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

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

COMP 103 : Test 4

2021, Feb 3 ** WITH SOLUTIONS **

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
- This test contributes 10% of your final grade
- You may use dictionaries.
- You may write notes and working on this paper, but make sure your answers are clear.

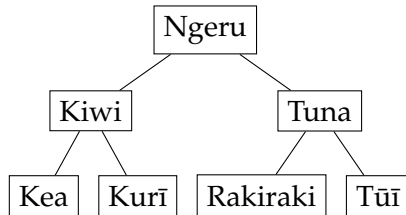| Questions	Marks	
1. Binary Trees	[10]	
2. General Trees	[20]	
	TOTAL:	

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 1. Binary Trees [10 marks]

(a) **[2 marks]** Suppose a binary tree is 5 layers deep. What is the *maximum* number of nodes that can fit in this tree before a new layer needs to be added?

$$1 + 2 + 4 + 8 + 16 = 31 \text{ (or } 2^5 - 1)$$

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

Kea, Kurī, Kiwi, Rakiraki, Tūī, Tuna, Ngeru

(c) **[6 marks]** Complete the following method, which should do an **in-order depth-first traversal** of a tree and print each node as it goes. It should visit the children from left to right.

The tree is made of BTNodes, which have the following methods available to you:

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

```
public void printTree(BTNode node) {
    if (node.getLeft() != null) {
        printTree(node.getLeft());
    }

    UI.println(node.getValue());

    if (node.getRight() != null) {
        printTree(node.getRight());
    }



}
```

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 [20 marks]

Suppose you are working on a video game that allows the player to have detailed text-based conversations with non-player characters. You have decided to use a Tree to represent each conversation, with player choices dictating which option happens next.
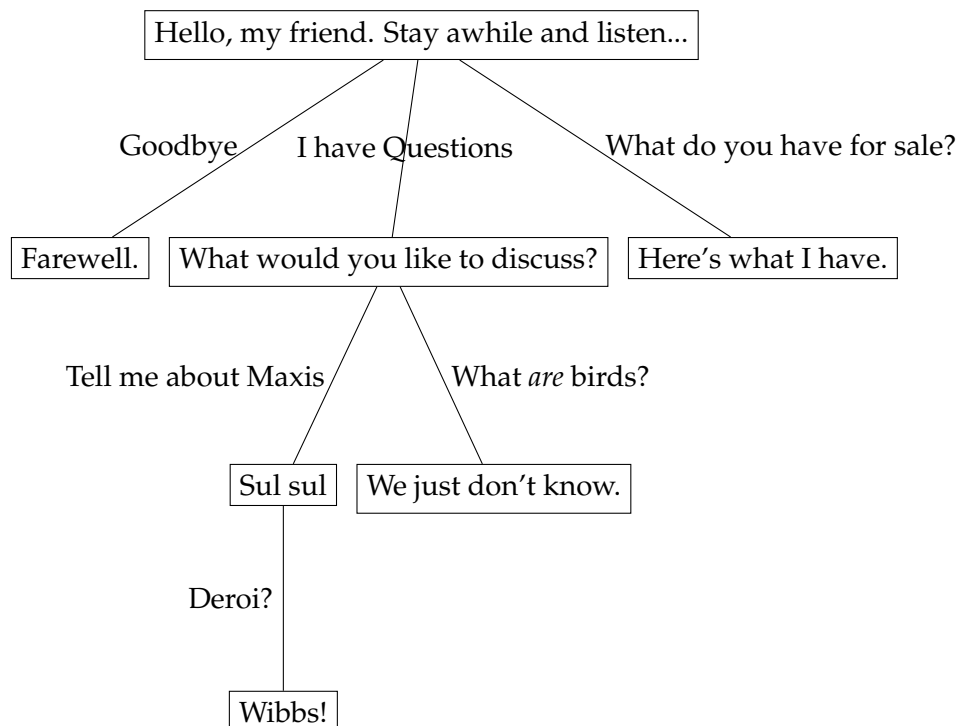
- The program uses GTNodes that contains String objects.
- The GTNode class is iterable, and iterates through the node's children.

```
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 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
    public void  Iterator <GTNode<E>> iterator();    //  get  iterator   for   children
```

One example conversation tree is shown below. The labels on the edges indicate the player's response and are provided only for clarity – you do not need to worry about them!

(Question 2 continued)

(a) **[10 marks]** Complete the following printConversationTree(...) method which is given the root node of the tree, and should <u>use recursion</u> to print out all the dialogue in the tree, using indentation to show the structure.

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

```
Hello, my friend. Stay awhile and listen...
    Farewell
    What would you like to discuss?
        Sul Sul
            Wibbs!
        We just don't know.
    Here's what I have.
```

Hint: You should create a recursive function with more arguments.

```java
public void printConversationTree (GTNode<String> node){
    printConversationTree (node, "");
}

public void printConversationTree (GTNode<String> node, String indent){
    UI. println (indent + node.getItem());
    for (GTNode<String> child : node) {
        printConversationTree ( child , indent + "   " );
    }
}
```
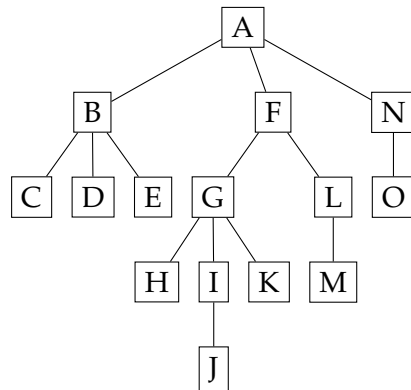
(b) **[4 marks]** what is the name for the tree traversal you used for printConversationTree

pre-order depth first traversal

(Question 2 continued)

(c) **[6 marks]** Suppose you have the general tree shown below:

In which order would be nodes the processed if you ran a method that performed a **breadth-first** traversal of the tree, adding the children of a node to the queue from **right-to-left**?

A N F B O L G E D C M K I H J

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

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
```