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

ID Number: ..... **Signature** .....

COMP 103: Test 3

5 October 2012

Model Solutions

Instructions

- Time allowed: **35 minutes**
- There are 25 marks in total.
- Answer **all** the questions.
- The appendix contains brief collection class documentation.
- If you think some question is unclear, ask for clarification.
- This test will contribute 10% of your final grade,
  (But your mark will be boosted up to your exam mark if that is higher.)
- You may use paper translation dictionaries, and calculators without a full set of alphabet keys.
- Write your answers in the boxes in this test paper and hand in all sheets. You may ask for additional paper if you need it.
- You may write notes and working on this paper, but make sure it is clear where your answers are.

Questions	**Marks**	
1. Linked Structures	[10]	
2. Implementing a Binary Search Tree	[10]	
3. Merging Linked List	[5]	
	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.

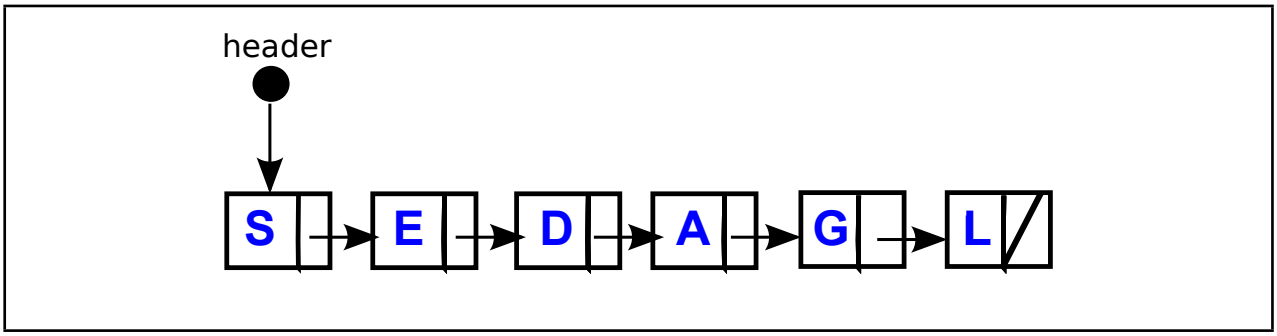
Question 1. Linked Structures [10 marks]

(a) [2 marks] Assume that you have a LinkedList class with a header field that contains a reference to the first of a chain of singly-linked LinkedNode objects.

The LinkedList class has the following insert method:

```
public void insert(String value){
    header = new LinkedNode(value, header);
}
```

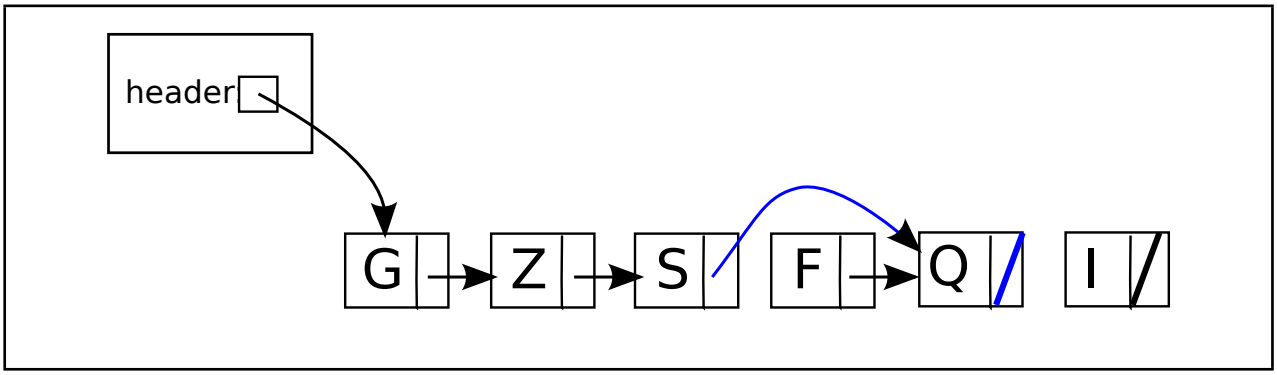
Show the final state of the LinkedList after the following values are inserted into the initially empty LinkedList: **L, G, A, D, E, S**.



(b) [2 marks] If a LinkedList contains n values, what is the cost of inserting a new value using the insert method above: O(1), O(log(n), O(n), or O(n^2)? Justify your answer.

O(1). It just has to create a new node and change one value, regardless of how large the list is.

(c) [2 marks] Draw the state of the LinkedList below, after the values **F** and **I** have been removed.



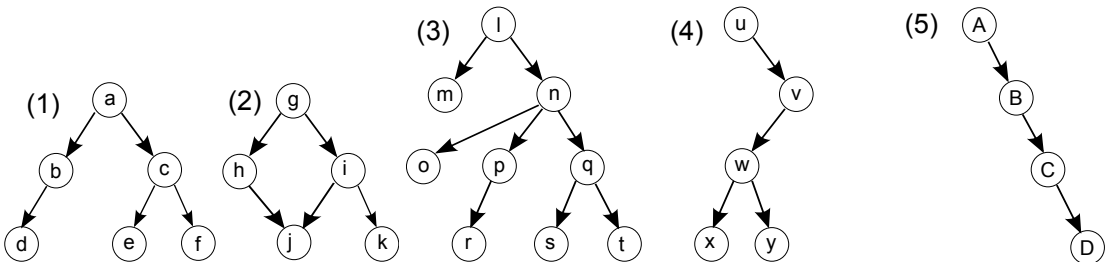
(Question 1 continued on next page)

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.

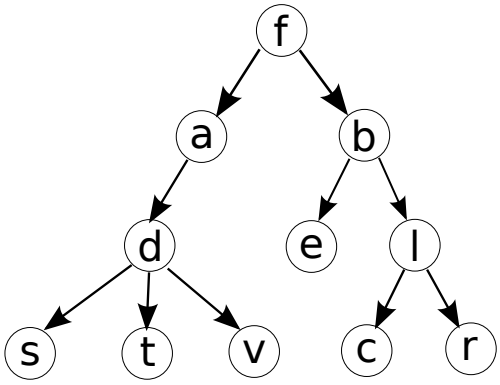
(Question 1 continued)

(d) [2 marks] Which of the following trees are **not** binary trees? Justify your answer.



(2) is not a binary tree because node j has two parents, and in a tree, a node can only have one parent (or none for the root).
(3) is not a binary tree because node n has three children; in a binary tree, nodes can have at most two children.

Consider the following tree and the output of a traversal of the tree:



Traversal: s t v d a e c r l b f

(e) [2 marks] Is this traversal a post-order, pre-order, in-order, or breadth-first traversal of the tree? Justify your answer.

Post-order. Each node is printed after its children , eg, `d` is printed after `s`, `t`, and `v` and all the other nodes are printed before the root node.

Question 2. Implementing Binary Search Trees [10 marks]

This question concerns a BinarySearchTree class which implements a Set of Strings.

BinarySearchTree objects have a field called root of type BSTNode that references the root node of the tree if the Set is non-empty, or otherwise is set to null.

Each BSTNode has three fields: item containing the value in the node, and left and right which contain references to the left and right children of the node. left and right will contain null if those children do not exist. The fields are public, so that they can be accessed by methods in the BinarySearchTree class.

The BinarySearchTree class has a size() method that returns the number of elements in the Set. This method calls a private helper method — sizeRec(BSTNode node) — to calculate the size recursively.

Part of the BinarySearchTree class is shown below:

```
public class BinarySearchTree {

    private BSTNode root;
    public int size() {
        return this.sizeRec(root);
    }

}

class BSTNode {
    public String item;
    public BSTNode left, right;
}
```

(a) [3 marks] Complete the following sizeRec(BSTNode node) method that should compute the size of the subtree whose root is node using a recursive approach. Note that if the node is null, the size should be 0.

```
    private int sizeRec(BSTNode node) {
        if (node == null) return 0;
        int leftSize = sizeRec(node.left);
        int rightSize = sizeRec(node.right);
        return leftSize + rightSize + 1;









    }
```

(Question 2 continued on next page)

(b) [5 marks] Complete the following contains(String target) method that returns true if the target string is in the BSTSet, or false if it is not. Your method should use an iterative rather than recursive approach (but don't use an Iterator!).
Note that Strings are Comparable so you can call the compareTo method on them.

```
public boolean contains(String target){
    if (target == null) throw new IllegalArgumentException("null argument");
    BSTNode node = root;
    while (node != null){
        int c = target.compareTo(node.item);
        if (c==0) return true;
        else if (c<0) node = node.left;
        else node = node.right;
    }
    return false;



}
```

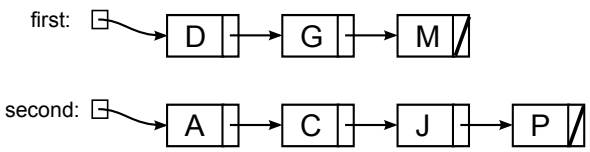
(c) [2 marks] Complete the following rootIsExactMiddle() method that should return true if and only if the string stored in the root node would appear at the exact middle of the sequence of strings printed out by an in-order traversal of the tree.
Hint: you may use the methods from the previous subquestions.

```
public boolean rootIsExactMiddle() {
    if (root == null) return false;
    return sizeRec(root.left)==sizeRec(root.right);









}
```

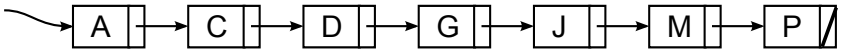
Question 3. [5 marks]

A key step in the MergeSort algorithm is to merge two sorted sublists into a single sorted list. The version of MergeSort in the first part of the course used arrays. It is also possible to implement MergeSort using linked lists. Complete the mergeLists method on the facing page that is passed two sorted linked lists, and returns a new sorted linked list that contains all the items from the two lists. You may assume that neither list is empty, and if there are duplicates in the two lists, then all the duplicates should be preserved in the final list.

For example, given the two lists:



MergeLists should return a new list:



Note that the linked lists in this case are simply represented by a reference to the first LinkedNode object in the sequence of LinkedNodes.

(Question 3 continued on next page)

(Question 3 continued)

```
    public LinkedNode mergeLists(LinkedNode first, LinkedNode second){

        LinkedNode head = new LinkedNode(null, null);
        LinkedNode tail = head;
        while ( first !=null  && second != null){
            if  ( first .value.compareTo(second.value)<0){
                tail .next = new LinkedNode(first.value, null);
                first  =  first .next;
            }
            else {
                tail .next = new LinkedNode(second.value, null);
               second = second.next;
            }
            tail  =  tail .next;
        }
        if  ( first !=null){
            tail .next =  first ;
        }
        else {
            tail .next = second;
        }
        return head.next;









    }

class LinkedNode {
    public String value;
    public LinkedNode next;

    public LinkedNode(String v, LinkedNode n){
        value = v;    next = n;
    }
}
```

****

Appendix: Collections

```
interface Collection<E>
    public boolean isEmpty()
    public int size()
    public boolean add(E item)
    public boolean contains(Object item)
    public boolean remove(Object element)
    public Iterator<E> iterator()

interface List<E> extends Collection<E>
    // Implementations: ArrayList, LinkedList
    public E get(int index)
    public E set(int index, E element)
    public void add(int index, E element)
    public E remove(int index)
    // plus methods inherited from Collection

interface Set extends Collection<E>
    // Implementations: ArraySet, HashSet, TreeSet
    // methods inherited from Collection

interface Queue<E> extends Collection<E>
    // Implementations: ArrayQueue, LinkedList
    public E peek ()                    // returns null if queue is empty
    public E poll ()                    // returns null if queue is empty
    public boolean offer (E element)    // returns false if fails to add

class Stack<E> implements Collection<E>
    public E peek ()                    // returns null if stack is empty
    public E pop ()                     // returns null if stack is empty
    public E push (E element)           // returns element being pushed

public class Collections
    public void sort(List<E>)
    public void sort(List<E>, Comparator<E>)
    public void shuffle(List<E>, Comparator<E>)
```