's than decreaseKey operations.

Optimal D-ary Heap degree

Q: What is the optimal D-ary heap degree to maximize performance of Dijkstra's algorithm?

A: In general **$D = E/V$** is the best degree to use to balance removals against decreaseKey operations improving Dijkstra's time complexity to **$O(E*\log_{E/V}(V))$** which is much better especially for dense graphs which have lots of decreaseKey operations.

The state of the art

The current state of the art as of now is the **Fibonacci heap** which gives Dijkstra's algorithm a time complexity of **O(E + Vlog(V))**

However, in practice, Fibonacci heaps are very **difficult to implement** and have a **large enough constant amortized overhead** to make them impractical unless your graph is quite large.

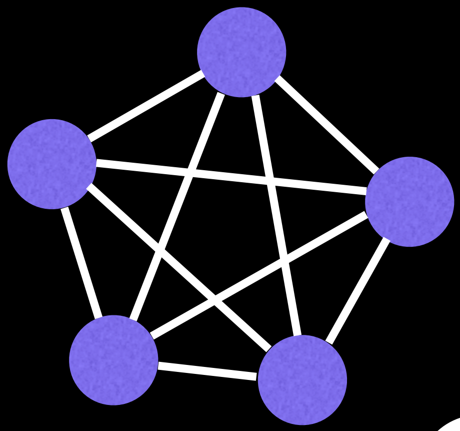
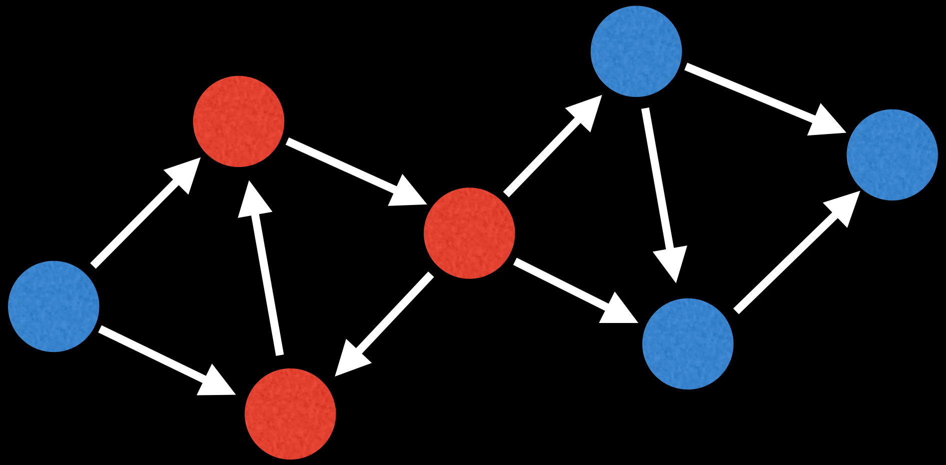
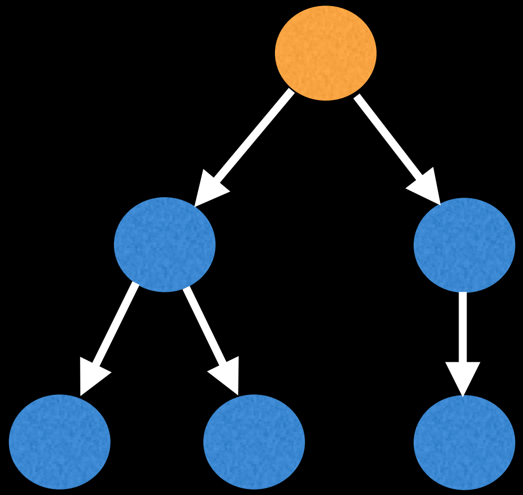
Source Code and Slides

Implementation **source code** and **slides** can be found at the following link:

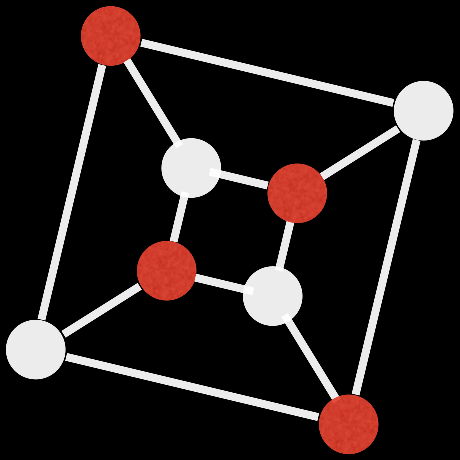
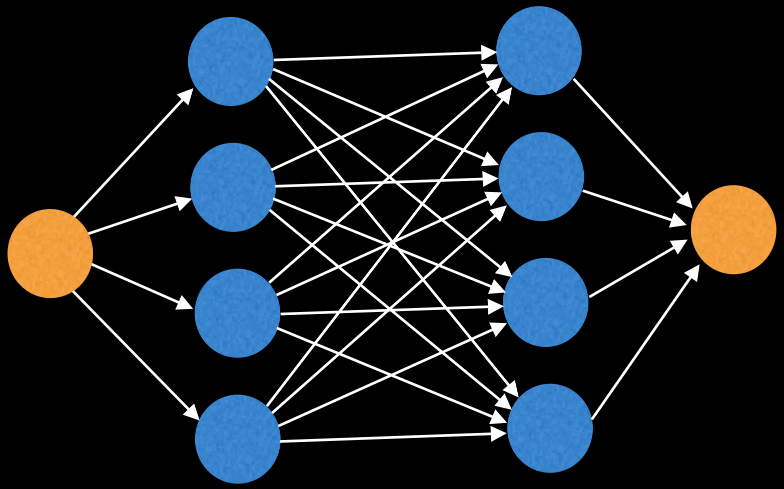
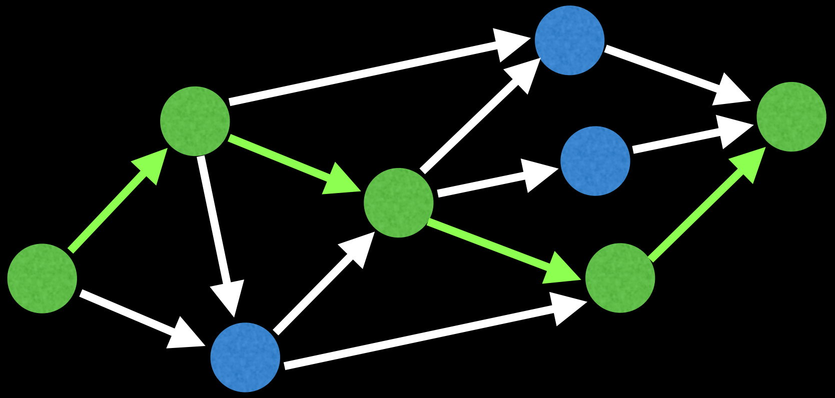
github.com/williamfiset/algorithms

Link in the description:

Next Video: Dijkstra source code

Graph Theory Video Series

Dijkstra’s Shortest Path Algorithm Source Code

William Fiset

Previous Video

Source Code and Slides

Implementation **source code** and **slides** can be found at the following link:

github.com/williamfiset/algorithms

Link in the description: