Assignment 2: Basic Java

Introduction
1. The assignment is due on 10.5.15 at 23:55.
2. Objective of this assignment: An introduction to programming with Java, including working with Iterators, Exceptions and Interface implementation.
3. The TA in charge of this assignment is Helal Assi. Questions should be sent to the mail helal.assi@gmail.com with the subject “236703 HW2”.
4. Read the instructions carefully, both in this document and in the provided code.
5. Make sure to write code that is clear, readable and reasonably documented.
6. Execution speed is not a central issue in the homework assignments in this course. In any case of uncertainty between simplicity and performance, choose the simpler implementation.
7. Try to avoid code duplication – use methods you’ve already written as much as possible.
8. In order to avoid mistakes please read the FAQ section in the course website which will be updated on an ongoing basis.

Preface
In this assignment you will need to implement a class for an “priority queue” data structure and classes defining a graph. These classes will be used to implement two graph traversal algorithms. The first algorithm is BFS – a Breadth First Search. The second algorithm is DFS – a Depth First Search. Both algorithms are explained in detail below.

Section A – Implementation of the Graph
This section describes the classes you must implement for a directed graph. Graphs in this assignment are described by the interfaces Vertex (for defining a graph node) and Edge (for defining an edge).

**Vertex**

You must implement the class `VertexImpl` which implements the interface `Vertex`.

- `VertexImpl(String name, int value)` – a constructor that receives a vertex’s name and number, and initializes the vertex with these values.
- `Value()` – returns the number of the vertex.
- `connect(Vertex other)` – Adds an edge to the graph from the current vertex to the vertex `other`. If such an edge already exists the method does nothing.
• size() – returns the number of vertexes in the sub-graph of which the current vertex is the root.
• find(Vertex target) – returns a collection containing the path (of Vertexes) from the current vertex to the target. If more than one path exists, an arbitrary path should be returned.
• getSuccessors() – returns a LinkedList containing all edges exiting the current vertex. **Important note:** The edges must be returned in the order in which they were added by the connect method. The automatic checker relies on this.

**Edge**

You must implement the class `EdgeImpl` which implements the interface `Edge`.

• getEndpoint() – returns the Vertex that the current edge points to. That is to say, if the current edge is $e = v \rightarrow u$ the method should return $u$.
• There are no special requirements for the c’tor.
Section B – Implementation of the Priority Queue
This section describes the class you must implement for the priority Queue.

An “Priority Queue” is a data structure that implements the same operations as a normal queue (enqueue, dequeue, peek) with a slight difference: the order in which elements are dequeued depends not only on the order of insertion into the queue, but also on the elements’ tags. The element removed each time the dequeue method is called should be the element with the minimal tag, or if there is more than one element with the minimal tag it should be the earliest element inserted into the queue with this tag.

An element’s tag is provided as a parameter to the enqueue method.

The PriorityQueueImpl class
The queue will include only vertices. Every Vertex enters with a tag. In addition, the class must implement the interface PriorityQueue, which is included in the provided code files, and also the interface Iterable<T>.

Methods

- enqueue(Vertex v, int tag) – inserts the vertex v into the queue with the tag value tag. The new vertex should be added after the last vertex with the same tag.
- dequeue() – removes an vertex from the queue. The vertex to be removed is the first vertex that was enqueued with the minimal tag. The method returns the vertex.
- peek() – returns the first vertex that was enqueued with the minimal tag without removing it from the queue.
- isEmpty() – returns true if and only if the queue is empty.
- Iterator() – returns an iterator. The iterator must traverse the vertex in the same order that the dequeue method would remove them from the queue. You can assume the queue will not change during the iteration.

Comments:

- The behavior of the Priority Queue is well defined and deterministic.
- In your implementation you may not use classes that implement the Queue<T> interface in java.util. You can read more about this in the Oracle site:

  [http://docs.oracle.com/javase/7/docs/api/java/util/package-summary.html](http://docs.oracle.com/javase/7/docs/api/java/util/package-summary.html)
Section C – Implementation of the Graph

This section describes the BFS and DFS algorithms and the classes you must define for them.

Both the BFS and DFS algorithms are based on the following generic algorithm:

1. Initialize two groups of vertexes: open and closed.
2. Insert the vertex from which the search will be started into open.
3. While open is not empty do:
   i. Remove a vertex from open (the choice of the vertex to be removed depends on the algorithm). We will call this vertex \( v \).
   ii. If \( v \) is in closed return to step i.
   iii. Open the vertex \( v \) (visit the vertex and insert all vertexes \( u \) for which an edge \( vu \) exists into open).
   iv. Insert \( v \) into closed.

**BFS (Breadth First Search) algorithm**

The algorithm traverses the graph widthwise, that is to say it first visits the starting vertex, then all neighboring vertexes (vertexes that are pointed to by an edge from the starting vertex). In the next stage the algorithm visits all vertexes at a distance of 2 from the start (neighboring vertexes of the vertexes in the previous stage) and so on.


**DFS (Depth First Search) algorithm**

The algorithm traverses the graph depthwise, that is to say it first visits the starting vertex, then after every vertex \( v \) visited by the algorithm, the next vertex is \( u \) such that the edge \( vu \) exists. If a vertex is reached for which there is no edge to a vertex that has not yet been closed, the algorithm returns to the last vertex with such an edge that it saw.


You should implement the classes `DFSGraphWalk` and `BFSGraphWalk` which will perform the DFS and BFS traversals accordingly. The classes should implement the interface `Iterable<Vertex>`, and each time the `next()` function of the iterator is called it should return the next step in the traversal.

**Instructions and hints for implementation of DFSGraphWalk and BFSGraphWalk**

- You must use the queue from section B to implement those algorithms.
- **Hint**: in order to implement those algorithms, you need to be able to get the priority (tag) of the vertex that was returned by the method dequeue. This can be done by using the method `Value`.
- The `Iterable<T>` interface:
  - In order for a class to implement the `Iterable<T>` interface it must contain a method `iterator()` that removes an object of a type that implements the interface `Iterator<T>`.
The Iterator defines a serial traversal of a group of objects; it does this with its methods hasNext() which returns true if the traversal has not yet finished, and next() which returns the next object.

- In this assignment the serial traversal of the iterator should be traversal according to the BFS or DFS algorithms.
- The Iterator<T> interface has a third method: remove(). This method will not be used in this assignment, and should have an empty implementation.
- There is an example of Iterable/Iterator implementation in recitation 3 and in the accompanying code for recitation 3.

- The c’tor should receive a parameter of type Vertex representing the starting vertex for the traversal of the graph.
- The behavior of the two algorithms should be deterministic. The requirements are formal, and the automatic tests rely on them. If your implementation of the queue and of getSuccessors() in VertexImpl are deterministic then the behavior of the algorithm will be deterministic.
- Both algorithms can be implemented as variants of the generic algorithm. What does this say about the class hierarchy in your implementation?
- Planning the implementation well will significantly reduce the length of the code and the implementation time.
- You may (and are encouraged to) add helper classes and/or abstract classes if your design requires it.
- You may assume that the graph will not change while the algorithms are traversing it.

**Automatic Testing Using JUnit**

You have been provided with a test class called Example to assist you in writing your own tests. This class works with the testing mechanism JUnit. We recommend that you run the test on your own implementation, and you should also write similar classes of your own with additional tests in order to help you test your work. Writing such tests of your own is not mandatory, and either way do not submit your tests.

**Compiling JUnit classes in Eclipse**

JUnit classes are written in Java and so are compiled automatically. However, it is necessary to add the JUnit library to the project. In order to do so, right click on the project in the project tree and choose **Build Path -> Add Libraries**. Click on the JUnit library and then press Next, choose **JUnit 4** in the window that opens.

**Running JUnit classes in Eclipse**

There are several ways to run the JUnit classes. The easiest way is to right click on the test file and then choose **Run As -> JUnit Test**. Another way is to choose **Run->Run Configurations** from the main menu and then double click on JUnit in the list on the left hand side, then fill in the run parameters (Choose which test classes will run). After running the tests a window will open with the results. If the line at the top of the window is green then all tests passed successfully, if
it is red then a list of failed tests will appear. Click on a specific test to receive further details on the failure.

**Writing JUnit classes**
A short explanation can be found [here](#). In short: you should write a class with a number of methods such that before each method definition write @Test. In the body of the class you can use static methods from the class `org.junit.Assert` to ensure the values you computed are as you expected them to be. It is recommended that you look at the code of the classes we provided, there are additional comments there that explain the meaning of relevant lines in the code.

**General Notes:**
1. You may not change the provided files. The automatic checker will overwrite these files with the original version.
2. Make sure that the method `equals` works "right" for all of the provided classes.
3. Add comments to all public classes and methods.
4. All of your classes should be implemented in the package `OOP2.Solution`.
5. You may not use external libraries in your implementation. You may use classes from `java.util` that are not specifically forbidden in part A.
6. Do not print anything to the standard output or the standard error. If you use printouts for your tests make sure to delete them before submitting your assignment.
7. You are provided with a file called Example.java that contains run examples for the assignment. You are required to ensure the test in the file compiles and succeeds with your implementation (if it doesn’t there is a good chance the official tests won’t succeed). Do not submit Example.java.

**Submission Details**
- Requests for postponement, for any reason, must be sent to the TA in charge of the course (Nurit). Note that the course has a late-submission policy, meaning you can submit your assignment late without a postponement approval. Details can be found in the course site under “General Info”.
- The assignment must be submitted electronically. (Save the submission confirmation).
- Your submission should consist of a zip file with the name format: `OOP2_<ID1>_<ID2>.zip` containing:
  - A file called readme.txt with the name, id and email of each of the submitters.
  - The zip file should hold all code files you wrote in order to solve the assignment.
- Do not submit the provided class files, and do not submit any tests.
- Points will be deducted for not conforming to the submission requirements (rar instead of zip, extra files, a readme file with the wrong name, etc.

*Good Luck!*