CIS 399 Data Visualization – Spring 2020, April 28th, Week 5

Homework Assignment 4

In this assignment you will use JavaScript to create a tree data structure, and then render that tree to the browser with the D3 library.

As in Homework 2, add your name, your e-mail address, and your UO-ID to the HTML elements at the top. Also make sure your submission is a valid HTML5 file. Check that it is valid by uploading it to the W3C HTML Validator.

Your project structure looks like this:

```
CIS399_HW4/
  Homework4.pdf
  hw4.html
  js/
    Node.js
    Tree.js
    script.js
  data/
   Tree.json
```

All your work for this HW will be in the `js/Tree.js` file. All other necessary code has been provided for you.

Remember, to be able to access the data files with JavaScript, **you will need to be *serving* the CIS399_HW4 directory**, not just opening the HTML file in a browser. If your development environment doesn't already launch a server for you, you can start one with one of these commands:

```
$ cd path/to/CIS399_HW4
# for python 2
$ python -m SimpleHTTPServer 8080
# for python 3
```

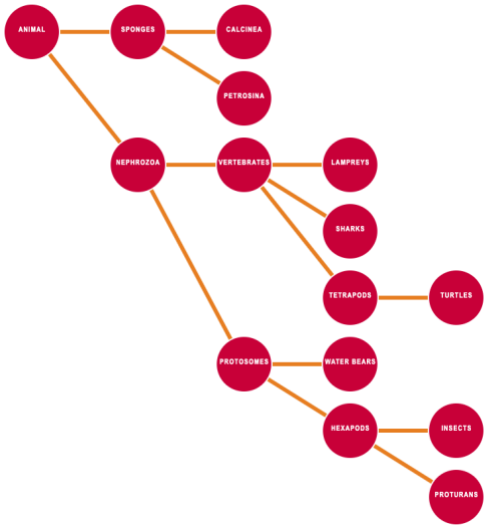
```
$ python -m http.server 8080
```

You can view the page at http://localhost:8080.

In this assignment we will be using JavaScript classes. The homework consists of two parts:

1. manipulating data with JavaScript, and
2. rendering elements to the browser using basic D3 functionality.

Your final tree should look similar to this one:



The Data and the Data Structure

The **`Node` class** has been provided in the `js/node.js` file and will be used as the **data structure of the nodes in the tree**.

The source data from which to construct the tree lives in the `data/Tree.json` file (see snippet below). This JSON file contains a list of objects, where each object has a name, and references the name of its parent.

```JSON
[{
   "name":"Animal",
    "parent":"root"
  },
  {
    "name":"Sponges",
    "parent":"Animal"
  },
  {
    "name":"Nephrozoa",
    "parent":"Animal"
  },...
```

Notice that **the constructor of the `Node` class takes in two parameters**: the name of the node as a string, and the name of the parent, also as a string.

```JS
constructor(nodeName, parentName) {

  /** String of Node Name */
  this.name = nodeName;

  /** String of Parent Name */
  this.parentName = parentName;

  /** Reference to parent Node Object. */
  this.parentNode = null;

  /** Array of Children. */
  this.children=[];

  /** Level of the node. */
  this.level=null;
```

```
  /**
    * Position of the node.
    * Initialize to -1
    */
  this.position=-1;
}

```

You will call this constructor from the `Tree` class for every node in the JSON to create an array of node objects. The constructor also initializes the remaining attributes of the `Node` class. Notice that there are both `level` and `position` attributes. The level refers to how **deep** in the tree a given node is (where the root is level 0), while the position refers to where a given node **is in relation to other nodes in the same level**. These two attributes will be important when rendering the tree to the browser and are discussed in more detail later in this spec.

The `Tree` class will have two main functions:

1. `buildTree()` - The **data wrangling** function, where we will create a tree data structure from the list of node objects
2. `renderTree()` - For **rendering** the tree to the screen.

Note that these functions will be called successively by the code you have been provided in `script.js`

```JS
// call fetchJSONFile then build and render a tree
// this is the function executed as a callback when parsing is done
fetchJSONFile('data/Tree.json', function(data) {
   let tree = new Tree(data);
   tree.buildTree();
   tree.renderTree();
});
```

The first part of the assignment will involve wrangling the data.

Part I: Data Wrangling

We have provided the skeleton of the tree class in `js/Tree.js`. Your job is to fill out the constructor and functions below.

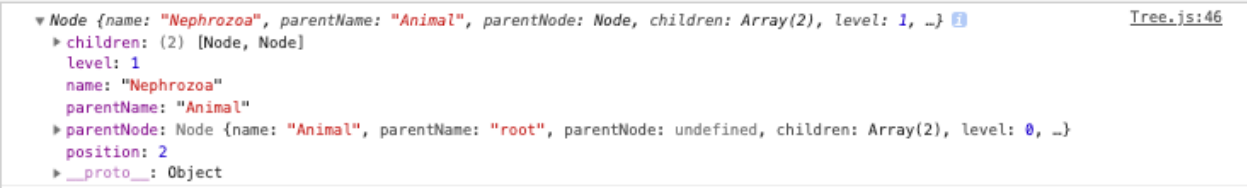
Our tree class will only need to have one attribute: a list (represented by an array) of objects of the class `Node`.

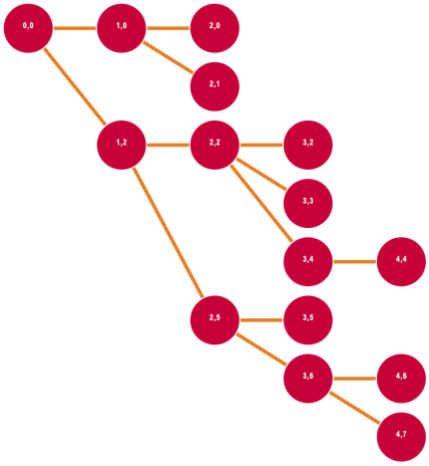
Let's start by filling out the constructor of the Tree class, which takes in an array of JSON objects.

```JS
/**
 * Creates a Tree Object
 * Populates a single attribute that contains a list (array) of Node objects to be used by the other functions in this class
 * note: Node objects will have a name, parentNode, parentName, children, level, and position
 * @param {json[]} json - array of json objects with name and parent fields
 */
constructor(json) {

}
```

A**dd code that will create this list of `Node` objects based on the input. This is a good place to populate the `parentNode` field of the `Node` objects as well.**
The next step is to complete the `buildTree()` function.

```JS
/**
 * Function that builds a tree from a list of nodes with parent refs
 */
buildTree() {

}
```

```

This function **should use the newly created list of `Node` objects and add a link to the node's children.** This function should **also call the `assignLevel()` and `assignPosition()` functions**, that we will implement in the next step.

**Your Node objects should look like this**:




Once all nodes have the correct parent and child references, we can proceed to the two recursive functions, `assignLevel` and `assignPosition`. Note that it is possible to do both, assigning levels and assigning positions in one sweep - feel free to do that if you prefer. Recursion is the easiest way to handle a hierarchical structure such as a tree.

The level of a node is how "deep" it is in the tree. The level of the root is 0, the children of the root are 1, the children of the children are 2, and so on. The position of a node is at least the position of its parent, and for every extra child of a parent the position increases. To be more specific, the position of a given node should be the lowest position held by any of it's children, and the nodes of a given level should never overlap. This lets us create a simple tree layout, without using any of D3's layout functions.
The figure below shows an example final tree for this homework, with labels on each node in the form **Level, Position**.
```



Both of these functions are best implemented recursively, so that you can recurse through the tree and assigning the level and position values as you go**. We recommend you implement the `assignLevel` function first since you may need the `level` attribute of a node in helping to determine its position.**

Part II: Rendering the Tree

Now that we have the data in the correct structure, we are ready to render this tree to the screen.

In order to do this, we will implement our last function, `renderTree()`. As the name implies, this function will render all the nodes in your instance of the `Tree` class to the screen. Note that we will not be using D3's build in tree layout functions – we've already calculated the necessary information for our layout in the first section.

The first step is to append an SVG element to the body of your page. We recommend sizing the SVG to something around 1200 x 1200 to give plenty of room for your nodes, edges, and labels.

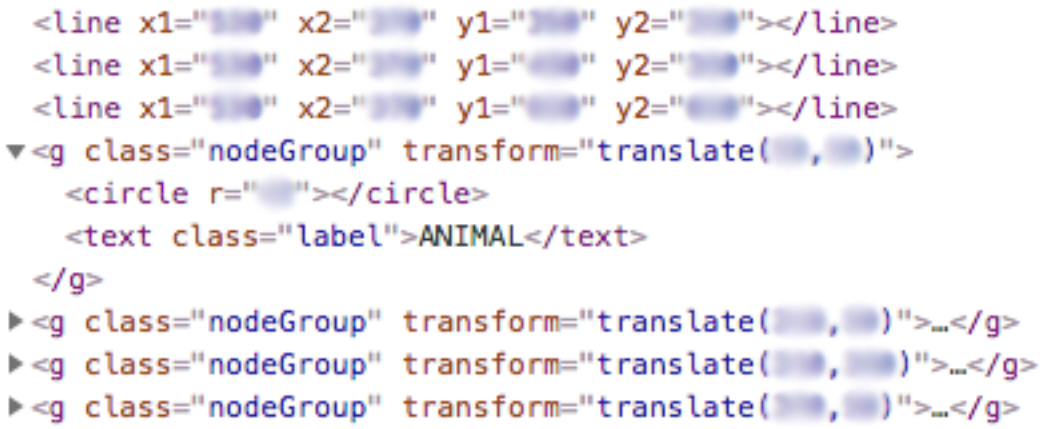
Once you have an SVG, you will use:

* `<circle>` elements to represent the nodes,
* `<line>` elements for the edges,
* `<text>` elements for the labels, and
* `<g>` elements as containers that move the nodes and labels together

You must use groups to group circles and text so that you can move them all at once.

Notice that we have provided the css rules in `hw4.html` for circle and line elements, as well as a class called `label` that you can apply to your text elements. We have also included css rule that changes the color of circles contained in a `nodeGroup` class object (i.e., an appropriately classed `<g>` element).

As illustrated in the figure below, your final `svg` element should contain both a set of `<line>` elements and a set of `<g>` elements. Each `<g>` element should contain both a `<circle>` and a `<text>` element, which have been translated together to their final location.



Scaling and Positioning

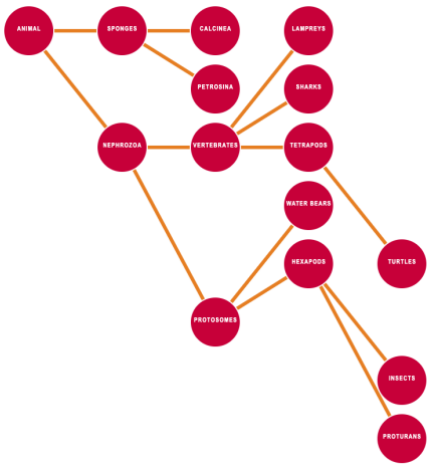
When positioning your SVG elements, use the `level` attribute of the node to calculate the x value of the circle and the `position` attribute to calculate the y value. Note that you need to scale the level and position values, in order to get reasonable values for the x and y attributes.

A few aesthetic things to keep in mind:

1. Make the Tree big enough so that it can be seen properly.
2. Render the edges connecting the nodes underneath the nodes.
3. Make sure there are no edge crossings.

4. Ensure all edges end at a position value greater or equal to that where they started (this is a result of the logic implemented in `assignPosition()`).
5. We don't assume the tree to be ordered, so any order of leafs is acceptable.

Here is an example of an *incorrect* tree, where the nodes of the fourth level are incorrectly positioned.



Have fun with your first D3 homework!

Rubric:
2 points: The constructor properly creates an array of `Node` objects.
2 points: The `buildTree` function properly populates the children.
2 points: The levels are correctly assigned.
2 points: The positions are correctly assigned.
2 points: The Tree is properly displayed in the browser, using groups and the appropriate D3 syntax in the renderTree function.
10 points total.

How and where to submit: Zip your files and submit on Canvas. Name your zip file FirstName_LastName.zip
Due date: May 11th, 2020 11:59pm

Questions + Concerns: Email me. I'd be happy to help.