```
graph.dependenciesOf('a'); // ['c', 'b']
graph.dependenciesOf('b'); // ['c']
graph.dependantsOf('c'); // ['a', 'b']

graph.overallOrder(); // ['c', 'b', 'a']
graph.overallOrder(true); // ['c']
graph.entryNodes(); // ['a']

graph.addNode('d', 'data');

graph.getNodeData('d'); // 'data'

graph.setNodeData('d', 'newData');

graph.getNodeData('d'); // 'newData'

var circularGraph = new
DepGraph({ circular: true });

circularGraph.addNode('a');
circularGraph.addNode('b');
circularGraph.addNode('c');
circularGraph.addNode('d');

circularGraph.addDependency('a', 'b');
circularGraph.addDependency('b',
'c'); // b depends on c
circularGraph.addDependency('c',
'a'); // c depends on a, which depends
on b
circularGraph.addDependency('d', 'a');

circularGraph.dependenciesOf('b'); //
['a', 'c']
circularGraph.overallOrder(); // ['c',
'b', 'a', 'd']
```

Dependency Graph

Simple dependency graph

Overview

This is a simple dependency graph useful for determining the order to do a list of things that depend on certain items being done before they are.

To use, `npm install dependency-graph` and then `require('dependency-graph').DepGraph`

API

DepGraph

Nodes in the graph are just simple strings with optional data associated with them.

- `addNode(name, data)` - add a node in the graph with optional data. If `data` is not given, `name` will be used as data
- `removeNode(name)` - remove a node from the graph

- `hasNode(name)` - check if a node exists in the graph
- `size()` - return the number of nodes in the graph
- `getNodeData(name)` - get the data associated with a node (will throw an `Error` if the node does not exist)
- `setNodeData(name, data)` - set the data for an existing node (will throw an `Error` if the node does not exist)
- `addDependency(from, to)` - add a dependency between two nodes (will throw an `Error` if one of the nodes does not exist)
- `removeDependency(from, to)` - remove a dependency between two nodes
- `clone()` - return a clone of the graph. Any data attached to the nodes will only be *shallow-copied*
- `dependenciesOf(name, leavesOnly)` - get an array containing the nodes that the specified node depends on (transitively). If `leavesOnly` is true, only nodes that do not depend on any other nodes will be returned in the array.
- `dependantsOf(name, leavesOnly)` (aliased as `dependentsOf`) - get an array containing the nodes that depend on the specified node (transitively). If `leavesOnly` is true, only nodes that do not have any dependants will be returned in the array.
- `directDependenciesOf(name)` - get an array containing the direct dependencies of the specified node
- `directDependantsOf(name)` (aliased as `directDependentsOf`) - get an array containing the nodes that directly depend on the specified node

- `overallOrder(leavesOnly)` - construct the overall processing order for the dependency graph. If `leavesOnly` is true, only nodes that do not depend on any other nodes will be returned.
- `entryNodes()` - array of nodes that have no dependants (i.e. nothing depends on them).

Dependency Cycles are detected when running `dependenciesOf`, `dependantsOf`, and `overallOrder` and if one is found, a `DepGraphCycleError` will be thrown that includes what the cycle was in the message as well as the `cyclePath` property: e.g. `Dependency Cycle Found: a -> b -> c -> a`. If you wish to silence this error, pass `circular: true` when instantiating `DepGraph` (more below).

Examples

```
var DepGraph = require('dependency-
graph').DepGraph;

var graph = new DepGraph();
graph.addNode('a');
graph.addNode('b');
graph.addNode('c');

graph.size() // 3

graph.addDependency('a', 'b');
graph.addDependency('b', 'c');
```