Programming assignment 4

PROJECT DESCRIPTION

For this assignment, you are to implement a graph that utilizes a hash table and then perform some analysis on that graph. The assignment can be broken into three chunks:

1) Create a HashTable
2) Graph building
3) Graph analysis

Hash Table

An interface–`HashTable.java`–is provided to you and contains methods for adding, retrieving, and removing. You are to create a class which implements this interface. Your hash table must support generics and, as part of this, use the default `hashCode()` Java method. You must use chaining to resolve collisions. For performance reasons, it is best that your table size be a prime number (a list of prime numbers is available here: http://primes.utm.edu/lists/small/1000.txt). You may assume, for this assignment that no more than 1000 items will be added to your hash table.

Graph building

An interface–`Node.java`–is given and defines how Node objects should behave. You must implement this interface in a class. A node has a unique name and unique integer id from 0 to $|V|$, that is, in the range of integers $[0, |V|)$. $|V|$ denotes the number of nodes in the graph.

An abstract class–`Graph.java`–is also provided. You are to create a class which extends this abstract class and implements all the abstract method stubs. The methods `addNode` and `addEdge` are used for graph construction. The methods `lookupNode(int)` and `lookupNode(String)` are used for node lookup based on their id and name, respectively. The lookup method based on the name must utilize a hash table, so its efficiency will be constant, on average. Finally, you must implement some other means of looking up a node based on its ID. This must have constant worst-case efficiency.

Analysis

The last part of the project is to perform some analysis on the graph. You are to determine, within the `isAcyclic()` method, if the graph has cycles or not (returns true if there are no cycles). Secondly, you are to perform a topological sort of the nodes in the graph. This is done in the `sort()` method, which returns an integer array containing the IDs of the sorted nodes. For the sort method, you may assume that it will only be called on an acyclic graph.

FACTORIES AND TESTING

In order to help us and you test your program, the last thing that you must do is implement the `create()` method in three factory classes. The `create()` method is simple, it will create an instance based on your implementation and return it. For example, if your graph class is named *SuperGraph* and its constructor doesn't take any arguments, then you would simple put:

```
return new SuperGraph();
```

Also, you are responsible for ensuring that your program works correctly by testing it. A simple testing script, ExampleTests, has been provided to give you an idea of how to go about testing your program. This is **not** meant to be a comprehensive test suite, but rather an aid to help you get started.

SCORING

- (15 points) The hash table works correctly (generics and put, get, and remove methods)
- (15 points) The hash table resolves collisions using chaining
- (5 points) Node IDs are auto-generated and unique
- (15 points) Graph is implemented using an adjacency list
- (10 points) Graph can look up a node by its ID with constant worst case efficiency
- (10 points) Graph can look up a node by its name with constant average case efficiency (via hash table)
- (15 points) Cycle detection
- (15 points) Topological sort

TESTING ASSUMPTIONS

Here are some answers to common questions that previous students have had about how your code will be tested.

1) We'll never try to get or remove items from the hash table that don't exist.
2) Only one graph will ever be created.
3) Within the `Graph` class, the `analyze` method may be called without calling the `sort` method.

If you have other questions, feel free to post to Piazza, email, or come to office hours.

PROVIDED CODE

The following are the files that are provided and short description about each.

ExampleTests.java
Some examples of how to test your program. Meant to help you get started.

Graph.java
Abstract class. Your graph implementation must extend this class.

GraphFactory.java
Factory used for creating graphs. Change the `create()` method to return your graph implementation.

HashTable.java
Interface. Your implementation must implement this interface.

HashTableFactory.java
Factory used for creating hash tables. Change the `create()` method to return your hash table implementation.

Node.java
Interface. Your implementation must implement this interface.

NodeFactory.java
Factory used for creating nodes. Change the `create()` method to return your node implementation.

If you are questions or are needing help concerning software engineering in Java–especially with implementing interfaces and extending abstract classes–please come to office hours.

Good luck!