Assignment 1

This assignment is worth 15% of the total course marks.

<u>**Due date:**</u> The assignment is due at **9am on Monday 09 September.** Late assignments will attract a penalty of 10% of the total assignment mark per day, and submissions more than 5 days late will not be marked. Only extensions on Medical Grounds or Exceptional Circumstances will be considered, and in those cases students need to submit an application for extension of progressive assessment form (http://www.uq.edu.au/myadvisor/forms/exams/progressive-assessment-extension.pdf) to the lecturer (email is acceptable) or the ITEE Enquiries desk on Level 4 of GPSouth Building) prior to the assignment deadline.

<u>Resources:</u>

The Assignment 1 zip file, available from Blackboard (under Learning Resources/Course Materials), contains the Java classes and interfaces that are mentioned in this handout. You will need those source files to complete this assignment.

Your Java Build Path will need to include the JUnit 4 Library to run the JUnit tests.

<u>Part 1:</u> Testing and implementing stacks and lists [2 marks]

The purpose of this part of the assignment is to familiarise you with array and linked-based implementations of linear abstract data types (ADTs), and to accustom you to using JUnit tests to uncover implementation errors.

Question 1(a) [1 mark]:

Class `part1.ArrayStack` (that is, class `ArrayStack` in the `part1` package) is an implementation of the `adt.IStack` interface that uses a fixed-length array.

Using the JUnit tests in `part1.test.ArrayStackTest` discover and correct the 2 ERRORS in the `part1.ArrayStack` implementation of the `adt.IStack` interface. When you find an error, insert a brief one-line "//" comment at that location, indicating where the error had been found, and why it occurred.

You may not modify the code other than to fix the 2 ERRORS and insert the required comments.

(Note that we do not consider it an error that the size of the stack is bounded above by the capacity of the underlying fixed-length array in this implementation.)

Question 1(b) [1 mark]:

Class `part1.LinkedList` is a doubly-linked list implementation of the `adt.PositionList` interface, that does not use header and trailer sentinels. It uses the `part1.Node` implementation of the `adt.Position` interface.

Using the JUnit tests in `part1.test.LinkedListTest` discover and correct the 4 ERRORS in the `part1.LinkedList` implementation of the `adt.PositionList` interface. When you find an error, insert a brief one-line "//" comment at that location, indicating where the error had been found, and why it occurred.

You may not modify the code other than to fix the 4 ERRORS and insert the required comments. Do not modify `part1.Node` in any way.

<u>Part 2:</u> Stock Portfolio [3 marks for COMP3506: 6 marks COMP7505]

The purpose of this part of the assignment is to give you experience implementing a simple class using linear data structures, and analysing your solution in terms of its space and time complexities.

Your task is to efficiently implement class `part2.Portfolio`, that represents a portfolio of company stocks. The class keeps track of the purchase price of each individual stock held, and the order in which they were purchased. Stocks may be purchased and added to the portfolio via the buy operation, and removed from the portfolio using the sell operation. The sell operation calculates the capital gain (or loss) from the sale.

Question 2(a) [2 marks]:

Your job is to implement the class skeleton `part2.Portfolio` efficiently according to its specification in that file.

You may use the linear data structures in `java.util` from the Java 7 SE API in your implementation (e.g. ArrayList, LinkedList, ArrayDeque etc.), but no other libraries should be used. (It is not necessary and makes marking hard.)

Any extra classes that you write should be included in the file `part2.Portfolio` as private nested classes. Any additional class methods should also be private.

To help you get your solution right, you should write you own JUnit tests in `part2.test.PortfolioTest` (some basic tests are included to get you started). Tests that you write will not be marked. We will test your implementation with our own tests.

Question 2(b) [1 marks]:

For your implementation, give the worst case time complexity of operations *buy* and *sell* in portfolio, given that *n* buy operations and *m* sell operations have already taken place (in any valid order, starting from an empty portfolio); and the worst case space complexity of a portfolio after *n* buy operations and *m* sell operations have taken place.

Express your answers in big-Oh notation, making the bounds as tight as possible, and simplifying your answer as much as possible.

Briefly justify your answers.

Question 2(c) [3 marks]: COMP7505 ONLY:

For your implementation, give the worst-case time complexity of performing *n* buy operations and *m* sell operations (in any valid order) on an initially empty portfolio; and use your result to give the *amortised running time* of the buy and sell operations.

Express your answers in big-Oh notation, making the bounds as tight as possible, and simplifying your answer as much as possible.

Justify your answers.

Hint: You may want to justify your amortised analysis of the running time of the buy and sell operations using the *accounting technique* described in pages 245-246, 543-546 of the textbook Data Structures and Algorithms in Java (5th Edition) by Goodrich and Tamassia.

Part 3: Trees [10 marks]

The purpose of this part of the assignment is for you to gain experience writing and analysing binary tree algorithms.

Given an individual's answers to a number of questions, the authorities use a decision tree to determine an appropriate reward, or penalty, for that person.

The *decision tree* is a non-empty proper binary tree in which the elements of internal nodes in the tree are integers that represent question numbers. Each internal node has exactly two children: the left child represents the remainder of the decision tree that is completed if the answer to the question is *yes* (true); and the right node represents the remainder of the decision tree that is completed if the answer to the question is *no* (false). The external nodes of the tree contain integers that represent the reward associated with reaching the conclusion that follows from having answered the questions (starting from the root node) in such a way as to reach that external node. Note that in the special case that the decision tree has only one node, the root, no questions have to be asked or answered to receive the reward given in that external node.

For convenience, if there are q different questions in the internal nodes of the decision tree, then those questions are each represented by an integer between 0 and q-1 (inclusive).

For example, the following decision tree happens to contain three questions: Questions 0, 1, 2 and 3. If a person has answered *yes* to Question 0, *no* to Question 1, *yes* to Question 2 and *yes* to Question 3, then their reward, as determined by the authorities using the decision tree, would be 1.

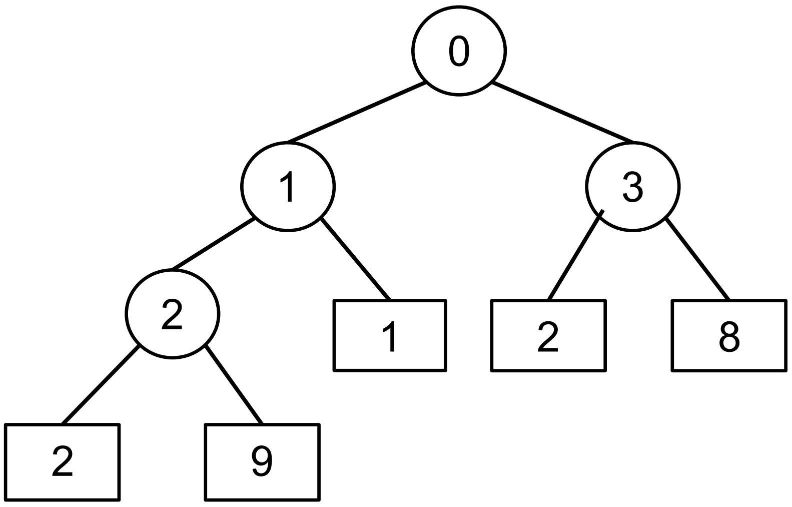

Figure 1: Example decision tree.

Question 3(a)[1 marks]:

Implement method `reward` in the class `part3.Question3` efficiently according to its specification in that file. The method `reward` takes a decision tree annotated with rewards for reaching the different outcomes; and an array containing a person's answers to all of the questions that appear anywhere in the decision tree. It returns the person's reward determined by the decision tree using the given answers.

You must implement this method yourself, "from scratch", using only methods provided by the `adt.Position`, and `adt.BinaryTree` interfaces. (This means that you should not add any additional import clauses.)

You may write supporting methods, but they must be declared private and included in the `part3.Question3` file. Similarly, if you choose to write any supporting classes, then they must be written as private nested classes and included in that file.

To help you get your solution right, you should write you own JUnit tests in `part3.test.Question3Test` (some basic tests are included to get you started). Tests that you write will not be marked. We will test your implementation with our own tests.

Question 3(b)[1 mark]:

What is the worst-case time complexity of your `reward` method in terms of the total number of nodes *n* in the decision tree? Briefly justify your answer.

Express your answers in big-Oh notation, making the bounds as tight as possible, and simplifying your answer as much as possible.

Question 4(a)[2 marks]:

The *indented representation* of the decision tree in Fig. 1 is given below.

```
Q0:
   Y->Q1:
      Y->Q2:
         Y->R2
         N->R9
      N->R1
   N->Q3:
      Y->R2
      N->R8
```

In general, the *indented representation* of a decision tree with only one node, the root, consists of one line without leading or trailing spaces containing the string “RX”, where X is the integer representation of the reward at the root.

For a decision tree with more than one node, the first line of the *indented representation* contains the string “QX:”, without any leading or trailing spaces, in which X stands for the integer representation of the question number at the root. Following that line comes the *indented representation* of the left subtree of the root, followed by the *indented representation* of the right subtree of the root.

The *indented representation* of a subtree of a decision tree is indented exactly three single space characters (‘ ‘) relative to its parent. It’s first line contains the string “A->Z” with no trailing spaces, where A is the character ‘Y’ if it is a left-subtree and ‘N’ if it is a right-subtree; and Z is either of the form “RX”, if the sub-tree only contains one node, and “QX:”, otherwise. If the sub-tree contains more than node, then following the first line of the sub-tree comes the *indented representation* of the left subtree, followed by the *indented representation* of the right subtree. There are no further lines in the representation of a subtree with only one node.

Give a recursive implementation of the method `printDecisionTree` in the class `part3.Question4` that can be used to print the indented representation (as described above) of decision trees.

You must implement this method yourself, "from scratch", using only methods provided by the `adt.Position`, and `adt.BinaryTree` interfaces. (This means that you should not add any additional import clauses.)

You may write supporting methods, but they must be declared private and included in the `part3.Question4` file. Similarly, if you choose to write any supporting classes, then they must be written as private nested classes and included in that file.

To help you get your solution right, you should write you own JUnit tests in `part3.test.Question4Test` (some basic tests are included to get you started). Tests that you write will not be marked. We will test your implementation with our own tests.

Question 4(b)[1 mark]:

What is the worst-case time complexity of your `printDecisionTree` method in terms of the total number of nodes *n* in the decision tree? Briefly justify your answer.

Express your answers in big-Oh notation, making the bounds as tight as possible, and simplifying your answer as much as possible.

Question 5(a)[3 marks]:

An individual is aware of the decision tree that is used by the authorities, and is prepared to revise their actual answers to each of the questions if it will benefit them. Unfortunately, there are costs associated with lying, and so the reward that they may receive by lying is offset by the cost of telling the lies. Your job is to write a method that will help them work out how to amend their answers in a way that will benefit them the most.

In particular, you have to implement method `revisedAnswers` in the class `part3.Question5` according to its specification in that file. You should make your implementation as efficient as you can.

To simplify your solution, you may assume that the same question does not appear more than once on any simple path from the root of the decision tree to one of its external nodes. (Note that despite this restriction the same question may appear more than once in the tree as a whole.)

You may use the linear data structures in `java.util` from the Java 7 SE API in your implementation (e.g. ArrayList, LinkedList, ArrayDeque etc.), but no other libraries should be used. (It is not necessary and makes marking hard.)

You may write supporting methods, but they must be declared private and included in the `part3.Question5` file. Similarly, if you choose to write any supporting classes, then they must be written as private nested classes and included in that file.

To help you get your solution right, you should write you own JUnit tests in `part3.test.Question5Test` (some basic tests are included to get you started). Tests that you write will not be marked. We will test your implementation with our own tests.

Question 5(b)[2 marks]:

Let *n* be the total number of nodes in the decision tree, and *q* be the number of distinct questions in the tree. Give the worst-case time and space complexity of your `revisedAnswers` method in terms of *q* and *n*. Briefly justify your answers.

Express your answers in big-Oh notation, making the bounds as tight as possible, and simplifying your answer as much as possible.

(N.B. The space complexity of your method should be measured in terms of how much additional space your method consumes. It does not include the size of the method parameters.)

Practical considerations:

If necessary, there may be some small changes to the files that are provided, up to 1 week before the deadline, in order to make the requirements clearer, or to tweak test cases. These updates will be clearly announced on the Announcements page of the course Blackboard site, and during the lectures.

For the coding tasks:

- Don't change the class names, specifications provided, or alter the method names, parameter types or return types or the packages to which the files belong.
- Don't write any code that is operating-system specific (e.g. by hard-coding in newline characters etc.), since we will batch test your code on a Unix machine.
- Your source files should be written using ASCII characters only.

Submission: Submit each of your source code files `ArrayStack.java`, `LinkedList.java, Portfolio.java, Question3.java, Question4.java and Question5.java` as well as your written answers to questions 2(b-c), 3(b), 4(b) and 5(b) in `report.pdf` electronically using Blackboard according to the exact instructions on the Blackboard website:

https://learn.uq.edu.au/

Answers to each of the questions 2(b-c), 3(b), 4(b) and 5(b) should be clearly labelled and included in file `report.pdf`.

You can submit your assignment multiple times before the assignment deadline but only the last submission will be saved by the system and marked. Only submit the files listed above. You are responsible for ensuring that you have submitted the files that you intended to submit in the way that we have requested them. You will be marked on the files that you submitted and not on those that you intended to submit. Only files that are submitted according to the instructions on Blackboard will be marked

Evaluation: Your assignment will be given a mark according to the following marking criteria. For COMP3506 the total marks possible is 15, and for COMP7505 the total marks possible is 18. The assignment is worth 15% for both courses.

Code must be clearly written, consistently indented, and commented. Java naming conventions should be used, and lines should not be excessively long (> 80 chars). Marks will be deducted in code analysis questions if we cannot read and understand your solution to check your analysis.

Part 1

Q1(a) [1 mark]:

- solution compiles and passes ALL part1.test.ArrayStackTest tests, and location and cause of all errors correctly marked — 1 mark
- otherwise or work has no academic merit — 0 marks

Q1(b) [1 mark]:

- solution compiles and passes ALL part1.test.LinkedListTest tests, and location and cause of all errors correctly marked — 1 mark
- otherwise or work has no academic merit — 0 marks

Part 2

Q2(a)[2 marks]:

Given that the solution does not violate any restrictions (using forbidden libraries etc), [0.5 marks] will be allocated for efficiency and the remaining [1.5 marks] will be allocated based on the outcome of running test cases:

• All of our tests pass	1.5 marks
• At least 2/3 of our tests pass	1 marks
• At least 1/3 of our tests pass	0.5 marks
• Work with little or no academic merit	0 marks

Note: code submitted with compilation errors will result in zero marks in this section. Code that violates any stated restrictions will receive zero marks.

Q2(b)[1 mark]

Analysis is correct and justified. [1 mark]

A mark of 0 will be given for this part if no reasonable attempt was made to complete part (a) or the solution to part (a) is difficult to read and comprehend.

Q2(c)[3 mark] – COMP7505 ONLY

Analysis is correct and justified. [3 marks]

A mark of 0 will be given for this part if no reasonable attempt was made to complete part (a) or the solution to part (a) is difficult to read and comprehend.

Part 3

Q3(a)[1 marks]:

Given that the solution does not violate any restrictions (using forbidden libraries etc) marks will be allocated based on the outcome of running test cases:

• All of our tests pass	1 mark
• At least 1/2 of our tests pass	0.5 marks
• Work with little or no academic merit	0 marks

Note: code submitted with compilation errors will result in zero marks in this section. Code that violates any stated restrictions will receive zero marks.

Q3(b)[1 mark]

Analysis is correct and justified. [1 mark]

A mark of 0 will be given for this part if no reasonable attempt was made to complete part (a) or the solution to part (a) is difficult to read and comprehend.

Q4(a)[2 marks]:

Given that the solution does not violate any restrictions (using forbidden libraries etc) [2 marks] will be allocated based on the outcome of running test cases:

• All of our tests pass	2 marks
• At least 3/4 of our tests pass	1.5 marks
• At least 1/2 of our tests pass	1 marks
• At least 1/4 of our tests pass	0.5 marks
• Work with little or no academic merit	0 marks

Note: code submitted with compilation errors will result in zero marks in this section. Code that violates any stated restrictions will receive zero marks.

Q4(b)[1 mark]

Analysis is correct and justified. [1 mark]

A mark of 0 will be given for this part if no reasonable attempt was made to complete part (a) or the solution to part (a) is difficult to read and comprehend.

Q5(a)[3 marks]:

Given that the solution does not violate any restrictions (using forbidden libraries etc), [1 mark] will be allocated for efficiency and the remaining [2 marks] will be allocated based on the outcome of running test cases:

• All of our tests pass	2 marks
• At least 3/4 of our tests pass	1.5 marks
• At least 1/2 of our tests pass	1 marks

• At least 1/4 of our tests pass	0.5 marks
• Work with little or no academic merit	0 marks

Note: code submitted with compilation errors will result in zero marks in this section. Code that violates any stated restrictions will receive zero marks.

Q5(b)[2 marks]

Analysis is correct and justified. [2 marks]

A mark of 0 will be given for this part if no reasonable attempt was made to complete part (a) or the solution to part (a) is difficult to read and comprehend.

School Policy on Student Misconduct: You are required to read and understand the School Statement on Misconduct, available on the School's website at:

http://ppl.app.uq.edu.au/content/3.60.04-student-integrity-and-misconduct

This is an individual assignment. If you are found guilty of misconduct (plagiarism or collusion) then penalties will be applied.

If you are under pressure to meet the assignment deadline, contact the course coordinator **as soon as possible**.