Problem: Implement breadth-first search on a graph.

Constraints

- **Is the graph directed?**
  - **Yes**
- **Can we assume we already have Graph and Node classes?**
  - **Yes**
- **Can we assume this is a connected graph?**
  - **Yes**
- **Can we assume the inputs are valid?**
  - **Yes**
- **Can we assume this fits memory?**
  - **Yes**

Test Cases

Input:

- **add_edge(source, destination, weight)**
- **graph.add_edge(0, 1, 5)**
- **graph.add_edge(0, 4, 3)**
- **graph.add_edge(0, 5, 2)**
- **graph.add_edge(1, 3, 5)**
- **graph.add_edge(1, 4, 4)**
- **graph.add_edge(2, 1, 6)**
- **graph.add_edge(3, 2, 7)**

graph.add_edge(3, 4, 8)

Result:

- **Order of nodes visited: [0, 1, 4, 5, 3, 2]**

Algorithm

We generally use breadth-first search to determine the shortest path.

- **Add the current node to the queue and mark it as visited**
- **While the queue is not empty**
  - **Dequeue a node and visit it**
  - **Iterate through each adjacent node**
    - **If the node has not been visited, add it to the queue and mark it as visited**

Complexity:

- **Time: O(V + E), where V = number of vertices and E = number of edges**
- **Space: O(V)**

Note on space complexity from Wikipedia:

- **When the number of vertices in the graph is known ahead of time, and additional data structures are used to determine which vertices have already been added to the queue, the space complexity can be expressed as O(V)**
- **If the graph is represented by an adjacency list it occupies O(V + E) space in memory**
- **If the graph is represented by an adjacency matrix representation, it occupies O(V^2)**