1 Java based partially ordered sets

While extending TextSeer, I ran into the problem of developing a framework to describe QoS values for activities. In work done to date, QoS values have been said to be measures of service from a c-semiring preference.

So naturally when I set about implementing a c-semiring preference like structure for TextSeer I naturally assumed that there would be a nice and simple poset class available for use online.

Turns out, there isn't. In this article I'll describe the implementation of a java POSET class.

A partially ordered set (poset) is a grounded set X with a partial order, induced by an operator $\leq$. Poset have the following properties:

- $a \leq a$ (reflexivity);
- if $a \leq b$ and $b \leq a$ then $a = b$ (antisymmetry);
- if $a \leq b$ and $b \leq c$ then $a \leq c$ (transitivity).

Our partial order class must hold with each of the above properties. To define instances of our class we want to pass pairs of values in, where the pairs show ordering. The set of pairs $\{\langle A, B\rangle, \langle B, C\rangle\}$ should be the grounded set $X = \{A, B, C\}$ with an ordering where $A \leq B$ and $B \leq C$ and $A \leq C$.

To start with, we need a simple structure to hold pairs of values. A quick visit to stack exchange resulted in some nice code `http://stackoverflow.com/questions/156275/what-is-the-equivalent-of-the-c-pairl-r-in-java`.

<code/Pair.java 1>

```java
public class Pair<A, B> {
  /* Variables */
    private A first;   // First Element
    private B second; // Second element

    /* General Functions*/
    public Pair(A first , B second); // Constructor
    public int hashCode();          // Return a hashcode of the
        pair
    public boolean equals(Object other); // Return true if this
         pair is equal another pair
    public String toString();    // Create a textual string of
        the pair

  /* Getters and Setters*/
    public A getFirst();
    public void setFirst(A first);
    public B getSecond();
    public void setSecond(B second);
}
```

We should be now ready to start work on our POSET class. We'll start by filling out a general class, leaving space for a datastructure and tests.

<code/POSET.java 2>

```

public class Poset {

    <<DATASTRUCTURE 3>>

    public <T> boolean lt(T a, T b){
      <LT>>
    }

    public <T> boolean leq(T a, T b){
      <<LEQ 6>>
    }

    public <T> boolean eq(T a, T b){
      <<EQ>
    }

    <<TESTCASE 9>>

}
```

INCLUDED BLOCKS: 3 on page 2, 6 on page 5, 9 on page 6

To store our pairs, the best structure to start with is a map. Where each each key represents the first value of a pair and the value represents the second value of a pair. This will cover typical cases where we get inputs of: $\langle A, B\rangle$ or $\langle B, C\rangle$; however, when we get multi-values e.g. $\langle A, B\rangle$ and $\langle A, D\rangle$ we will need to make sure we don't write over old map pairs. When new data comes in we'll need to run it through a data preparation function.

<DATASTRUCTURE 3>

```
    protected TreeMap<?, ?> orders;
    protected TreeMap<?, ?> backwards;

    // Simplify the creation of a new generic pair.
    // Requirement is that the type of values coming in
    // must be the same.
    static <T> Pair<T,T> P(T a, T b){
      return new Pair<T,T>(a,b);
    }

    // When a new list is created, store the data in a map
    public <T> Map<T,HashSet<T>> List(T... elements) {
    // To ensure that we don't write over existing map keys,
        process the data first.
```

```
14        return computeOrders((java.util.List<Pair<T, T>>) new
              LinkedList<T>(Arrays.asList(elements)));
15    }
16
17    <<DATAPREP 4>>
```

USED IN: code/POSET.java on page 2 INCLUDED BLOCKS: 4 on page 3

The above list function allows us to input an n-ary list of generic elements, though looking at line 7, shows that the elements should be generically typed pairs. For example, a use of this function could be:

$$List(p('A', 'B'), p('B', 'C'))$$

Due to the use of these generic types, whatever datatype that is used for the pairs should be accepted by the list and poset.

The next stage of the development is relatively straight forward. We are simply going to create a map with a set of values from incoming pairs. So that the first element of a pair is the key and the second element is added to the values set.

<DATAPREP 4>

```
public  <T> Map<T,HashSet<T>> computeOrders(List<Pair<T,T>> s){
      orders = new TreeMap<T,HashSet<T>>();
      backwards = new TreeMap<T,HashSet<T>>();
      for(Pair<T,T> p : s){
         if(orders.containsKey(p.getFirst())){
            ((HashSet<T>)orders.get(p.getFirst())).add((T) p.
                getSecond());
         }else{
            HashSet<T> h = new HashSet<T>();
            h.add(p.getSecond());
            ((TreeMap<T,HashSet<T>>)orders).put(p.getFirst(), h);
         }
      }

   <<TRANSITIVE CLOSURE 5>>
   }
```

USED IN: DATASTRUCTURE on page 2 INCLUDED BLOCKS: 5 on page 4

Once the map exists a key will have an associated set of values that are *worse*, i.e., $key \leq value$. Though, stopping implementation misses the property of transitivity.

To implement transitivity, a second map is created that will hold the reverse of the order map. That is a key will have an associated list of values that are better than it. We then have an extra function that will loop to ensure transitive closure for all elements of the new map.

<TRANSITIVE CLOSURE 5>

```
      // Do transitive closure
      /*Setup some variables to store hash of the existing maps
          */
      int backHash;
      int orderHash;
      int _backHash;
      int _orderHash;
      // Loop through a closure function until all transitive
          elements are included in backwards map
      do{
        backHash = backwards.hashCode();
        orderHash = orders.hashCode();
        doClosure();
        _backHash = backwards.hashCode();
        _orderHash = orders.hashCode();
      }while(backHash != _backHash || orderHash != _orderHash);
      return (Map<T, HashSet<T>>) orders;
    }

    // Not completely optimal, this function will search all
        orders and build a transitive set of values.
    public <T> void doClosure(){
      for(T key : ((TreeMap<T,HashSet<T>>)orders).keySet()){
        for(T ele : ((HashSet<T>)orders.get(key))){
          if( ((TreeMap<T,HashSet<T>>)backwards).containsKey(
              ele)){
              ((TreeMap<T,HashSet<T>>)backwards).get(ele).add(
                  key);
              if( ((TreeMap<T,HashSet<T>>)backwards).containsKey
                  (key))
                ((TreeMap<T,HashSet<T>>)backwards).get(ele).
                    addAll(((TreeMap<T,HashSet<T>>)backwards).get
                    (key));
          }else{
            HashSet<T> h = new HashSet<T>();
            h.add(key);
            h.add(ele); // Ensure satisfaction of reflexivity
            ((TreeMap<T,HashSet<T>>)backwards).put(ele, h);
          }
        }
      }
```

USED IN: DATAPREP on page 3

At this point our data structure is complete, we can create new poset (without operators) using by first declaring a new instance of poset:
Poset p = new Poset(); and then filling a list of pairs, e.g.,

```
p.List( P("a", "b"), P("b","c"), P("a","d"), P("d","c") );
```

Which will result in the following structure in the *backward* map of values:

```
{b=[a], c=[d, b, a], d=[a]}
```

To complete the poset class we will implement three operators, less than (LT), less than or equal to (LEQ), and equal to (EQ).

<LEQ 6>

```
    public <T> boolean leq(T a, T b){

      if(backwards != null && (backwards.containsKey(a) ||
          backwards.containsKey(b))){
        if(
            (
              (backwards.containsKey(b) &&
              ((HashSet<T>)backwards.get(b)).contains(a))
            ))
          return true;
      }
      return false;
    }
```

USED IN: code/POSET.java on page 2

Less than or equal to (LEQ) is the first implemented function, and is relatively simple to understand. If the second parameter has a key entry in the reversed map then open the value set and search for the first parameter. If the first parameter is in the value set then it has a relative ordering to the second parameter, either less than (LE) or (EQ).

From here the remaining operators are trivial. Based on the property of antisymmetry: if $a \leq b$ and $b \leq a$ then $a = b$ we can show equivalence with the functional definition of EQ.

<EQ 7>

```
    public <T> boolean eq(T a, T b){
      return leq(a,b) && leq(b,a);
    }
```

The less than method (LE) is relatively straight forward as well. To compute less than (LT), we evaluate true for any pair that has the relation LEQ and that does not have the EQ relation.

<LT 8>

```
    public <T> boolean lt(T a, T b){
      if(leq(a,b) && !eq(a,b)) return true;
      else return false;
    }
```

Our class is now completed. To finalize we'll attempt to test our code against the original poset properties of reflexivity, antisymmetry and transitivity.

<TESTCASE 9>

```
Poset p = new Poset();
p.List(P("a", "b"),
    P("1", "a"),
    P("a", "1"),
    P("b","c"),
    P("a","d"),
    P("d","c"));
System.out.println("leq(a,a) = " + p.leq("a","a") + " (
    reflexivity)");
System.out.println("leq(a,1) = " + p.leq("a","1") + " & leq(1,a
    ) = " + p.leq("1", "a") + " finally eq(1,a) = " + p.eq("1",
    "a") + " (antisymmetry)");
System.out.println("leq(a,b) = " + p.leq("a","b") + " & leq(b,c
    ) = " + p.leq("b","c") + " finally leq(a,c) = " + p.leq("a",
     "c") + " (transitivity)");
// Extra tests for good measure
System.out.println("eq(a,b) = " + p.eq("a","b"));
System.out.println("leq(b,a) = " + p.eq("b","a"));
System.out.println("lt(a,c) = " + p.lt("a","c"));
System.out.println("lt(c,a) = " + p.lt("c","a"));
```

USED IN: code/POSET.java on page 2

The output of the tests are as follows:

```
leq(a,a) = true (reflexivity)
leq(a,1) = true & leq(1,a) = true finally eq(1,a) = true (antisymmetry)
leq(a,b) = true & leq(b,c) = true finally leq(a,c) = true (transitivity)
eq(a,b) = false
leq(b,a) = false
lt(a,c) = true
lt(c,a) = false
```

All code demonstrated in this example is available through the TextSeer github repository at `https://github.com/edm92/TextSeer/`.