Algorithms & Data Structures (M)

Assessed Exercise 2 (2013–14)

This exercise is about design, implementation, and testing of an abstract data type (ADT).

Its weighting is *0.2* in the ADS(M) course assessment.

The deadline is *Friday 21 March 2014* at 16:30.

Design

A *binary relation* is a collection of pairs (*x*, *y*). The pairs are in no fixed order. Individual *x* and *y* values may be duplicated, but no pair (*x*, *y*) may be duplicated.

Example:

The following binary relation relates countries to their official languages:

FR	French
DE	German
IT	Italy
BE	French
BE	Flemish
NL	Dutch
UK	English
IE	English
IE	Irish

Note that some countries are duplicated, and some languages are duplicated, but no (country, language) pair is duplicated.

Design an abstract data type, `Relation`, such that each `Relation` value represents a binary relation. `Relation` must be equipped with operations that enable application code to:

1. test whether the relation contains a given pair (*x*, *y*)
2. given *x*, determine all values *y* such that the relation contains (*x*, *y*)
3. given *y*, determine all values *x* such that the relation contains (*x*, *y*)
4. make the relation empty
5. add a given pair (*x*, *y*) to the relation
6. remove a given pair (*x*, *y*) from the relation
7. given *x*, remove all pairs (*x*, *y*) from the relation
8. given *y*, remove all pairs (*x*, *y*) from the relation
9. render the relation's contents as a string, in a suitable format.

Express your design in the form of a Java interface, `Relation`. Each operation must be accompanied by a comment specifying its observable behaviour.

You can choose to make your interface support *either* heterogeneous *or* homogeneous relations:

- In a heterogeneous relation, the *x* values and the *y* values may be arbitrary objects.
- In a homogeneous relation, all the *x* values must be of type *X*, and all the *y* values must be of type *Y*. Here the interface has two type parameters, *X* and *Y*.

Implementation

Now write a Java class that implements your `Relation` interface:

- If your interface supports heterogeneous relations, your class must also do so.
- If your interface supports homogeneous relations, your class must also do so.

You will need to choose a suitable data structure, and algorithms to implement all the operations of the interface.

Your class must contain implementation notes as comments. The implementation notes must (a) briefly describe your data structure; and (b) for each operation, briefly describe your algorithm (if it is a standard algorithm, just name it) and state its time complexity.

Important note: In this exercise, you must represent a relation using your own data structure (such as an array, linked-list, binary-search-tree, or hash-table). *Do not* use any of the Java collection classes to represent a relation.

Testing

Test your class thoroughly. Your `main` method must construct a relation, then call all the operations that meet requirements 1–8 above. At each call to an operation, your `main` method must output the name of the operation, its arguments, and its results. After each operation that *updates* the relation, your `main` method must output the relation's contents.

Submission

By the deadline stated above, you must submit your deliverables through the ADS(M) Moodle page. (Click "Assessed exercise 2 submission".)

The deliverables are all your Java source files, class files, input data files (if any), and output(s). The output(s) are essential to show that you have tested your program thoroughly. Submit all the deliverables as a single `.zip` file.

Important: You must also sign the School's "declaration of originality" form.

Assessment

Your program will be marked against the following criteria:

Design	9 marks
Implementation correctness	9 marks
Implementation efficiency	6 marks
Testing	6 marks

Your mark for implementation efficiency will depend on the time complexities of your `Relation` operations. In particular, the operations that meet requirements 1, 5, and 6 should be better than $O(n)$ to earn a high mark for implementation efficiency.

Your total mark will be converted to a grade on the University's 22-point scale. Your mark will be reduced if your code is clumsy or hard to read, or if you do not follow the above submission instructions.