VICTORIA UNIVERSITY OF WELLINGTON
TE HERENGA WAKA

EXAMINATIONS – 2024
TRIMESTER 2 (TEST 2)
FRONT PAGE

COMP 103

SOLUTIONS

2024-10-19

Time allowed: **120 MINUTES**

Permitted materials: **CLOSED BOOK**
You are allowed to use a printed language dictionary during the test. Electronic dictionaries, apps, and other digital resources are not permitted.

Instructions:
Attempt ALL **7** questions
The test will be marked out of a total of **120** marks.
You will be provided with a *concise Java documentation for Collections* and a *brief Java documentation*.
Record your answers in their designated spaces — 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 booklet. If you need additional space, you may request extra white paper sheets from the invigilators.
If you find any question unclear, request clarification from the invigilator.
Assume that all necessary java libraries are already imported.

Question	#1	#2	#3	#4	#5	#6	#7	**Total**
Max Points	20	20	16	26	14	10	14	**120**
Mark								

Question 1: Properties of Collections **[20 marks]**

For questions (a) and (b), specify which of the four collection types – Set, List, Map, Deque – have the given properties. (Your answer may include more than one type).

a) Collection(s) that maintain insertion order of elements: **[4 marks]**

```
Your answer:

List, Deque
```

b) Collection(s) that do not allow `null` elements: **[4 marks]**

```
Your answer:

None of the collections implemented in JCF prohibits null element but, it is
↪  recommended not to use nulls with Queues (and Deques) because null can be the
↪  return type some methods.
```

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

i. ☑ true / ☐ false : Elements are stored in the natural order of their keys.
ii. ☐ true / ☑ false : The time complexity for retrieving an element is O(1).
iii. ☑ true / ☐ false : It allows duplicate values for different keys.

d) Consider the following code snippet using ArrayDeque: **[8 marks]**

```
ArrayDeque<String> aDeque = new ArrayDeque<>();
aDeque.offerLast("A");
aDeque.offerLast("B");
aDeque.offerFirst("C");
aDeque.offerLast("D");
aDeque.pollFirst();
aDeque.offerFirst("E");
```

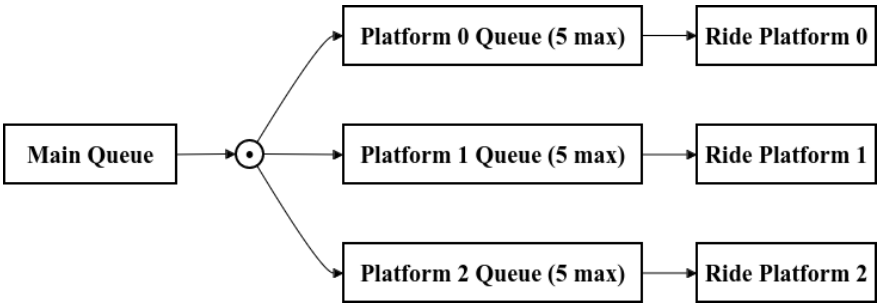
What will be the returned value of executing `aDeque.peekFirst()` and `aDeque.peekLast()` after these operations? And, what will be final state of the deque?

```
Your answer:

aDeque.peekFirst() returns: ->  E

aDeque.peekLast() returns: ->   D

Final state of the deque after executing aDeque.peekFirst() and aDeque.peekLast():

->  E A B D
```

Question 2: Using Collections – Simulation **[20 marks]**

This question is about simulating the queue system for a popular roller coaster ride in a park.

The roller coaster has 3 loading platforms. Each platform has space for a queue of up to 5 people (referred to as visitors). Each platform loads visitors into the roller coaster cars. There is also a main queue area where visitors wait until there is space in one of the platform queues. The following figure shows the roller coaster queue system:



The `Visitor` class that represents individual park visitors has the following methods:

```
public class Visitor {
    public void boardRide(); // Visitor boards the ride
    public boolean hasRidden(); // Has the visitor completed the ride?
    public static Visitor newVisitor(); // Returns either null or a new visitor
}
```

The `RideSimulation` class has the following fields and methods:

```
public class RideSimulation {
    private static final int NUM_PLATFORMS = 3;
    private static final int PLATFORM_SIZE = 5;
    private List<Deque<Visitor>> pfQueues; // List of platform queues
    private Deque<Visitor> mainQueue;

    public void initialiseQueues(){...} // To be completed

    public void runSimulation(){...} // To be completed
}
```

Question 2 (continued)

a) Complete the `initialiseQueues()` method which initialises `mainQueue` and the three platform queues in `pfQueues` list. All the queues should exist but should be empty. **[8 marks]**

```
public void initialiseQueues() {
    // YOUR CODE HERE
    // Initialise the main queue as an ArrayDeque
    mainQueue = new ArrayDeque<>();



    // Initialise the platform queues as ArrayDeques
    pfQueues = new ArrayList<>();
    for (int i = 0; i < NUM_PLATFORMS; i++) {
        pfQueues.add(new ArrayDeque<Visitor>());
    }



}
```

b) Complete the `runSimulation()` method to run the simulation. **[12 marks]**

At each time tick of the simulation:

- The visitor at the head of each platform queue who has completed the ride is removed from the platform.
- The visitor at the head of each platform queue who has not completed the ride (if there is one) boards the ride.
- Visitors waiting in the main queue are moved to fill up all the platform queues (if there are any spaces).
- A new visitor (if there is one, i.e., if it is not `null`) is added to the main queue.

```
public void runSimulation() {
    // YOUR CODE HERE
    // Board visitors on the ride and remove those who have completed the ride
    for (Deque<Visitor> pfQueue : pfQueues) {
        if (!pfQueue.isEmpty()) {
            Visitor visitor = pfQueue.peek();
            if (visitor.hasRidden()) {
                pfQueue.poll();
            } else {
                visitor.boardRide();
            }
        }
    }



    // Move visitors from the main queue to the platform queues
    for (Deque<Visitor> pfQueue : pfQueues) {
        while (pfQueue.size() < PLATFORM_SIZE && !mainQueue.isEmpty()) {
            pfQueue.offer(mainQueue.poll());
        }
    }



    // Add new visitor to the main queue
    Visitor newVisitor = Visitor.newVisitor();
    if (newVisitor != null) {
        mainQueue.offer(newVisitor);
    }



}
```

Question 3: Complexity – Big-O Costs **[16 marks]**

For each piece of code given in parts (a) and (b) 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.
- in both parts **(a)** and **(b)**, assume that `list` is an ArrayList of size n.

a) [7 marks]

```
Set<Integer> result = new TreeSet<Integer>();   // cost = O(  1   )     times = 1
for (int i=list.size()-1; i>=0; i--) {
    int count = 0;                              // cost = O(  1   )     times = n
    for (int j=list.size()-1; j>=0; j--){
        if (list.get(i) == list.get(j)){        // cost = O(  1   )     times = n^2
        count = count + 1;                      // cost = O(  1   )     times = n^2
        }
    }
    if (count > 3){                             // cost = O(  1   )     times = n
    result.add(list.get(i));                    // cost = O( log(n) )     times = n
    }
}
                                                // Total Cost = O(  n^2   )
```

b) [4 marks] (Note how the for loop is incremented!)

```
Queue<Integer> queue = new PriorityQueue<Integer>();// cost = O(  1  )   times = 1
for (int i=1; i<list.size(); i=i*2){
    if(list.get(i) == list.get(i-1)){               // cost = O( 1  )    times = log(n)
        queue.offer(list.get(i));                   // cost = O(log(n))  times = log(n)
    }
}
                                                    // Total Cost = O( [log(n)]^2  )
```

c) [5 marks]

We are using a java program to process and maintain student grades in a course. To this end, we use a HashMap where the key is the Student ID and the value is the student's grade. When the number of students is 1,000 ($\approx 2^{10}$) students, the program takes 10 nanoseconds to find a student's grade given the student ID. If the number of students were 20,000 students, how long would you expect the program to take to find a student's grade given the student ID? Explain why.

```
Your answer:
Still 10 nanaseconds.
Finding a student from a HashMap is O(1) (assuming a few number of collisions).
```

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: Binary Trees **[26 marks]**

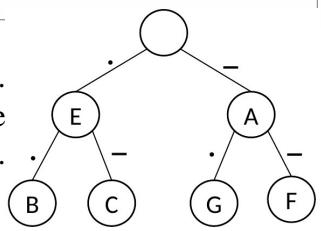
Suppose we are using a binary tree to encode characters as sequences of dots (`"."`) and dashes (`"-"`). In the tree, the root node contains an empty string and any other node contains either a letter or an empty string. In this tree, starting at any node:

- moving left along a branch represents a dot and
- moving right along a branch represents a dash.

The `CharNode` class represents the tree nodes. It has the following constructor and methods:

```
// CharNode class:
// Creates a new CharNode containing a string (single letter or empty) and two child
↪  nodes for the dot and dash branches.
public CharNode(String letter, CharNode dot, CharNode dash);
public String getLetter(); // Returns the letter stored in the current CharNode.
public CharNode getDot(); // Returns the left child (dot branch).
public void setDot(CharNode node); // Sets the left child (dot branch) to the provided
↪  node.
public CharNode getDash(); // Returns the right child (dash branch).
public void setDash(CharNode node); // Sets the right child (dash branch) to the
↪  provided node.
```

The figure on the right illustrates a binary tree used for encoding letters. In this tree, empty circles represent nodes containing an empty string, the letter "C" is encoded as · – and the letter "F" is encoded as – – (two dashes).



a) Complete the following `listString` method, which takes the root node of a tree and returns a `Set` containing all unique letters (excluding empty strings) encoded in the tree. For example, for the tree above, `listString` method should return the set `{E, B, C, A, G, F}`. **[13 marks]**

```
public Set<String> listString(CharNode node) {
    Set<String> results = new HashSet<String>();
    // YOUR CODE HERE

    listString(node, results);
    return results;
}


public void listString(CharNode node, Set<String> results) {
    if (node == null) { return; }

    if (!node.getLetter().isEmpty()) {
        results.add(node.getLetter());
    }

    listString(node.getDot(), results);
    listString(node.getDash(), results);



}
```

Question 4 (continued)

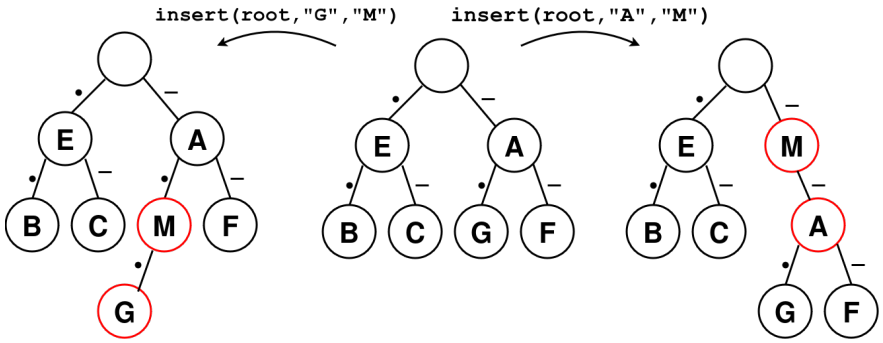
b) Complete the `insert()` method below. **[13 marks]**

The method takes three parameters:

- The root node of the binary tree.
- An original letter `oriL`.
- A new letter `newL`.

The task is to insert `newL` into the tree before `oriL`, making `oriL` a child of `newL`, without changing `oriL`'s original parent. Additionally, `newL` should follow the same connection (dot or dash link) as `oriL` did. You can assume that the tree always contains `oriL`.

The figure below illustrates two examples of trees after the insertion of nodes.




```
public void insert(CharNode node, String oriL, String newL) {
    // YOUR CODE HERE




    if (node == null) {
        return;
    }

    if (node.getDot() != null && node.getDot().getLetter().equals(oriL)) {
        CharNode newNode = new CharNode(newL, node.getDot(), null);
        node.setDot(newNode);
        return;
    }

    if (node.getDash() != null && node.getDash().getLetter().equals(oriL)) {
        CharNode newNode = new CharNode(newL, null, node.getDash());
        node.setDash(newNode);
        return;
    }

    insert(node.getDot(), oriL, newL);
    insert(node.getDash(), oriL, newL);


}
```

Question 5: Traversing General Trees **[14 marks]**

a) *pre-order traversal of a file system*: **[5 marks]**

Imagine you are designing a basic file system for an operating system. The file system is represented as a tree where each node is either a file or a directory. A directory can contain other directories and files, but a file cannot contain other nodes. Below is the `FSNode` class, representing a **F**ile **S**ystem **Node**, which you will use.

```
class FSNode {
    String name;
    boolean isDirectory;
    FSNode[] children;

    public FSNode(String name, boolean isDirectory, FSNode[] children) {
        this.name = name;
        this.isDirectory = isDirectory;
        this.children = children;
    }

    public String getName() { return name;}
    public boolean isDirectory() { return isDirectory;}
    public FSNode[] getChildren() { return children;}
}
```

Write a method `preTraversal` that performs a *pre-order traversal* of this file system, starting from the root directory, and returns a list of file and directory names in the order they are visited.

```
public List<String> preTraversal(FSNode root) {
    // YOUR CODE HERE




        List<String> result = new ArrayList<>();
    if (root != null) {
        result.add(root.getName());   // Visit the current node
        if (root.isDirectory()) {     // Only directories have children
            for (FSNode child : root.getChildren()) {
                result.addAll(preTraversal(child));
            }
        }
    }
    return result;









}
```

b) *Depth of a file system:* **[5 marks]**

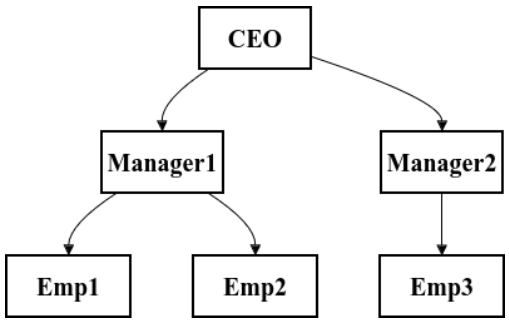
Continuing from the file system representation as a tree, we want to find the depth of the deepest component in the file system. The depth is defined as the number of edges on the longest path from the root directory to any file or subdirectory. If the tree consists of only one node (the root), the depth is 0.

Using the `FSNode` class defined in part (a), complete the following method to calculate the depth of the file system given the root node.

```
public int getDepth(FSNode node){
    if (node == null) return 0;
    int maxDepth = 0;
    // YOUR CODE HERE (note that only directories have children)



    if (node.isDirectory()) {
        for (FSNode child : node.getChildren()){
            int childDepth = getDepth(child);
            if (childDepth > maxDepth){
                maxDepth = childDepth;
            }
        }
    }
    return maxDepth+1;




}
```

c) *Post-order Traversal of an Organizational Hierarchy:* **[4 marks]**

A company's organizational hierarchy is represented by the following tree (see diagram below). Using a post-order traversal, what is the order of employees visited? (Note: post-order, not pre-order).




```
YOUR ANSWER:

Emp1, Emp2, Manager1, Emp3, Manager2, CEO
```

Question 6: Graph Traversals **[10 marks]**

You are designing a feature for a social networking platform to suggest friends to users. The network is represented as a graph where each node is a user, and an edge between two users indicates they are *direct* friends. Two users are *connectable* if there is a path between them through any number of intermediate users.

The users are represented by the `UserNode` class which has the following methods:

```
// UserNode class methods
String getName() // returns the name of the user.
List<UserNode> getFriends() // returns a list of the user's direct friends.
```

Your task: Complete the following method. The method should return the list of all user names that are connectable to the given `UserNode`. Note that the starting user should not be included in the result list. You can use any approach you want.

```
public List<String> findAllConnectable(UserNode start) {
    List<String> connectables = new ArrayList<>();
    // YOUR CODE HERE
////////////////////////// Version 1: BFS
    Set<UserNode> visited = new HashSet<>();
    Queue<UserNode> queue = new LinkedList<>();

    // Start with the given user
    queue.offer(start);
    visited.add(start);

    while (!queue.isEmpty()) {
        UserNode current = queue.poll();

        // Get friends of the current user
        for (UserNode friend : current.getFriends()) {
            // If the friend has not been visited yet
            if (!visited.contains(friend)) {
                visited.add(friend);
                queue.offer(friend);

                // Add to the connectables list (avoid adding the start user)
                if (!friend.equals(start)) {
                    connectables.add(friend.getName());
                }
            }
        }
    }

    return connectables;
}
////////////////////////// Version 2: DFS

    Set<UserNode> visited = new HashSet<>();

    // Start DFS from the given user
```

```
    dfs(start, visited, connectables);

    return connectables;
}

private void dfs(UserNode current, Set<UserNode> visited, List<String> connectables) {
    visited.add(current);

    // Explore each friend of the current user
    for (UserNode friend : current.getFriends()) {
        if (!visited.contains(friend)) {
            // Add the friend's name to connectables list (avoid adding the start user)
            if (!friend.equals(current)) {
                connectables.add(friend.getName());
            }
            // Recursive DFS call
            dfs(friend, visited, connectables);
        }
    }
}
```

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 7: Binary Search and Heaps **[14 marks]**

The university library maintains a digital catalog of all its books, sorted by ISBN. Students often search for books by ISBN to locate them quickly.

Consider the following code snippet for performing a binary search on an array of ISBN numbers:

```
public int searchISBN(int[] catalog, int targetISBN) {
    int left = 0;
    int right = catalog.length - 1;

    while (left <= right) {
        int mid = (left + right) / 2;
        if (catalog[mid] == targetISBN) {
            return mid;
        } else if (catalog[mid] < targetISBN) {
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }
    return -1;
}
```

a) Explain in your own words how this binary search algorithm works. What is the significance of updating left and right in the loop? **[4 marks]**

```
Your answer:


alg repeatedly divides the array in half by comparing the target ISBN to the middle
↪  element (`mid`). If the target is found, it returns the index. If the target is
↪  less than the middle element, it searches the left half by updating `right`; if
↪  greater, it searches the right half by updating `left`. Continues until the target
↪  is found or left meets right...
```

b) If the array has 1024 books (i.e., `catalog.length = 1024`), what is the maximum number of comparisons the algorithm will need to make? Explain your reasoning. **[5 marks]**

```
Your answer:

10, because binary search reduces the search space by half each time, leading to a
↪  logarithmic time complexity of O(log n), and log2(1024) = 10.
```

c) A task management app uses a max-heap to keep the most urgent task at the top of the list, based on its deadline. Consider the following code snippet for inserting a new task into the max-heap:

```
public void insertTaskIntoHeap(int[] heap, int deadline) {
    heap[heap.length] = deadline; // Add the new task at the end of the heap
    int currentIndex = heap.length;

    while (currentIndex > 0) {
        int parentIndex = (currentIndex - 1) / 2;
        if (heap[currentIndex] > heap[parentIndex]) {
            // Swap the current node with its parent
            int temp = heap[currentIndex];
            heap[currentIndex] = heap[parentIndex];
            heap[parentIndex] = temp;
            currentIndex = parentIndex; // Move up to the parent's position
        } else {
            break; // Heap property is satisfied
        }
    }
}
```

Explain the purpose of the while loop in this code. Why do we compare the current node with its parent? **[5 marks]**

```
Your answer:

While loop restores the max-heap property after a new task is added. The new task is
↪ initially placed at the end of the heap, and the loop moves it up the tree if
↪ necessary by comparing it with its parent. If the new task has a higher priority
↪ (in this context, a closer deadline) than its parent, the two nodes are swapped.
↪ This process continues until the new task is either the root of the heap (where it
↪ has no parent) or its parent has a higher priority, ensuring the max-heap property
↪ is maintained.
```

* * * * * * * * * *