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

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

COMP 103 : Test 4

2022, Feb 9

Instructions

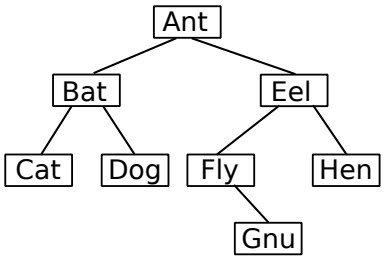
- Time allowed: **50 minutes**
- Attempt **all** questions. There are 30 marks in total.
- Write your answers in this test paper and hand in all sheets.
- If you think some question is unclear, ask for clarification.
- Brief Java documentation is provided with the test
- You may use dictionaries.
- You may write notes and working on this paper, but make sure your answers are clear.

Questions	**Marks**	
1. Binary Trees	[14]	
2. General Trees	[16]	
	TOTAL:	

Question 1. Binary Trees **[14 marks]**

(a) **[2 marks]** Suppose a binary tree contains 45 nodes. What is the *minimum* number of layers that the tree could have?

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



(Question 1 continued on next page)

(Question 1 continued)

(c) **[5 marks]** Complete the following printTreeDF method to print all the nodes of a tree in **post-order depth-first** order, always printing the left child before the right child.
Note: printTreeDF must be recursive.
The tree is made of BTNodes, which have the following methods available to you:

```
class BTNode {
        String getValue();     // return the value of this node
        BTNode getLeft();      // return the left child (null if none)
        BTNode getRight();     // return the right child (null if none)
}
```

```
public void printTreeDF(BTNode node) {




}
```

(d) **[5 marks]** Complete the following searchTreeBF method to search the nodes of a tree in **breadth-first** order to find (and return) the first node whose value starts with "go". It returns null if there are no nodes which start with "go".
The tree is made of BTNodes, as above.

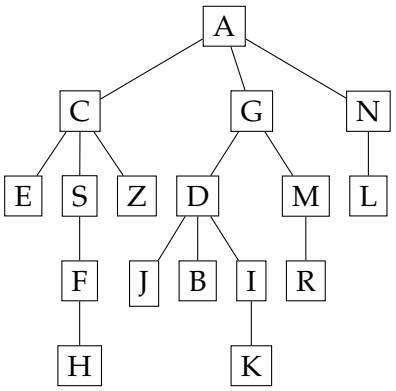
```
public BTNode searchTreeBF(BTNode node) {




}
```

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. General Trees **[16 marks]**

Consider the general tree shown below:



(a) **[2 marks]** Which order would the nodes be processed in if you performed a **breadth-first** traversal of the tree, adding the children of a node to the queue from **left-to-right**?

(b) **[2 marks]** Which order would the nodes be processed in if you performed a **post-order depth-first** traversal of the tree.

(c) **[2 marks]** Why doesn't it make sense to ask for an **in-order depth-first** traversal of the tree?

(Question 2 continued on next page)

(Question 2 continued)

An investment management program uses general trees to store information about companies and their subsidiaries.

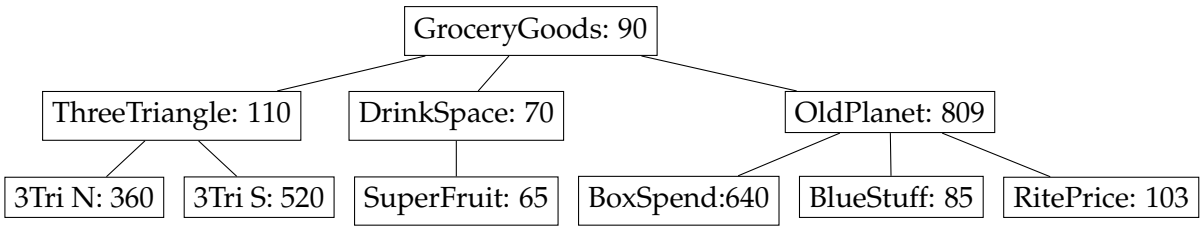
Each company is described by a Company object which has at least the following two methods:

```
class Company{
    public String getName();
    public int getRevenue();
}
```

A company tree is represented using GTNodes<Company>. The GTNode class is iterable, and iterates through the node's children.

```
class GTNode<E> implements Iterable<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 Iterator <GTNode<E>> iterator();    // get iterator for children
```

An example company tree (with the name and revenue of each company) is shown below.



(Question 2 continued on next page)

(Question 2 continued)

(d) **[10 marks]** Complete the following findOld(...) method which is given the root node of a company tree and a minimum revenue, and should return a Set of all the companies in the tree whose revenue is greater than the minimum.
You may assume that the company has at least one node.

For example, if tree is the company tree above, findOld(tree, 600) should return a set containing the "OldPlanet" and "BoxSpend" companies.

Hint: You should create a recursive "helper" method with more arguments.

```
public Set<Company> findOld(GTNode<Company> root, int minRev){




}
```

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

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

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 all standard collection classes
   public int size()                           // cost: O(1) for all 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 is close to the end.)
   public E remove(int index)                  // cost: O(n) (unless index is close to the 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

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 the key is not present)
   // (get put and remove return the old value, if any)
```

```
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)

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)


class Collections
   public void sort(List<E> list);                          // cost = O(n log(n)), but O(n) if almost sorted
   public void sort(List<E> list, (E e1, E e2)->{...});     // cost = O(n log(n)), but O(n) if 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> // All Comparable objects have a compareTo method:
   public int compareTo(E other);
          // returns
          //     -ve if this comes before other;
          //     +ve if this comes after other,
          //       0 if this and other are the same
```