Graphs and Hypergraphs

Graph definitions

- A directed graph consists of zero or more nodes and zero or more edges
- An edge connects an origin node to a destination node
  - The origin and destination nodes need not be distinct, but they must be on the same graph
- An undirected graph can be represented as a directed graph in which all edges come in pairs

APIs for ADTs

- Requirements of an API:
  - The *constructors* and *transformers* must together be able to create all legal values of the ADT
    - Or, at least, any needed by the applications that use it
    - For a general use ADT, this means *all* legal values
  - The *accessors* must be able to extract any data
    - Or, at least, any needed by the applications that use it
- Desirable properties of an API:
  - The API should be simple
  - Convenience methods should be provided ***if:***
    - They are likely to be used often
    - The user would expect them to be present (e.g. `toString()`)
    - They simplify the use of the API more than they add complexity to it
    - They provide serious gains in efficiency

My graph API

- In my design:
  - There are three classes: `Graph`, `Node`, and `Edge`
  - The `Graph` class has a `print()` method
- My goals for my design were *completeness* and *simplicity* (in that order)
  - I am not claiming that my design is the best possible, but I think it's very good

Graph methods

- Constructor
  - public **Graph**(Object value)
- Mutative transformers
  - public void **add**(Node n)
  - public void **delete**(Node node)
  - public void **delete**(Edge edge)
- Accessors
  - public Set<Node> **nodes**()
  - @Override public String **toString**()
  - public void **print**()
  - public void **dump**() // debugging aid

Node methods

- Constructor
  - public **Node**(Object value,
    Graph g)
- Mutative transformer
  - public void **delete**()
- Accessors
  - public Graph **getGraph**()
  - public Set<Edge> **getOutpointingEdges**()
  - public Set **getInpointingEdges**()
  - @Override public String **toString**()

Edge methods

- Constructor
  - public **Edge**(Node fromNode, Object value, Node toNode, Graph g)
- Mutative transformer
  - public void **delete**()
- Accessors
  - public Graph **getGraph**()
  - public Node **getOrigin**()
  - public Node **getDestination**()
  - @Override public String **toString**()

Where is...?

- Where's the data? (Node names, edge labels, etc.)
  - My classes all extend Plex, which has public Object value
  - You could have getValue and setValue methods
- Where are the equals(Object obj) methods?
  - I claim node equality means: ***Same node, same graph.*** In other words, ==
  - Similarly for edges
  - Equality for graphs *could* mean graph isomorphism: There exist $Node_{G1} \rightarrow Node_{G2}$ and $Edge_{G1} \rightarrow Edge_{G2}$ mappings that make the graphs have identical structure
    - This is an intractable (exponential) problem, and I don't deal with it
- Where are the hashCode() methods?
  - The inherited hashCode methods are consistent with equals meaning ==

Why do I have...?

- A deleteEdge(Edge edge) method in Graph, when I already have a delete() method in Edge?
  - It's just a convenience method
  - Since I have deleteNode (which is necessary), it's reasonable for the user to expect a corresponding deleteEdge method
- A getInpointingEdges() in Node?
  - Most programs only use outpointing edges
  - If the method *is* needed, it's easy for me to provide, *much much* more complex for the user
- All those toString() methods?
  - I think toString() is always a good idea—for debugging, if nothing else

Hypergraphs

- A hypergraph is a collection of zero or more graphs, with many fewer restrictions.
- There is no generally accepted definition of a hypergraph, but here are some of the things that might be allowed
  - Nodes may reside simultaneously on many graphs, or on none at all
  - Edges may originate from multiple nodes, or none at all
  - Edges may terminate at (point to) multiple nodes, or none at all
  - The origin and destination nodes of an edge need not be on the same graph
  - Edges may originate from and/or point to graphs or other edges
  - Graphs may contain other graphs as nodes. Nodes may contain graphs Even edges may contain graphs
- Obviously, a hypergraph is a much more complex structure than a simple directed graph
  - With the right approach, hypergraphs are actually much simpler than "ordinary" graphs

Plex

- A plex consists of four sets:
  - **containers**: The other plexes in which this plex occurs
    - For example, nodes and arcs may occur in a graph
  - **contents**: The other plexes contained in this plex
    - For example, a graph may contain nodes and arcs
  - **origins**: The other plexes "from which" this plex comes
    - For example, an edge comes from a node
  - **destinations**: The other plexes "to which" this plex goes
    - For example, an edge goes to a node
- There are two simple validity rules:
  - If plex X is a container of plex Y, then plex Y is a content of plex X, and vice versa
  - If plex X is a destination of plex Y, then plex Y is an origin of plex X, and vice versa
  - This redundancy is for reasons of efficiency

Plex data and constructors

```
public class Plex {
    Set<Plex> containers   = new HashSet<Plex>();
    Set<Plex> contents     = new HashSet<Plex>();
    Set<Plex> origins      = new HashSet<Plex>();
    Set<Plex> destinations = new HashSet<Plex>();
    public Object value;

    protected Plex() { }

    protected Plex(Object value) {
        this.value = value;
    }
```

Plex methods

```
void addContainer(Plex that) {
   this.containers.add(that);
   that.contents.add(this);
}

void removeContainer(Plex that) {
   this.containers.remove(that);
   that.contents.remove(this);
}
```

- Similarly for addContent, removeContent, addOrigin, removeOrigin, addDestination, and removeDestination

Implementing hypergraphs with plexes

- A plex can represent a graph
  - Its contents can hold the nodes on this graph
- A plex can represent a node
  - Its containers can hold the graph(s) on which it occurs
  - Its origins can hold its inpointing edges
  - Its destinations can hold its outpointing edges
- A plex can represent an edge
  - Its containers can hold the graph(s) on which it occurs
  - Its origins can hold its origin node(s)
  - Its destinations can hold its destination node(s)
- Aside from what we call things, once we have implemented plexes, we have implemented hypergraphs!

Implementing graphs with plexes

- We can model graphs, nodes, and edges by putting restrictions on their plexes
- Graph:
  - The origins, destinations, and containers sets are empty
  - The nodes of the graph are in the contents set
- Node:
  - The contents set is empty
  - The containers set contains a single element, the graph that the node resides on
  - The origins set contains the edges that point to the node
  - The destinations set contains the edges that come out of the node.
- Edge:
  - The contents set is empty
  - The containers set contains only the graph that the edge resides on
  - The origins set contains only the one node from which the edge originates
  - The destinations set contains only the one node that the edge points to

The Graph class

- **public class Graph extends Plex {**
  // containers : the graph containing
  // this node
  // contents : empty
  // origins : inpointing edges
  // destinations : outpointing edges

- **public Graph(Object value) {**
  this.value = value;
  }

- **public Set<Node> nodes() {**
  HashSet<Node> nodes =
  new HashSet<Node>();
  for (Plex plex : contents) {
  nodes.add((Node)plex);
  }
  return nodes;
  }

- **public void add(Node n) {**
  this.addContent(n);
  }

  public void delete(Node node) {
  removeContent(node);
  }

- **public void delete(Edge edge) {**
  edge.delete();
  }

- @Override
  public String toString() {
  return "Graph " + value;
  }

The Node class

```
public class Node extends Plex {
//    containers   : the graph containing
//                   this node
//    contents     : empty
//    origins      : inpointing edges
//    destinations : outpointing edges
public Node(Object value, Graph g) {
    this.value = value;
    g.add(this);
}
public Graph getGraph() {
    return
          (Graph)Extractor.getOne(containers);
}
public Set<Edge> getOutpointingEdges() {
    HashSet<Edge> edges =
                        new HashSet<Edge>();
    for (Plex plex : destinations) {
        edges.add((Edge)plex);
     }
    return edges;
 }
```

```
public Set getInpointingEdges() {
    HashSet<Edge> edges =
                        new HashSet<Edge>();
    for (Plex plex : origins) {
        edges.add((Edge)plex);
    }
    return edges;
}
public void delete() {
    Graph g = (Graph) Extractor.
                    getOne(this.containers);
    for (Plex edge : origins) {
        ((Edge)edge).delete();
    }
    for (Plex edge : destinations) {
        ((Edge)edge).delete();
    }
    g.delete(this);
}
@Override
public String toString() {
    return "Node " + value;
}
```

The Edge class

```
public class Edge extends Plex {
// containers   : empty
// contents     : empty
// origins      : node this edge comes from
// destinations : node this edge points to
```

```
public Edge(Node fromNode, Object value,
            Node toNode, Graph g) {
    this.value = value;
    this.addOrigin(fromNode);
    this.addDestination(toNode);
}
```

```
public void delete() {
    Plex fromNode(Node)Extractor.
                        getOne(origins);
    removeOrigin(fromNode);
    Plex toNode = (Node)Extractor.
                        getOne(destinations);
    removeDestination(toNode);
}
```

```
public Node getOrigin() {
    return (Node)Extractor.getOne(origins);
}
```

```
public Node getDestination() {
    return
      (Node)Extractor.getOne(destinations);
}
```

```
public Graph getGraph() {
    Node node = getOrigin();
    return node.getGraph();
}
```

```
@Override
public String toString() {
    ArrayList<Object> nodes =
                new ArrayList<Object>();
    for (Plex node : destinations) {
        nodes.add(((Node)node).value);
    }
    return "Edge " + value + " to Node "
            + nodes.get(0);
}
```

A slightly amusing helper method

- Suppose you have a Set containing at most one element--how do you get that element?
- ```
  class Extractor {
      protected static Plex getOne(Set<Plex> set) {
          for (Plex plex : set) {
              return plex;
          }
          return null;
      }
  }
  ```
- Why should we put this in a class of its own?
  ```