Project 1: Java ADTs and the Adapter Pattern

COMP 180, Spring 2021

Test case (part 3) due: Tue, Feb 16 @ 11:59pm

Main project due: Mon, Feb 22 @ 11:59pm

Introduction

This project will introduce you to programming in Java by asking you to develop an abstract data type for graphs, and then adapt it for a slightly different interface.

A graph consists of a set of *nodes* (or *vertices*) N and a set of edges $E \subseteq N \times N$. For this project, graphs are *directed*, meaning there could be an edge from a to b without there being an edge from b to a.

We will define graphs as an abstract data type that implements the following interface:

```
public interface Graph {
    boolean addNode(String n);
    boolean addEdge(String n1, String n2);
    boolean hasNode(String n);
    boolean hasEdge(String n1, String n2);
    boolean removeNode(String n);
    boolean removeEdge(String n1, String n2);
    List<String> nodes();
    List<String> succ(String n);
    List<String> pred(String n);
    Graph union(Graph g);
    Graph subGraph(Set<String> nodes);
    boolean connected(String n1, String n2);
}
```

Here, nodes are labeled by strings, and if two strings are `equals` then they always refer to the same node. The methods do the following:

- `boolean addNode(String n)` adds the node `n` to the graph. It returns `true` if the node was not previously in the graph (i.e., it was added by the call), and `false` if the node was already present.
- `boolean addEdge(String n1, String n2)` adds an edge from the node `n1` to the node `n2`. It returns `true` if the edge was not previously in the graph, and `false` otherwise. This method should throw `NoSuchElementException` if `n1` or `n2` were not previously added as nodes.
- `boolean hasNode(String n)` returns `true` if the node `n` was added to the graph previously (and not removed), and `false` if not.
- `boolean hasEdge(String n1, String n2)` returns `true` if the edge from `n1` to `n2` was added to the graph previously (and not removed), and `false` if not.
- `boolean removeNode(String n)` removes node `n` from the graph and all edges to or from `n`. It returns `true` if `n` was previously in the graph and `false` otherwise.
- `boolean removeEdge(String n1, String n2)` removes the edge from `n1` to `n2` from the graph, returning `true` if the edge was previously in the graph and `false` otherwise. This method should throw `NoSuchElementException` if `n1` or `n2` were not previously added as nodes.
- `List<String> nodes()` return a list containing all the nodes in the current graph, in some unspecified order.
- `List <String> succ(String n)` returns a list of all nodes `n2` such that there is an edge from `n` to `n2` in the graph, i.e., it returns the *successors* of `n`. This method type uses the `List` interface. This is in a *generic* type, meaning you can have lists of strings, list of integers, lists of lists of strings, etc. We'll go into detail about how this works later, but for now, all you need to know is that `List<String>` is a list of strings, and in the documentation, anywhere you see the *type parameter* `E` you can mentally substitute `String`. This method should throw `NoSuchElementException` if `n` was not previously added as a node.
- `List <String> pred(String n)` returns a list (a `List<String>`) of all nodes `n2` such that there is an edge from `n2` to `n` in the graph, i.e., it returns the *predecessors* of `n`. This method should throw `NoSuchElementException` if `n` was not previously added as a node.
- `Graph union(Graph g)` returns a new graph that includes all the nodes and edges of the current graph and all the nodes and edges of `g`. Nodes identified by the same string in both graphs are coalesced to be the same node in the returned graph. *Note:*

You may **not** assume that `g` is implemented by an `ListGraph`, i.e., code that casts `g` to an `ListGraph` will receive no credit. This requirement means this method will be extremely annoying to write, which sometimes happens when interfaces and APIs are not ideal.

- `Graph subGraph(Set<String> nodes)` returns a new graph containing the nodes of the current graph that are included in `nodes` and the current graph's edges among them. For example, if the current graph has nodes `[A,B,C,D]` and edges `A→B`, `A→C`, `C→D` and the `nodes` argument to `subGraph` is `[A, B, C, E]`, then the resulting graph would contain edges `A→B` and `A→C`, and nodes `A`, `B`, and `C`.
- `boolean connected(String n1, String n2)` returns `true` if and only if there is a *path* from `n1` to `n2`, meaning there is a sequence of edges from `n1` to some `na` to some `nb` etc to `n2`. If `n1.equals(n2)`, this method should return true, i.e., a path of length 0 counts. To implement this method, you'll probably want to use either breadth-first search or depth-first search (either will work). For this method to work correctly in the presence of cycles in the graph (i.e., the case when one node is `connected` to itself), you might want to use a `HashSet<String>`. This method should throw `NoSuchElementException` if `n1` or `n2` were not previously added as nodes.

Here is the code skeleton for the project.

Part 1: Graphs with Adjacency Lists

A key algorithmic design choice in implementing graphs is how to represent edges in the graph. For the first part of the project, you will implement the `Graph` interface using *adjacency lists*. More specifically, write an implementation of `Graph` in the file `ListGraph.java` in which the nodes and edges of the graph are represented using the following field:

```
  private HashMap<String, LinkedList<String>> nodes;
```

Here, the graph is represented as a mapping from nodes to lists of their successors in the graph. The map itself is a hash table (a `HashMap`), and the lists are `LinkedList`s, which are just like linked lists in C.

For example, here is some code that creates a few nodes and edges in the graph and does some tests to see what the graph contains.

```
nodes.put("a", new LinkedList<String>());  // add node "a" to the graph
nodes.put("b", new LinkedList<String>());  // add node "b"
nodes.put("c", new LinkedList<String>());  // add node "c"
nodes.containsKey("a"); // returns true, "a" is a graph node
nodes.containsKey("d"); // returns false, "d" is not a graph node
nodes.get("a").add("b"); // add edge from "a" to "b"
nodes.get("c").add("a"); // add edge from "c" to "a"
nodes.containsKey("c") &&
  nodes.get("c").contains("a") &&
  nodes.containsKey("a") &&
  nodes.get("a").contains("b")  // returns true, there's a path from "c" to "b"
```

Hint: If you want to iterate through a `LinkedList` in Java, you can use a `while` loop, or you can use the following syntactic sugar; we'll explain why this works later.

```
for (String n : nodes.get("a")) { // iterate through elements of nodes.get("a")
  // code that uses n
}
```

If you want to iterate over a `HashMap`, you can do the following:

```
for (String n : nodes.keySet()) { // iterate over key of HashMap
  LinkedList<String> s = nodes.get(n); // get corresponding successor lists
}
```

It is also possible to avoid the call to `get` by iterating over the `entrySet` of the `HashMap`. If you're interested, you can find out how to use that approach by searching online.

Part 2: A Different Graph API

The `Graph` interface we gave you above is just one possible interface for graphs. For example, here is a class that implements an immutable representation of graph edges:

```
public class Edge {
    private String src, dst; // source, destination
    Edge(String src, String dst) {
        this.src = src; this.dst = dst;
    }
    String getSrc() { return src; }
    String getDst() { return dst; }
}
```

We can then use this class to define a different interface for graphs:

```
public interface EdgeGraph {
    boolean addEdge(Edge e);
    boolean hasNode(String n);
    boolean hasEdge(Edge e);
    boolean removeEdge(Edge e);
    List<Edge> outEdges(String n);
    List<Edge> inEdges(String n);
    List<Edge> edges();
    EdgeGraph union(EdgeGraph g);
    boolean hasPath(List<Edge> l);
}
```

with the following methods:

- `boolean addEdge(Edge e)` adds an edge to the graph, returning `true` if the edge was not already in the graph or `false` if not. Note that, in this API, nodes are not added separately from edges. Node `n` is automatically added to the graph if an edge containing `n` is added.
- `boolean hasNode(String n)` returns true if and only if some edge has been added to the graph (and not removed) with `n` as either the source or destination fo the edge.
- `boolean hasEdge(Edge e)` returns true if edge `e` has been added to the graph (and not removed).
- `boolean removeEdge(Edge e)` removed edge `e` from the graph, returning `true` if it was previously in the graph and `false` otherwise. If this method removes the last edge to or from a given node in the graph, that node should also be removed. Note: This method does not throw an exception even if one or the other end of the `Edge` is not in the graph.
- `List<Edge> outEdges(String n)` returns a list of all edges that start at node `n`.
- `List<Edge> inEdges(String n)` returns a list of all edges that end at node `n`.
- `List<Edge> edges()` returns a list of all the edges in this graph, in some unspecified order.
- `EdgeGraph union(EdgeGraph g)` returns a new graph that includes all the nodes and edges of the current graph and all the nodes and edges of `g`. Nodes identified by the same string in both graphs are coalesced to be the same node in the returned graph. *Note:* You may **not** assume that `g` is implemented by an `EdgeGraphAdapter`, i.e., code that casts `g` to an `EdgeGraphAdapter` will receive no credit.
- `boolean hasPath(List<Edge> l)` returns true if the path `l` (a `List<Edge>`) is in the graph, meaning if `l = e1, e2, ..., en` then all edges `ei` are in the graph. The method should also check if the argument is a path, i.e., if `ei.getDst() == e(i+1).getSrc()` for every edge in the middle of the list. If this is not the case, this code should raise the exception `BadPath`, which is defined in file `BadPath.java`. Note that every graph includes the empty path (since if a graph contains a path, it should contain every sub-path).

Your task for the second part of the project is to implement an `EdgeGraph`. But, like any good software engineer, you don't want to start from scratch. You already have perfectly good `Graph` implementation. So, for this part of the project, you will write an *adapter* that, given a `Graph`, will adapt it to be an `EdgeGraph`. More specifically, implement `EdgeGraphAdapter`, which looks like the following:

```
public class EdgeGraphAdapter implements EdgeGraph {

    private Graph g;

    EdgeGraphAdapter(Graph g) { this.g = g; }

    // methods of EdgeGraph
}
```

So, to implement the methods of `EdgeGraphAdapter`, you will write code that *delegates* graph operations to `g`. You'll notice that for some methods of `EdgeGraph`, you can call corresponding methods of `g` with no change. With other methods, you'll need to change the arguments a bit. And with still other methods, you'll need to implement new functionality that's not part of `Graph`.

Notice also that your code will work with *any* implementation of `Graph`, not just `ListGraph`. Cool!

Part 3: Write and Share a Test Case

To help you test your code, we've created a simple method `main` in class `Main` so you can run some test cases. The body of `main` looks like this:

```
    public static void main(String[] args) {
        test1();
        test2();
    }
```

It just calls a couple of simple tests we wrote. Though it won't count in the grading of your project, you should right a bunch more tests (`test3`, `test4`, etc., or call them whatever you want since we won't evaluate these methods) and add calls to them in `main`.

Here is the definition of `test1`:

```
    public static void test1() {
        Graph g = new ListGraph();
        assert g.addNode("a");
        assert g.hasNode("a");
    }
```

There are a bunch of ways to write and design tests, which we'll talk about later in the semester, but this method just creates a new graph, adds a node to it, and checks that the node is in the graph. To check that methods are returning the correct values, it uses Java's `assert` statement, which takes a boolean argument and raises an exception if the argument doesn't evaluate to `true`.

Important: To run the code with assertions enabled, you need to invoke `java -ea Main`, i.e., you need to pass the `-ea` (or `-enableassertions`) argument to the JVM. Otherwise, by default, the JVM will ignore the assertion statements completely and not run them.

Although you can't generally share code for this class, we will make one exception for this project: You **must** write one test case, in the style of `test1` above, and post it publicly to campuswire to share with the class. **Your test case must be for Part 1 only.** You may not share test cases for Part 2. Your test case can test any functionality of Part 1, as much or as little as you like. It need not be substantively different than others' test cases, but you must come up with it on your own. In fact, you should come up with a test case (or many test cases) right now, before you've even started implementing the project. This is called Test-driven development, which is a popular software engineering approach.

What to turn in

Put all your code in the code skeleton we have supplied, and upload your solution using Gradescope.