TE WHARE WĀNANGA O TE ŪPOKO O TE IKA A MĀUI

VICTORIA
UNIVERSITY OF WELLINGTON

EXAMINATIONS – 2018

TRIMESTER 3

COMP 103
INTRODUCTION TO DATA STRUCTURES AND ALGORITHMS

Time Allowed: TWO HOURS

CLOSED BOOK ******** **WITH SOLUTIONS** **********

Permitted materials: Silent non-programmable calculators or silent programmable calculators with their memories cleared are permitted in this examination.

Printed foreign language–English dictionaries are permitted.

No other material is permitted.

Instructions: Attempt ALL Questions.
The examination will be marked out of 120 marks.
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.

Questions:

1. Collection Types	[16]
2. Lists, Maps, and Sorting	[28]
3. Complexity, Big-O costs	[24]
4. Simulation with Collections	[22]
5. Traversing General Trees	[20]
6. Traversing Graphs	[10]

Date of revision: February 21, 2019

Question 1. Collection Types **[16 marks]**

(a) **[4 marks]** What is the key property of the Java Stack type that distinguishes it from the more general Collection type?

Elements can only be added or removed from one end

(b) **[4 marks]** Suppose you are writing a program to keep track of your reviews of books you have read.

- What collection type would you use to store the reviews?
- Justify your choice.

A map of book to review, as you can quickly check what your review is based on the book

(c) **[4 marks]** Suppose you are writing a program to keep track of all the items a store has in stock.

- Why would it be a bad idea to store this information in a Set?
- What would be a better Collection type?

A set can't contain duplicates so can't have multiple of an item.
A map would let you keep track of quantity better.

(Question 1 continued on next page)

(Question 1 continued)

(d) **[4 marks]** Suppose you are writing a program to keep track of incoming maintenance requests for a building management company. A request might be something like "the lightbulb on level 3 is out" or "a group of people are stuck in the Cotton elevator". You have decided to use a Queue.

- What might go wrong?
- What is a simple way to solve the problem?

Urgent requests would have to wait for less urgent requests to be cleared.
Fix it by changing to a priority queue.

Question 2. Lists, Maps, and Sorting **[28 marks]**

Suppose you are writing part of a video game that handles the rewards from fights with monsters. Every Monster is worth points, and has an item which they drop if they are defeated.

The program has an allFights field containing a List of Fights.

- Each Fight has a list of Monsters that appear in the fight;
- Each Monster has an itemId which is the name of the item they have.

The program also has an allItems field that contains a Map containing information about each of the Items. The key of the allItems map is the itemId (a String), and the values are the actual Item objects.

```
    private List<Fight> allFights;
    private Map<String, Item> allItems;
```

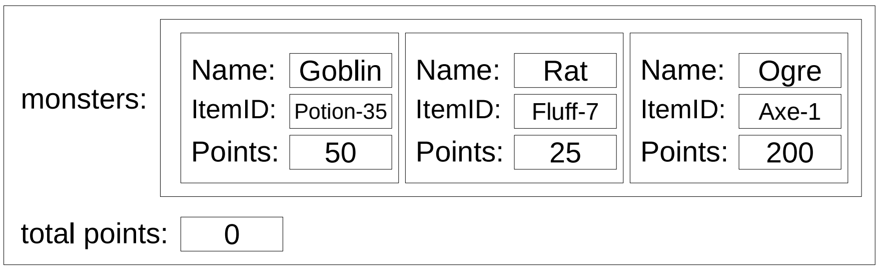
Here are descriptions of the methods of the Fight, Monster, and Item classes:

```
public class Fight {
    public List<Monster> getMonsters()      // returns the List of Monsters in the fight
    public void setTotalPoints(int total)   // records how many points the fight is worth
}
```

```
public class Monster {
    public String getName()         // returns the name of this monster
    public String getItemId()       // returns the itemId of the monster's item
    public int    getPoints()       // returns how many points the monster is worth
}
```

```
public class Item {
    public String getID()               // returns the itemId of the item
}
```

Here is a sketch of one possible Fight in allFights. The Fight contains three Monsters: a goblin, a rat, and an ogre.



(Question 2 continued on next page)

(Question 2 continued)

(a) **[10 marks]** Complete the following computePointTotals method. For each Fight in the allFights field, it should compute the total points gained in the encounter, based on the points value of each monster present in the encounter, and then store the total in the Fight.

```
public void computePointTotals(){
    for (Fight fight : allFights ){
        int total = 0;
        for (Monster monster : fight .getMonsters()){
            total = total + monster.getPoints();
        }
        fight . setTotal ( total );
    }
}
```

(b) **[10 marks]** Complete the following calculateDroppedItems method which should return a List of Items that will be dropped by the monsters after a fight.

Note: You will need to look up the actual Item using the itemID

```
public List<Item> calculateDroppedItems(Fight fight){
    List<Item> items = new ArrayList<Item>();
    for (Monster monster : fight .getMonsters()){
        String itemID = monster.getItemID();
        Item item = allItems .get(itemID);
        items.add(item);
    }
    return items;
}
```

(Question 2 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 2 continued)

(c) **[8 marks]** Complete the following sortMonsterList method that should sort the List of Monsters in a Fight. Monsters with higher point values come before Monsters with lower point values. Monsters with the same point value should be sorted by their itemId in standard String order.
Note that Monsters are not Comparable. You may use a lambda or define a compareMonsters method.
Strings are comparable, and can be compared with the compareTo method.

```
    public void sortMonsterList (Fight fight){
        Collections.sort( fight.getMonsters(), (Monster m1, Monster m2) ->{
                if (m1.getPoints() > m2.getPoints()) {return -1;}
                if (m1.getPoints() < m2.getPoints()) {return 1;}
                return m1.getItemID().compareTo(m2.getItemID());
        });
Version 2:
        Collections.sort(encounter.getMonsters(), this::compareMonsters));
    }

    public int compareMonsters(Monster m1, Monster m2){
        if (m1.getPoints()==m2.getPoints()){
            return it1.getItemID().compareTo(it2.getItemID());
        }
        return (m2.getPoints()-m1.getPoints());
    }
```

Question 3. Complexity, Big-O costs **[24 marks]**

(a) **[10 marks]** The two fragments of code below perform operations on the elements of a List. Assume the size of the list is n.

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.

```
for (int k = 0;  k < list.size() / 2;  k++) {
        int other = list.size() - 1 - k;       // cost= O( 1 )  times= n/2
        int temp = list.get(k);                // cost= O( 1 )  times= n/2
        list.set(k, list.get(other));          // cost= O( 1 )  times= n/2
        list.set(other, temp);                 // cost= O( 1 )  times= n/2
}
                                               // Total Cost = O( n )
```

```
for (int k = 0;  k < list.size() - 1;  k++) {
    int minIndex = k;                                       // cost= O( 1 )  times= n

    for (int j = k+1; j < list.size();  j++) {
        if ( list.get(j) < list.get(minIndex)) {            // cost= O( 1 )  times= n^2
            minIndex = j;                                   // cost= O( 1 )  times= n^2
        }
    }
    if ( minIndex != k ) {                                  // cost= O( 1 )  times= n
        list.set(k, list.set(minIndex, list.get(k)));       // cost= O( 1 )  times= n
    }
}                                                           // Total Cost = O( n^2 )
```

(Question 3 continued on next page)

(Question 3 continued)

(b) **[10 marks]** Assume that allWords is a Collection containing n Strings. The following code fragment prints the Strings out in alphabetical order, along with how many times they occured. Work out the cost in Big-O notation.

```
Map<String,Integer> map;
map = new TreeMap<String,Integer>();           // cost= O(   1    )    times= 1
for (String word : allWords){
     int count = 1;                            // cost= O(   1    )    times= n
     if (map.containsKey(word)) {              // cost= O( log(n) )    times= n
          count = count + map.get(word);       // cost= O( log(n) )    times= n
     }
     map.put(word, count);                     // cost= O( log(n) )    times= n
}
for ( String word : map.keyset() ){
    int count = map.get(word);                 // cost= O( log(n) )    times= n
    outFile.println(word + " (" + count + ")"); // cost= O(   1    )    times= n
}
                                               // Total Cost = O( n log(n)    )
```

(c) **[4 marks]** A simulation program uses a Queue to store all the incoming data transmission messages that need to be processed.

When the Queue has 10,000 messages, the program takes 12 microseconds to process every message in the queue.

If the Queue had 10,000,000 messages, how long would you expect the program to take to process every message? Explain why.

About 12,000 microseconds = 12 milliseconds
Removing from a queue is an O(1) operation, but we need to do it for every item in the queue, so the entire process is O(n).
So the program will have to process about 1000 times as many items, so it should take about 1000 times longer

Question 4. Simulation with Collections **[22 marks]**

Suppose you are writing a program to simulate customers using the self-checkout machines at a supermarket. Each checkout machine has a separate queue of customers. When a customer arrives at the checkout, they join the shortest available queue. Each customer has a basket that contains a number of items.

At each timestep, the program

- decides whether a customer arrives, and if so, adds them to the back of the shortest queue.
- Each customer at the head of a queue for a checkout-machine processes one item from their basket.
- Any customer with an empty basket has finished, and leaves the queue.

You are to write the addCustomer and advanceAllCheckouts methods.

Hint: sketch a diagram of the content of allCheckouts.

The Customer class has the following constructor and methods:

```
Customer class:
   public Customer(int time, int numItems); // make a new customer, recording arrival time
                                            // and the number of items in their basket
   public void processOneItem();            // processes one item in the customer's basket
   public boolean completedAllItems();      // true if customer has finished all their items
   public int getArrivalTime();             // returns time tick when customer arrived
```

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

```
public class CheckoutSimulation{
   private Set<Queue<Customer>> allCheckouts;

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

   public void run (){
       int time = 0;
       while (true){
            time++;
            if (Math.random()<0.05) {                            // decide if there is a new customer
                 int numItems = (int) Math.ceil(Math.random()*15); // generate number of items
                 addCustomer(new Customer(time, numItems));      // subquestion (a)
            }
            advanceAllCheckouts();                               // subquestion (b)
       }
   }
}
```

(Question 4 continued on next page)

(Question 4 continued)

(a) **[11 marks]** Complete the following addCustomer method which should add the customer to the shortest queue (fewest number of customers) in allCheckouts:

```
public void addCustomer(Customer cust){
    Queue<Customer> minQ = null;
    int min = Integer.MAX_VALUE;
    for (Queue<Customer> queue : allCheckouts){
        if (queue.size () < min) {
            min = queue.size();
            minQ = queue;
        }
    }
    minQ.offer(cust);
}
```

(b) **[11 marks]** Complete the following advanceAllCheckouts method which should

- Process one item of each customer at the head of a queue in allCheckouts.
- dequeue any customer who has completed their checkout

```
public void advanceAllCheckouts(){
    for (Queue<Customer> queue : allCheckouts){
        if (!queue.isEmpty()) {
            Customer c =queue.peek();
            c.processOneItem();
             if (c.completedAllItems()){
                 queue. poll ();
             }
        }
    }
}
```

SPARE PAGE FOR EXTRA ANSWERS

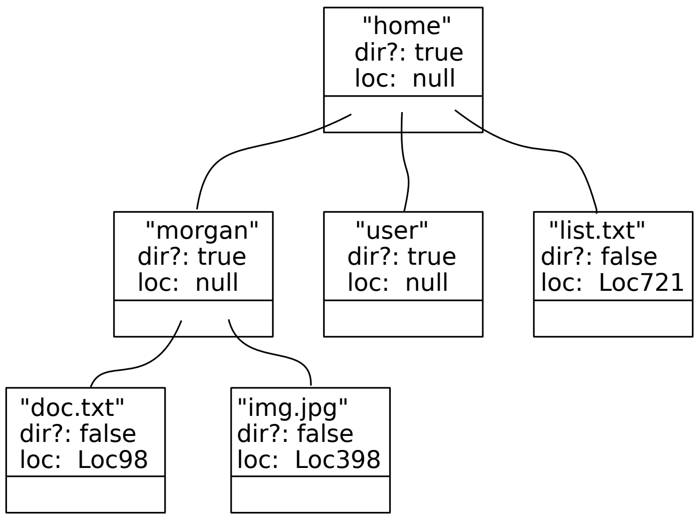
Cross out rough working that you do not want marked.
Specify the question number for work that you do want marked.

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

In the lectures and assignment 6, we used a general tree implemented with GTNode to represent expressions for a calculator with a String in each node.

This question uses a general tree implemented with GTNode to represent the directory (folder) structure of the files and directories on a computer. Each GTNode contains a FileDescriptor which contains the name of the file or directory, a boolean field to record whether it is a directory or not, and the location on the disk where the file contents are stored.

The diagram below shows a little directory structure with three directories and three files.



The methods for the GTNode and FileDescriptor classes are shown below.

Note that this version of GTNode is **not** Iterable: You must use

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

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
  public void addChild(GTNode<E> child);      // add a child
  public GTNode<E> getChild(int i);           // return i'th child
  public void removeChild(int i);             // remove i'th child
```

```
class FileDescriptor
  public FileDescriptor(String name, boolean dir);  // constructor
  public String getName();                          // return item in the node
  public boolean isDir();                           // True if this is a directory
  public DiskLoc getLoc();                          // Location of file contents on disk
```

(Question 5 continued on next page)

(Question 5 continued)

(a) **[10 marks]** The full pathname of a file or a directory is the name of the file or directory, prefixed with all the directory names from the top of the tree down to the file or directory, separated by “/”.
For example, the full path names of all the directories in the figure above are:

```
/home
/home/morgan
/home/morgan/doc.txt
/home/morgan/img.jpg
/home/user
/home/list.txt
```

Complete the following printFullPathNames method which should print out the full pathname of each directory and file in the tree, as in the example above.

Note Each pathname should start with a “/”

```
public void printFileTree (GTNode<FileDescriptor> root){
    printFileTree (root, "");
}

public void printFileTree (GTNode<FileDescriptor> fileNode, String prefix) {
     FileDescriptor  file  = fileNode.getItem();
    UI. println ( prefix  + "/" + file.getName());

    for (int i=0; i<node.numChildren(); i++){
        printFileTree (node.getChild(i),  prefix  + "/" + file.getName());
    }




}
```

(Question 5 continued on next page)

(Question 5 continued)

(b) **[10 marks]** Complete the following searchLoc method which should search a directory structure for a file (not a directory) that matches the supplied filename and should return the location on disk of the file. If there is no file with a matching filename, it should return null.

```
public DiskLoc search(GTNode<FileDescriptor> tree, String filename){
        FileDescriptor  file  = tree.getItem();
        if (! file . isDir ()  && file.getName().equals(filename)) {
            return  file .getLoc();
        }

        for ( int  i=0; i<node.numChildren(); i++){
            FileLoc  loc  = search(node.getChild( i ),  filename );
            if  (loc!=null)  {  return  f;  }
        }

        return  null;
}
```

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

Suppose you are writing a program to sift through your social media connections and figure out how many steps there are between you and famous actor Kevin Bacon.

Your program stores information about all the people in the social network in a Collection of Person objects:

```
    private Collection<Person> allPeople;    // All Persons in the network
```

It also contains a field called kevinBacon that holds the Person object for Kevin Bacon.

Each Person object contains a Set of its friends, and Person is Iterable so that you can use a foreach loop to iterate through the friends of a Person. You do not need to know any of the other fields or methods of the Person class.

(a) **[5 marks]** Complete the following getConnected method which should return true if the given Person is connected to kevinBacon.
Note: It uses a visited Set to keep track of Persons it has visited.

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

public boolean isConnected(Person p, Set<Person> visited){
     if ( p.equals(kevinBacon) ) { return true; }   // or if (p==kevinBacon) ....
     visited .add(p);
     for (Person friend : p){
         if (! visited .contains( friend )) {
             if (isConnected( friend , visited )) return true;
         }
     }
     return false ;




}
```

(Question 6 continued on next page)

(Question 6 continued)

(b) **[5 marks]** There is a common saying that everybody is connected by 6 degrees of separation–you can connect to anybody else on earth by following at most six links. The easiest way to test this is to write a new version of isConnected that takes a person p, a target person and a small integer n (like 6) and returns true only if p is within n links of target (for example, whether you are within 6 degrees of Kevin Bacon).

```
public boolean isConnected(Person p, Person target, int n){
    Set<Person> visited = new HashSet<Person>();
    return isConnected(p, target, n, visited);
}

public boolean isConnected(Person p, Person target, int n, Set<Person> visited) {
    if (n<0)                     {return false;}
    if (visited.contains(p))     {return false;}
    if (p.equals(target))        {return true;}
    visited.add(p);
    for(Person friend : p.getFriends()){
        if (isConnected(friend, target, n-1, visited)) { return true; }
    }
    visited.remove(p);
    return false;
}
```

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 Examination

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)
```

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

```
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:
```

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