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

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

COMP 103: Practice Exam

2024, June 12

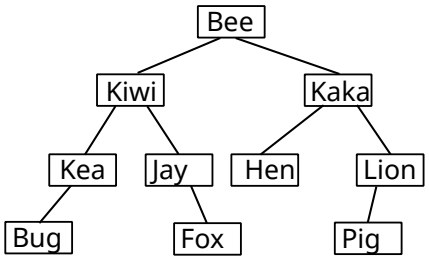
Instructions

- Time allowed: **ONE HOUR**
- Attempt ALL Questions.
- The examination will be marked out of 50 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.
- If you think some question is unclear, ask for clarification.
- You may use unmarked paper Chinese-English translation dictionaries.

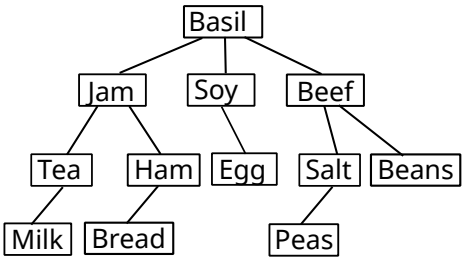
Questions	**Marks**	
1. Tree Traversal Orders	[5]	
2. Binary Trees	[15]	
3. General Trees	[10]	
4. Traversing Graphs 1	[10]	
5. Traversing Graphs 2	[10]	
	TOTAL:	

Question 1. Tree Traversal Orders **[5 marks]**

(a) **[2 marks]** For the binary tree below, give the order the nodes would be printed in if they were printed via an **pre-order depth-first traversal.**



(b) **[3 marks]** For the general tree below, give the order the nodes would be printed in if they were printed via an **post-order depth-first traversal**, assume children are processed left to right.



Question 2. Binary Trees **[15 marks]**

This question uses binary trees built from LabelNode objects:

```
public class LabelNode {
    private String label;      // the label
    private LabelNode left;    // left child node.
    private LabelNode right;   // right child node.

    public LabelNode(String label){ this.label = label; }

    public String getLabel(){     return label; }
    public LabelNode getLeft(){   return left ; }
    public LabelNode getRight(){ return  right ; }

    public void setLeft(LabelNode child){   left  = child; }
    public void setRight(LabelNode child){  right  = child;}
}
```

(a) **[5 marks]** Complete the following printPostOrder method to print all the labels in a Binary Tree of LabelNodes in an **post-order depth-first** order.
Hint: printPostOrder should be recursive.

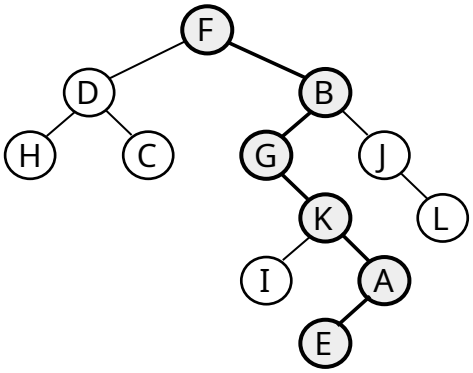
```
public void printPostOrder(LabelNode node) {




}
```

(Question 2 continued on next page)

(Question 2 continued)

(b) **[3 marks]** The following printFirstPath(..) method should print out the labels in a path from the root of a Binary Tree of LabelNodes. At each node, the path should follow the child with the alphabetically earlier label.

For example, with the tree below, printFirstPath(..) should print:
F B G K A E




```
    public void printFirstPath (LabelNode root){

1       LabelNode node = root;

        while (node!=null){

2           UI.print(node.getLabel()+" ");

3           if (node.getLeft().getLabel().compareTo(node.getRight().getLabel())<=0){
4               node = node.getLeft();
            }
5           else {
6               node = node.getRight();
            }
        }

    }
```

printFirstPath(..) is broken. What labels will it print out before it crashes with an error, and what line will cause the error?

labels:

Line with error:

(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) **[7 marks]** Complete the following addLabels method which will add a Set of labels to a Binary tree of LabelNodes. For each label in the newLabels Set, the method makes a LabelNode containing the label and should then follow a random path down the tree from the root, until it reaches a node that has a null child (left, or right, or both). It should then add the new LabelNode as a child of that node.

To follow a random path, it should choose the left child if (Math.random()<0.5), and choose the right child otherwise.

You may assume that the root is not null.

```
public void addLabels(LabelNode root, Set<String> newLabels){
    for (String label : newLabels){
        LabelNode newNode = new LabelNode(label); // new node to add
    }
}
```

Question 3. General Trees **[10 marks]**

This question uses general trees built from Person nodes. The trees represent the employment structure in a company. Every person in the company (except the CEO) has a manger, and a set of "employees".

Note: The nodes have links up the tree (in the manager field) as well as links down the tree (in the employees field).

```
public class Person {
    private String name;
    private Person manager;
    private Set<Person> employees = new HashSet<Person>();

    public Person(String name){ this.name = name; }

    public String getName(){ return name; }

    public Set<Person> getEmployees(){ return Collections.unmodifiableSet(employees); }
    public void addEmployee(Person m){ employees.add(m); }
    public void removeEmployee(Person m){ employees.remove(m); }

    public Person getManager(){ return manager; }
    public void setManager(Person m){ manager = m; }
}
```

(a) **[7 marks]** Complete the following findPerson method which will search a tree of Person nodes for a Person with the specified name and will return the Person node. If there is no Person in the tree with that name, it returns null.

```
public Person findPerson(Person person, String name){




}
```

(Question 3 continued on next page)

(Question 3 continued)

(b) **[3 marks]** The following changeManager(..) method should move a Person from under their current manager to be a employee of a new manager. The method isn't complete. Add code to complete the move.

Note: The employees of person should move with the person.

```
public void changeManager(Person person, Person newManager){
    Person oldManager = person.getManager();
    newManager.addEmployee(person);









}
```

Question 4. Traversing Graphs 1 **[10 marks]**

You are writing a program to keep track of computers that are infected with a computer worm. A computer worm is a harmful computer program that copies itself to spread to other computers.

Your program stores information about all computers in a network of computers in List of Compter objects.

```
private List<Computer> allComputers; // all computers in a network of computers
```

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

The Computer class has a method called isDetected, which is true if the computer in infected by the worm. Otherwise, it is false.

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

(a) **[5 marks]** Complete the following findVulnerableComputers method. It should return a Set of the Computers that are directly or indirectly connected to infectedComputer which is infected by the worm.

```
public Set<Computer> findVulnerableComputers(Computer infectedComputer){




}
```

(Question 4 continued on next page)

(Question 4 continued)

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

```
public boolean potentialThreat(Computer suspectedComputer){
    Set<Computer> visited = new HashSet<Computer>();
    return potentialThreat(suspectedComputer, visited );
}

public boolean potentialThreat(Computer suspectedComputer, Set<Computer> visited){









}
```

Question 5. Traversing Graphs 2 **[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; }









}
```

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

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;
```