COMP250 Assignment 5

Adam Hooper <adam.hooper@mail.mcgill.ca>

Question 1

We were asked to implement a class which does a Heapsort of an array. The class notes outlined the requisite algorithms very clearly; they are implemented in `Sorter.java`.

```
import java.util.Comparator;
import java.util.Arrays;
import java.util.List;

public class Sorter
{
        private int heapSize;
        private Object[] a;
        private Comparator c;

        public static void sort (Object[] a, Comparator c)
        {
                Sorter o = new Sorter (a, c);
                o.heapSort ();
        }

        public static void sort (List L, Comparator c)
        {
                Object[] a;

                a = L.toArray ();
                sort (a, c);

                L.clear ();
                L.addAll (Arrays.asList (a));
        }

        private Sorter (Object[] a, Comparator c)
        {
                this.a = a;
                this.c = c;
        }

        private void heapSort ()
        {
                buildHeap ();

                for (int i = heapSize - 1; i >= 1; i--)
                {
                        exchange (0, i);
                        heapSize--;
                        heapify (0);
                }
        }

        private int parent (int i)
        {
                return (i + 1) / 2 - 1;
        }

        private int left (int i)
```

```
        {
                return 2 * (i + 1) - 1;
        }

        private int right (int i)
        {
                return 2 * (i + 1);
        }

        private void exchange (int i, int j)
        {
                Object t = a[i];
                a[i] = a[j];
                a[j] = t;
        }

        /*
         * Iterates up heap (after inserting an element at i) to keep it a heap
         */
        private void heapify (int i)
        {
                int l = left (i);
                int r = right (i);
                int largest;

                largest = i;

                if (l < heapSize && c.compare (a[l], a[largest]) > 0)
                {
                        largest = l;
                }

                if (r < heapSize && c.compare (a[r], a[largest]) > 0)
                {
                        largest = r;
                }

                if (largest != i)
                {
                        exchange (i, largest);
                        heapify (largest);
                }
        }

        /*
         * Makes the array a into a heap
         */
        private void buildHeap ()
        {
                heapSize = a.length;

                for (int i = heapSize / 2; i >= 0; i--)
                {
                        heapify (i);
                }
        }
}
```

I wrote a simple test program in `TestHeapsort.java` which sorts all the Strings passed on the command line.

```
import java.util.Comparator;

public class TestHeapsort
```

```
{
        public static void main (String[] a)
        {
                if (a.length == 0)
                {
                        System.err.println ("usage: java TestHeapsort "
                                            + "[string1] [string1] ...");
                        return;
                }

                Sorter.sort (a, String.CASE_INSENSITIVE_ORDER);

                for (int i = 0; i < a.length; i++)
                {
                        System.out.println (a[i]);
                }
        }
}
```

Here is a sample run of the test:

```
$ java TestHeapsort a 8 five seven two l pep dee whee pep a
8
a
a
dee
five
l
pep
pep
seven
two
whee
```

Question 2

Question 2 was certainly more difficult. We were asked to determine if two trees from a file are isomorphic. The algorithms were all given (with small mistakes often) so all we had to do was understand them and implement them in Java. Much easier said than done....

I made a `Vertex` class to represent tree nodes:

```
import java.util.LinkedList;
import java.util.Iterator;

public class Vertex
{
        private String label;
        protected Vertex parent;
        protected LinkedList children;
        private int distance = -1;
        protected int orderLabel = -1;

        public Vertex (String label, Vertex parent)
        {
                this.label = label;
                this.parent = parent;
                this.children = new LinkedList ();

                if (parent != null)
                {
```

```
                parent.children.add (this);
        }
}

/*
 * Returns the height of the tree rooted at this Vertex
 *
 * A tree with 1 Vertex has height 1 -- 2 Vertices has height 2.
 * Any more will be >= 2
 */
public int getHeight ()
{
        Iterator i;
        LinkedList curList;
        LinkedList nextList;
        Vertex v;
        int height = 0;

        curList = new LinkedList ();
        curList.add (this);

        while (true)
        {
                height++;

                nextList = new LinkedList ();

                for (i = curList.iterator (); i.hasNext (); )
                {
                        v = (Vertex) i.next ();

                        nextList.addAll (v.children);
                }

                if (nextList.isEmpty ()) break;

                curList = nextList;
        }

        return height;
}

/*
 * Returns the depth of the Vertex
 *
 * The root Vertex has depth 0. Any lower will be >= 1
 */
public int getDepth ()
{
        int depth = 0;
        Vertex v = this;

        while (v.parent != null)
        {
                v = v.parent;
                depth++;
        }

        return depth;
}

/*
 * Implements first half of RootedTree.calculateDistanceFromLeaves ()
 *
```

```
 * calculates the distance from this Vertex to the closest null below
 */
protected void calculateDistanceFromLeaves_children ()
{
        Vertex child;
        int t;

        distance = 0;

        if (children.isEmpty ())
        {
                return;
        }

        // Calculate (minimum distance of a child) + 1
        for (Iterator i = children.iterator (); i.hasNext ();)
        {
                child = (Vertex) i.next ();

                child.calculateDistanceFromLeaves_children ();

                t = child.distance + 1;

                if (distance == 0 || t < distance)
                {
                        distance = t;
                }
        }
}

/*
 * Implements second half of RootedTree.calculateDistanceFromLeaves ()
 *
 * If the parent is closer to null than children, recalculate children
 */
protected void calculateDistanceFromLeaves_parent ()
{
        Vertex child;
        int t;

        if (parent == null && children.isEmpty ())
        {
                distance = 0;
        }

        if (parent != null)
        {
                t = parent.distance + 1;

                if (t < distance)
                {
                        distance = t;
                }
        }

        for (Iterator i = children.iterator (); i.hasNext (); )
        {
                child = (Vertex) i.next ();

                child.calculateDistanceFromLeaves_parent ();
        }
}

/*
```

```
         * Returns a string representing the rooted tree at this Vertex
         */
        public String toString ()
        {
                Iterator i;
                Vertex v;
                String ret;

                ret = getLabel ();

                for (i = children.iterator (); i.hasNext (); )
                {
                        v = (Vertex) i.next ();
                        ret += "(" + v.toString () + ")";
                }

                return ret;
        }

        protected int getDistance ()
        {
                return distance;
        }

        public String getLabel ()
        {
                return this.label;
        }
}
```

Next, a `Map` class to compile and print the answer when we get it:

```
import java.util.LinkedList;
import java.util.Iterator;

public class Map
{
        private LinkedList map = new LinkedList ();

        /*
         * Declare 2 Vertices to be isomorphic
         */
        public void add (Vertex a, Vertex b)
        {
                Vertex[] n = new Vertex[2];
                n[0] = a;
                n[1] = b;

                map.add (n);
        }

        /*
         * Return a string list of all isomorphic relations
         */
        public String toString ()
        {
                String r = null;
                Vertex[] entry;
                Vertex a;
                Vertex b;

                for (Iterator i = map.iterator (); i.hasNext (); )
                {
```

```
                        entry = (Vertex[]) i.next ();
                        a = entry[0];
                        b = entry[1];

                        if (r == null)
                        {
                                r = "{";
                        }
                        else
                        {
                                r += ", ";
                        }

                        r += "(" + a.getLabel () + ", " + b.getLabel () + ")";
                }

                r += "}";

                return r;
        }
}
```

Of course, the main problem was the `RootedTree` class, which implements the algorithms given.

```
import java.util.LinkedList;
import java.util.Iterator;
import java.util.Comparator;

public class RootedTree
{
        private LinkedList vertices;
        private Vertex root;

        public RootedTree (String representation)
        {
                vertices = new LinkedList ();
                buildFromRepresentation (representation);
        }

        /*
         * Populates this RootedTree from a given string
         * e.g., "a(b(c))(d(e)(f))" -->
         *           a
         *          / \
         *         b   d
         *        /   / \
         *       c   e   f
         */
        private void buildFromRepresentation (String repr)
        {
                int nextLeftParen;
                int nextRightParen;
                int nextParen;
                Vertex parent = null;
                Vertex child = null;

                while (repr.length () > 0)
                {
                        nextLeftParen = repr.indexOf ('(');
                        nextRightParen = repr.indexOf (')');

                        if (nextLeftParen >= 0
                            && nextLeftParen < nextRightParen)
```

```
                {
                        nextParen = nextLeftParen;
                }
                else
                {
                        nextParen = nextRightParen;
                }

                // handle single-node tree
                if (nextParen == -1)
                {
                        root = new Vertex (repr, null);
                        vertices.add (root);
                        return;
                }

                if (nextParen != 0)
                {
                        child = new Vertex
                                (repr.substring (0, nextParen),
                                 parent);

                        vertices.add (child);
                }

                if (nextParen == nextLeftParen)
                {
                        // Go down a level
                        parent = child;
                        child = null;
                }
                else // nextParen == nextRightParen
                {
                        // Return up a level
                        child = parent;
                        parent = parent.parent;
                }

                repr = repr.substring (nextParen + 1);
        }

        root = child;
}

public String toString ()
{
        return root.toString ();
}

/*
 * Populates all vertices' "distance"
 */
private void calculateDistanceFromLeaves ()
{
        if (root == null) return;

        root.calculateDistanceFromLeaves_children ();
        root.calculateDistanceFromLeaves_parent ();
}

/*
 * Returns 1 or 2 center vertices (farthest from any edge)
 */
private Vertex[] findCenter ()
```

```
{
        Iterator i;
        Vertex v;
        Vertex v1 = null;
        Vertex v2 = null;
        Vertex[] ret;
        int maxDistance = -1;

        if (root == null) return null;

        calculateDistanceFromLeaves ();

        for (i = vertices.iterator (); i.hasNext ();)
        {
                v = (Vertex) i.next ();

                if (v.getDistance () == maxDistance)
                {
                        v2 = v;
                }

                if (v.getDistance () > maxDistance)
                {
                        maxDistance = v.getDistance ();
                        v1 = v;
                }
        }

        if (v1 != null && v2 != null
            && v1.getDistance () == v2.getDistance ()) // two root nodes
        {
                ret = new Vertex[2];
                ret[0] = v1;
                ret[1] = v2;
        }
        else
        {
                ret = new Vertex[1];
                ret[0] = v1;
        }

        return ret;
}

/*
 * Makes v the root of this tree
 */
private void reRoot (Vertex v)
{
        Vertex p, g;

        if (v == null) return;

        root = v;

        p = v.parent;
        v.parent = null;

        while (p != null)
        {
                p.children.remove (v);
                v.children.add (p);

                g = p.parent;
```

```
                p.parent = v;

                v = p;
                p = g;
        }
}

/*
 * Zeroes all vertices' order labels
 */
private void initializeVertices ()
{
        Iterator i;
        Vertex v;

        for (i = vertices.iterator (); i.hasNext (); )
        {
                v = (Vertex) i.next ();

                v.orderLabel = 0;
        }
}

/*
 * Returns a LinkedList of all vertices at depth d
 */
private LinkedList verticesAtDepth (int d)
{
        Iterator i;
        Vertex v;
        LinkedList curList, nextList;

        curList = new LinkedList ();
        curList.add (root);

        while (d-- > 0)
        {
                nextList = new LinkedList ();

                for (i = curList.iterator (); i.hasNext (); )
                {
                        v = (Vertex) i.next ();

                        nextList.addAll (v.children);
                }

                curList = nextList;
        }

        return curList;
}

/*
 * Sets the order labels on all vertices at the given depth.
 * Returns the vertices at that depth
 *
 * This function also rearranges the vertices below this depth
 */
private LinkedList labelLevel (int depth)
{
        LinkedList ret;
        Iterator i;
        Vertex v;
        Vertex prevV;
```

```
        int newLabel;
        Vertex[] sortVertices;
        Comparator comp;

        ret = verticesAtDepth (depth);

        // sort the children of every vertex at this depth
        comp = new ComparatorByVertexOrderLabels ();

        for (i = ret.iterator (); i.hasNext (); )
        {
                v = (Vertex) i.next ();

                Sorter.sort (v.children, comp);
        }

        // sort the whole list at this depth
        comp = new ComparatorByVertexChildOrderLabels ();

        Sorter.sort (ret, comp);

        // label these ones
        v = null;
        newLabel = -1;
        for (i = ret.iterator (); i.hasNext (); )
        {
                prevV = v;
                v = (Vertex) i.next ();

                if (prevV == null || comp.compare (prevV, v) != 0)
                {
                        // v > prevV
                        v.orderLabel = ++newLabel;
                }
                else
                {
                        // v == prevV
                        v.orderLabel = newLabel;
                }
        }

        return ret;
}

static class ComparatorByVertexOrderLabels implements Comparator {
        public int compare (Object a, Object b)
        {
                return ((Vertex) a).orderLabel
                        - ((Vertex) b).orderLabel;
        }
}

/*
 * Compares two vertices by the lists of their children's order labels
 *
 * Vertices' lists of children must already be in order
 */
static class ComparatorByVertexChildOrderLabels implements Comparator {
        public int compare (Object a, Object b)
        {
                LinkedList aChildren = ((Vertex) a).children;
                LinkedList bChildren = ((Vertex) b).children;
                Iterator ai, bi;
                Vertex av, bv;
```

```
                int diff;

                diff = aChildren.size () - bChildren.size ();
                if (diff != 0) return diff;

                for (ai = aChildren.iterator (),
                      bi = bChildren.iterator ();
                     ai.hasNext () && bi.hasNext (); )
                {
                        av = (Vertex) ai.next ();
                        bv = (Vertex) bi.next ();

                        diff = av.orderLabel - bv.orderLabel;

                        if (diff != 0) return diff;
                }

                return 0;
        }
}

/*
 * Returns a Map if the given rooted trees are isomorphic
 */
static private Map rootedIsomorphic (RootedTree t1, RootedTree t2)
{
        LinkedList L1, L2;
        Iterator i1, i2;
        Vertex v1, v2;
        Comparator comp = new ComparatorByVertexChildOrderLabels ();
        Map ret;

        int h = t1.root.getHeight ();

        if (h != t2.root.getHeight ()) return null;

        t1.initializeVertices ();
        t2.initializeVertices ();

        L1 = t1.verticesAtDepth (h - 1);
        L2 = t2.verticesAtDepth (h - 1);

        for (int depth = h - 2; depth >= 0; depth--)
        {
                L1 = t1.labelLevel (depth);
                L2 = t2.labelLevel (depth);

                if (L1.size () != L2.size ())
                {
                        return null;
                }

                for (i1 = L1.iterator (), i2 = L2.iterator ();
                     i1.hasNext () && i2.hasNext (); )
                {
                        v1 = (Vertex) i1.next ();
                        v2 = (Vertex) i2.next ();

                        if (comp.compare (v1, v2) != 0)
                        {
                                return null;
                        }
                }
        }
```

```
                ret = new Map ();
                generateMapping (t1.root, t2.root, ret);
                return ret;
        }

        /*
         * Recursively constructs Map m, given that v1 and v2 are isomorphic
         */
        static private void generateMapping (Vertex v1, Vertex v2, Map m)
        {
                Iterator i1, i2;
                Vertex child1, child2;

                m.add (v1, v2);

                for (i1 = v1.children.iterator (),
                      i2 = v2.children.iterator ();
                     i1.hasNext () && i2.hasNext (); )
                {
                        child1 = (Vertex) i1.next ();
                        child2 = (Vertex) i2.next ();

                        generateMapping (child1, child2, m);
                }
        }

        /*
         * Returns a Map if t1 and t2 are isomorphic
         *
         * This will re-root the trees if necessary, then call isomorphic()
         */
        static Map isomorphic (RootedTree t1, RootedTree t2)
        {
                Map ret;
                Vertex[] c1, c2; // Arrays of at most 2 vertices each

                c1 = t1.findCenter ();
                t1.reRoot (c1[0]);
                c2 = t2.findCenter ();
                t2.reRoot (c2[0]);

                ret = rootedIsomorphic (t1, t2);

                if (ret == null && c1.length == 2)
                {
                        t1.reRoot (c1[1]);

                        ret = rootedIsomorphic (t1, t2);
                }

                return ret;
        }
}
```

Finally, I wrote a test program in `TestIsomorphic.java` to ensure success.

```
import java.util.StringTokenizer;
import java.io.*;

public class TestIsomorphic
{
        public static void main (String[] args)
```

```
                throws IOException, FileNotFoundException
        {
                FileReader fr;
                BufferedReader br;
                String treeStr1, treeStr2;
                RootedTree t1, t2;
                int spaceIndex;
                Map map;

                if (args.length != 1)
                {
                        System.err.println ("usage: "
                                            + "java TestIsomorphic [file]");
                        System.exit (1);
                }

                fr = new FileReader (args[0]);
                br = new BufferedReader (fr);

                treeStr1 = br.readLine (); // first tree as a string
                treeStr2 = br.readLine (); // second tree as a string

                if (treeStr2 == null) // trees separated by ' ', not '\n'
                {
                        spaceIndex = treeStr1.indexOf (' ');

                        if (spaceIndex == -1)
                        {
                                System.err.println ("Cannot parse file");
                                System.exit (1);
                        }

                        treeStr2 = treeStr1.substring (spaceIndex + 1);
                        treeStr1 = treeStr1.substring (0, spaceIndex - 1);
                }

                // Generate trees from strings
                t1 = new RootedTree (treeStr1);
                t2 = new RootedTree (treeStr2);

                // Test them
                System.out.println ("Tree 1: " + t1);
                System.out.println ("Tree 2: " + t2);

                map = RootedTree.isomorphic (t1, t2);

                if (map == null)
                {
                        System.out.println ("The trees are not isomorphic.");
                }
                else
                {
                        System.out.println ("An isomorphism exists:");
                        System.out.println (map);
                }
        }
}
```

I wrote several test trees to make sure my program was all right. What follows are the test files I used and the results obtained.

```
$ cat ../tests/andrei
root1(a1(c1)(d1))(b1(e1)(f1))
a2(c2)(d2)(root2(b2(e2)(f2)))
```

```
$ java TestIsomorphic ../tests/andrei
Tree 1: root1(a1(c1)(d1))(b1(e1)(f1))
Tree 2: a2(c2)(d2)(root2(b2(e2)(f2)))
An isomorphism exists:
{(root1, root2), (b1, a2), (f1, d2), (e1, c2), (a1, b2), (d1, f2), (c1, e2)}
$ cat ../tests/degenerate
a
b
$ java TestIsomorphic ../tests/degenerate
Tree 1: a
Tree 2: b
An isomorphism exists:
{(a, b)}
$ cat ../tests/failure
a(b(c(d)))
a(b)(c)(d)
$ java TestIsomorphic ../tests/failure
Tree 1: a(b(c(d)))
Tree 2: a(b)(c)(d)
The trees are not isomorphic.
$ cat ../tests/given
a(b(e(j)(k))(f(l)))(c(g(m))(h))(d(i))
n(o(r)(s(w)))(p(t))(q(u(x))(v(y)(z)))
$ java TestIsomorphic ../tests/given
Tree 1: a(b(e(j)(k))(f(l)))(c(g(m))(h))(d(i))
Tree 2: n(o(r)(s(w)))(p(t))(q(u(x))(v(y)(z)))
An isomorphism exists:
{(a, n), (d, p), (i, t), (c, o), (h, r), (g, s), (m, w), (b, q), (f, u), (l, x),
(e, v), (k, z), (j, y)}
$ cat ../tests/mine
root(dee(bee))(buy(go)(moo))
zuy(zo)(zoo)(zoot(zee(b0)))
$ java TestIsomorphic ../tests/mine
Tree 1: root(dee(bee))(buy(go)(moo))
Tree 2: zuy(zo)(zoo)(zoot(zee(b0)))
An isomorphism exists:
{(root, zoot), (dee, zee), (bee, b0), (buy, zuy), (moo, zoo), (go, zo)}
```