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

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

COMP 103: Makeup Exam

8 September, 2019

Instructions

- Time allowed: **2 Hours** Attempt ALL Questions.
- The examination will be marked out of 100 marks.
- This exam contributes 76% of your final grade.
- Brief Documentation is at the end of the examination script
- Answer in the appropriate boxes if possible — if you write your answer elsewhere, make it clear where your answer can be found.
- There are spare pages for your working and your answers in this examination, but you may ask for additional paper if you need it.
- If you think a question is unclear, ask for clarification.
- You may use Chinese-English translation dictionaries, and calculators without a full set of alphabetic keys.

Questions:

1. Collection Types [14]
2. Lists, Maps, and Sorting [20]
3. Complexity, Big-O costs [16]
4. Simulation with Collections [20]
5. Traversing General Trees [20]
6. Traversing Graphs [10]

Question 1. Collection Types **[14 marks]**

(a) **[2 marks]** Which type of collection has all the following properties?

- Duplicates are NOT allowed,
- the elements are unordered,

(b) **[4 marks]** What are the key properties of the Stack type?

(c) **[4 marks]** Suppose you are writing a program to plan the treatments of patients in a hospital. Patients can have different priorities, which are assigned when they arrive at the emergency room.

- What collection type would you use to store the patients and their order in the hospital?
- Justify your choice.

(d) **[4 marks]** Suppose you are writing a program to manage requests for an on-line shopping website.

- Why would it be a bad idea to store the requests in a Stack?
- What would be a better Collection type?

Question 2. Lists, Maps, and Sorting **[20 marks]**

In this question, you will be writing part of a program that keeps track of all the items in different shops in a shopping mall.

Each Shop has a list of the names of the items in the shop.

Here are the methods of the Shop class:

```
public class Shop {
    public List<String> getItems() //Returns list of names of all items in the shop
}
```

The program has an allItemInfo field that contains a Map containing ItemInfo objects that contain information about each type of item: weight, color, and price.

```
    private Map<String, ItemInfo> allItemInfo;
```

- The key of the allItemInfo map is the name of the item,
- the values of the map are ItemInfo objects that contain information about items.

Here are the methods of the ItemInfo class:

```
public class ItemInfo {
    public String getColor()       // returns the color of this item
    public double getWeight()      // returns the weight of this item
    public double getPrice()       // returns the price of this item
}
```

(Question 2 continued on next page)

(Question 2 continued)

(a) **[10 marks]** Complete the following computeTotalPriceShop method which should return the total price of all the items in a given Shop.

Hint:

- step through each item in the Shop,
  - look up the item in allItemInfo,
  - find the price of the item, and add it to the total.
- Finally, return the total price.

```
public double computeTotalPriceShop (Shop shop) {









}
```

(Question 2 continued on next page)

(Question 2 continued)

(b) **[10 marks]** The program has an allShops field containing a Set of all the Shops in the store:

```
private Set<Shop> allShops;
```

Complete the following listWeights method which should return a List of weights of all the different items in all the shops in the shopping mall.

Hint:

- Create an empty List of Weights for the answer
  - Step through each Shop in the shopping mall
  - Step through each item in the Shop,
- Find the Weight of the item (using allItemInfo), and add it to the answer.
- Finally, return the answer.

```
public List <Double> listExpireDates () {

}
```

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 3. Complexity: Big-O costs **[16 marks]**

(a) **[12 marks]** For each fragment, 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.

This fragment of code below operate on the elements of a List. Assume the size of the list is n.

```
for (int k = 0; k < list.size(); k++) {
    for (int j = 0; j < list.size(); j++) {
        UI.println(list.get(j));          // cost= O(      ) times=
    }
}                                         // Total Cost = O(      )
```

In this fragment of code below, allPatients is a set of Patient.

```
Queue<Patient> pq = new PriorityQueue<String>();
for (int i = 0; i<allPatients.size(); i++){
    pq.offer(allPatients.get(i));         // cost= O(      ) times=
}
while (!pq.isEmpty()){                    // cost= O(      ) times=
    Patient item = pq.poll();             // cost= O(      ) times=
}
                                          // Total Cost = O(      )
```

(b) **[4 marks]** A program uses an ArrayList to store all the products in a shop.

When the List has 1000 items, the program takes 5 nanoseconds to remove the first item from the list.

If the List had 10,000 items, how long would you expect the program to take to remove the first word? Explain why.

Question 4. Simulation with Collections **[20 marks]**

Suppose you are writing a program to simulate passengers going through an airport passport control. Each officer has a separate queue of passengers. When a passenger arrives at the area, they join the shortest queue.

At each timestep, the program

- decides whether a passenger arrives, and adds them to the back of the longest queue.
- advances the checking of the passenger at the front of each queue by one "tick".
- any traveller who has now finished is removed from the queue.

You are to write the addPassenger and advanceAllChecking methods.

Hint: Sketch a diagram of the content of the allQueues field.

The Passenger class has the following constructor and methods:

```
Passenger class:
    public Passenger( int time );                 // make a new passenger, recording arrival time
    public void advanceCheckingByTick();  // removes 1 tick from remaining checking time
    public boolean completedChecking();   // true if passenger has completed checking passport
    public int getArrivalTime();                  // returns time tick when passenger arrived
```

The PassportCheckingSimulation class has the following field, constructor and run method.

```
public class PassportCheckingSimulation{
    private Set<Queue<Passenger>> allQueues;

    public SecuritySimulation(){
        allQueues = new HashSet<Queue<Passenger>>();
        for( int i = 0; i<5; i++){                          // initialise queues
            allQueues.add(new ArrayDeque<Passenger>());
        }
    }

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

(Question 4 continued on next page)

(Question 4 continued)

(a) **[10 marks]** Complete the following addPassenger method which should find the longest queue in allQueues and then add the customer to that queue.

```
public void addPassenger(Passenger p){




}
```

(b) **[10 marks]** Complete the following advanceAllChecking method which should

- advance the checking of each passenger at the head of a queue in allQueues by one tick, (check for empty queues!)
- Remove from the queue any passenger who has completed their checking.

```
public void advanceAllChecking(){




}
```

Question 5. Traversing General Trees **[20 marks]**

This question uses a general tree implemented using GTNode to represent expression for a calculator. The item in a GTNode is a String that is either an operator or a number.

Note, this version of GTNode is Iterable: You can use

```
for ( GTNode child : node ){... child ...}|
```

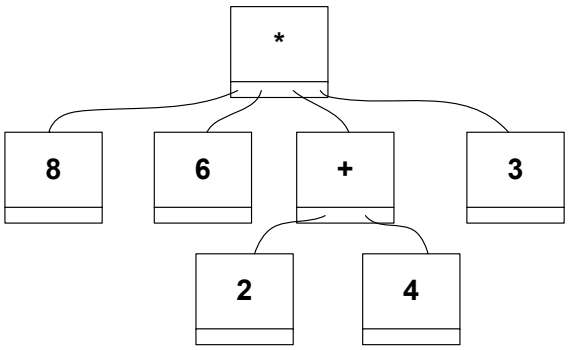
to iterate through the children of a node.

```
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
```

(a) **[9 marks]** Complete the following printExpression method which should print out an expression in "Cambridge-Polish notation with brackets". For example, the expression tree below should be printed as:

should be printed as: `( * 8 6 ( + 2 4 ) 3 )`

For example, the expression tree



- To print a leaf node (which has no children), it just prints the item in the node.
- To print a subtree, it prints an open bracket, then the operator, then the children, then a closing bracket.

```
public void printExpression (GTNode<String> node){




}
```

(Question 5 continued on next page)

(Question 5 continued)

(b) **[11 marks]** Complete the following evaluate method which evaluates an expression:

- To evaluate a leaf node, it just returns the value in the node.
  Hint: You can turn a String into a double using Double.valueOf(...)
- To evaluate a subtree, it should evaluate its children, then apply the operator to the values of the children and return the result.
- It needs to support the operators: + and *

```
public double evaluate(GTNode<String> node) {




}
```

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

You are writing a program to check bus connections in Xiamen. Your program needs a method to check if bus stop in XMUT is connected to other bus stops in the city.

Your program stores information about all the bus stops in a List of BusStop objects:

```
private List<BusStop> allStops; // List of all bus stops in Xiamen
```

Complete the following isConnectedToXMUT method which should find if BusStop a is connected to XMUT.

Note:

- Each BusStop object contains a Set of its neighbour Bus Stops that have a direct bus connection.
- BusStop is Iterable so that you can use a foreach loop to iterate through the neighbours of a BusStop:

  ```
  for(BusStop neighbour : stop) { ...
  ```

- isConnectedToXMUT does NOT find the shortest path
- You do not need to know any of the other fields or methods of the BusStop class.

```
public boolean isConnectedToXMUT(BusStop a){
        Set<Stop> visited = new HashSet<BusStop>();
        boolean ans = isConnectedToXMUT(a, visited);
        return ans;
}
/** Adds to visited stops connected to a */
public boolean isConnectedToXMUT(BusStop a, Set<BusStop> visited){
    if ( a.equals(XMUT) ) { return true; }









}
```

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

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.
                                 // The String class is Comparable, and has this method
   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
```

```
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;
```