VICTORIA UNIVERSITY OF WELLINGTON
TE HERENGA WAKA

EXAMINATIONS – 2022

TRIMESTER 2

COMP 103

INTRODUCTION TO DATA STRUCTURES AND ALGORITHMS

21/10/2022

Time Allowed: TWO HOURS

CLOSED BOOK

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– there are 6 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	**Marks**	
1. Questions about Collections	[18]	
2. Using Maps	[20]	
3. Complexity: Big-O costs	[16]	
4. Binary Trees	[26]	
5. Traversing General Trees	[30]	
6. Traversing Graphs	[10]	
	TOTAL:	

Question 1. Questions about Collections **[18 marks]**

(Terminology: A collection is an object that groups multiple elements into a single unit.)

For these questions, circle "true" or "false" for each property.

(a) **[4 marks]** For a collection which is an implementation of the Stack interface, state whether each property is true or false. (Circle the correct one.)

true/false: The last element added to the collection is the first element to be removed

true/false: Items must have a natural ordering to be stored in the collection

true/false: An element can be removed by specifying its index

true/false: The collection cannot contain duplicate elements

(b) **[4 marks]** For a collection which is an implementation of the List interface, state whether each property is true or false.

true/false: The first item added is the last item to be removed

true/false: The order of items can be reversed

true/false: Two elements can contain the same value

true/false: The elements can always be put in sorted order

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

true/false: The cost of the method get() is independent of the size of the Map

true/false: Two different keys must have different values associated with them

true/false: A foreach loop through the keys steps through them in their natural order.

true/false: Adding a new value associated with a key will delete any previous value associated with the key.

(d) **[2 marks]** For a class to have a "natural ordering" it needs to have a 'compareTo()' method. Which *other* method(s) should be checked for consistency with compareTo?

(e) **[2 marks]** Joe wants to create a FIFO queue of music files where each element has type MusicFile. Fix the line of code below.

Queue myMusicFiles = **new** *Queue*<MusicFile>();

(f) **[2 marks]** A HashSet has complexity $\mathcal{O}(1)$ for the contains method. What is the complexity of add(), and of remove()?

The complexity of the add() method is ..... , and

the complexity of the remove() method is .....

Question 2. Using Maps. **[20 marks]**

Suppose you are writing a program for a website that lets students vote for their class representative ("Class rep"). To vote, users enter the name of their preferred candidate. Your program uses a Map (stored in the votes field) to keep track of all the names that have been entered and the number of votes for each candidate. The candidate names (Strings) are the keys of the map, and the number of votes for the candidates are the values.

```
private Map<String, Integer> votes = new HashMap<String, Integer>();
```

(a) **[7 marks]** Complete the following addVote method that will record another vote in the votes map. The parameter is the name of the candidate. Note that if this is the first vote for a candidate, the name of the candidate will not yet be in the votes map, and must be added.

```
public void addVote(String candidate){




}
```

(Question 2 continued on next page)

(Question 2 continued)

(b) **[13 marks]** We want to know which candidate got the most votes, but of course there might have been a "tie", in which top count was scored by *several* candidates. Complete the following getTopCandidates method that will return a Set containing the names of those who got the highest number of votes.

```
public Set<String> getTopCandidates(){




}
```

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]**

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 (worst-case) costs of the fragments of code below where list is a List of size n.

```
List<Integer> result = new ArrayList<Integer>();      // cost = O(      ),   times =
for (int i=list.size()-1; i>=0; i--) {
        for (int j=list.size()-1; j>=0; j--){
                if ( list.get(i)/ list.get(j) > 5){    // cost = O(      ),   times =
                        result.add(0, list.get(i));    // cost = O(      ),   times =
                }
        }
                                                       // Total Cost = O(          )
}
```

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

```
Set<Integer> set = new TreeSet<Integer>();          // cost = O(      ),   times =
for (int i=list.size()-1; i>=1; i=i/2){
        if( list.get(i) > list.get(i-1)){           // cost = O(      ),   times =
                set.add( list.get(i));              // cost = O(      ),   times =
        }
}
                                                    // Total Cost = O(          )
```

(Question 3 continued on next page)

(Question 3 continued)

(c) **[4 marks]** A car dealer uses a TreeMap to maintain their car database. In this database, each car has a unique identity number which is used as the key and the car's price is the value.

If the databade has 1,000 (approximately 2^{10}) cars with unique identity numbers, the program takes 10 milliseconds to add a new car into the database.

If the dealer had 8,000 cars in their database, how long would you expect the program will take to add a new car to the database? Explain why.

(d) **[4 marks]** A university uses HashMap to store all the students. The key is the Student ID and the value is the student's academic achievement.

When the university has 20,000 students, the program takes 100 nanoseconds to find a student's achievement given the student ID.

If the university had 200,000 students, how long would you expect the program to find a student's achievement given the student ID? Explain why.

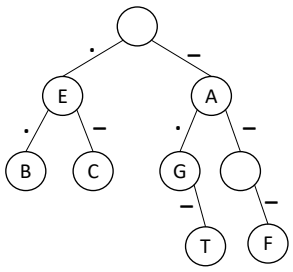
Question 4. Binary Trees **[26 marks]**

Morse code is a method used in telecommunication to encode text characters as standardized sequences of two different signal durations, called *dots (".")* and *dashes ("-")*. Morse code can be presented as a *binary tree*. The root node contains an empty string. Other nodes contain either a letter or an empty string. Starting at this root, or any other internal nodes: moving left along a link signifies a *dot*, while moving right is a *dash*.

The MorseNode class has the following constructor and methods:

```
MorseNode class:
// make a new MorseNode containing a given letter and 2 nodes on the dot and dash sides
public MorseNode(String letter, MorseNode dotNode, MorseNode dashNode);
public String getLetter();                  // gets the letter stored in the current Morse node
public MorseNode getDotNode();              // returns the child on the dot link
public void setDotNode(MorseNode node);     // sets the child on the dot link to the node object
public MorseNode getDashNode();             // returns the child on the dash link
public void setDashNode(MorseNode node);    // sets the child on the dash link to the node object
```

The following figure shows an example of a binary tree for Morse code where empty circles present nodes containing empty strings (i.e., "").



(a) **[13 marks]** Complete the following lookUp() method which is given the root node and a list of "." and "-" (called mCode). The method should find the letter corresponding to the given list. If there is no corresponding letter, the method should print "Not found" and if it is an empty string it should print "empty string".

For example, for the list { "-", ".", "-"}, the method should print "T". For the list { ".", "-", "."}, the method should print "Not found", and for the list {"-", "-"} it should print "empty string".

(Write your answer in the answer box on the following page.)

Hint: you can use list.subList(int start, int end) to extract a sub-list from start (inclusive) to end (exclusive).

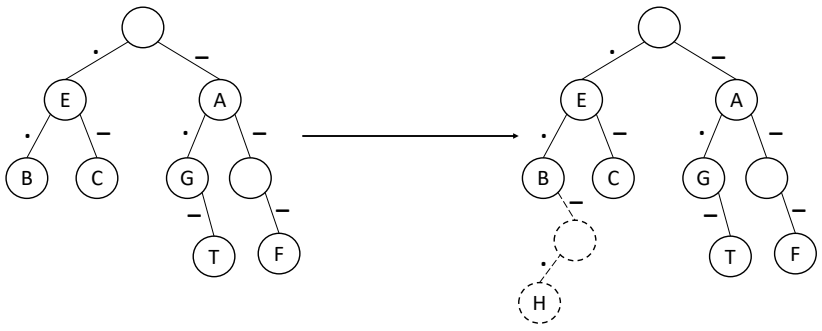
```
public void lookUp(MorseNode node, List<String> mCode){




}
```

(b) **[13 marks]** Complete the following extend() method which is given the root node, a list of "." and "-" (called mCode) and a letter. The method should add a new node containing the given letter. The position of the new node is defined by mCode. Along the path defined by mCode, any missing node should be filled by a node containing an empty string. You can assume that mCode always specifies a new node and all the Strings in mCode are either "." or "-".

For example, calling extend(root, { ".", ".", "-", "."}, "H") results in the following extension:



```
public void extend(MorseNode node, List<String> mCode, String letter){




}
```

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 **[30 marks]**

This question concerns a solitaire card game with cards laid out in a tree structure. The program uses GTNodes containing Card objects.

The Card class implements a toString() method that returns a description of the card (e.g. `"S-1"` for Ace of Spades, or `H-8` for the 8 of Hearts)

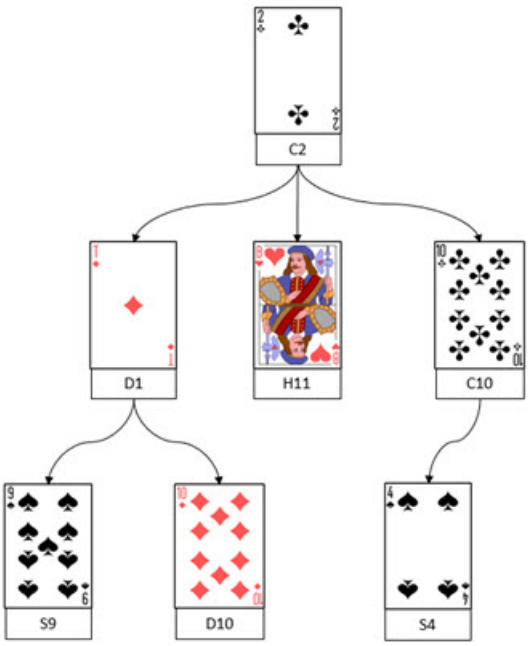
Note, this version of GTNode is **not** Iterable: to iterate through the children of a node, use:

```
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 Card:
public Card(String suit, int value) // returns a card with the specified suit and value
public int getValue()               // returns the numeric value of the card
public String getSuit()             // returns the suit of the card
public String toString()            // returns a String describing the card,
// eg "D-13" for King (13) of Diamonds
```

An example tree:



Note: The suits are shortened "Hearts" to "H" , "Diamonds" to "D", "Clubs" to "C", "Spades" to "S". Values are 1(for ace), 2, 3, $\cdots$, 10, 11 (for jack), 12 (for queen) and 13 (for king).

(Question 5 continued on next page)

(Question 5 continued)

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

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

```
        S-9
        D-10
    D-1
    H-11
        S-4
    C-10
C-2
```

Hint: You can declare and use a recursive function with more arguments.

```
public void printCardTree(GTNode<Card> node){




}
```

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

(Question 5 continued on next page)

(Question 5 continued)

(c) **[9 marks]** One action in the game is to find cards. Complete the findCard(...) method which is given the root of a tree, a suit, and a value. It should return from the tree the node storing the card with the corresponding suit and value.

```
public GTNode<Card> findCard(GTNode<Card> node, String suit, int value) {




}
```

(Question 5 continued on next page)

(Question 5 continued)

(d) **[10 marks]** [HARD] Playing the card game the player can remove cards following some rules. If the player remove a card (C), all of the children of C becomes children of C's parent. If C is the root of the tree nothing is removed from the tree.

For example, in layout above (in part (a)) when C10 is removed correctly, S4 becomes a new child of C2.

Complete the following removeCard(...) method which is passed the root of a tree and correctly removes a card with the corresponding suit and value.

The method returns the node of the removed card, null if no card was removed.

Note: Specific cards can only appear in the tree once. Tip: You can use the method from part (b) to find the node.

```
public GTNode<Card> removeCard(GTNode<Card> node, String suit, int value){




}
```

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

"Airline Inc." is an airline company that have flights between a number of airports. Each airport that can be reached from another airport with a direct flight is considered a "connection" of that airport. There are no one-way connections, so if there is a flight from XXX to YYY then there is also a flight from YYY to XXX.

"Airline Inc." use software which features an Airport class with the following methods:

```
Airport class:
private Set<Airport> connections;
public String getCode();   // returns the code of the Airport (e.g. WEL)
public void addConnection(Airport n);   // add n as a possible destination
public void removeConnection(Airport n);   // remove n as a possible destination
```

Note: you can use a foreach loop to iterate through the connections of an airport:

```
for ( Airport neighbour : airport ){...}
```

(a) **[2 marks]** Complete the following removeConnection(...) method in the Airport class that removes a flight connection between two airports in both directions.

```
public void removeConnection(Airport airpt) {




}
```

(Question 6 continued on next page)

(Question 6 continued)

(b) **[4 marks]** Complete the following findAirport(...) method. The method returns the Airport with the provided airport code, but only if the airport can be reached from the starting airport (sp).

If no airport can be reached with the corresponding code, return null.

Note, airport codes are unique.

```
public Airport findAirport(Airport sp, String code) {
        return findAirport(sp, code, new HashSet<Airport>());
}

public Airport findAirport(Airport airport, String code, Set<Airport> visited) {









}
```

(Question 6 continued on next page)

(Question 6 continued)

(c) **[4 marks]** Complete the following possible(...) method which returns a boolean. It returns true, when there is a route through the airline's network from the starting point (sp) to the destination (dest) using a maximum number (num) of individual flights. Otherwise, it returns false. (Note, the method will also return true if fewer flights than the maximum number connects the airports.)

```
public boolean possible(Airport sp, Airport dest, int num) {
        return possible(sp, dest, num, new HashSet<Airport>());
}

public boolean possible(Airport airport, Airport dest, int num, Set<Airport> visited) {









}
```

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

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;
```