Final exam 2014, questions and answers

Introduction to Computing II (University of Ottawa)

Université d'Ottawa
Faculté de génie

École d'ingénierie et de
technologie de l'information

uOttawa

L'Université canadienne
Canada's university

University of Ottawa
Faculty of Engineering

School of Information
Technology and Engineering

Introduction to Computing II (ITI 1121)
Final Examination: Solutions

Instructor: Marcel Turcotte

April 2014, duration: 3 hours

Identification

Last name: _____ First name: _____

Student #: _____ Seat #: _____ Signature: _____ Section A or B

Instructions

1. This is a closed book examination.
2. No calculators, electronic devices or other aids are permitted.
   (a) Any electronic device or tool must be shut off, stored and out of reach.
   (b) Anyone who fails to comply with these regulations may be charged with academic fraud.
3. Write your answers in the space provided.
   (a) Use the back of pages if necessary.
   (b) You may not hand in additional pages.
4. Write comments and assumptions to get partial marks.
5. Do not remove the staple holding the examination pages together.
6. Beware, poor hand writing can affect grades.
7. Wait for the start of the examination.

Marking scheme

Question	Maximum	Result
1	10	
2	10	
3	5	
4	20	
5	15	
6	15	
7	15	
8	10	
Total	**100**	

Question 1 (10 marks)

A. Despite autoboxing, in Java version 1.5 and above, the following statement is invalid.
True or **False**

```
Float value = 1.6;
```

B. The execution of the method **foo** is significantly faster than that of **bar**.
True or **False**

```
public static void foo() {
    long s1 = (long) 0;
    for (int j=0; j <10000000; j++) {
        s1 = s1 + (long) 1;
    }
}
```

```
public static void bar() {
    Long s2 = (long) 0;
    for (int j=0; j <10000000; j++) {
        s2 = s2 + (long) 1;
    }
}
```

C. To declare the class **Dog** a subclass of **Animal**, one writes:
True or **False**

```
public class Dog implements Animal {

}
```

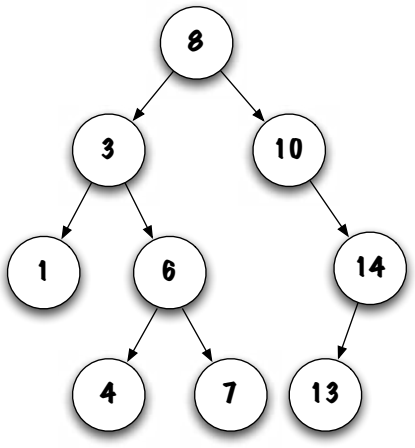
D. The **finally** block always executes when the **try** block exits, with or without exception.
True or **False**

E. The methods of an iterator for a singly linked list must have access to the implementation of the classes **Node** and **SinglyLinkedList**.
True or **False**

F. The value of the following (RPN) postfix expression is 14.
True or **False**

```
3  8  *  4  16  8  /  +  −
```

G. The following **Binary Search Tree** is balanced.
True or **False**



H. The **postorder** traversal of the above tree is 1, 4, 7, 6, 3, 13, 14, 10, and 8.
True or **False**

I. The simplest way to implement a stack is by using a circular array.
True or **False**

J. Dummy nodes can only be used in a singly linked list.
True or **False**

Question 2 (10 marks)

A. The Java program below will cause a compilation error because:

(a) There is no **catch** expression

(b) There is no **throws** statement

(c) Exception is an unchecked expression

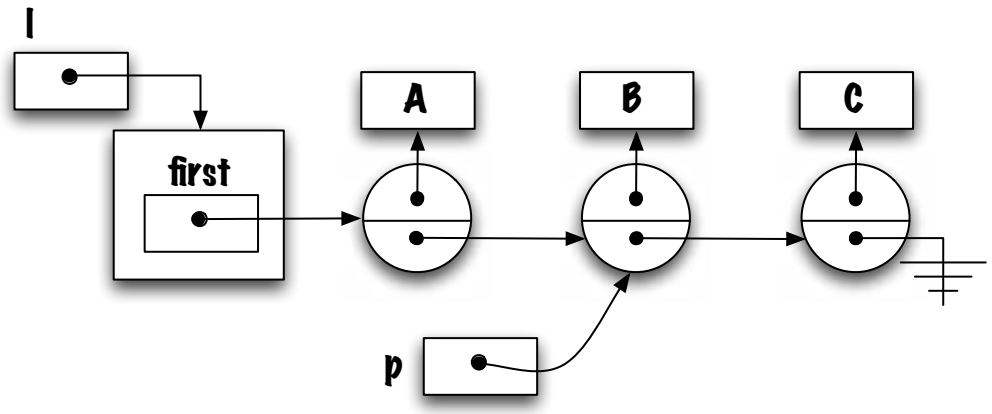
(d) [(a) or (b)]

(e) (a), (b) and (c)

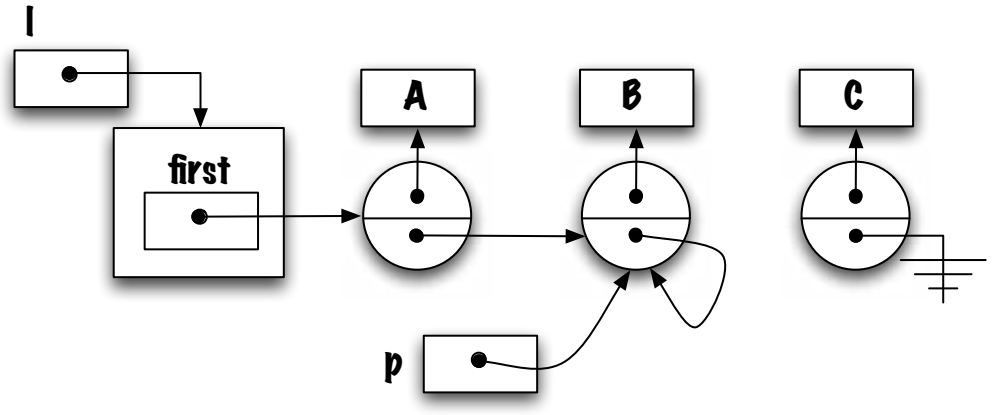
```
public class Test {
    public static void foo(int arg) {

        if (arg == -1) {
            throw new Exception( "an Exception" );
        }

    }
}
```

B. Modify the memory diagram below to represent the content of the memory after the execution of the following statement:

```
p.next = p;
```



Solution:



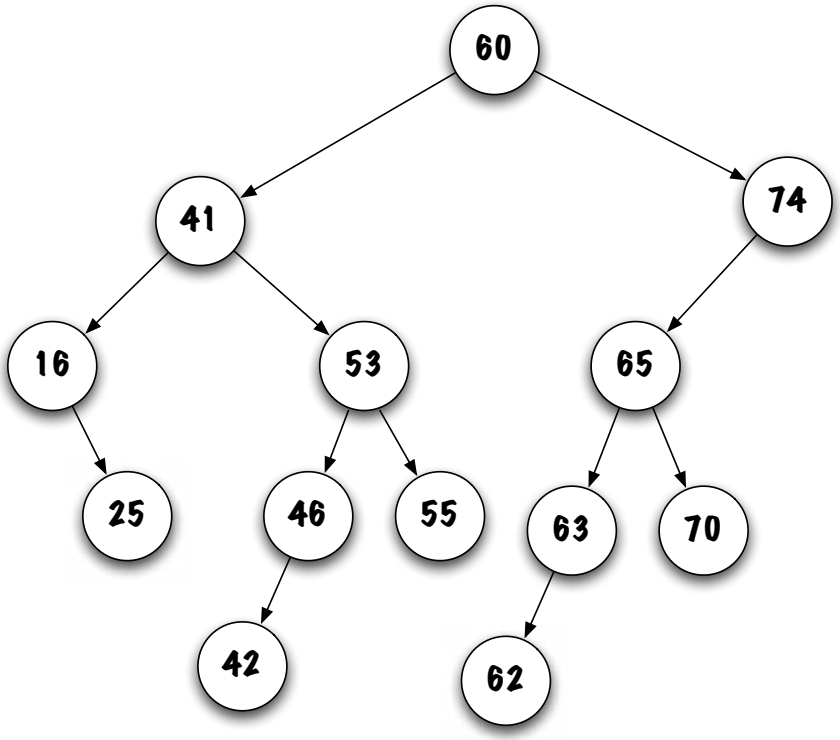
C. Which of the following operation(s) in a singly linked list can never take constant time?

(a) addFirst
(b) addLast
(c) removeFirst
(d) [get]
(e) (b) and (d)

D. Which of the following statements is incorrect.

(a) [Queues are needed to implement method calls in a virtual machine]
(b) Queues can be used to implement breadth-first search
(c) Queues are useful for asynchronous treatment
(d) Stacks can be used to implement depth-first search
(e) The evaluation of an expression in Reverse Polish Notation (RPN) is best done with a stack

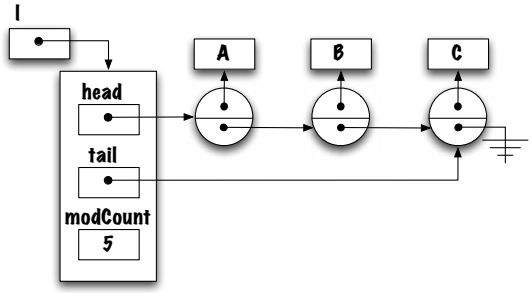
E. The **depth** of the following tree is.



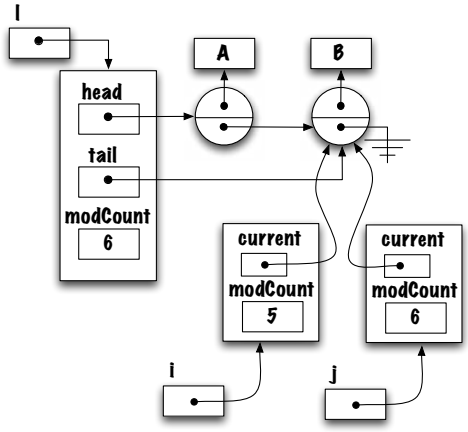
(a) 2
(b) [4]
(c) 6
(d) 13
(e) 14

Question 3 (5 marks)

This question is about the "fail-fast" implementation of an iterator seen in class, as well as Assignment #4. Given the following memory diagram:



Knowing that **i** and **j** are both of type **Iterator**, which of the following sequences of method calls yields the following memory diagram:



(a)

```
j = l.iterator(); i = l.iterator();
i.next(); i.next(); i.next(); i.remove();
j.next(); j.next();
```

(b)

```
i = l.iterator(); j = l.iterator();
j.next(); j.next(); j.next(); j.remove();
i.next(); i.next();
```

(c)

```
i = l.iterator(); i.next(); i.next();
j = l.iterator(); j.next(); j.next(); j.next(); j.remove();
```

(d)

```
j = l.iterator(); j.next(); j.next(); j.next(); j.remove();
i = l.iterator(); i.next(); i.next();
```

(e)

```
j = l.iterator(); j.next(); j.next(); j.next(); j.remove();
i = j;
```

Question 4 (20 marks)

Write a class, named **Interval**, to represent the set of all points on a line greater than or equal to left and smaller than or equal to right.

- The class **Interval** has exactly one constructor. This constructor has two parameters, the left and right values of this interval.
- The class has two getters, **getLeft** and **getRight**, returning the left and right values of this interval, respectively.
- The method **contains** returns **true** if and only if the value of the parameter is a point that belongs to the interval.
- The method **intersect** returns **true** if and only if this interval and that of the interval designated by the parameter of the method intersect. In other words, these two intervals have points in common.
- Finally, the execution of the test program below, **IntervalTest**, should produce the following result.

```
Interval: {12.0,20.0}
i1.getLeft() -> 5.0
i2.getRight() -> 8.0
i1.contains(2.0) -> false
i2.contains(5.0) -> true
i1.intersect(i2) -> true
i1.intersect(i3) -> false
caught IllegalArgumentException: left (10.0) is larger than right (5.0)
```

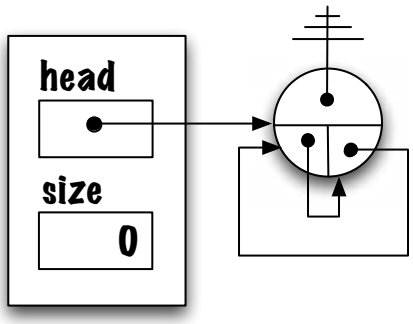
```
public class IntervalTest {
  public static void main(String args[]) {
    Interval i1, i2, i3, i4;

    i1 = new Interval(5.0, 10.0);
    i2 = new Interval(4.0, 8.0);
    i3 = new Interval(12.0, 20.0);

    System.out.println(i3);
    System.out.println("i1.getLeft() -> " + i1.getLeft());
    System.out.println("i2.getRight() -> " + i2.getRight());
    System.out.println("i1.contains(2.0) -> " + i1.contains(2.0));
    System.out.println("i2.contains(5.0) -> " + i2.contains(5.0));
    System.out.println("i1.intersect(i2) -> " + i1.intersect(i2));
    System.out.println("i1.intersect(i3) -> " + i1.intersect(i3));

    try {
        i4 = new Interval(10.0, 5.0);
        System.out.println("i4.getRight() -> " + i4.getRight());
    } catch (IllegalArgumentException e) {
        System.out.println("caught IllegalArgumentException: " + e.getMessage());
    }

  }
}
```

Solution:

```
public class Interval {

    private fnal double left;
    private fnal double right;

    public Interval(double left, double right) {
        if (left > right) {
            throw new IllegalArgumentException("left is larger than right");
        }
        this.left = left;
        this.right = right;
    }

    public double getLeft() {
        return left;
    }

    public double getRight() {
        return right;
    }

    public boolean contains(double v) {
        return v >= left && v <= right;
    }

    public boolean intersect(Interval other) {
        return other != null && this.right >= other.left && this.left <= other.right;
    }

    public String toString() {
        return "Interval: {" + left + "," + right + "}";
    }
}
```

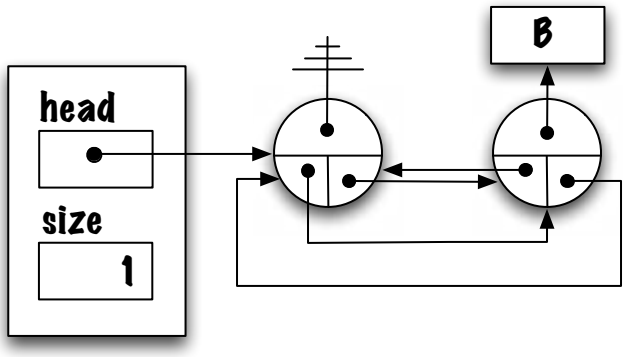
Question 5 (15 marks)

A. Complete the implementation of the class **LinkedList** below by filling the two rectangles so that when creating a **LinkedList** object this memory diagram is produced:




```
public class LinkedList<E> {

    private static class Node<T> {

        private T value;
        private Node<T> previous;
        private Node<T> next;

        private Node( T value, Node<T> previous, Node<T> next ) {
            this.value = value;
            this.previous = previous;
            this.next = next;
        }
    }

    private Node<E> head;
    private int size;

    public LinkedList() {

        head = new Node<E>(null, null, null);
        head.next = head;
        head.previous = head;
        size = 0;

    }

}
```

B. For the class **LinkedList** on the previous page, complete the implementation of the method **addLast**. Adding an element when the list is empty should produce this memory diagram:




```
public void addLast( E elem ) {
```

```
if (elem == null) {
    throw new NullPointerException();
}
Node<E> before, after;
before = head.previous;
after = before.next;
before.next = new Node<E>(elem,before,after);
after.previous = before.next;
size++;
```

```
}
```

Question 6 (15 marks)

For this question, a circular array is used to implement a queue. You must implement a special method **dequeue** that takes one parameter (an integer) specifying how many elements need to be removed from the queue. The method **dequeue** returns a reference to a list containing all the elements that have been removed, in reverse of order of removal.

For this question, you are allowed to use the predefined class **LinkedList**. In particular, it has a constructor **LinkedList()**, stores an arbitrarily large number of elements, and implements the following methods.

```
public interface List<E> {
    // Add the element at the specified position of the list
    public abstract boolean add(int pos, E o);
    // Returns the element at the specified position in this list.
    public abstract E get(int index);
    // Returns the number of elements in this list
    public abstract int size();
}
```

- For your implementation of the method **dequeue**, you cannot use the methods of the class **CircularQueue**, accordingly these methods are not shown on the next page. Your code needs to manipulate directly the instance variables, **elems**, **front**, **rear**, and **size**.

The execution of the Java program below displays "[2,1,0]".

```
CircularQueue<Integer> q;
q = new CircularQueue<Integer>(100);

for (int i=0; i<8; i++) {
    q.enqueue(i);
}

List<Integer> l;
l = q.dequeue(3);

System.out.print("[");
for (int i=0; i<l.size(); i++) {
    if (i>0) {
        System.out.print(",");
    }
    System.out.print(l.get(i));
}
System.out.println("]");
```

```
import java.util.List;
import java.util.LinkedList;

public class CircularQueue<E> {
    private E[] elems;
    private int front, rear, size;

    public CircularQueue(int capacity) {
        if (capacity < 0) {
            throw new IllegalArgumentException("negative number");
        }
        elems = (E[]) new Object[capacity];
        front = 0; // front must be 0 whenever the queue is empty
        rear = -1; // rear must be -1 whenever the queue is empty
    }

    // Solution:

    public List<E> dequeue(int num) {
        if (num<0 || num>size) {
            throw new IllegalArgumentException("larger than size");
        }

        List<E> xs;
        xs = new LinkedList<E>();

        while (num > 0) {
            xs.add(0, elems[front]);
            elems[front] = null;
            front = (front + 1) % elems.length;
            num--;
            size--;
        }

        if (size==0) {
            front = 0;
            rear = -1;
        }

        return xs;
    }









} // End of CircularQueue
```

Question 7 (15 marks)

Write a method named **frequency** that takes as input a list of objects containing a **character** value and a **boolean** flag initially set to **false**, and prints the frequency of each character of the list. Every time an element is **counted**, its flag is set to **true** so that it is not printed or counted a second time.

```
List<Tuple> l;
l = new LinkedList<Tuple>();

l.add(new Tuple('a')); l.add(new Tuple('b')); l.add(new Tuple('a'));
l.add(new Tuple('c')); l.add(new Tuple('b')); l.add(new Tuple('a'));
l.add(new Tuple('c')); l.add(new Tuple('a')); l.add(new Tuple('d'));
l.add(new Tuple('d')); l.add(new Tuple('b'));

Frequency.frequency(l);
```

Executing the above program produces the following output "a : 4, b : 3, c : 2, d : 2". Below you will find a schematic representation of the list for the execution of the above program. Each set of parentheses contains a character and boolean value. Here **t** and **f** are used to represent **true** and **false**, respectively. This is the list before the execution of the program.

`(a,f)->(b,f)->(a,f)->(c,f)->(b,f)->(a,f)->(c,f)->(a,f)->(d,f)->(d,f)->(b,f)`

The method **frequency** will first display **a : 4**. The list will have been transformed as follows.

`(a,t)->(b,f)->(a,t)->(c,f)->(b,f)->(a,t)->(c,f)->(a,t)->(d,f)->(d,f)->(b,f)`

Next, the method displays **b : 3**. The list will have been transformed as follows.

`(a,t)->(b,t)->(a,t)->(c,f)->(b,t)->(a,t)->(c,f)->(a,t)->(d,f)->(d,f)->(b,t)`

Next, the method displays **c : 2**. The list will have been transformed as follows.

`(a,t)->(b,t)->(a,t)->(c,t)->(b,t)->(a,t)->(c,t)->(a,t)->(d,f)->(d,f)->(b,t)`

Finally, the method displays **d : 2**. The list will have been transformed as follows.

`(a,t)->(b,t)->(a,t)->(c,t)->(b,t)->(a,t)->(c,t)->(a,t)->(d,t)->(d,t)->(b,t)`

Your implementation must comply with the following directives:

- You need to use iterators to traverse the list. In fact, the only method of the **List** that you can use is the method **iterator**, which returns an iterator on the list.
- The frequency table should not be saved, just printed. In particular, you cannot use arrays, lists, stacks or queues to store counts. The counts are simply printed.
- You will find the source code for the class **Tuple** and the interface **Iterator** on page 16.

```
public class Frequency {

    public static void frequency(List<Tuple> l) {

        // Solution using two iterators:

        if (l == null) {
            return;
        }

        Iterator<Tuple> i1;
        i1 = l.iterator();

        while (i1.hasNext()) {
            Tuple t1 = i1.next();

            if (! t1.visited() ) {
                int count = 1;
                t1.toggle();

                Iterator<Tuple> i2 = l.iterator();
                while (i2.hasNext()) {
                    Tuple t2 = i2.next();
                    if (! t2.visited() && t2.getChar() == t1.getChar()) {
                        t2.toggle();
                        count++;
                    }
                }

                System.out.println(t1.getChar()+" -> " + count);
            }

        }

    } // End of frequency

} // End of Frequency
```

```
// alternative solution

public class Frequency {

    public static void frequency(List<Tuple> l) {

        // Solution using one iterator:

        if (l == null) {
            return;
        }

        boolean done = false;

        while (!done) {
            Iterator<Tuple> i;
            i = l.iterator();

            char current = 0;
            int count = 0;

            while (i.hasNext()) {
                Tuple t = i.next();
                if (count == 0) {
                    if (!t.visited()) {
                        current = t.getChar();
                        t.toggle();
                        count++;
                    }
                } else if (t.getChar() == current) {
                    t.toggle();
                    count++;
                }
            }

            if (count == 0) {
                done = true;
            } else {
                System.out.println(current + " -> " + count);
            }

        }
    } // End of frequency
} // End of Frequency
```

Objects of the class **Tuple** are used to store a **character** and a **boolean**. The value of the **boolean** is initially **false**. The method **toggle** is used to invert the value of **visited**.

```
public class Tuple {

    private char c;
    private boolean visited;

    public Tuple(char c) {
        this.c = c;
        visited = false;
    }

    public void toggle() {
        visited = ! visited;
    }

    public boolean visited() {
        return visited;
    }

    public char getChar() {
        return c;
    }

    public String toString() {
        if (visited) {
            return "(" + c + ",t)";
        } else {
            return "(" + c + ",f)";
        }
    }
}
```

This question is about the abstract data type **List** and iterators. The interface **List** declares a method named **iterator**, which has a return value of type **Iterator**. The interface **Iterator** declares the following methods;

```
public interface Iterator<E> {

    // Returns true if the iteration has more elements.
    public abstract boolean hasNext();

    // Returns the next element in the iteration.
    public abstract E next();

}
```

Question 8 (10 marks)

Implement the method **int count(E low, E high)** for the binary search tree presented in class. The method returns the number of elements in the tree that are greater than or equal to **low** and smaller than or equal to **high**.

- The elements stored in a binary search tree implement the interface **Comparable<E>**. Recall that the method **int compareTo(E other)** returns a negative integer, zero, or a positive integer as the instance is less than, equal to, or greater than the specified object.
- A method that is visiting too many nodes will get a maximum of 9 marks.
- Given a binary search tree, **t**, containing the values **1, 2, 3, 4, 5, 6, 7, 8**, the call **t.count(3,6)** returns the value **4**.

```
public class BinarySearchTree<E extends Comparable<E> > {

    private static class Node<T> {

        private T value;

        private Node<T> left;
        private Node<T> right;

        private Node( T value ) {
            this.value = value;
            left = null;
            right = null;
        }
    }

    private Node<E> root = null;

    public int count(E low, E high) {

        // Solution:

        if (low == null || high == null) {
            throw new NullPointerException();
        }

        if (low.compareTo(high)>0) {
            throw new IllegalArgumentException();
        }

        return count(root, low, high);









    } // End of count

    // BinarySearchTree continues on the next page...
```

```
    // Solution:

    private int count(Node<E> current, E low, E high) {
        int count = 0;
        if (current != null) {
            boolean greather = current.value.compareTo(low)>=0;
            boolean less = current.value.compareTo(high)<=0;
            if (greather && less) {
                count++;
            }
            if (current.left != null && greather) {
                count = count + count(current.left, low, high);
            }
            if (current.right != null && less) {
                count = count + count(current.right, low, high);
            }
        }
        return count;
    } // End of count


    // Non-recursive solution. Here, a stack is used to keep track of the unprocessed
    // (opened) nodes. The stack could be replaced by a queue since the order for
    // visiting the nodes is not important. Here, I am using java.util.Stack.

    public int count(E low, E high) {
        if (low == null || high == null) {
            throw new NullPointerException();
        }
        if (low.compareTo(high)>0) {
            throw new IllegalArgumentException();
        }
        int count = 0;
        if (root != null) {
            Stack<Node<E>> open = new Stack<Node<E>>();
            open.push(root);
            while (! open.isEmpty()) {
                Node<E> current = open.pop();
                boolean greather = current.value.compareTo(low)>=0;
                boolean less = current.value.compareTo(high)<=0;
                if (greather && less) {
                    count++;
                }
                if (current.left != null && greather) {
                    open.push(current.left);
                }
                if (current.right != null && less) {
                    open.push(current.right);
                }
            }
        }
        return count;
    } // End of count

} // End of BinarySearchTree
```