Family Name:..... **Other Names:**.....

Student ID:..... **Signature**.....

Introduction to Data Structures and Algorithms (COMP 103): Final

2020, July 22

Instructions

- Time allowed: **4 Hours**
- Starts: Wednesday July 22, 080:00 AM in China (12:00 PM NZ time)
- Finishes: Wednesday July 22, 12:00 PM in China (16:00 PM NZ time). **Late submissions will receive Zero**
- Attempt **all** the questions.
- The examination will be marked out of 110 marks.
- If you think some question is unclear, ask for clarification through WeChat account: *nekooei1984*
- Brief documentation is at the end of the examination script.
- You must submit your work in **ONE** PDF (*COMP103.pdf*) file into the online submission system.

Questions:

1. Properties of Collections [16]
2. Lists, Maps, and Comparable [24]
3. Complexity: Big-O costs [13]
4. Simulation with Collections [27]
5. Traversing General Trees [20]
6. Traversing Graphs [10]

SPARE PAGE FOR EXTRA ANSWERS

Cross out rough working that you do not want marked.
Specify the question number for work that you do want marked.

Question 1. Properties of Collections **[16 marks]**

For these questions, circle "true" or "false" for each property. (**Note** some properties may be true for more than one type.)

(a) **[4 marks]** For a Queue, state whether each property is true or false.

true/false	:	The first item added is the last item to be removed
true/false	:	Items can be added at any specified position
true/false	:	The collection can have duplicate items
true/false	:	The last item added is the last item to be removed

(b) **[4 marks]** For a Set, state whether each property is true or false.

true/false	:	The order of values in the collection is important
true/false	:	Items must be added with a key
true/false	:	The time an item is added affects when it is removed
true/false	:	The collection can have duplicate items

(c) **[4 marks]** For a Map, state whether each property is true or false.

true/false	:	The order of items in the collection can be changed
true/false	:	The time an item is added does not affect when it is removed
true/false	:	The order of values in the collection is important
true/false	:	The collection can same values with different keys

(d) **[4 marks]** For a TreeSet, state whether each property is true or false.

true/false	:	Items are stored in a sorted order
true/false	:	Items must have a natural ordering to be stored in the collection.
true/false	:	Items must implement consistent hashCode and equals methods
true/false	:	Adding an item to a TreeSet is less efficient than adding to a HashSet

Question 2. Lists, Maps, and Comparable **[24 marks]**

Suppose you are writing part of a program for an University. The program keeps track of students who are in courses. The program also keeps track of students who have passed courses.

The program has an allStudent field that contains a Map containing objects with information about each Student known to the system. The key of the allStudents map is the studentID (eg 1813446320. Each Student object contains a name and a set of courses.

```
private Map<Integer, Student> allStudents;
```

The program also has an allCourses field containing a List of Courses.

```
private List<Course> allCourses;
```

Each Course has fields storing

- a List of RegStudents *i.e.,* students who were registered to the course;
- the number of students passing *i.e.,* students who have passed their course;

Documentation of the methods of the Course and Student classes:

```
public class Course {
    public List<RegStudent> getRegStudents() // returns the List of registered students in the course
    public void setNumberPassing(int num)    // sets the number of students who have passed
    public int getNumbersPassing()           // returns the number of students who have passed
}
```

```
public class Student {
    public String getName()                  // returns the name of the student
    public Set<String> getCourses()          // returns a set of Courses
}
```

Implementation of the RegStudent class:

```
public class RegStudent{
    private final int studentID;             // unique student ID (key for the allStudents Map)
    private boolean isPassed = false;        // whether the student has passed the course or not

    public RegStudent(int studentID) {
        this.studentID = studentID;
    }

    public int getStudentID() {return studentID;}

    public boolean hasPassed() {return isPassed;}

    public void recordPassed() {isPassed = true;}

}
```

(Question 2 continued on next page)

(Question 2 continued)

(a) **[7 marks]** Complete the following updateAllNumbersPassing() method. For each Course in the allCourses field, it should count the number of registered students passing each course, store the number in the course.

```
public void updateAllNumbersPassing(){




}
```

(b) **[7 marks]** Complete the following studentsOfCourse(...) method which should return a collection of the studentIDs of everyone who is in the course but has NOT passed it.

```
public Set<Integer> studentsOfCourse(Course course){




}
```

(Question 2 continued on next page)

(Question 2 continued)

(c) **[10 marks]** It has been decided that the RegStudent class needs to implement the Comparable interface (see documentation sheet). The natural order of the class should be by studentID, from lowest to highest. Provide the code that does this.

The revised RegStudent class should work correctly with both TreeSets and HashSets: *i.e.*, hash code, equals and comparisons should be consistent.

```
public class RegStudent implements Comparable {

    private final int studentID;        // studentID cannot change once set
    private boolean isPassed = false; // this initial value can be changed later
    public RegStudent(int studentID) {
        this.studentID = studentID;
    }
    public int getStudentID() {return studentID;}

    public boolean hasPassed() {return isPassed;}

    public void recordPassed() {isPassed = true;}

    public int compareTo(RegStudent other) {









    }
    public boolean equals(Object other) {







    }
    public int hashCode() {



    }

}
```

Question 3. Complexity: Big-O costs **[13 marks]**

For each question below, work out the cost (in Big-O notation) by

- working out the cost of performing each line once.
- working out the number of times each line will be performed.
- computing the total cost.

(a) **[4 marks]** What are the Big-O costs of the fragments of code below? Assume the size of the list is n.

```
for (int i=0; i<list.size(); i++) {
    int idx = (int)(Math.random)*(i+1))        // cost= O(    ) times=
    if(idx != i){                              // cost= O(    ) times=
        int x = list.remove(idx);              // cost= O(    ) times=
        list.add(i, x);                        // cost= O(    ) times=
    }
}
// Total Cost = O(      )
```

(b) **[4 marks]** What are the Big-O costs of the fragments of code below. Assume the size of the list is n.

```
List<Double> ans = new ArrayList<Double>();    // cost= O(    ) times=

for (Double num : list) {
    for (int j = 0; j <= list.size(); j++) {
        if(num==list.get(j)) {                 // cost= O(    ) times=
            ans.add(j, num);                   // cost= O(    ) times=
        }
    }
}
 // Total Cost = O(      )
```

(Question 3 continued on next page)

(Question 3 continued)

(c) **[5 marks]** A program has a List to store all the names in a phone book.

When the List has 1,000,000 names, the program takes 12 microseconds to find a name in the list.

If the List had 8,000,000 names, how long would you expect the program to find a name in the set? Explain why.

Question 4. Simulation with Collections **[27 marks]**

Suppose you are writing a program to simulate students placing online orders on a University canteen. The canteen has multiple restaurants and each restaurant has it own queue.

At each timestep, the program

- decides whether to create a new order, and if so adds the new order to the back of the queue of the correct restaurant.
- advances the preparation of the order at the front of each queue by one time "tick".
- removes any order that has now completed preparation from its queue and notifies the student that the order is ready.

You are to write the addOrder and advanceAllOrders methods.

The CanteenSimulation class has the following field, constructor and run() method.

```
public class CanteenSimulation{
    private Map<String, Queue<Order>> allRestaurants;

    public CanteenSimulation(){
        allRestaurantQueues = new HashMap<String, Queue<Order>>();
    }

    public void run (){
        int time = 0;
        while (true){
            time++;
            if (Math.random()<0.05) {
                addOrder(new Order(time)); // subquestion (a)
            }
            advanceAllOrders();              // subquestion (b)
        }
    }
}
```

The Order class has the following constructor and methods:

```
Order class:
    public Order(int time);
    public void advanceOrderByTick();
    public boolean completedPreparation();
    public String getRestaurant();
    public void notifyStudent();
```

Hint: sketch a diagram of the content of allRestaurants.

(Question 4 continued on next page)

(Question 4 continued)

(a) **[10 marks]** Complete the following addOrder(Order o) method which should find out which restaurant the order is for, and then add the order to the queue for that restaurant, creating the queue if there isn't one already.

```
public void addOrder(Order o){




}
```

(b) **[12 marks]** Complete the following advanceAllOrders() method which should

- advance all the orders that are at the head of their queue.
- remove any orders that have been completed

```
public void advanceAllOrders(){




}
```

(Question 4 continued on next page)

SPARE PAGE FOR EXTRA ANSWERS

Cross out rough working that you do not want marked.
Specify the question number for work that you do want marked.

(Question 4 continued)

(c) **[5 marks]** The canteen wants to introduce a "preferred students" system, where a student who has a better GPA can get higher priority in the order queue and therefore get their food faster.

Describe below, using bullet points, what changes would be needed in the program for it to implement this system.

Question 5. Traversing General Trees **[30 marks]**

This question is about a management system for course records of XMUT courses. The course records are organised in a tree structure. The items in GTNodes are files objects.

The File class implements a toString() method that returns a description of the file (e.g. `"Project:2020"` for the results of projects in 2020, or `Exam:2019` for the exam questions in 2020)

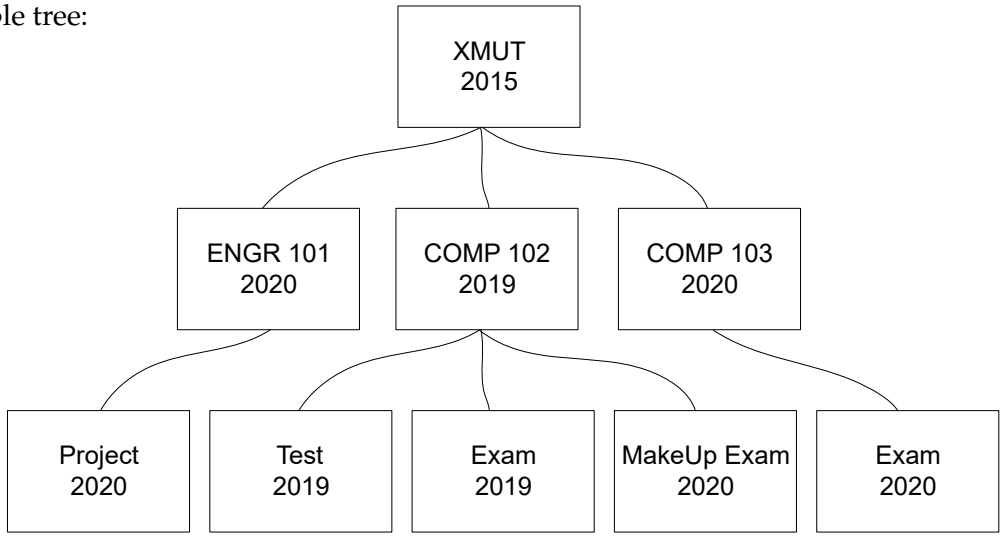
Note, this version of GTNode is **not** Iterable: to iterate through the children of a node, use:

```
for ( int i=0; i<node.numChildren(); i++){... node.getChild(i) ...}
```

```
class GTNode<E>:
    public GTNode(E item);                                  // constructor
    public E getItem();                                     // return item in the node
    public int numChildren();                               // return number of children of the node
    public void addChild(GTNode<E> child);                  // add a child
    public void addChild(int pos, GTNode<E> child);         // add a child at a specific position
    public GTNode<E> getChild(int i);                       // return i'th child
    public void removeChild(int i);                         // remove i'th child
```

```
class CourseRecord:
    public CourseRecord(String name, int year)  // returns a course record with the specified name and date
    public int getYear()                        // returns the year of the course record
    public String getName()                     // returns the name of the course record
    public String toString()                    // returns a String describing the course record,
                                                // eg "Exam:2020" for exam questions of 2020
```

An example tree:



(Question 5 continued on next page)

(Question 5 continued)

(a) **[9 marks]** Complete the following printCourseRecords(...) method which is given the root node of the tree, and should print out all the course records in the tree, using indentation to show the structure.

For example, the tree on the previous page should be printed as:

```
XMUT
   ENGR 101
      Project:2020
   COMP 102
      Test:2019
      Exam:2019
      MakeUp Exam:2020
   COMP 103
      Exam:2020
```

Note: You should print the leaves **with** years and should print other nodes **without** years.

Hint: You can create a recursive function with more arguments.

```
public void printCourseRecords(GTNode<CourseRecord> node){




}
```

(b) **[2 marks]** what is the name for the tree traversal you used for printFileTree

(Question 5 continued on next page)

(Question 5 continued)

(c) **[9 marks]** One action in the course record management system is to remove certain course records from the system. Complete the removeUnderCourseRecord(...) method which is given the root of a tree and a course record name year. It should remove from the tree all the course records in any subtree under a course record of the specified name.

For the example tree on the previous page,

- removeUnderCourseRecord(root,"COMP103", 2020) should remove the Exam:2020 because its is under the COMP103:2020.
- removeUnderCourseRecord(root, "XMUT", 2015) should remove all the course records except the XMUT at the root.

```
public void removeUnderCourseRecord(GTNode<CourseRecord> node, String name, int year){




}
```

Question 6. Traversing Graphs **[10 marks]**

You are writing a program to keep track of people who are infected with COVID-19.

Your program stores information about all people in a region in List of Person objects.

```
private List<Person> allPeople;    // all people in a region
```

Each Person object contains a Set of persons it is connected to directly, and Person is Iterable so that you can use a foreach loop to iterate through the person's direct neighbours.

The Person class has a method called isInfected, which is true if a COVID test for the person was positive. Otherwise, it is false.

You do not need to know any of the other fields or methods of the Person class.

(a) **[5 marks]** Complete the following findPotenialInfected method. It should return a Set of the Persons that are directly or indirectly connected to infectedPerson whose test result is positive.

```
public Set<Person> findPotenialInfected(Person infectedPerson){




}
```

(Question 6 continued on next page)

(Question 6 continued)

(b) **[5 marks]** Complete the following inDanger method which returns true if a Person is connected to an infected person, directly or indirectly, in the network.

```
public boolean inDanger(Person p){
    Set<Person> visited = new HashSet<Person>();
    return inDanger(p,  visited );
}

public boolean inDanger(Person p,  Set<Person> visited){









}
```

* * * * * * * * * * * * * * *

SPARE PAGE FOR EXTRA ANSWERS

Cross out rough working that you do not want marked.
Specify the question number for work that you do want marked.

Documentation for COMP 103 Exam

Brief, simplified specifications of some relevant Java collection types and classes.

Note: *E* stands for the type of the item in the collection.

```
interface Collection<E>
    public boolean isEmpty()                    // cost: O(1) for standard collection classes
    public int size()                           // cost: O(1) for standard collection classes
    public void clear()
    public boolean add(E item)
    public boolean contains(Object item)
    public boolean remove(Object element)
```

```
interface List<E> extends Collection<E>
    // Implementations: ArrayList
    public boolean isEmpty()
    public int size()
    public void clear()
    public E get(int index)                     // cost: O(1)
    public E set(int index, E element)          // cost: O(1)
    public boolean contains(Object item)        // cost: O(n)
    public void add(int index, E element)       // cost: O(n) (unless index close to end.)
    public E remove(int index)                  // cost: O(n) (unless index close to end.)
    public boolean remove(Object element)       // cost: O(n)
```

```
interface Set extends Collection<E>
    // Implementations: HashSet, TreeSet
    public boolean isEmpty()
    public int size()
    public void clear()
    public boolean add(E item)                  // cost: O(1)       for HashSet
                                                //       O(log(n))  for TreeSet
    public boolean contains(Object item)        // cost: O(1)       for HashSet
                                                //       O(log(n))  for TreeSet
    public boolean remove(Object element)       // cost: O(1)       for HashSet
                                                //       O(log(n))  for TreeSet
```

```
class Stack<E> implements Collection<E>
    public boolean isEmpty()
    public int size()
    pubic void clear()
    public E peek()                 // cost: O(1)
    public E pop()                  // cost: O(1)
    public E push(E element)        // cost: O(1)
    // (peek and pop return null if the queue is empty)
```

```
interface Queue<E> extends Collection<E>
   // Implementations: ArrayDeque, LinkedList, PriorityQueue
   public boolean isEmpty()
   public int size()
   public void clear()
   public E peek()                    // cost: O(1)      for ArrayDeque, LinkedList
                                      //       O(1)      for PriorityQueue
   public E poll()                    // cost: O(1)      for ArrayDeque, LinkedList
                                      //       O(log(n)) for PriorityQueue
   public boolean offer(E element)    // cost: O(1)      for ArrayDeque, LinkedList
                                      //       O(log(n)) for PriorityQueue
   // (peek and poll return null if the queue is empty)
```

```
interface Map<K, V>
   // Implementations: HashMap, TreeMap
   public V get(K key)                // cost: O(1)      for HashMap
                                      //       O(log(n)) for TreeMap
   public V put(K key, V value)       // cost: O(1)      for HashMap
                                      //       O(log(n)) for TreeMap
   public V remove(K key)             // cost: O(1)      for HashMap
                                      //       O(log(n)) for TreeMap
   public boolean containsKey(K key)  // cost: O(1)      for HashMap
                                      //       O(log(n)) for TreeMap
   public Set<K> keySet()             // cost: O(1)
   public Collection<V> values()      // cost: O(1)
   // get returns null if key not present; put & remove return the old value, (if any)
```

```
class Collections
   public void sort(List<E> list);                          // cost = O(n log(n)) in general
                                                            //        O(n)        almost sorted
   public void sort(List<E> list, (E e1, E e2)->{..});      // cost = O(n log(n)) in general
                                                            //        O(n)        almost sorted
   public void swap(List<E> list, int i, int j);            // cost = O(1)
   public void reverse(List<E> list);                       // cost = O(n)
   public void shuffle(List<E> list);                       // cost = O(n)
```

```
interface Comparable<E>          // Items can be compared for sorting or a priority queue.
   public int compareTo(E other); // Comparable objects must have a compareTo method:
    // returns     -ve if this comes before other;
    //             +ve if this comes after other,
    //              0  if this and other are the same
    // Note: The String class is Comparable, and has this method
```

```
interface Iterable<E>            // Can use a foreach loop on these items
   public Iterator<E> iterator(); // Iterable objects must have an iterator method:
```

```
Integer and Double constants:
   Integer.MAX_VALUE; Integer.MIN_VALUE;
   Double.MAX_VALUE; Double.NaN; Double.POSITIVE_INFINITY; Double.NEGATIVE_INFINITY;
```