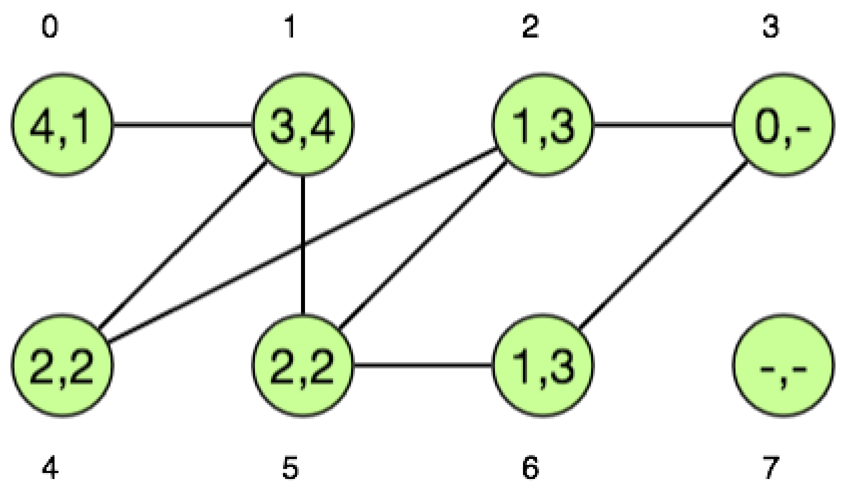
The breadth-first search algorithm

Breadth-first search assigns two values to each vertex **v**:

- A **distance**, giving the minimum number of edges in any path from the source vertex to vertex **v**.
- The **predecessor** vertex of **v** along some shortest path from the source vertex. The source vertex's predecessor is some special value, such as *null*, indicating that it has no predecessor.

If there is no path from the source vertex to vertex **v**, then **v**'s distance is infinite and its predecessor has the same special value as the source's predecessor.

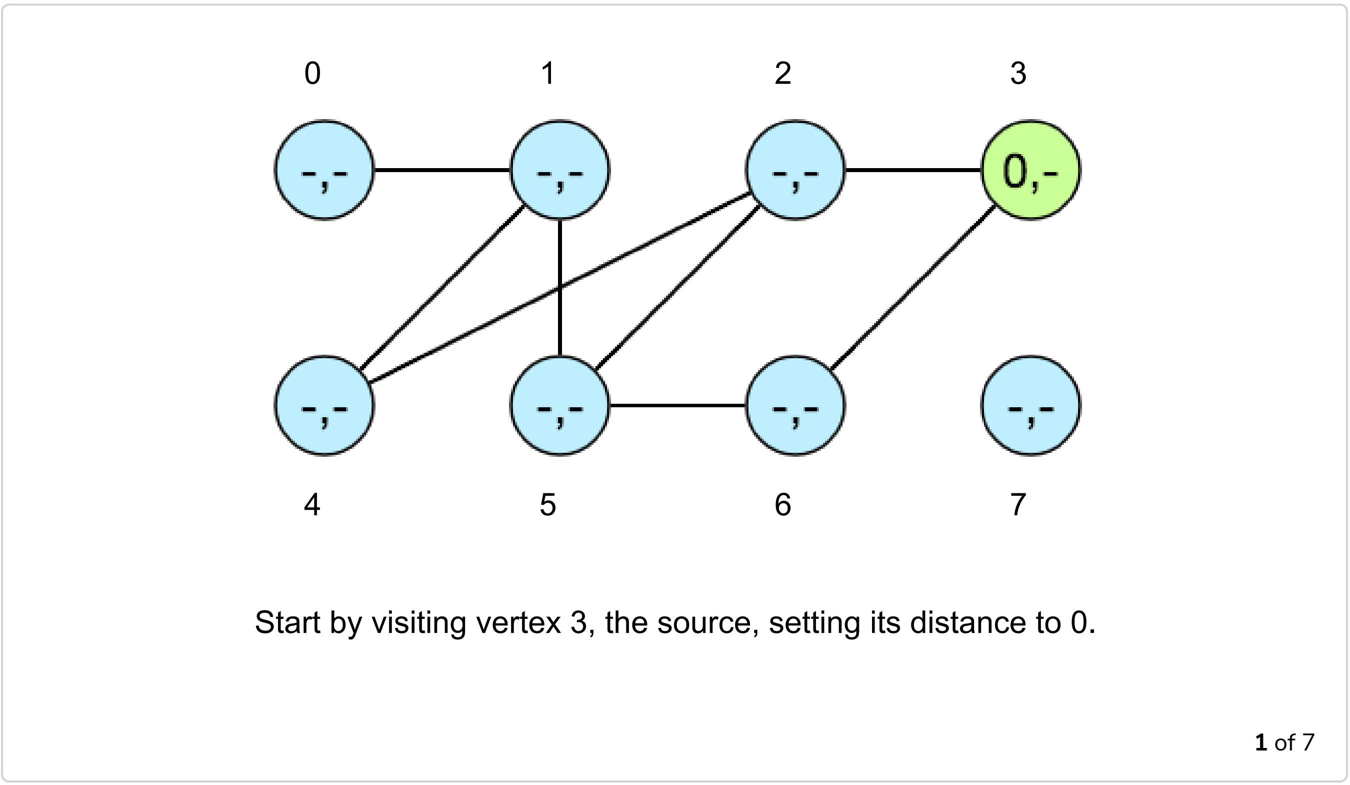
For example, here's an undirected graph with eight vertices, numbered 0 to 7, with vertex numbers appearing above or below the vertices. Inside each vertex are two numbers: its distance from the source, which is vertex 3, followed by its predecessor on a shortest path from vertex 3. A dash indicates *null*:



In BFS, we initially set the distance and predecessor of each vertex to the special value (*null*). We start the search at the source and assign it a distance of 0. Then we visit all the neighbors of the source and give each neighbor a distance of 1 and set its predecessor to be the source. Then we visit all the neighbors of the vertices whose distance is 1 and that have not been visited before, and we give each of these vertices a distance of 2 and set its predecessor to be vertex from which we visited it. We keep going until all vertices reachable from the source vertex have been visited, always visiting all vertices at distance **k** from the source before visiting any vertex at distance **k**+**1**.

Given the example above, here are the steps plus a visualization to play through each step:

- Start by visiting vertex 3, the source, setting its distance to 0.
- Then visit vertices 2 and 6, setting their distance to 1 and their predecessor to vertex 3.
- Start visiting from vertices at distance 1 from the source, beginning with vertex 2. From vertex 2, visit vertices 4 and 5, setting their distance to 2 and their predecessor to vertex 2. Don't visit vertex 3, because it has already been visited.
- From vertex 6, don't visit vertex 5, because it was just visited from vertex 2, and don't visit vertex 3, either.
- Now start visiting from vertices at distance 2 from the source. Start by visiting from vertex 4. Vertex 2 has already been visited. Visit vertex 1, setting its distance to 3 and its predecessor to vertex 4.
- From vertex 5, don't visit any of its neighbors, because they have all been visited.
- Now start visiting from vertices at distance 3 from the source. The only such vertex is vertex 1. Its neighbors, vertices 4 and 5, have already been visited. But vertex 0 has not. Visit vertex 0, setting its distance to 4 and its predecessor to vertex 1.
- Now start visiting from vertices at distance 4 from the source. That's just vertex 0, and its neighbor, vertex 1, has already been visited. We're done!



Start by visiting vertex 3, the source, setting its distance to 0.

Notice that because there is no path from vertex 3 to vertex 7, the search never visits vertex 7. Its distance and predecessor remain unchanged from their initial values of *null*.

A couple of questions come up. One is how to determine whether a vertex has been visited already. That's easy: a vertex's distance is *null* until it has been visited, at which time it gets a numeric value for its distance. Therefore, when we examine the neighbors of a vertex, we visit only the neighbors whose distance is currently *null*.

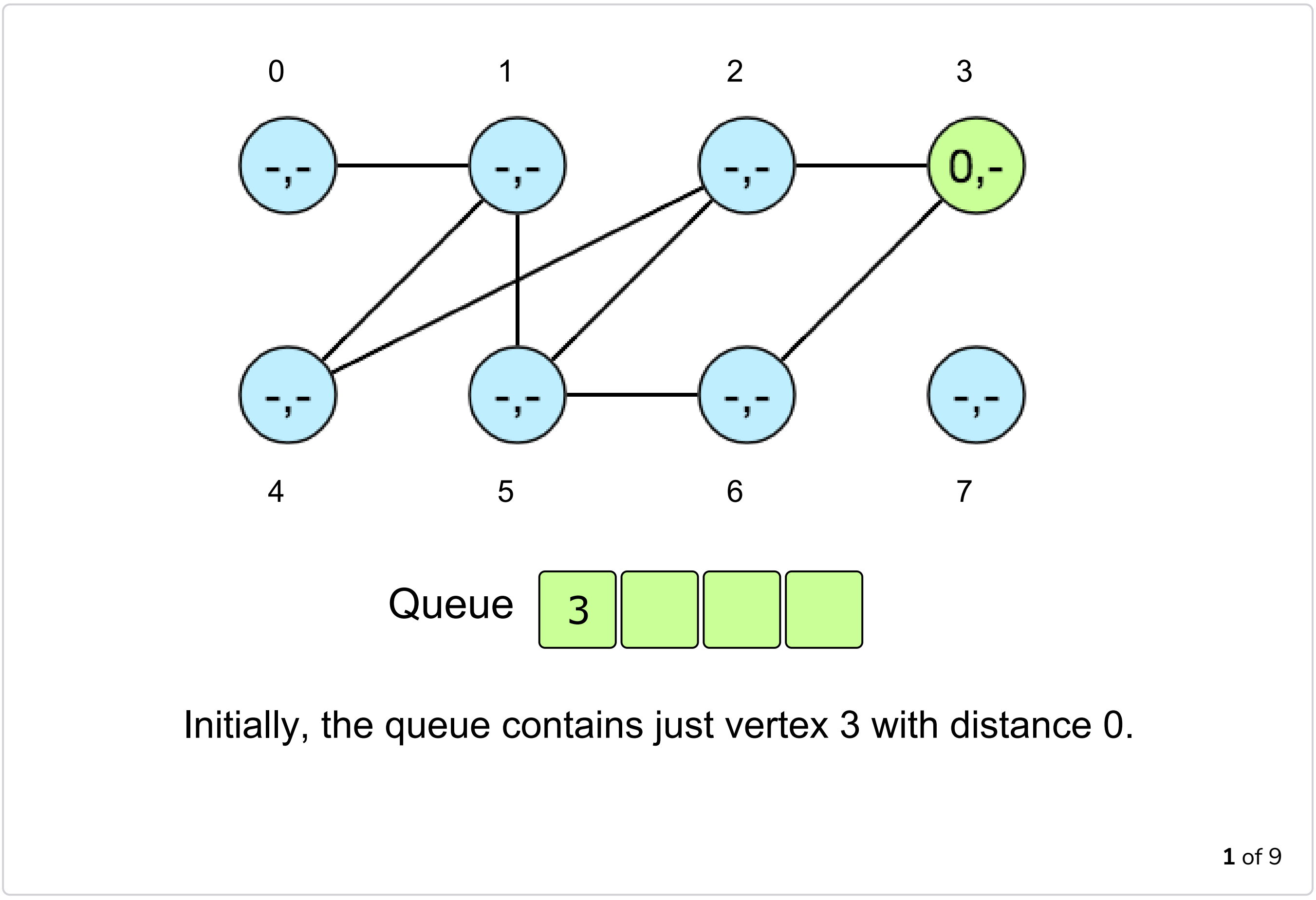
The other question is how to keep track of which vertices have already been visited but have not yet been visited from. We use a **queue**, which is a data structure that allows us to insert and remove items, where the item removed is always the one that has been in the queue the longest. We call this behavior **first in, first out**. A queue has three operations:

- ***enqueue(obj)*** inserts an object into the queue.
- ***dequeue()*** removes from the queue the object that has been in it the longest, returning this object.
- ***isEmpty()*** returns true if the queue currently contains no objects, and

false if the queue contains at least one object.

Whenever we first visit any vertex, we enqueue it. At the start, we enqueue the source vertex because that's always the first vertex we visit. To decide which vertex to visit next, we choose the vertex that has been in the queue the longest and remove it from the queue—in other words, we use the vertex that's returned from *dequeue**()***. Given our example graph, here's what the queue looks like for each step, plus the previous visualization shown with the queue state:

- Initially, the queue contains just vertex 3 with distance 0.
- Dequeue vertex 3, and enqueue vertices 2 and 6, both with distance 1. The queue now contains vertex 2 with distance 1 and vertex 6 with distance 1.
- Dequeue vertex 2, and enqueue vertices 4 and 5, both with distance 2. The queue now contains vertex 6 with distance 1, vertex 4 with distance 2, and vertex 5 with distance 2.
- Dequeue vertex 6, and don't enqueue any vertices. The queue now contains vertex 4 with distance 2 and vertex 5 with distance 2.
- Dequeue vertex 4, and enqueue vertex 1 with with distance 3. The queue now contains vertex 5 with distance 2 and vertex 1 with distance 3.
- Dequeue vertex 5, and don't enqueue any vertices. The queue now contains just vertex 1 with distance 3.
- Dequeue vertex 1, and enqueue vertex 0 with distance 4. The queue now contains just vertex 0 with distance 4.
- Dequeue vertex 0, and don't enqueue any vertices. The queue is now empty. Because the queue is empty, breadth-first search terminates.



Initially, the queue contains just vertex 3 with distance 0.

Notice that at each moment, the queue either contains vertices all with the same distance, or it contains vertices with distance **k** followed by vertices with distance **k+1**. That's how we ensure that we visit all vertices at distance **k** before visiting any vertices at distance **k+1**.