A4 Disease. Writing and Testing Methods

1. Introduction

This assignment is more complex than A3, for a few reasons. First, the methods are more complicated. Some methods are not only recursive but will have a loop, e.g. to process children of root of the DiseaseTree. Second, it takes more thought to create the trees to be tested –the test-case trees. Third, testing and debugging is harder. In A3, it was easy to test all fields, using toString(), toStringRev, and size(). Here, it is harder. One could use DiseaseTree.toString(), but there the difficulty of manually constructing the String that you expect that function to produce.

This document hopes to help you through the method writing, testing, and debugging.

2. Fields of DiseaseTree

DiseaseTree has a few constants (static, final, public variables), which you don't have to deal with, and only two fields.

(1) root: the Person who is the root of the tree. Note that all people have different names. That means you can distinguish people by their names.
(2) children: the Set of the children of the root —could be 0, 1, 2, 3, … any number of children. Look at the constructor to see that this field will contain a HashSet.

3. The methods of this class

It's important that you have a good understanding of the methods that are already written, as well as those you have to write, because you may be able to *use* already written ones when you write yours. Seeing a list of them may help. To the left below is a list of already written methods; to the right are the methods you have to write, in the order to be written. *Study their specifications.* Learn *what* they do, not *how* they do it. In the lists, we abbreviate "DiseaseTree" by "DT" to save space.

DT(DT) and DT (Person)	1 and 2. add(Person, Person) and size()
contains(Person)	3. depthOf(Person)
addToWidthMap(int, HashMap<Integer, Integer>)	4. widthAtDepth(int)
addToWidths(int, int[])	5. getDiseaseRouteTo(Person)
getChildren() and getChildrenCount()	6. getSharedAncestorOf(Person, Person)
getTree(Person) and getParentOf(Person)	7. equals(Object)
getRoot()	
hashCode()	
maxDepth() and maxWidth()	
toString() and toStringVerbose() and toStringVerbose(int)	

Now, it is a good idea to look at a few of the recursive method bodies in order to gain some familiarity with how these recursive methods are written. Look at contains(…) and getTree(…), for example.

You probably have to implement add(Person, Person) and size() together and hen test/debug them.

For the rest of the ones you have to do, it is best to do one at a time and test/debug it.

4. Setting up a JUnit testing class

The single-parameter constructor of DiseaseTree has a Person for a parameter. The Person constructor requires a String (person's name), a Network (which you will not use in testing), and the person's health (you can use 0). We suggest you use the following code in your JUnit testing class. It introduces two new things to you: static variables in the JUnit testing class and the "@BeforeClass" annotation.

```
private static Network n;
private static Person[] people;
```

```
@BeforeClass
public static void setup() {
  Network n= new Network();
  people= new Person[]{
        new Person("A", n, 0), new Person("B", n, 0), new Person("C", n, 0),
        new Person("D", n, 0), new Person("E", n, 0), new Person("F", n, 0),
        new Person("G", n, 0), new Person("H", n, 0), new Person("I", n, 0),
        new Person("J", n, 0), new Person("K", n, 0), new Person("L", n, 0)
  };
    }
```

You will not use a Network —the health simulation uses that, so we create it in the JUnit testing class but we never change it. @BeforeClass means that this method will be called first, and we want that done to set up the Network and an array of people to use in testing. Note that these static variables won't be changed.

Don't use static variables in a JUnit testing class to pass variables that change among different testing procedures. You do not know the order in which testing procedures will be called.

5. One way to test add

The usual way to test a method like add is to call add and then test that fields are correct. As with A3 DlinkedList, it gets complicated to determine what fields to test and how to test them. Here are some tests you might trey. Note how you call various methods which you know already work:

```
//Test add to root
DiseaseTree dt2= dt.add(people[0], people[1]);
assertEquals(people[1], dt2.getRoot());
assertEquals(1, dt.getChildrenCount());
assertEquals(0, dt2.getChildrenCount());
assertTrue(dt.getChildren().contains(dt2));
assertTrue(dt2.getChildren().isEmpty());

//Test add to non-root
DiseaseTree dt3= dt.add(people[1], people[2]);
assertEquals(people[2], dt3.getRoot());
assertEquals(1, dt.getChildrenCount());
assertEquals(1, dt2.getChildrenCount());
assertEquals(0, dt3.getChildrenCount());
assertTrue(dt.getChildren().contains(dt2));
assertTrue(dt2.getChildren().contains(dt3));
assertTrue(dt3.getChildren().isEmpty());

//Test add second child
DiseaseTree dt4= dt.add(people[0], people[3]);
assertEquals(people[3], dt4.getRoot());
```

5. A more systematic way to test

It would be nice to have a way to test where we don't have to think about which fields to test but just test all of them, as we did with DLInkedList. That could mean having a function that gave back a string representation that would be easy for us to produce manually. A problem is here is because the children are in a Set variable, there is no ordering of the children, and it is therefore difficult to produce a canonical representation of the children, perhaps in order of the names of their roots.

A4 Disease. Writing and Testing Methods

Unfortunately, classes Person and DiseaseTree were not set up to make this easy! There is no simple way simple way in the testing class to produce a string representation of a tree with the children in a sorted order without writing our own sort routine —which we will show you below.

The main point to take away from this is that when one is designing a large, complicated, class, one should consider how it will be tested. Testing should be thought of before hand and built into the class.

So, in our testing class, we write a new function to produce a very compact representation of a tree, one that one can manually construct fairly easily, in order to test. This will require a sorting procedure; we use a version of selection sort. These are at the end of this document; you can copy them into your JUnit testing class. With those methods, here is how we would begin testing add(Person). Note that we test adding at level 0, first, then at other levels.

```
@Test
public void testOneNodeTree() {
    DiseaseTree t1= new DiseaseTree(people[0]);
    assertEquals("A", toStringBrief(t1));
}

@Test
public void testAddAtLevel0() {
    DiseaseTree t1= new DiseaseTree(people[0]);
    t1.add(people[0], people[1]);
    assertEquals("A[B]", toStringBrief(t1));

    t1.add(people[0], people[2]);
    assertEquals("A[B C]", toStringBrief(t1));

    t1.add(people[0], people[3]);
    assertEquals("A[B C D]", toStringBrief(t1));
}

@Test
public void testAddAtLevel1() {
    DiseaseTree t1= new DiseaseTree(people[0]);
    t1.add(people[0], people[1]);
    t1.add(people[0], people[2]);
    t1.add(people[0], people[3]);

    t1.add(people[1], people[5]);
    assertEquals("A[B[F] C D]", toStringBrief(t1));

    t1.add(people[1], people[4]);
    assertEquals("A[B[E F] C D]", toStringBrief(t1));

    t1.add(people[4], people[6]);
    assertEquals("A[B[E[G] F] C D]", toStringBrief(t1));

    t1.add(people[0], people[7]);
    assertEquals("A[B[E[G] F] C D H]", toStringBrief(t1))
}
```

```
/** Return a representation of this tree. This representation is:
 *  (1) the name of the person at the root, followed by
 * (2) followed by the representations of the children.
 * There are two cases concerning the children.
 *
 * No children? Their representation is the empty string.
 * Children? Their representation is the representation of each child, with
 * a blank between adjacent ones, and delimited by "[" and "]".
 * Examples:
 * One-node tree:  "A"
 * A root A with children B, C,D:   "A[B C D]"
 * A root A with children B, C, D and B has a child F: "A[B[F] C D]"
 */
 public String toStringBrief(DiseaseTree t) {
     String res= t.getRoot().getName();

     Object[] childs= t.getChildren().toArray();
     if (childs.length == 0) return res;
     res= res + "[";
     selectionSort1(childs);

     for (int k= 0; k < childs.length; k= k+1) {
         if (k > 0) res= res + " ";
         res= res + toStringBrief(((DiseaseTree)childs[k]));
     }
     return res + "]";

 }

 /** Sort b --put its elements in ascending order.
  * Sort on the name of the person at the root of the DiseaseTree
  * Throw cast-class exception of b's elements are not DiseaseTree */
 public static void selectionSort1(Object[] b) {
     int j= 0;
     // {inv P: b[0..j-1] is sorted and b[0..j-1] <= b[j..]}

     //        0---------------j--------------- b.length
     // inv : b |  sorted, <=   |   >=         |
     //        --------------------------------

     while (j != b.length) {
         // Put into p the index of smallest element in b[j..]
         int p= j;
         for (int i= j+1; i != b.length; i++) {
             String bi= ((DiseaseTree)b[i]).getRoot().getName();
             String bp= ((DiseaseTree)b[p]).getRoot().getName();
             if (bi.compareTo(bp) < 0)  {
                 p= i;

             }
         }

         // Swap b[j] and b[p]
         Object t= b[j]; b[j]= b[p]; b[p]= t;
         j= j+1;
     }
 }
```