The University of Western Ontario

Computer Science CS1027b

Final Examination - FEB GYM - 9am-noon - Wednesday, April 29^{th}, 2015

Last Name:	
Given Names:	
Student Number:	

PLEASE CIRCLE ONE

Section I
John Barron
Tuesday 11:30am-1:30pm B&GS 0153
Thursday 11:30am-12:30pm B&GS 0153

Section II
Beth Locke
Tuesday 3:30pm-5:30pm 3M 3250
Thursday 3:30pm-4:30pm 3M 3250

DO NOT TURN THIS PAGE UNTIL INSTRUCTED TO DO SO!

Instructions

- Fill in your name and student number above immediately. Please use the last name and given names the university has for you (as on your student card).
- You have **3 hours** to complete the exam.
- For multiple choice questions, circle your answers on this exam paper.
- For other questions, write your answers in the spaces provided in this exam paper.
- The marks for each individual question are given.
- Relevant Java interfaces are at the back of the exam.
- There are also pages for rough work at the back of the exam. You may detach them if you wish, but hand them in with the rest of the exam paper.
- Calculators, phones and laptops are not allowed!

Mark Summary

1	2	3	4	5	6	7	8	9	10	total
/20	/10	/15	/20	/15	/20	/35	/10	/15	/30	/175

Problem 1: true/false (20 marks)

Choose **one** answer for each question.

1.	Encapsulation encourages data and the operations on that data be combined in a single class definition.	true	false
2.	A queue is an example of a linear collection.	true	false
3.	Abstract data types are programming language specific.	true	false
4.	The first item placed in a stack would be the last item removed.	true	false
5.	The first item placed in a binary search tree would be the last item removed.	true	false
6.	An array in Java can change in size once allocated.	true	false
7.	Static variables are stored on the call stack.	true	false
8.	When a method is called, a call frame is allocated on the call stack for use during its execution.	true	false
9.	When Java runs recursive methods, it will reuse the call frame of a method each time that same method calls itself.	true	false
10.	The sum of all numbers from 1 to n can be solved recursively.	true	false
11.	The sum of all numbers from 1 to n can be solved iteratively.	true	false
12.	The n^{th} Fibonacci number is best computed with a recursive solution.	true	false
13.	The Towers of Hanoi problem is $O(2^n)$, but only in the iterative case.	true	false
14.	Given an element and an ordered list, the process of inserting that element is $O(1)$ when using a linked data structure.	true	false
15.	Given an element and an unordered list, the process of inserting that element is $O(1)$ when using an array data structure.	true	false
16.	In a binary search tree, the leftmost node is a leaf and contains the smallest element.	true	false
17.	The degree of a tree node refers to the number of children it has.	true	false
18.	The base case for recursive tree traversal methods is: this node has no children.	true	false
19.	Balancing a binary search tree is key in maintaining its time complexity benefits.	true	false
20.	The time complexity of Insertion Sort is $O(n^2)$.	true	false

Problem 2 (10 marks)

In each of the following situations, use big-O notation to express the amount of work being done in terms of n.

1. (2%) The last element is removed from an UnorderedArrayList of size n.
   Answer: $O(1)$

2. (2%) A level order traversal of a linked tree of of size n is performed.
   Answer: $O(n)$

3. (2%) We remove the smallest element from a balanced binary search tree of size n.
   Answer: $O(log(n))$

4. (2%) We execute the following code segment

```
for (int i = 1; i < n; i++)
  for (int j = n; j > 1; j=j/2)
    System.out.println(i+j);
```

   Answer: $O(n * log(n))$

5. (2%) We execute the following code segment

```
static int func(int n){
   if (n <= 0)  return 1;
   else return (n+func(n-2));
}
```

   Answer: $O(n)$

Problem 3 (15 marks)

1. (2 %) What is the time complexity of an Insertion sort if the items are already in ascending order?

   $O(n^2)$

2. (2 %) What is the time complexity of a Selection sort if the items are already in ascending order?

   $O(n^2)$

3. (2 %) What is the time complexity of a Selection sort if the items are not already sorted?

   $O(n^2)$

4. (2 %) What is the time complexity of a Quick sort if the pivot is always in the middle?

   $O(nlog(n))$

5. (2 %) What is the time complexity of searching in a binary search tree if the tree is balanced?

   $O(nlog(n))$

6. (5 %) What is the time complexity of the following sorting method, if the array parameter toSort is already sorted when the method is called with `toSort` = { 1, 2, 4, 7, 9 })?

```
public static void sortIntArray (int[] toSort)
{
boolean sorted = false;
for(int x = toSort.length-1;!sorted && x > 0;x--) {
    sorted=true;
    for(int y=0;y < x;y++)
        if(array[y] > array[y+1]) {
          swap(array[y],array[y+1]);
          sorted=false;
        }
    }
}
```

   $O(n)$

Problem 4 (20 marks)

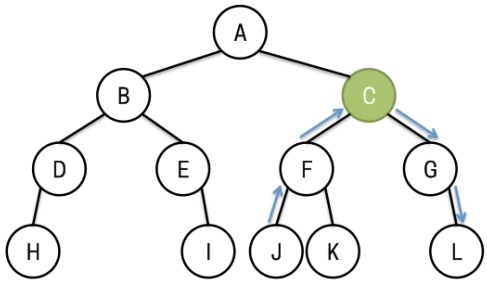
This question is related to your fourth assignment. Assume that you have completed the method `pathToRoot` and `lowestCommonAncestor`.

```
/**
 * Finds the target elements and returns a path from the targetElement to the root.
 * It calls a recursive method to do this.
 * @param targetElement
 * @return Iterator which will iterate over elements on the path from the targetElement
 * to the root of the tree (includes both).
 */
public Iterator<T> pathToRoot(T targetElement) throws ElementNotFoundException

/**
  * Will add to the pathToRoot ArrayUnorderedList<T> visitor if the node
  * is on the path from the targetElement to the root of the tree
  * @param targetElement
  * @param node
  * @param pathToRoot
  */
protected void pathToRootAgain(T targetElement, BinaryTreeNode<T> node,
                          ArrayUnorderedList<T> pathToRoot)

/**
  * Finds the lowest (ie. deepest) or maximal-level node common to both the path
  * from target1 to the root and the path from target2 to the root.
  * @param target1 The first element to find
  * @param target2 The second element to find
  * @return the element found in the lowest common ancestor node
  * @return the element found in the lowest common ancestor node
  * @throws ElementNotFoundException
  */
public T lowestCommonAncestor( T target1, T target2 ) throws ElementNotFoundException
```

Write a method to add to `LinkedBinaryTree.java` that will return an iterator which will iterate over the elements along the shortest path from the element target1 to the element target2. The shortest path from target1 to target2 will pass through the lowestCommonAncestor, that is to say, the shortest path with be from target1 up to the to the lowestCommonAncestor, then from the lowestCommonAncestor back down the tree to target2. You may call any of the above methods, as well as those defined for the ADT, in your solution.



1. Write your method here. The header and some code has been provided for you.

```
public Iterator<T> shortestPath( T target1, T target2 ) throws ElementNotFoundException {
        T commonAncestor = lowestCommonAncestor(target1, target2);
        ArrayUnorderedList<T> pathOne = new ArrayUnorderedList<T>();
        pathToRootAgain(target1, root, pathOne);
        ArrayUnorderedList<T> pathTwo = new ArrayUnorderedList<T>();
        pathToRootAgain(target2, root, pathTwo);

        if(pathOne == null || pathTwo == null)
            throw new ElementNotFoundException("binary tree");

        ArrayUnorderedList<T> shortestPath = new ArrayUnorderedList<T>();

        //fill in the shortestPath list processing here and return an iterator

        T current = pathOne.removeFirst();
        while(current!=null && !current.equals(commonAncestor)){
            shortestPath.addToRear(current);
            current = pathOne.removeFirst();
        }

        shortestPath.addToRear(commonAncestor);

        while(!pathTwo.removeLast().equals(commonAncestor));

        while(!pathTwo.isEmpty()){
            current = pathTwo.removeLast();
            shortestPath.addToRear(current);
        }

        return shortestPath.iterator();
}
```

Problem 5 (15 marks)

The following two methods, `mystery` and `mysteryAgain` have been added to the `LinkedOrderedList` class:

```
public class LinkedOrderedList<T> extends LinkedList<T>
                            implements OrderedListADT<T> {
     protected int count;
     protected LinearNode<T> head, tail;

     ...

     public int mystery() {
          return mysteryAgain(head);
     }

     private int mysteryAgain(LinearNode<T> current){
          if( current == null)
               return 0;
          else
               return 1 + mysteryAgain( current.getNext() );
     }

     ...

}
```

Assume that all methods of the LinkedList and LinkedOrderedList have been implemented correctly. Also, assume the LinkedList class defines a toString method which will put the contents of the list onto a single line separated by a space.

Consider the following small program:

```
1. public class TestMyLists {
2.     public static void main(String args[]){
3.          LinkedOrderedList<Integer> myList = new LinkedOrderedList<Integer>();
4.          myList.add(5);
5.          myList.add(9);
6.          System.out.println(myList.toString());
7.          System.out.println(myList.mystery());
8.          myList.add(2);
9.          myList.add(10);
10.         System.out.println(myList.toString());
11.         System.out.println(myList.mystery());
12.     }
13. }
```

These questions are about the code found on the previous page.

1. (2 %) What would be printed by line 6?

   `Answer: 5 9`

2. (2 %) What would be printed line 7?

   `Answer: 2`

3. (2 %) What would be printed by line 10?

   `Answer: 2 5 9 10`

4. (2 %) What would be printed by line 11?

   `Answer: 4`

5. (3 %) What does the mystery method return, when invoked on a list?

   `Answer: The size of the list`

6. (4 %) If the size of the list is n, what is the time complexity of invoking the mystery method (in big O notation)?

   Answer: $O(n)$

Problem 6 (20 marks)

(6a) (7%) Suppose we have a number of Person objects in an array and we wish to insert them in an unordered list. Using the methods in `ListADT` and `UnorderedListADT`, insert these person objects into an unordered list `personList`. Print out the number of persons in the `personList` list and its contents using list methods. to print out the contents of this list.

```
public static void main(String[] args) {
ArrayUnorderedList<Person> personList=
                    new ArrayUnorderedList<Person>();
Person[] p = {new Person("Andrew",  34),
              new Person("Betty",   22),
              new Person("Weixin",  24),
              new Person("Hanna",   29),
              new Person("Betty",   22),
              new Person("Andrew",  23),
              new Person("Hanna",   31),
              new Person("Andrew",  23),
              new Person("Weixin",  28),
              new Person("Mohamed", 33),
              new Person("Weixin",  88),
              new Person("Andrew",  25)};

    for(int i=0;i<p.length;i++) {
       personList.addToRear(p[i]);
    }
    System.out.println("Number of persons in the list:" + personList.size());
    System.out.println("\npersonList:");
    System.out.println(personList.toString());

}
```

(6b) (4%) What will the main method print? outputs:

```
Number of persons in the list:12

personList:
Andrew     34
Betty      22
Weixin     24
Hanna      29
Betty      22
Andrew     25
Hanna      31
Andrew     34
Weixin     28
Mohamed    33
Weixin     88
Andrew     25
```

(6c) (7%) Suppose you have the unordered list `personList`, of type `Person` created above. Write a `int` method, `countDuplicates`, that returns an integer giving the number of duplicates in the list. Again use methods in the ListADT and UnorderedListADT to do this calculation.

```
public static int countDuplicates(ArrayUnorderedList<Person> personList) {
int numDuplicates=0; // number of duplicates
while(!personList.isEmpty())
   {
   Person personOfInterest=personList.removeFirst();
   if(personList.contains(personOfInterest))
      {
      numDuplicates++;
      }
   }
return(numDuplicates);
}
```

(6d) (2%) For the list `personList` from (6a), what does the statement:

```
System.out.println("\nNumber of duplicates: " + countDuplicates(personList) +"\n");
```

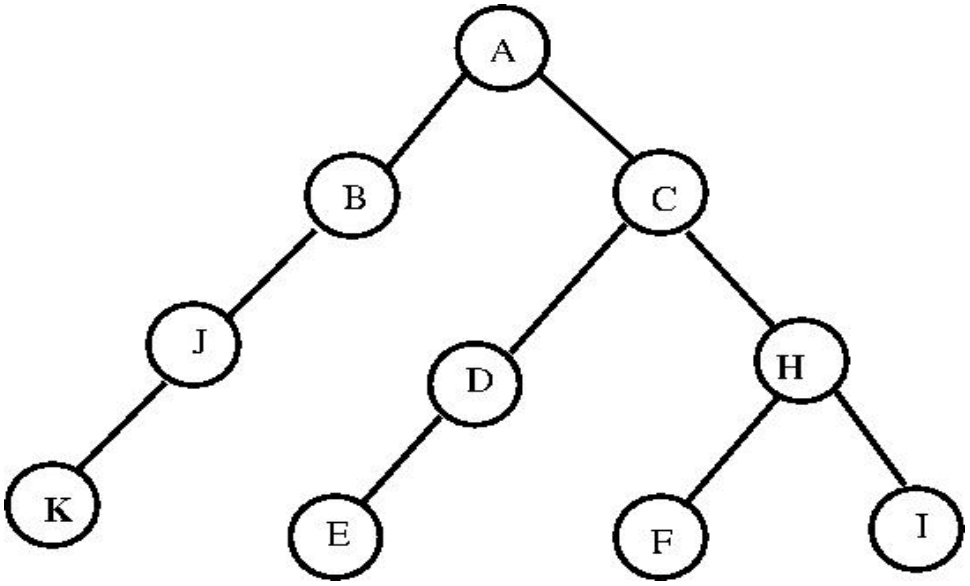
print?

```
Number of duplicates: 3

They are Andrew 34, Andrew 25 and Betty 22
```

Problem 7 (35 marks)

Consider the following binary tree:



(7a) (5%) Give the **inorder** traversal of this tree:

K J B A E D C F H I

(7b) (5%) Give the **preorder** traversal of this tree:

A B J K C D E H F I

(7c) (5%) Give the **postorder** traversal of this tree:

K J B E D F I H C A

(7d) (5%) Give the **level order** traversal of this tree:

A B C J D H K E F I

(7e) (5%) Consider the mystery traversal below.

```
public static void mystery1(BinaryTreeNode<String> node) {
    if(node==null) return;
    if(node.getLeft()!=null || node.getRight()!=null)
        System.out.format("%3s ",node.getElement());
    mystery1(node.getLeft());
    mystery1(node.getRight());
}
```

What is printed by this method for the binary tree on the previous page? A B J C D H

(7f) (5%) Consider the mystery traversal below.

```
public static void mystery4(BinaryTreeNode<String> node) {
    if(node==null) return;
    if(node.getLeft()==null && node.getRight()!=null)
        System.out.format("%3s ",node.getElement());
    mystery4(node.getLeft());
    mystery4(node.getRight());
}
```

What is printed by this method for the binary tree on the previous page? nothing

(7g) (5%) Consider the mystery traversal below.

```
public static void mystery5(BinaryTreeNode<String> node) {
    if(node==null) return;
    if(node.getLeft()!=null || node.getRight()!=null)
        System.out.format("%3s ",node.getElement());
    mystery5(node.getRight());
    mystery5(node.getLeft());
}
```

What is printed by this method for the binary tree on the previous page? A C H D B J

Problem 8 (10 marks)

Write a recursive boolean method `mommy` that returns true or false depending on whether node A is the mommy of node B in a binary tree rooted by node *root* or not. The BinaryTreeNode definition is given in the appendix. It has getter or setter methods for getting and setting the element data, and the left and right children.

```
public static boolean mommy(BinaryTreeNode<String> A,
                            BinaryTreeNode<String> B,
                            BinaryTreeNode<String> node) {
    if(node==null)
        return(false);
    if(node==A && node.getLeft()==B)
        return(true);
    if(node==A && node.getRight()==B)
        return(true);
    return (mommy(A,B,node.getLeft()) || mommy(A,B,node.getRight()));
}
```

Problem 9 (15 marks)

Given two binary trees containing strings, write a recursive boolean method **identicalTree** that returns true if they are both structurally identical and have the same String values at the corresponding nodes (this is state equivalence). Use the getters and setters described in the BinaryTreeNode class. The private variable element is a String, use String's equals to test for equality is necessary.

```
public static boolean identicalTree(BinaryTreeNode<String> node1,
                                    BinaryTreeNode<String> node2) {
if(node1==null && node2==null) return(true);
else
if((node1==null && node2!=null) ||
   (node1!=null && node2==null)) return(false);

if(!(node1.getElement().equals(node2.getElement()))) return(false);
return(identicalTree(node1.getLeft(),node2.getLeft()) &&
       identicalTree(node1.getRight(),node2.getRight()));
}
```

Problem 10 (15 marks + 15 bonus marks)

You are required to solve one of these problems for full marks. Solving both will give you up to 15% bonus marks.

(10a) (15%) Consider the identical tree problem posed in Problem 9. Write an iterative (non-recursive) method using **queues** to determine if two trees have identical structure and state equivalence for corresponding node values. Use the getters and setters described in the BinaryTreeNode class. The private variable element is a String, use String's equals to test for equality is necessary. Do **not** use Iterators!

```
public static boolean identicalTreeQueue(BinaryTreeNode<String> node1,
                                         BinaryTreeNode<String> node2) {
QueueADT<BinaryTreeNode<String>> queue1 = new LinkedQueue<BinaryTreeNode<String>>();
QueueADT<BinaryTreeNode<String>> queue2 = new LinkedQueue<BinaryTreeNode<String>>();
if(node1==null && node2==null) return(true);
else
if((node1==null && node2!=null) ||
   (node1!=null && node2==null)) return(false);

// at this point, node1 and node2 are non-null
queue1.enqueue(node1);
queue2.enqueue(node2);
while (!queue1.isEmpty() && !queue2.isEmpty())
    {
    BinaryTreeNode<String> q1 = queue1.dequeue();
    BinaryTreeNode<String> q2 = queue2.dequeue();
    if(!q1.getElement().equals(q2.getElement())) return(false);
    // check is one of the children is null and the other
    // is not null or vice versa
    if((q1.getLeft()==null && q2.getLeft()!=null) ||
       (q1.getLeft()!=null && q2.getLeft()==null)) return(false);
    if((q1.getRight()==null && q2.getRight()!=null) ||
       (q1.getRight()!=null && q2.getRight()==null)) return(false);
    // At this point left and right children either are both
    // non-null or are both null (and the elements are equal)
    if(q1.getLeft()!=null) // or q2.getLeft()!=null
       {
       queue1.enqueue(q1.getLeft());
       queue2.enqueue(q2.getLeft());
       }
    if(q1.getRight()!=null) // or q2.getRight()!=null
       {
       queue1.enqueue(q1.getRight());
       queue2.enqueue(q2.getRight());
       }
    }
if(!queue1.isEmpty() || !queue2.isEmpty()) return(false);
return(true); // got to the end, the trees must be state equivalent
}
```

(10a) continued:

(10b (15%) Consider the identical tree problem posed in Problem 9. Write an iterative (non-recursive) method using **stacks** to determine if two trees have identical structure and state equivalence for corresponding node values. Use the getters and setters described in the BinaryTreeNode class. The private variable element is a String, use String's equals to test for equality is necessary. Do **not** use Iterators.

```
public static boolean identicalTreeStack(BinaryTreeNode<String> node1,
                                         BinaryTreeNode<String> node2) {
StackADT<BinaryTreeNode<String>> stack1 = new LinkedStack<BinaryTreeNode<String>>();
StackADT<BinaryTreeNode<String>> stack2 = new LinkedStack<BinaryTreeNode<String>>();
if(node1==null && node2==null) return(true);
else
if((node1==null || node2!=null) ||
   (node1!=null || node2==null)) return(false);

// at this point, node1 and node2 are non-null
stack1.push(node1);
stack2.push(node2);
while (!stack1.isEmpty() && !stack2.isEmpty())
    {
    BinaryTreeNode<String> q1 = stack1.pop();
    BinaryTreeNode<String> q2 = stack2.pop();
    // put right references first, so that
    // the left references are popped first
    if(!q1.getElement().equals(q2.getElement())) return(false);
    // check is one of the children is null and the other
    // is not null or vice versa
    if((q1.getLeft()==null && q2.getLeft()!=null) ||
       (q1.getLeft()!=null && q2.getLeft()==null)) return(false);
    if((q1.getRight()==null && q2.getRight()!=null) ||
       (q1.getRight()!=null && q2.getRight()==null)) return(false);
    // At this point left and right children either are both
    // non-null or are both null (and the elements are equal)
    // Push the right children before the left children, so that
    // the left children are popped before the right children - Now the
    // nodes are processed in the same order as the queue solution
    if(q1.getRight()!=null) // or q2.getRight()!=null
       {
       stack1.push(q1.getRight());
       stack2.push(q2.getRight());
       }
    if(q1.getLeft()!=null) // or q2.getLeft()!=null
       {
       stack1.push(q1.getLeft());
       stack2.push(q2.getLeft());
       }
   }
if(!stack1.isEmpty() || !stack2.isEmpty()) return(false);
return(true);  // got to the end, the trees must be state equivalent
}
```

(10b) continued:

Interfaces and Classes

```
public interface StackADT<T>{
  // Adds one element to the top of this stack.
  public void push (T element);

  // Removes and returns the top element from this stack.
  public T pop();

  // Returns without removing the top element of this stack.
  public T peek();

  // Returns true if this stack contains no elements.
  public boolean isEmpty();

  // Returns the number of elements in this stack.
  public int size();

  // Returns a string representation of this stack.
  public String toString();
}

public interface QueueADT<T>{
   // Adds one element to the rear of this queue.
   public void enqueue (T element);

   // Removes and returns the element at the front of this queue.
   public T dequeue();

   // Returns without removing the element at the front of this queue.
   public T first();

   // Returns true if this queue contains no elements.
   public boolean isEmpty();

   // Returns the number of elements in this queue.
   public int size();

   // Returns a string representation of this queue
   Public String toString();
}
```

```
public interface ListADT<T> extends Iterable<T>{
  // Removes and returns the first element from this list.
  public T removeFirst();

  // Removes and returns the last element from this list.
  public T removeLast();

  // Removes and returns the specified element from this list.
  public T remove(T element);

  // Returns a reference to the first element in this list.
  public T first();

  // Returns a reference to the last element in this list.
  public T last();

  // Returns true if this list contains the specified target element.
  public boolean contains(T target);

  // Returns true if this list contains no elements.
  public boolean isEmpty();

  // Returns the number of elements in this list.
  public int size();

  // Returns an iterator for the elements in this list.
  public Iterator<T> iterator();

  // Returns a string representation of this list.
  public String toString();
}

public interface OrderedListADT<T> extends ListADT<T>
{
   /**
    * Adds the specified element to this list at the proper location
    *
    * @param element  the element to be added to this list
    */
   public void add (T element);
}
```

```
public interface UnorderedListADT<T> extends ListADT<T>
{
   //  Adds the specified element to the front of this list
   public void addToFront (T element);

   //  Adds the specified element to the rear of this list
   public void addToRear (T element);

   //  Adds the specified element after the specified target
   public void addAfter (T element, T target);
}


public interface Iterator<T>{
  // Returns true if the iterator has more elements.
  boolean hasNext();

  // Returns the next element in the iterator.
  T next();
}


public class BinaryTreeNode<T>{
   protected T element;
   protected BinaryTreeNode<T> left, right;

   // the getters you will need
   public T getElement();
   public BinaryTreeNode<T> getLeft();
   public BinaryTreeNode<T> getRight();
}
```

Rough work 1/4

Rough work 2/4

Rough work 3/4

Rough work 4/4