

Intro Software Engineering Problem Set 3

- Exercise 6.2. Refer to Figures 5.6 and 6.5. You do not need to be able to compile your code — in particular, you do not have to reproduce the rest of `IntSet`.
- (This is based on Exercise 6.6.)
Implement the following generic iterator method:

```
/** Apply a filter to an Iterator object.
 * @return an Iterator object that produces, in order, each
 * exactly once, all elements e produced by g for which
 * x.checker(e) is true.
 * @throws NullPointerException if either g or x is null.
 */
static <E> Iterator<E> filter(Iterator<E> g, Check<E> x)
```

Here `Check` is defined as follows:

```
interface Check<E> { public boolean checker(E x); }
```

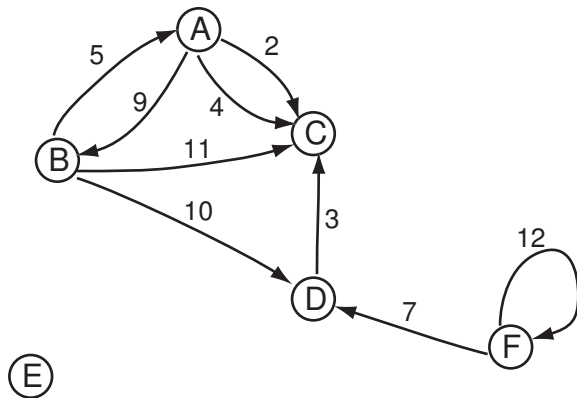
Implement the `filter` method as a static method of a public class that can be compiled for testing. Include a file `Check.java` containing the definition of `Check` just given.

- Do exercises 7.8, 7.9 and 7.10. Then, if you answered “no” to either 7.9 or 7.10, describe how you could change the specification of `Counter` to make the subtype legitimate.

Midterm Project Part 1: Graphs and a Graph Viewer

A *graph* consists of a set V of *vertices* (also called *nodes*) and a set E of *edges*. Each edge $e \in E$ has two endpoints, which are members of V . For our purposes, graphs are *directed*, which means that one of an edge’s endpoints is its *source* and the other is its *target*. The graphs you will work with in this project are *weighted* graphs, in which every edge has a number (for our purposes, an integer) that is its *weight*.

A graph is usually represented pictorially by a diagram in which each vertex is a circle and each edge is an arrow from its source to its target. For example, the graph below has six vertices and nine edges:



A Graph Abstraction

Download the file `Graphs.tgz` and create an Eclipse project with its contents. The file `Graph.java` contains the generic interface `Graph<V>`, which corresponds to a (directed, integer-weighted) graph whose vertices are objects of type `V`. The file `Edge.java` defines a class `Edge<V>` whose objects represent integer-weighted directed edges between vertices of type `V`; the `edgesFrom` method in `Graph<V>` returns a collection of `Edge<V>` objects.

Provide a class that implements `Graph<V>`. The most convenient way to represent graphs is to use so-called *adjacency lists*: for each vertex in the graph, store a collection (preferably some kind of `java.util.Collection`) of the edges originating at that vertex. In particular, this makes the `edgesFrom` method fairly easy to implement. You should consider carefully the operations your graph implementation must support and try to design a representation that can support them efficiently. Discuss the efficiency of your implementation.

PointNode Graphs

In the next part of the project, you will write a program that must draw graphs in a window. The problem of taking an arbitrary graph and deciding where to draw its vertices to produce an aesthetically pleasing result is very hard. This project avoids the issue by working with graphs whose vertices have an inherent notion of location.

The file `PointNode.java` defines the class `PointNode`. A `PointNode` has a name, which is a string, and a pair of integer coordinates specifying where it is to be drawn in a window. `PointNode` objects are immutable. You may also notice that the equality, comparison and hashing methods of a `PointNode` simply delegate their tasks to its name. For the rest of this project, you will be working with graphs with `PointNode` vertices, that is, graphs of type `Graph<PointNode>`.

Reading Graphs from Files

The file `GraphParsing.java` contains a specification for a standalone procedure to read a textual representation of a `Graph<PointNode>` from a file. Complete the implementation of

this method.

The graphs to be read by `parsePointNodeGraph` have as their vertices `PointNode` objects whose names are nonempty strings that contain no whitespace. The file format for graphs is as follows. The file begins with a list of the vertices, **one per line**, in the following format:

```
name1 x1 y1
name2 x2 y2
etc.
```

That is, each line starts with the (whitespace-free) name of the vertex, followed by its x -coordinate, followed by its y -coordinate. The coordinates are integers; the tokens on each line are separated by whitespace.

The first blank line in the file signals the end of the list of vertices. Each subsequent line consists of the name of a vertex, followed by the targets and weights of zero or more edges beginning at that vertex; the items in a line are separated by whitespace. The end of the graph information is marked either by the end of the file or by the second blank line in the file. The `parsePointNodeGraph` procedure should not attempt to read past whichever of these two conditions is encountered first.

For example, the following text:

```
A 20 30
B 100 80
C 70 90
```

```
A B 1
A C 2
C B 3
```

corresponds to a graph that looks like this:

