Assignment 1

20 Marks

Due: Monday, 27 August 2018 at 6:00pm

Introduction

Data structures are a core part of the design of most applications. In this first assignment you will implement three concrete data types that will be used in a simulation of an air traffic management system. The simulation is based on the OneSKY Australia project, which will provide integrated air traffic management for all of Australia's airspace.

Details

You will implement and test three different data structures. The data structures are a `Cube` (a three-dimensional array-like structure), `IterableQueue` and an `Iterator` that can iterate over the `IterableQueue`. You are provided with Java interfaces to define these abstract data types (ADTs). You are also provided with initial shells of the classes for the concrete data types (CDTs) that you need to implement. You are provided with some initial JUnit tests for the CDTs. You will need to implement further JUnit tests to fully test your CDTs.

The Cube ADT is a collection that holds items in a three-dimensional grid layout. Each item in the grid is accessed by an (x, y, z) coordinate tuple. The Cube<T> interface defines the public methods and behaviours of the ADT. You need to implement the BoundedCube<T> class to provide a concrete implementation of the data structure.

The `IterableQueue` ADT is a collection that holds items in the order in which they are added. A queue is commonly referred to as a "first-in, first-out" collection, meaning that the first item added to the queue will be the first item removed from the queue. This queue is iterable meaning that it is possible to access each element in the queue, starting at the front of the queue and progressing towards the end of the queue. The `IterableQueue<T>` interface defines the public methods and behaviours of the ADT. `IterableQueue<T>` extends the `Iterable<T>` interface, meaning that you will need to implement the iterator method defined in `java.lang.Iterable`. You do not need to implement the `forEach` and `spliterator` default methods defined in the `Iterable<T>` interface. You need to implement the `TraversableQueue<T>` class to provide a concrete implementation of the data structure.

The iterator method in the `TraversableQueue<T>` class will need to return an object that implements the `java.util.Iterator` interface. This object will need to be able to iterate over all the elements in an `IterableQueue` and provide access to each element in turn. The iterator will start at the front of the queue and progress towards the end of the queue. If elements are dequeued while the iterator is iterating over the queue, it is possible that there will no longer be an element at the iterator's position in the queue. If this happens the iterator's next method should throw a `java.util.NoSuchElementException`. One approach to implementing the iterator is to make it a private inner class in the `TraversableQueue` class.

Two JUnit test classes are provided: `BoundedCubeTest` and `TraversableQueueTest`. They are written using JUnit 4 and use the Hamcrest matcher framework. These provide a few initial tests of the CDTs you need to implement. You will need to implement further JUnit tests of your CDTs. Your own JUnit tests should be in classes called: `MyBoundedCubeTest` and `MyTraversableQueueTest`. You do not need to duplicate the provided tests in your test classes. Do **not** add tests to the provided JUnit test classes. Your JUnit tests **must** be written using JUnit 4 but do not need to use the Hamcrest matcher framework.

You are also provided with a very simple air traffic management simulator that makes use of the CDTs. It provides both an automated and interactive mode of operation. Interactive mode is started by launching the application with no command line parameters (e.g. `java OneSky`). Automatic mode is started by launching the application with two command line parameters, one to indicate auto mode and the other being the number of iterations of the simulation (e.g. `java OneSky auto 1000`). You do not need to implement any features in the simulator. It is provided simply as an example of how the ADTs could be used.

Constraints

You may not change anything in the provided ADT interfaces. You may not move any provided classes or interfaces to different packages. You may not provide any public methods in the CDT classes that are not implementations of the methods defined in their interfaces. You may not use anything from the Java Collections framework in the JDK, aside from the `Iterable` and `Iterator` interfaces that are used by the provided ADT interfaces. You may make use of the `Comparable` or `Comparator` interfaces, if you wish, in your solution. Your implementations of the CDTs must be based on **basic** Java language constructs and **not** data structure or algorithm libraries. You may not add or remove methods to or from the provided JUnit test classes. You must provide public constructors for the CDT classes that correspond to the CDT object creation in the provided JUnit test classes.

Your code must compile using the standard Oracle JDK version 8 or Open JDK version 8. It is not acceptable that your code only works in your IDE. Note that IntelliJ provides inaccurate code coverage data for test cases.

You may provide as many other public, and non-public, constructors as you see fit for your CDT classes. You may add extra non-public data members or methods to the CDT classes. Use these to keep the method logic simple and to avoid repeating code. You may add non-public classes to the design to help with the implementation of your CDTs.

Design Considerations

The OneSky simulation application is meant to be able to represent the entire Australian airspace. For simplicity we will say that the Australian airspace is a cube that is 5321 kilometres from west to east, by 3428 kilometres from north to south, by 35 kilometres high. This corresponds to latitudes 8 to 44 degrees south and longitudes 108 to 156 degrees east. It also accounts for the maximum altitude at which jet aircraft can fly.

For the purposes of the simulation, we will say that aircraft are meant to be separated at all times by one kilometre. Each cell in the BoundedCube will represent one cubic kilometre. We will also say that the Australian airspace will contain a maximum of 20,000 aircraft at any one time. Your BoundedCube design should take into consideration efficient use of memory and must be able to represent the entire Australian airspace on a computer with 8 GB of memory.

Each cell in the Cube must be able to accommodate more than one aircraft because it is possible for aircraft to encroach on each other's space. In the simulation, the `addAircraft` method in `AirSpace` checks to see an aircraft is encroaching on another aircraft's space.

The purpose of the `IterableQueue` is to provide an ability to access all elements in the queue. For the simulation this could be to simulate contacting all aircraft in a single cell of the airspace to warn that they are too close together. Or, as is done in the interactive mode of the simulation, it allows simulation of searching for aircraft that have been identified by radar but not yet plotted in the airspace.

The OneSky simulation is intentionally incomplete and only creates aircraft that are populated into the airspace. You are not expected to add any new functionality to the simulation. It is provided solely as a context to be considered when designing the CDTs.

Analysis

In the JavaDoc comments for your CDT classes indicate the run-time efficiency of each public method. Use big-O notation to indicate efficiency (e.g. O(nlogn)). This should be on a separate line of the method's JavaDoc comment, as the last line of the description and before any `@param`, `@return` or `@throws` tags. When determining run-time efficiency of a method, take into account any methods that it may call.

In the JavaDoc comments for your CDT classes indicate the memory usage efficiency of the data structure. This should be in the JavaDoc comment for the class and should be on a separate line, which is the last line of the description and before any `@author`, `@see` or any other tags.

In a comment at the end of each CDT class, provide a justification for the design choices that you made in implementing the CDT. The justification should consider the run-time and memory space efficiency of the data structure and methods in the context of the air traffic management simulation.

For the BoundedCube class, ensure that your justification clearly considers memory usage in the context of the OneSky simulation. In this simulation there will be a proportionally very small number of occupied cells in the cube data structure. Your justification should compare alternative approaches to implementing the data structure. If you think your implementation has limitations, describe an alternative implementation that would be more suitable for the OneSky simulation.

Provide proper bibliographic references and citations for any resources that you use in designing your implementations of the CDTs or in analysing the CDT design. Follow IEEE referencing standards and place the bibliographic references at the end of your justification comment in each CDT file.

Submission

You must submit your completed assignment electronically through Blackboard. You should submit your assignment as a single zip file called **`assign1.zip`** (use this name – all lower case). This zip file **must** contain the same directory structure provided in the original **`assign1.zip`** that contained the initial code for the assignment. Submissions that **do not** conform to the requirements (e.g. different directory structures, packages or classes) **may not** be marked.

You may submit your assignment multiple times before the deadline – only the last submission will be marked.

Late submission of the assignment will **not** be marked. In the event of exceptional personal or medical circumstances that prevent you from handing in the assignment on time, you may submit a request for an extension.

See the course profile for details of how to apply for an extension:
http://www.courses.uq.edu.au/student_section_loader.php?section=5&profileId=94208

Requests for extensions **must** be made no later than 48 hours prior to the submission deadline. The application and supporting documentation (e.g. medical certificate) **must** be submitted via my.UQ. You **must** retain the original documentation for a minimum period of six months to provide as verification should you be requested to do so.

Please read the section in the course profile about plagiarism. Submitted assignments will be electronically checked for potential plagiarism.

Assessment and Marking Criteria

This assignment assesses course learning objectives:

1.1 Understand the internal workings of fundamental data structures and algorithms.
1.2 Determine the running time and memory space usage of common algorithms.
2.1 Adapt or invent new algorithms and data structures for software engineering problems.
2.2 Analyse the performance of algorithms built on fundamental data structures and algorithms.
2.3 Select and justify appropriate combinations of data structures and algorithms to solve software engineering problems.

Completeness

The marking criteria below are based on the assumption that the entire assignment is completed. A partially complete assignment will be accepted and marked but the final mark for the assignment will be capped, based on the amount of work completed. The maximum marks that can be achieved are:

- 5 marks, if the assignment does not compile, based solely on the analysis provided.
- 8 marks, if only the `TraversableQueue`, without an iterator, is implemented.
- 12 marks, if only the `TraversableQueue`, with an iterator, is implemented.
- 10 marks, if only the BoundedCube is implemented.
- 20 marks, if all parts of the assignment are implemented.

Criteria	Excellent	Good	Satisfactory	Poor
Functionality	▪ Passes at least 90% of test cases, only failing sophisticated or tricky tests.	▪ Passes at least 80% of test cases, only failing one or two simple tests.	▪ Passes at least 70% of test cases.	▪ Passes less than 70% of test cases.
	7	6	5 4	3 2 1 0
Code & Design Quality	▪ Code is well structured with excellent readability and clearly conforms to a coding standard. ▪ Comments are clear, concise and comprehensive, enhancing comprehension of method usage and implementation details. ▪ All design choices are appropriate for the context and the ADT.	▪ Code is well structured with very good readability and conforms to a coding standard. ▪ Comments are clear, mostly concise and comprehensive, and in most cases enhancing comprehension of method usage and implementation details. ▪ All, but one, design choice is appro-priate for the context and the ADT.	▪ Code is well structured with good readability and mostly conforms to a coding standard. ▪ Comments are clear and fairly comprehensive, providing useful information about method usage and implementation details. ▪ Most design choices are appropriate for the context and the ADT.	▪ Parts of the code are not well structured, are difficult to read, or do not conform to a coding standard ▪ Comments at times are not clear, or provide little useful information about method usage or implementation details. ▪ Most design choices are inappropriate for the context or the ADT.
	2	1.5	1	0.5 0
Testing	▪ All public methods in the CDTs are comprehensively tested. (e.g. 90% of branches and almost all boundary conditions are tested.) ▪ Tests are clearly designed with an exemplary understanding of the algorithm and data structure design.	▪ Almost all public methods in the CDTs are well tested. (e.g. 80% of branches and most boundary conditions are tested.) ▪ Tests seem to be designed with a very good understanding of the algorithm and data structure design.	▪ Most public methods in the CDTs are moderately well tested. (e.g. 70% of branches and some boundary conditions are tested.) ▪ Tests seem to be designed with a fairly good understanding of the algorithm and data structure design.	▪ Some public methods in the CDTs are adequately tested. (e.g. less than 70% of branches and a few boundary conditions are tested.) ▪ Tests seem to be designed with little understanding of the algorithm and data structure design.
	4	3	2	1 0
Analysis	▪ Correctly determined run-time and space efficiency of every method and CDT. ▪ Justification of CDT design choices are valid, clearly expressed and relevant to the application. ▪ BoundedCube discussion demon-strates a good understanding of the issues related to its memory usage in this context and the trade-offs of different potential implementations.	▪ Correctly determined run-time efficiency of almost all methods and space efficiency of all CDTs. ▪ Justification of CDT design choices are valid and relevant to the application. ▪ BoundedCube discussion demon-strates a good understanding of the issues related to its memory usage in this context and some trade-offs of different potential implementations.	▪ Correctly determined run-time and space efficiency of most methods and CDTs. ▪ Justification of CDT design choices are mostly valid and somewhat relevant to the application. ▪ BoundedCube discussion demon-strates a good understanding of the issues related to its memory usage in this context.	▪ Correctly determined run-time and space efficiency of some methods and CDTs. ▪ Justification of CDT design choices are at best partially valid or not relevant to the application. ▪ BoundedCube discussion demon-strates little understanding of the issues related to its memory usage in this context.
	7	6	5 4	3 2 1 0
Total:				