Introduction to Graphs

6 min

In this lesson, we'll take an object-oriented approach to build an implementation of the graph data structure in JavaScript. With three classes, Edge, Vertex, and Graph, we can implement a variety of

Preview: Docs Loading link description

graphs

that satisfy the requirements of many algorithms. Remember that a Graph consists of vertices and their corresponding edges.

For this lesson, we want our Graph class to be flexible enough to support directed, undirected, weighted, and unweighted graphs. We will provide you with an Edge class that connects two vertices, along with the weight of the connection (to support weighted graphs).

With this in mind, we will create our Graph with the following requirements:

- A Vertex can store any data.
- A Vertex maintains a list of connections to other vertices, represented by a list of Edge instances.
- A Vertex can add and remove edges going to another Vertex.
- A Graph stores all of its vertices, represented by a list of Vertex instances.
- A Graph knows if it is directed or undirected.
- A Graph knows if it is weighted or unweighted.
- A Graph can add and remove its own vertices.
- A Graph can add and remove edges between stored vertices.

Let's start with familiarizing ourselves with the classes that we will build in **Vertex.js** and **Graph.js**. We already set up .print() methods for you that will print out the state of the graph structure. Don't worry about the class in **Edge.js** yet. We will use it to connect the vertices in a later exercise.

To keep the concepts grounded in a real-world application, we'll build a transportation network of railroads and train stations as we go.

Instructions

1. Checkpoint 1 Passed

1.

Let’s start by setting up the constructor for our Vertex class. When a vertex is first created, it should hold any given data, and it should have an empty list of edges because it does not have any connections.

In the constructor, expect a data parameter and set it to the data class property. Then, set the edges class property to an empty array.

Hint

Remember, classes are created using the class expression, and constructor is a method that runs when an instance of the class is first created.

```
class MyClass {
  constructor(arg) {
    this.someValue = arg;
  }
}
```

To check that the constructor initializes the Vertex correctly, you can create an instance of a Vertex with any value (e.g. string, boolean, number, etc).

Print out the instance using the Vertex’s .print() function to see what it looks like. We should see a single vertex with the given value and no edge connections.

2. Checkpoint 2 Passed

2.

Moving on to the constructor for our Graph class, a graph is essentially a collection of vertices and edges. Our graph only needs to keep track of a list of vertices.

In the Graph class in **Graph.js**, create a constructor that takes no parameters. Since a graph doesn’t have any vertices when it is first created, set the vertices property to an empty array in the constructor.

Hint

Inside **Graph.js**, we should define the Graph‘s constructor to set the vertices property to an empty array, just like how we did for the Vertex class.

Vertex.js

```
const Edge = require('./Edge.js');


class Vertex {
  constructor (data) {
    this.data = data;
    this.edges = [];
  }


  print() {
    const edgeList = this.edges.map(edge =>
        edge.weight !== null ? `${edge.end.data} (${edge.weight})` : edge.end.data) || [];


    const output = `${this.data} --> ${edgeList.join(', ')}`;
    console.log(output);
  }
}


module.exports = Vertex;
```

Graph.js

```
const Edge = require('./Edge.js');
const Vertex = require('./Vertex.js');


class Graph {
  constructor() {
    this.vertices = [];
  }
  print() {
    const vertexList = this.vertices || [];
    vertexList.forEach(vertex => vertex.print());
  }
}

module.exports = Graph;
```

Edge.js

```
class Edge {
  constructor(start, end, weight = null) {
    this.start = start;
    this.end = end;
    this.weight = weight;
  }
}

module.exports = Edge;
```