CSCU9A3 Assignment: University Management System

Due 4pm, 26th November 2021

You will implement components of a university management system, which stores information about students, professors, and modules. You will be given partly complete code, and it is your job to complete the code. Comments containing "TODO" mark placeholders in the existing code for you to complete.

Test code is supplied showing you the expected output. Look at the expected output to understand what the code should do. The code for the assignment is available as a zip file on Canvas.

You are required to make the following enhancements to the code, each identified by numbers in square brackets below. The complete assignment is worth 45% of the overall grade for CSCU9A3.

Refactoring the code

Analysing the code, you will see that there is a lot of code replication between classes *Student* and *Professor*.
[1] Refactor/restructure the code in order to **reduce code replication**.

Improving Student Class

Nowadays, universities need to track more information about the students. You are given a basic *Student* class, but it needs the following functionality added.

[2] Create a new attribute, called *enrolledModules*, that will store the current modules in which the student is enrolled.
[3] Create a new attribute, called *isARUAA*, that will store whether a student is ARUAA or not.
[4] Add the necessary methods for the new attributes, according to the encapsulation concept.
[5] Create a new constructor for *Student* with all parameters.
[6] Amend old constructors to instantiate the new attributes; if needed, initialise them as empty (eg.: 0, if *Integer*, false, if *Boolean*, "", if *String*, …).
[7] Ensure that *addModule*() method checks adding new modules for a student will not exceed the maximum number of modules (MAX_NUM_MODULES) before adding them, and returns an appropriate value to confirm whether the module number could be altered.
[8] Change the *addModule*() method to also add the module to recently created *enrolledModules* attribute.
[9] Ensure *addModules*() only allows the same module to be added once.
[10] Add a method to remove modules from *enrolledModules*; remember to also remove it from the *numberModules* attribute.

Adding a MScStudent Class

The universities also have post-graduate students, such as MSc. To handle these students, create a new class, called *MScStudent*.

[11] Add a *MScStudent* class – avoid code replication.
[12] Add the following attributes to this class: string *researchTitle* and Professor *supervisor*.
[13] Create constructor and methods for the attributes, according to the encapsulation concept.
[14] Create a method to print all attributes of this class. This method must allow printing of objects of type *MScStudent* using *System.out.println*.

Binary Tree Walk

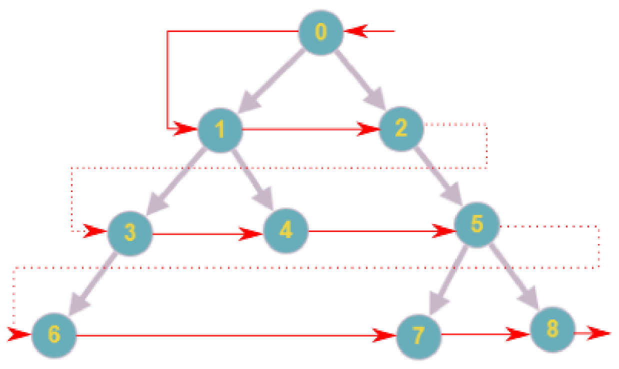
In our system, we have a class *Module* that represents the modules offered by the university. We also have a class *Cohort* that associates Module, Students, and Professor. Precisely, this class stores a Module, the list of students enrolled in that module, and the Professor teaching that module. We are using a Binary Tree to store the list of students. A *Node* class containing a reference to a *Student* object has been provided for you together with a partially implemented *BinaryTree* class consisting of *Node* objects. The *BinaryTree* class contains the implementation necessary to add *Student* objects to the tree, method definition to walk, and a method definition for a find method.

The class *University* has a list (arraylist) of *Cohort*. This class implements important methods that handle the students enrolled in a specific cohort:

1) *walkTree*(val), which calls a walk over the binary tree of students associated with the cohort in index *val*
2) *find*(val, name), which calls the method find for the cohort in index *val*

[15] Your task is to complete the implementation of the *walkTree* method in BinaryTree.java

This method should visit the nodes as shown below.

Input binary tree	**Expected output**
	0,1,2,3,4,5,6,7,8

Those methods will return a String object containing a comma separated list of strings representing the Students, by calling *getNodeObjectName*() on each Student object.

The initial implementation of the walk methods will result in only the root node of the tree being returned. You must finish the implementation of the method. As a starting point and to provide some clues as to how you might write your tests, the *go*() method of *UniversityTest*.java contains calls to print out the student list of a specific module, print out the tree, and calls the traversal method.

Searching a Binary Tree
It is not much use having a more efficient structure for searching for items in a binary tree if we don't actually have a method to find the items. An outline for a *find*() method has been supplied for you in the class *BinaryTree*. The *go*() method in *UniversityTest*.java makes a call to the above private find method (by calling the find method in the *University*.java for a specific module) and provides code that will determine if the Student with the name 'John' or 'Jack' has been located in a specific module.

[16] Your task is to complete the *find*() method in the BinaryTree.java, such that it finds a Student with the given name or returns null if it couldn't find it.

Remember that, to call the *find*() method in the *BinaryTree*, you need to use the public method find(val, name) in the *University*.java, which will then call the find() method in the *BinaryTree* for the module in index *val*.

The solution only involves about 8-10 lines of code. The challenge is handling the recursive search process rather than writing many lines of code.

Sorting
To allow the universities to properly track the current offered modules, we need to sort the original ArrayList of Modules in the University class by their name and code.

[17] Your task is to implement the *QuickSort* algorithm in the University class. The sorting should be based on:

- name or code for each Module, depending on the value passed to the "attr" parameter. You can assume this is always a string with value "name" or "code",
- ascending or descending, depending on the value passed to the "asc" parameter.

Outline code to check if the list is sorted correctly, and to print the results of attempting to do this is provided for you in the *UniversityTest* class.

Graph
As can be seen in the *Module* class, each module has a list of prerequisites. However, to avoid issues, there should not be a cyclic reference between modules and prerequisites, i.e., a situation in which "module A is prerequisite of B and module B is prerequisite of A" or "module A is prerequisite of B, module B is prerequisite of C, and module C is prerequisite of A" should **never** happen. To detect this type of problem, we are going to

use a directed graph. A class Graph has been implemented for you. In the class University, there is a graph that represents the relationship of the module and pre-requisites.

[18] Implement the method *checkForCycles* in the *University* class to detect cycles in the graph.

Again, the solution only involves about 8-10 lines of code and the challenge is handling the recursive search process rather than writing many lines of code.
NOTE: you just need recursion to implement this. However, lectures of Week 10 (which will be released before the assignment submission deadline) might help.

Testing
[19] Write appropriate test methods for the (i) tree walk, (ii) find method, (iii) quick sort, and (iv) graph cycle, in *UniversityTest*.java to check that they work as intended.

Summary of Requirements
To summarise, you are required to:

- Refactor the code (task [1]) (**15%**)
- Modify the *Student* class (tasks [2]-[10]) (**12%**)
- Create a *MScStudent* class (tasks [11]-[14]) (**7%**)
- Implement the walk and find methods in *BinaryTree* (tasks [15]-[16]) (**20%**)
- Implement the *quickSort* method in *University* (task [17]) (**16%**)
- Implement the *checkForCycles* method in *University* (task [18]) (**12%**)
- Write tests in *UniversityTest* (task [19]) (**8%**) - you may wish to create additional unit tests to the ones required to ensure that your code is working as intended although these will not be graded
- Marks will also be allocated for appropriate commenting, sensible variable names, and general quality of code – efficient, concise, readable (**10%**)

Your submission will be graded according to the Common Mark Scheme.

Submission
Please ensure that all your code compiles. Only a minimal attempt will be made to try to get code to compile if it appears faulty and this will incur a penalty.

You must submit your code (all .java files, **as a zip file**) on Canvas no later than Friday the 26th of November at 4pm.

Plagiarism
Work which is submitted for assessment must be your own work. All students should note that the University has a formal policy on plagiarism which can be found at:
https://www.stir.ac.uk/about/professional-services/student-academic-and-corporate-services/academic-registry/academic-policy-and-practice/quality-handbook/assessment-and-academic-misconduct/#eight Plagiarism means presenting the work of others as though it were your own. The University takes a very serious view of plagiarism, and the penalties can be severe (ranging from a reduced grade in the assessment, through a fail

for the module, to expulsion from the University for more serious or repeated offences). Specific guidance in relation to Computing Science assignments may be found in the Computing Science Student Handbook. We check submissions carefully for evidence of plagiarism, and pursue those cases we find.

Late submission

If you cannot meet the assignment hand-in deadline and have good cause, please see the module coordinator to explain your situation and ask for an extension. Coursework will be accepted up to seven days after the hand-in deadline (or expiry of any agreed extension) but the mark will be lowered by three marks per day or part thereof. After seven days the work will be deemed a non-submission and will receive an X.