CMSC420 Project - Spring 2017
Part 3

The BIG 420 Project*
Part 3 will be due at 11:59PM on max(syll, submit server) (plus 48 hour grace period)

Last Modified March 3, 2017

Contents

*Participation in this project may prove HAZARDOUS to your health. Unfortunately, failure to participate early and often will definitely adversely affect your GPA. Take my advice. Start now, because you're already behind. If you don't believe me, ask anyone who took CMSC 420 with Dr. Hugue.

1 Introduction and General Overview

Welcome to Part 3 of the Spring, 2017, edition of Dr. Hugue's CMSC 420 project. Note that you may need to refer to the part 1 and part 2 specifications since parts 3 and 4 build on it. A substantial portion of your grade in this course will be determined by your performance on several programming assignments (collectively referred to as *The Project*). The time commitment required on your part varies from student to student, but expect to spend a good bit of time planning as well as coding and debugging each part of the projects.

The primary motivation of this project is to give you the experience of building portions of a real world application using the data structures and algorithms that we will study during the semester. This semester's long term goal is to mimic some of the functionality of MapQuest [1]. By the end of this course, if you have worked hard and done your job well, you will have a program that is capable of drawing maps of a general area and supporting the following functions: displaying a highlighted route; calculating shortest routes (based on time or Distance); generating driving instructions (complete with correct "turn left and then go straight for 2.34 miles" annotations); determining closest points of interest, such as all of the "Internet Cafes" within a 20 mile radius of College Park, MD; and just generally impressing friends and family with its awesome ability to tell you how to get to where you want to be in both plain text and picture form. And, even if things don't go smoothly, you will have had the opportunity to think about data structures, and programming in general, in new and interesting ways.

2 MeeshQuest Components

There are four major components to this project each of which will upgraded with each part: a dictionary data structure, a spatial data structure, an adjacency list, and a mediator.

2.1 Dictionary Data Structure

A dictionary data structure is one which is capable of storing objects in sorted order based on key such as a string or an integer. For instance, you might have several hundred City objects which consist of the name of the city, the latitude and longitude at which it is located, and the radius or expanse of the city limits. One way of storing these cities is to sort them by name; another is to store them in decreasing order by population; yet another is in increasing order by latitude.

[1]

To manage a structure based on the names of cities, you would not need a comparator for cities; but, rather, since the keys are city names (strings), your comparator would need to handle `strings`. So to compare cities by name in Java 1.5, your comparator might look like this:

```
public class CityNameComparator implements Comparator<String>
{
            public int compare (String s1, String s2) {

                     (however you want to do string comparison)
                        return

            }

}
```

Specifically in this project we'll use a dictionary to hold and sort cities and attractions by name and a spatial data structure to hold and sort by coordinate. The data dictionary will be required in all project parts.

New for part 3 will be the requirement that you implement the SortedMap interface using an AVL-G tree which *could* be used as the basis of your data dictionary in parts 2 and 3. However, if we demanded that you use that structure and it didn't work, we'd be unable to grade any of your project if the AVL-G was broken. So, keep using the Treemap/Treeset from part1 as well.

2.2 Spatial Data Structure

A spatial data structure is one which is capable of storing and sorting objects based on multidimensional keys. The two-dimensional structures, such as the PR quadtree and PM quadtrees, can store objects based on two values, such as the longitude and latitude (x, y) of a city. In Java this is not a big deal, but in other languages, such as C and C++, it is. Note that latitude lines are horizontal, north and south of the equator, and correspond to lines with a fixed `y` coordinate, such as $y = 30N$. Longitude lines are vertical, east and west of Greenwich, England, and correspond to lines with a fixed `x` coordinate, such $x = 40W$. Three-dimensional structures, such as a K-d tree of order 3 or a PM octree, can store objects based on three values, such as the longitude, latitude, and sea level of a city (x, y, z).

2.3 Adjacency List

This is a rather simple component which will not be changing much over the semester. There are many ways to implement an adjacency list, not just an array or arrays or a simple matrix like you may have learned in the lower levels! You can do a skiplist of skiplists, or a TreeMap, or maybe even an Arraylist of Arraylists– any number of various objects which provide the right traversal capabilities, meaning that searching for an element in the adjacency list needs to be in $O(\log n)$ where n is the number of vertices.

Regardless of the specific implementation of the adjacency list, its purpose is to provide us with a quick and easy way to perform shortest path calculations so we will be able to produce MeeshQuest functionality. Combined with the spatial data structure and a simple Java GUI library called Canvas Plus which we will be providing for you, you will not only be able to print out detailed instructions of how to get to your destination, but you will also be able to generate a colored map which highlights the route you have calculated on a map. We will not grade it explicitly, but you might prefer to keep an adjacency list representation of the edges that are inserted into the PM quadtree than to derive it on the fly.

2.4 Mediator

A Mediator is a design pattern that is described in the famous book *Design Patterns* by Gamma et Al, also referred to as the Gang of Four (GOF) book.[1]

The intent of a Mediator is to **define** an object that encapsulates how a set of objects interact. Mediator promotes a loose coupling among objects by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently[2]

In other words, the idea is to have one or more objects (hint: more!) which are capable of reading in commands from the input stream and executing them by calling the right functions in the dictionary, spatial, and adjacency list data structures to perform the requested action(s).

Minimally, the Mediator could be a class named CommandParser which would read commands from the standard input stream, parse the data, pass it onto the correct component for further processing, analyze the return values from this component, print the correct success or failure message back to the user, and then loop until the EXIT() command is read.

It would be wise (hint hint) to break this functionality into several classes which perform one or more of these tasks. These objects, when combined together, form the abstract notion of a Mediator for our MeeshQuest program.

3 Roadmap

This is a roadmap of the major component that we will use in each part of the project. An asterisk (*) indicates that we will be reusing the structure from the previous project with little or no modification. "i" stands for `insert`, "f" stands for `find`, and "d" stands for `delete`. (The command parser will have new commands added with each project, but the overall design need not change after project 1 unless you want to make it more efficient or elegant.)

Part	Dictionary	Spatial	Adj. List	Application	Me
1	Treemaps (i/f)	n/a	n/a	n/a	Any imp
2	Treemaps (i/f/d)	PR Quad (i/f/d)	n/a	n/a	Any imp
3	AVL-G (i/f)	PM3 Quadtree (i/f)	Any (i/f/d) in $O(\log n)$	Dijkstra/Shortest Path	
4	AVL-G (i/f/d)PR of	PM1/3 Quadtrees (i/f/d)	*	MST	

****NOTE: This roadmap is subject to change as the semester progresses.****

4 Project I/O: XML and Conventions

Just a reminder that the rules from Part 1 and Part 2 may still hold; so, check the Part 1 and Part 2 specifications if you can't find what you want in this document: www.cs.umd.edu/users/meesh/cmsc420/part1.

4.1 Part 2 Update: Data dictionaries and notes about Cities

Please see the corresponding sections in the specification for Part 1 and Part 2 if you remain unsure about the objects and elements of the JAVA API appropriate for representing cities. The bulk of this section deals with the modifications to the data dictionary required for parts 3 and 4.

We have alluded to the fact that you are going to have to maintain a dictionary (a collection) of all the cities we create using the `createCity` command throughout the entire project. For both Part 2 and Part 3, need to be able to find cities based on their (x, y) coordinates, since aside from two cities bearing the same name being illegal, two cities cannot occupy the same (x, y) coordinate. The treemap and comparator for two-dimensional coordinates from Part 2 is sufficient to check for uniqueness of the coordinate–city name

[2]While this design pattern is presented on page 273 of [1], a Google search using "design patterns mediator" is cheaper than buying the book.

assignment for for Part 3 and Part 4. However, you may see a new command which requires you to do a search for cities with coordinates satisfying a given rule in Part 3, and will definitely see it in Part 4. The comparator for city coordinates will remain the unchanged. That is, it *must* sort (x, y) coordinates in the following fashion: sort on the y coordinate first in ascending order; if two cities have the same y coordinate, compare their x coordinates, and place the one with the smallest x coordinate first.

5 Part 2: Comparators, Treemaps, Cities, PR Quadtrees and Range Searches

For your convenience, the portion of the specification associated with Part 2 commands has been removed from this document. However, please feel free to visit it: www.cs.umd.edu/users/meesh/cmsc420/ProjectBook/part2.

6 Part 3: Sorted Map Insert, the AVL-G tree, Polygonal Maps, and Road Adjacency Lists

For the remainder of the semester, you will have the pleasure of implementing the data dictionary with both the TreeMap from part 2[3] and an *AVL-G* tree to implement the SortedMap interface. In part 3, you will be required to insert a key into the data dictionary and do single-item queries. In part 4, you will be required to support both insert and delete, and may choose to implement range queries for the data dictionary.

The new spatial structure for part 3 and part 4 is the PM quadtree (PM1 or PM3). Part 3 requires you to implement only insert for the PM3. Of course, there will be spatial range searches, and the usual annoying input and output formats.

Plus, we're building on the last part. You're probably insanely bored of `TreeMap`s and `Comparator`s by now, but now that you've had some practice with them in the minors, it's time to use your mastery of modifying pre-existing data structures to your advantage. In addition, all you mathphobes will finally get over your fear, since part of this project might involve the derivation of some formulas.

The individual commands for part 3 appear in Section 7.3, after detailed discussions of the structures involved and implementation requirements and hints. In general, part 3 requires you to implement insert and `find_key` operations for the dictionary and spatial structures, while part 4 adds in the delete operations. Please don't be scared–remember how far away college (and graduation!) once seemed. It gets worse out there; so, at least try to kick up your performance rate now while it's safe.

***** Adopting any source code or pseudocode of a PM quadtree from Internet search, that is, any code that is not provided or linked as resources of this course, is explicitly prohibited. You have to build your own.**

6.1 AVL-g tree requirements

In Part 3, you will be asked to *consider* replacing the treemap of city names which is used as an index to the city objects with some other structure. For this semester of 420, the "other structure" will be a modified AVL tree. You are all familiar with the traditional definition of an AVL tree, or if not, you can become so:

> The AVL tree (named for its inventors Adelson-Velskii and Landis) should be viewed as a BST with the following additional property: For every node, the heights of its left and right subtrees differ by at most 1.

This is Shaffer's definition for an AVL tree; we modify the definition in one very simple way. Our AVL-g tree takes a parameter on creation—the so-named `g` parameter—which refers to the maximum height by which the subtrees differ. Apart from this, our trees are identical in implementation to the conceptual trees

[3]This is not for production code, but to make sure that we can grade code that requires some form of the SortedMap that relies only on the JAVA api code.

given by Shaffer; we rebalance the tree using identical rotations to the basic tree, both single and double. You can find ample resources online for AVL trees; the problem as we pose it is in adapting the resources you can find for our AVL-g trees.

Astute readers will note that for any $k < g$, any AVL-k tree is a valid AVL-g tree. Thus, since a regular AVL tree is just an AVL-1 tree by our definition, you could just grab source for an AVL from the internet and point out that it's a valid AVL-g tree for any g. However, we're not satisfied with such trickery. Note that given a number g and an ordered input set K, we can always find some number m which is the maximal balance factor found somewhere in the tree. For example, consider an AVL-1 tree with two nodes. It is always true that such a tree has at least one node with balance factor 1—the tree can never be balanced better than that. Consider similarly an AVL-2 tree with three nodes, where the nodes were inserted in sorted order. See a pattern? Of course it's trivial to find the pattern for $g + 1$ nodes in sorted order, but if you want a fun challenge, try and come up with a way of finding m for any arbitrary input set and g. Anyway, our tests will use this property—all of our tests verify that the maximal balance factor of the tree is greater than m but less than g.

However, since we're using Java and a fundamental feature of Java is its interfaces, your code will be required not only to properly implement the AVL-g specifications, but also to implement Java's SortedMap interface. This will allow you to fully replace the TreeMap with your SortedMap as you desire. Your AVL tree should be generic just like the TreeMap, and it should *eventually* implement every one of the SortedMap methods except `headMap(), tailMap(), keySet()` and `values()`. However, we will not test `remove()` until Part 4. So, you can choose to delay it until after you've finished Part 3.

But, since you don't implement `remove` in Part 3, this prevents you from properly implementing `subMap().clear()` and related functions. Therefore, your behavior for `subMap().clear()` won't be tested—but we advise that at this point you forward to `AvlGTree.this.clear()`. If that sounded confusing to you, don't worry about it for now, but come back to it later. If you would like guidance on how to begin implementing SortedMap, there are two resources to which you can turn: we provide you with an example, SortedMapExample.jar, which shows a SkipList that has been used to implement SortedMap. You can find SortedMapExample.jar in the top level directory of the course files. Similarly, you can easily find the source code for Java's `TreeMap` online; Josh Bloch's implementation strategies may prove useful and relevant to your code. As always, if you take inspiration or structure from either of these sources please cite that in your README file.

Any portion of the TreeMap that uses the red-black tree explicitly (or the skiplist version) must be modified to use the AVL-G tree instead. Failure to do this will result in a deduction of the points that you tricked the submit server into giving you. It will not be treated as an honor code violation since the red-black tree is not the most intuitive creature that you will ever meet.

Although this probably won't be a problem for AVL trees as much as some of the structures in the past, we expressly forbid you from actually using a skiplist or other complex data type inside your AVL tree code. That is, although you may be inspired (or more) by the code you see in TreeMap or SkipList, you may not actually use a TreeMap or SkipList in your implementation. The same goes for TreeSet (which, interestingly, is just a wrapper around a TreeMap—cool trivia!), HashSet or HashMap; basically, if you're using a structure that you feel probably duplicates the functionality of your SortedMap, you should not be using it.

We now proceed into a formal definition of an AVL-g tree. In order to do this, we must define the height of a subtree—we do so recursively, saying that a leaf node has height 0, and an internal node has a height equal to the height of its tallest child, plus one. We define the height of an empty tree to be -1.

Thus, an AVL-g tree is defined as a binary search tree where at any node, the height of its two subtrees (left and right) differ by at most a positive constant `g`. The constant is given at instantiation, as a parameter to the constructor. Although quite a bit of literature exists on maintaining the balance factor in an AVL tree, our tests only require that your tree maintain the above property, as well as obviously retaining any key-value mappings passed into it. **Note: do not waste time searching for additional information regarding the AVL-G tree because there is none. This structure is unique to MeeshQuest.**

These rules ARE **mandatory**, meaning that no credit will be given for code for which these rules are not observed. Obviously, there can be multiple AVL trees of a given order that are correct. Since grading

by merely diff-ing is impossible, we will grade your tree by checking that the structure you produce satisfies the above properties.

6.1.1 Important Notes about the SortedMap

In a development environment, it is prudent to isolate new code from tested code, especially after refactoring. Furthermore, you will often be required to modify a version of a well-studied data structure to suit your application. Finally, avoiding double jeopardy in grading is a meta-requirement for any course managed by Darth Hugue. Thus, we provide the following guidelines for the design, implementation, and use of the AVL-g tree in MeeshQuest.

- While you are free to implement this tree in whatever manner you choose, the explicit use of the TreeMap or TreeSet class is strictly prohibited!

  However, you are encouraged to look at the code used to implement them to see how you might proceed, as well as the `SortedMapExample.jar`, where the skiplist plays the role that the AVL-g tree will in your project.

- *Warning.* The Roadmap says that the AVL-g tree should be used for the data dictionary, and, indeed, is expected to be populated when a `create_city` command is processed However, you are encouraged and expected to maintain your treemap-based data dictionary as well to protect your other structures from failing tests because a faulty AVL-g tree corrupted your city database. Again, it is *expected* that tests of, say, the `map_road` command will use a data dictionary that is correct, irrespective of SortedMap implementation. The AVL-g will be evaluated by checking its compliance with the SortedMap Interface.

6.2 AVL-g Tree Implements Sorted Map Requirements

You are required to implement the `SortedMap` interface in the java.util package using the AVL-g tree. A `Map`, in short, is:

> An object that maps keys to values. A map cannot contain duplicate keys; each key can map to at most one value. [4]

The `SortedMap` extends the `Map` interface to further specify:

> A map that further guarantees that it will be in ascending key order, sorted according to the natural ordering of its keys (see the Comparable interface), or by a comparator provided at sorted map creation time. [5]

Since any Binary Search Trie structure guarantees that the actual data values, in this case what we've called "search keys", will be stored in sorted order as recovered through in-order traversal, the SortedMap interface is the most appropriate choice for implementation using the AVL-g, so that it can be used in place.

Your tree is not required to support null keys or values, however you may choose to do so. If a null values are ever passed in for a "key" or "value" parameter you are permitted to throw a NullPointerException if your implementation does not support null values.

You can find the exact specifications for the SortedMap interface in the JAVA API at:

`http://docs.oracle.com/javase/6/docs/api/java/util/SortedMap.html`

One hint if anyone really likes a class name other than AvlGTree: you can just add an extra AvlGTree.java file with the following contents and you'll be set:

[4] http://docs.oracle.com/javase/6/docs/api/java/util/Map.html
[5] http://docs.oracle.com/javase/6/docs/api/java/util/SortedMap.html

```
//file AvlGTree.java
package cmsc420.sortedmap

public class AvlGTree<K, V> extends YOURNAME {
  public AvlGTree() {
    super();
  }
  public AvlGTree(final Comparator<? super K> c) {
    super(c);
  }
  public AvlGTree(final int g) {
    super(g);
  }
  public AvlGTree(final Comparator<? super K> c, final int g) {
    super(c,g);
  }
}
```

As per the Java specifications for entrySet, keySet, and values, note, your structure should return a Set or Collection that is backed by the tree. When a Set/Collection is "backed" by another data structure the set/collection always reflects the state of the backed data structure. That means that if someone called "AvlGTree.entrySet()" then "AvlGTree.put()", the Set obtained from entrySet would allow you to access the element added from the put operation even though the entrySet was returned prior to the call to put.

This is not as hard as you might think it is. There are many approaches to this problem but may we suggest that you take a peek at the source code for TreeMap. You are allowed to *adapt* the code from TreeMap for entrySet as you see fit as long as no red-black tree based TreeMap code is used.

A hint: if you implement AbstractSet, the problem of writing a Set for entries, keys, and values becomes reduced to the problem of writing their iterators. With a little careful thought, you could even reduce that problem to one rather than three! For more, look at how TreeMap implements its entrySet and keySet.

Your AvlGTree class must be called `AvlGTree`, and it must exist in `AvlGTree.java`. The AvlGTree class must be placed in the package `cmsc420.sortedmap`. Your class must implement the following constructors:

```
// If this generic wildcard confuses you, reread the end of the
// part 1 spec for the introduction to generics.
public AvlGTree();
public AvlGTree(final Comparator<? super K> c);
public AvlGTree(final int g);
public AvlGTree(final Comparator<? super K> comp, final int g);
```

You're already quite familiar with the `SortedMap` interface—you've been using it every time you use a `TreeMap`, which is Java's stock implementation of `SortedMap`. `SortedMap` is a sub-interface of the `Map` interface, defined in the `java.util` package as part of the **Java Collections Framework**. In this project, you are going to get some exposure to implementing a Sun-provided Java interface. This will give you some practical experience coding to a standard and will also teach you leaps and bounds about how to solidly implement an object.

The first thing you'll need to do is have a look at the API documentation for the `Map` and `SortedMap` interfaces. Bookmark these pages now, since you'll be referring to them again and again throughout this project. You might even find it useful to print out copies. The URLs are

- docs.oracle.com/javase/6/docs/api/java/util/Map.html
- docs.oracle.com/javase/6/docs/api/java/util/SortedMap.html

Note that you are responsible for implementing every method from the SortedMap interface *except for remove(), headMap(), keySet(), tailMap(), and values()*. In Eclipse, you will be given the option to auto-fill these methods when you implement the interface, so you only have to remember not to implement the methods listed above. A wise implementation would throw an `UnsupportedOperationException` in case the user is unaware that we don't require remove or any of the other methods—that's an example of what's known as "fail-fast programming." The opposite camp of programmers would, of course, say that the tree should gracefully just return null and do nothing for unimplemented operations; however, we think that fail-fast programming is important and we will be testing your tree to see that you throw the desired exception. Many of these methods are one line, and most of the others are very short also. The majority of the work will be in implementing the `put()` method, which amounts to writing the algorithms for building an AVL-g tree. The details of your class are mentioned in the Avl-g portion of the spec (e.g., the name of the class and its required constructors).

6.2.1 Caution: `entrySet()` is tricky

There is one method that deserves considerable attention and is the most difficult to implement. It's vastly important that you understand this concept since it is required for most of the other methods (which you don't have to implement until the next project). This method is `entrySet()`.

As you can see from the prototype of the method, it returns a `Set` object. When students see this, they immediately assume that the method is supposed to create a new `Set`, add all of the `Map`'s entries into it, and return that `Set`. **That is completely wrong.** Observe the API description of the `entrySet()` method:

> Returns a set view of the mappings contained in this map. Each element in the returned set is a `Map.Entry`. The set is backed by the map, so changes to the map are reflected in the set, and vice-versa.

Note that changes in the `Set` are reflected in the `Map` and vice versa. Some students will read that and *still* think that they are supposed to create a new `Set`, copy all of the elements of the `Map` into it, return it, but include some way of managing that relationship (for instance, keep a pointer to the `Map` in the `Set` object, and store a list of all of the entry sets created and returned by that method in the map, updating all as necessary). **This is also completely wrong** (although it's closer to accurate).

How, then, does one implement this relationship in Java without making copies? The `Set` object that is returned from `entrySet()` does not itself contain any data. The `Set` that is returned is simply an object whose methods call the methods in the "backing map" (i.e., the map from which it was created). The simplest way to implement this behavior is by creating an inner class inside your `SortedMap` implementation that itself implements the `Set` interface. For example

```
public class AvlGTree<K, V> implements SortedMap<K, V> {
    public Object remove(Object key) { ... }
    public boolean containsKey(Object key) { ... }
    // We're leaving out generics here because otherwise
    // the code would be a mile long
    public Set entrySet() { return new EntrySet(); }
    protected class EntrySet implements Set {
        public boolean remove(Object o) {
            Map.Entry me = (Map.Entry)o;
                // throws a ClassCastException if this fails,
                // as per the API for Set

            boolean b = AvlGTree.this.containsKey(me.getKey());
            AvlGTree.this.remove(me.getKey());
            return b;
        }
```

```
        public boolean contains(Object o) {
            Map.Entry me = (Map.Entry)o;
            return AvlGTree.this.containsKey(me.getKey()) &&
                (me.getValue() == null ?
                    AvlGTree.this.get(me.getKey()) == null :
                    me.getValue().equals(AvlGTree.this.get(me.getKey())));
        }
    }
}
```

Observe that the `Set` object returned by `AvlGTree`'s `entrySet()` method is of type `EntrySet`, an inner class of `AvlGTree`, whose `remove()` method simply calls the `AvlGTree`'s `remove()` method, and containment in the returned set is checked by existence in the `AvlGTree`. Thus, if the `AvlGTree` changes *after* the `Set` is created, the `contains()` method of that `Set` will still return accurate results. **You must implement `entrySet()` to this standard.** We will easily discover whether or not your `entrySet()` is implemented correctly. It's very easy to grade:

```
SortedMap<String, String> m = new AvlGTree<String, String>();
m.put("Auto","Fail");
Set<Map.Entry<String, String> > s = m.entrySet();
m.put("F","---");
if (!s.contains(new Map.Entry<String, String>("F","---")))
System.out.println("You fail!");
```

Note that this wouldn't compile since `Map.Entry` is an interface, so you can only instantiate it anonymously. Substitute that with a concrete implementer of `Map.Entry`. ;) But you get the idea.

All methods in `Set` must be implemented by the `Set` object you return via your `entrySet()` method. However, there are certain methods, as detailed in the description of the `entrySet()` method, which are explicitly not supported by the entry set. Those include `add()` and `addAll()`. A good exam question might be why the authors of the `SortedMap` interface (our friend Josh Bloch) made this decision. There's a good reason—think about it!

6.2.2 Testing SortedMap: `equals()`, `hashCode()` must work

Unlike the structures you implemented in Part 2, your `SortedMap` implementation will be tested directly as well as through your XML. That is, we're going to link your `AvlGTree` into our code and run a `SortedMapTester` program on it. Your `SortedMap` implementation should always agree with `TreeMap`. Thus, **it is extremely important that your `equals()` method works correctly. Furthermore, it is equally important that the `TreeMap` `equals()` method will work with your AvlGTree. Look at `TreeMap`'s source code, see what it extends, and make sure it will be able to check your AvlGTree for equality, since we will invoke the `equals()` method bidirectionally.** And don't try to cheat and make your `equals()` method return true all the time, because we can (and will) test that–quite easily, in fact.. **Your `SortedMap` will fail all of the tests if `equals()` does not work** and that is *not* a valid excuse for a regrade. **You have been warned.**

In addition, your SortedMap implementation must have properly working `hashCode()` and `toString()` methods that agree with TreeMap. All returned objects must also have `hashCode()` and `toString()` methods that agree with TreeMap. For example, your EntrySet and SubMap must have `hashCode()` and `toString()` methods agreeing with TreeMap.

Also, **RTFAPI**. The API is very specific about what exceptions need to be thrown when, what needs to be returned when, etc. You are expected to throw exactly the exceptions the API defines. This applies both to the methods in the `Map`/`SortedMap` interface but also the methods in `Set` for the object returned by `entrySet()`. I am imposing another requirement that is commonly done in the Java Collections Framework but not clearly defined in the API, and that refers to the `Iterator` object returned

by a call to `entrySet().iterator()`. In addition to the exceptions described by the API for `Iterator` (specifically `NoSuchElementException` if `next()` is invoked when `hasNext()` returns `false`), **your iterator must throw a** `ConcurrentModificationException` if the map changes after an `Iterator` has been created. There is a very easy way to do this—just keep a `modCount` variable that gets incremented whenever a structure-changing method (i.e., `put()` and `remove()`) is called. When you create an iterator, save the `modCount` according to the `Map` when the iterator was created, and each time `next()` is called, check the saved `modCount` against the `modCount` of the `Map`. If the two numbers are not the same, it means the map has changed since the `Iterator` was created, and at that moment the `next()` method of the iterator should throw a `ConcurrentModificationException`.

6.2.3 Anonymous Inner Classes

One more thing: let me teach you anonymous inner classes. If you are looking at others code (like Josh's when you look at `TreeMap`'s source code) you might see this and be confused. An anonymous inner class is a way of shorthand-ing the creation of an inner class or the implementation of an interface. The syntax looks like this:

```
public Set<Map.Entry<K, V>> entrySet() {

  return new AbstractSet<Map.Entry<K, V>>() {
    public boolean add(final Object o) {
      throw new UnsupportedOperationException();
    }
  };
}
```

Notice two things—first, we are sort of "instantiating" an abstract class (we could do the same with an interface), but instead of a semicolon we open a curly brace and begin defining its methods. The compiler will not let you compile this code unless you implement all of the methods in `Set` within the braces (for obvious reasons). What actually happens behind the scenes is that the compiler changes that statement into code that looks like this:

```
// Generics don't exist once the compiler has its say; see Part 1
public Set entrySet() { return new AnonymousInnerClass1(); }
public class AnonymousInnerClass1 implements Set {
  public boolean add(Object o) {
    throw new UnsupportedOperationException();
  }
}
```

Pretty handy, huh?

The source code for a full implementation of `SortedMap` using the skiplist is in the `SortedMapExample.jar` file in the top level cmsc420 directory in my user pages. And, you also have the `TreeMap`'s source code) to study. Again, you are to replace the use of the SkipList in our example file and the use of the Red-Black tree in built-in TreeMap and TreeSet implementations. You will lose full credit if the AVL-G tree is not used appropriately.

6.3 PM Quadtrees

As promised, this project introduces a vastly more powerful spatial data structure, the beloved PM Quadtree, where PM stands for Polygonal Map. A polygonal map[6] consists of a set of line segments that can intersect only at the endpoints. PM Quadtrees have orders, which you will learn about in class and will read about

[6]This is not the same as the maps associated with JAVA or SortedMap, but the colloquial usage of the term map.

it in Samet. You will also learn that the X in PMX, where X is either 1, 2, or 3, is just there to indicate which partitioning rules are to be used when representing the polygonal map.

For part 3 of the project, you must implement a PM3 to receive full credit. In part 4 you will be able to adopt a working PM3 quadtree from part 3; so, a working PM3 quadtree will be required, and test-able with the primary input, before you submit it.

PM2 Quadtree not required in the projects.

For this project, and depending on which option you implement, we have a few new fun operations we can perform on spatial maps. Since the PM quadtrees also store line segments, we're creating roads.

We will also have *airports* and *terminals*. And once we have roads in this map, we can party.

The radius search is still in, plus this time around the radius search will also report any roads that exist in that radius. We also introduce two new types of operations:

Nearest City to Segment: Given a road S, locate the nearest city. Java gives you the geometric calculations so you won't have to muck around with the math (like the folks did in C++ years ago) but you will still need to come up with a suitable algorithm to do this efficiently.

Nearest Segment to Point: Given a point P, locate the nearest road. If you're clever you can roll this operation and the previous one into a single operation where the type of geometry is irrelevant (my geometry classes attempt to make the type transparent by providing common functions like intersects() and distance() – if you use these or roll your own with similar characteristics, you can write one "nearest object to object" where it's irrelevant whether those objects are points or line segments. It's all a matter of distance, after all.

One of the convenient features of PM quadtrees is that they are deterministic, meaning all quadtrees ARE created equal. The order in which objects are inserted is irrelevant to the resulting structure, and that structure is invariant. All your quadtrees should look exactly the same, which means we can and will be using diff to test your quadtrees. Thus, you're not required to implement any specific interface, Java provided or otherwise. You can implement your quadtrees however you want.

To make your quadtrees convenient for Part 4 the spatialWidth and spatialHeight attributes of the `<commands>` tag will always be powers of 2. e.g. 128, 256, 512, 1024, so on. This makes dividing the regions a very quick operation. If the values are negative or not numbers use the standard "illegal attribute" error message. Java has bit shift operations that you are welcome to take advantage of (although javac probably converts `x/2` into `x>>1`). Recall shifting the bits of a positive integer one to the right has the effect of dividing it by 2 (and rounding down) – similarly `x<<1` multiplies x by 2; `x<<2` multiplies x by 4, and so forth. Using the `>>` operator on ints for division makes you look smart and doesn't leave compiler optimization to chance. And obviously it doesn't work on floats. Also, for this part we will not create partitions in your PM Quadtree deep enough to create regions whose area is less than 1x1. Thus we are limiting the height of the tree based on `max(spatialWidth,spatialHeight)`. (Note that I said "based on" not "exactly equal to").

Note: For our PM Quadtrees, if a city or any portion of a line segment lands on any of a region's borders, that city or line is added to that region. This is consistent with every quadrant being closed on all four boundaries

For instance, suppose you had a simple instance where your quadtree occupied an area defined by (0,0), (1024,1024). The center point of the first partition that will be created, therefore, would be at (512,512). If the first road you add contains a city at (512,512), this city would be added to all four of the children of the resulting gray node, since the city touches the corners of each of those four regions.

Our test files will include a new attribute in the `<commands>` tag, *pmOrder*, whose value can be one of the following:

```
1 - PM1 Quadtree
3 - PM3 Quadtree
```

We will run your code through tests of various orders, and give you credit for all of the ones you passed, scaling to reflect the total amount of credit we're offering for each of the options. If you are not supporting certain orders in your project, you can print anything you want (or nothing at all) when you encounter a *pmOrder* attribute whose value indicates something you didn't do. In the end you'll just fail the test anyway regardless of what you print since we'll be checking via diff. Regardless of which PM options you implement,

always print out a `<results/>` tag so that the XMLParser doesn't throw an exception when we reroute your output back into the parser to auto-format it for diffing.

We will not speed test your PM Quadtree specifically because its complexity is not good as far as add and remove are concerned. We will speed test your "nearest city/nearest segment" in the same way we speed tested your spatial map from the previous part; so you'll have to spend some time coming up with a reasonable average case algorithm. Note that no matter what you do the complexity will still be O(n), but on average this should take drastically less time.

We will grade your PM primarily by using `<printPMQuadtree />`, since all PM Quadtrees given the same geometry should be identical. Another nice feature of the PM is that they are also identical regardless of the order of operations. In other words their structure is deterministic for inputs. That is, of course, because they are tries.

6.3.1 PM Pseudocode

If you're having trouble understanding this, be sure to check the on-line PM lectures.

```
class PMQuadtree {
   class Node; // define this yourself; superclass for Gray and Black;
               // evan used an interface, but this is pseuedocode.

   White SingletonWhiteNode = new White();

   class Gray extends Node {
      Node[] children;
      Rectangle[] regions; // a gray node is a partition
                   //into 4 regions; these rects are those regions

      // parameters here are stuff like the geometry you're adding
      //and any other info you need on the stack

      Node add(Geometry g, ...) {
         if (g intersects regions[0]) children[0] = children[0].add(...)
         ...
         if (g intersects regions[3]) children[3] = children[3].add(...)
      }

      Node remove(Geometry g, ...) {
         if (g intersects regions[0]) children[0] = children[0].remove(...);
         ...
         if (g intersects regions[3]) children[3] = children[3].remove(...);
         if (all children are white) return SingletonWhiteNode;
         if (three children white, one child black) return the black child;
         if (not all children are gray) {
            Black b = new Black;
            add all geometry in this subtree into b;
            if (b is valid) return b;
            else {
               // this gray is still necessary, so keep it in the tree
               return this;
            }
         }
      }
```

```
   }

   class Black extends Node {
      List geometry;
      Node add(Geometry g, ...) {
         add g to geometry
         if (this node is valid) return this;
         else return partition(...);
      }

      Node remove(Geometry g, ...) {
         remove g from geometry
         if (geometry is empty) return SingletonWhiteNode;
            else return this;
      }

      Node partition(...) {
         Gray g = new Gray(...);
         for each item i in geometry {
            g.add(i);
         }
         return g;
      }
   }

   class White extends Node {
      Node add(Geometry g, ...) {
         return new Black(g);
      }

      Node remove(...) { error }
   }
}
```

6.3.2 Hints for using PM pseudocode

Now, naturally it's up to you to fill in the ellipsis (the parameter lists) with whatever you need. You will need info about the depth of tree, primarily, so you can figure out how to make the rectangles for the gray. Note that you don't necessarily have to store those regions. I do it as an efficiency issue (space vs. time). You could compute the rectangles on the fly each time, but I think that's wasteful; so I chose to store them.

Also note: the key to making your PM Quadtree a PM1, PM3, etc is the line in `Black.add()` regarding node validity :

```
if (this node is valid) ...
```

This is the only part of the code that differs among the PM1 and PM3 (well, the "validity check" is done in the `Gray.remove()` also, but you get the idea). A PM3, for instance, is valid as long as it only has zero or one vertices in it. It is allowed to have any number of q-edges as long as they don't intersect each other. When you attempt to insert a second dot into this list, however, the node becomes invalid, and you must partition the node into 4 nodes, each corresponding to one fourth of the original block.

The key to doing this efficiently is to use OO design here. is key. There are some details I omitted from the pseudocode. For instance, the way I setup my code to allow for easily adapting the generic PM Quadtree

into a PM1 and PM3 was not straight subclassing as you might expect. The reason for this becomes apparent when you try.

Suppose you subclass PMQuadtree into PM1Quadtree and PM3Quadtree. The problem here is that the PMQuadtree class is not the one that changes, its inner classes do. What you really need to subclass is the black and gray nodes, since they are the ones that contain the `valid()` code. The easiest way to do that to prevent you from having to rewrite the entire `add()` and `remove()` methods of Black and Gray respectively to change only a few of its lines would be to isolate the validation operation into a method, say `valid()`, that you can override and let dynamic dispatch handle it. But there's a problem with subclassing the inner classes.

The Gray and Black classes refer to each other. Also, presuming you write the methods for PMQuadtree itself (such as add, remove, etc. – I showed only the add and remove for the node classes), you will refer to Gray and Black by name in the base class. This former is really the bigger problem because, suppose you subclass Gray and Black and only change their `valid()` method, for instance in a PM1Gray and PM1Black class. If you only changed the `PM1Black.valid()` method, then when it calls `partition()` it will return a new Gray, not a new PM1Gray. You'd have to override that method to change the type it instantiates when it partitions, and now you'll quickly realize that you're rewriting the code.

6.3.3 Validator Object

What you really want is to isolate the validation operation and be able to subclass that without changing anything else.

You've heard of comparators: external objects that can be passed into sorted structures to change their behavior.

Why not create a validator object? Make your PMQuadtree contain a Validator object as a member, and make your PMQuadtree constructors require that you pass in one of these Validator objects. Make it an interface. Now create three implementations: PM1Validator and PM3Validator. The code for the validator's `valid()` method is only about 10 lines long. You've just magically done what I've done (because I told you how) and now you have a PMQuadtree that becomes one of three orders with only one method changing (and very little code). The best thing to do is create three subclasses, PM1Quad and PM3Quad, who contain private static inner classes which implement the validator and hard code the PM1's default constructor (the one that takes zero parameters) to pass its private validator to the super-class constructor (via a call to super(...) as the first line of the constructor), and make PMQuadtree an abstract class. Note that you can still provide constructors for abstract classes (for the purpose of their subclasses calling them).

This is technically an instance of the Decorator design pattern. Read more about it if you'd like. Design patterns actually do rock. Take CMSC433.

6.4 Adjacency List

An adjacency list is a graph representation stored as a list of lists—the first list contains and is indexed by each of the connected vertices in the graph. Each vertex v in this list points to another list containing all other vertices connected to v by some edge.

A list indexed by something other than a number (its offset from the array's virtual origin in memory) is called many things in different languages: the generic term is "associative array". In Perl this is called a "hash". In Java, this is called a `Map`. A `Map` is any structure that associates one object with another. So for your adjacency list, you want a `Map` which associates a named vertex with a list of its neighbors. `Map` is a Java interface which contains methods that accomplish this goal. The folks at Sun have also implemented a guaranteed logarithmic `Map` called `TreeMap` (located in the `java.util` package).

A `Map` is a collection of entries. An entry is a key-value pair. A value is retrieved by searching the map for its corresponding key. The map is sorted by its keys, so for a `TreeMap`, which sorts the entries for logarithmic-time retrieval, either the keys must be `Comparable` or a `Comparator` must be provided. For an adjacency list, the keys would be the names of the vertices in the graph and the values would be their corresponding neighbor lists. A map is described as "mapping x to y" where x and y are types; since our

vertices will have names, we will represent them with strings. The neighbor lists can really be any collection (a list or a set; edges in our graphs will be unique based on their endpoints, so we will never have more than one edge connecting two vertices x and y). So our adjacency list is really just a map which maps `String`s to `List`s.

The goal for our adjacency list is to discover as efficiently as possible whether an edge (x, y) exists. With `TreeMap`, which is guaranteed to behave logarithmically, given a key k, we can get its corresponding value in logarithmic time. However, our adjacency list maps strings to lists. We can get the list of neighbors for a given vertex in logarithmic time, but unless we can also then search that list for the other endpoint in logarithmic time, our overall complexity will still be linear. Remember that our lists are really sets due to uniqueness of edges. Java has another great class for us called `TreeSet` which is, as you've probably guessed, a guaranteed logarithmic set. If your adjacency list is built using a `TreeMap` which maps `String`s to `TreeSet`s, the overall complexity of locating an edge in your adjacency list will be $O(\log v + \log e)$. Not bad!

You don't need to write any new structures to implement a logarithmic adjacency list—just use the existing Java structures to your advantage. Your `AdjacencyList` class only needs a `TreeMap` which maps `String`s to `TreeSet`s and methods that interact with that `TreeMap`. (Note that as of Java 1.4, there is no formal way, as in C++ templates, to specify that you are creating a `TreeMap` that specifically maps `String`s to `TreeSet`s—that invariant is enforced only by what you put in the map. Java 1.5 addresses this issue). Your adjacency list has no formal requirements about how it is implemented—we will interact with it only via your command parser, so feel free to name the methods anything you want. You are not even required to write an `AdjacencyList` class if you don't feel the need to write one. The real requirement in place is that you must be able to acquire some collection object containing all vertices adjacent to a given vertex in logarithmic time or better. As long as this requirement is met, you have complete liberty in implementing this however you please.

6.5 Shortest Path, and Heaps

As if we haven't assigned enough already, in addition we would like you to implement a Dijkstra's algorithm. What's that I hear? Yes... that's right, a ghost of projects past... Dijkstra's is back, revived from 212. And, please, feel free to adopt and correctly attribute some Dijkstra code, as long as you understand how the algorithm works, as described below for completeness.

**** IMPORTANT ** Although our edges are technically undirected, it is silly to rewrite Dijkstra's algorithm for undirected edges. So, you must implement your adjacency list as if the graph were bidirectional, with each vertex acting, alternately, as the start vertex so that Dijkstra's algorithm will work without modification. The rules about printing out lists of edges (roads) in descending asciibetical order do not apply to shortest path. The path should be printed in the order that the vertices occur in the path.**

6.5.1 Dijkstra's algorithm

This one just keeps coming back to haunt you, doesn't it? As the mastermind of this generation of projects once said, *"you should not be awarded a diploma unless you can adequately explain how Dijkstra's works."* Just Google it. A good resource for Dijkstra's algorithm for Java is here:

renaud.waldura.com/doc/java/dijkstra

Some students may have correctly implemented an efficient shortest/fastest path algorithm in the past. Others have implemented it, but have not made it efficient. Therefore, in order to support shortest path calculations on large data sets within a reasonable amount of time, you will be required to run your shortest path algorithm in $O(E \log V)$ time. In order to achieve this, you may need to restructure parts of your adjacency list. Note that, in general, insertion/deletion from this structure is $O(\log n + \log m)$, where n is the number of nodes in the graph and m is the degree of the graph (the max number of edges incident to a

single vertex). This represents a binary search to find the correct row of the list to find the starting vertex, followed by a binary search to check for existence of the ending vertex.

Recall that in general, graphs are traversed in *depth first order* by expanding a start node, PUSHing all the kids onto a stack, and choosing the next node to expand by POPing from the stack. Similarly, traversing a graph in *breadth first order* is performed by inserting the children of a node into a FIFO queue, and choosing the next node by dequeuing the first element. Furthermore, *greedy* algorithms attempt to produce a global minimum (or maximum) value by choosing the locally optimal value at every step, and not all greedy algorithms actually converge.

Fortunately, Dijkstra's algorithm is a *convergent greedy algorithm* which finds minimum length (cost, or weight as appropriate) paths from a given start vertex to all possible destination vertices for graphs with positive edge weights. Since the algorithm converges, if one is only interested in the minimum cost to a single destination, it may be possible to stop the algorithm early. However, it will be possible to pass the stress test without making this type of an optimization. Note that Dijkstra's algorithm visits or expands vertices (our loci) in priority order, where the priority for our project is the weight.

To implement an efficient Dijkstra's algorithm you will need a priority queue, which is implemented Java version 1.5 (java.util). Or, in the absence of Java 1.5, this can be implemented through clever use of a `TreeSet`/`Map` and `Comparator`s (the problem with a plain set is that you can't have two elements with the same priority, so you need a way to always break ties) to implement a binary heap (the mere *heap* of CMSC 351).

In Part 3, you'll use the graph to implement Dijkstra's algorithm to find the shortest path from a given start vertex to all other vertices from that one (often referred to as single-source all-destination algorithm. The core of the algorithm uses a priority queue to supply vertices in increasing order of distance from the vertices that have already been processed. Since the algorithm works for directed graphs and in our project we will use undirected graphs,

The execution time of the algorithm depends on the method used to implement the priority queue, as discussed briefly in the excerpt from a prior spec. Using the binary heap, the expected runtime of Dijkstra's is $(V + E)\log(V)$, where V is the number of vertices and E is the number of edges. If we instead use a Fibonacci heap, which should appear in Part 3, the expected runtime will e (amortized) $O(V \log V + E)$. It would be nice if you understand where this bound comes from. We've invited Kevin Conroy, 2004 Dorfman Research Prize winner, to explain:

> Allow me to sketch the algorithm to explain the running time (I'm not trying to teach the algorithm here—see your book/class/newsgroup/Google). Every iteration of this algorithm you are guaranteed to find the correct shortest path to exactly one node—so we know right away there will be V iterations. At each state your graph is split in two sections—the 'solved[7]' section, for which you know the correct distances, and the rest, which have some distance values associated with them which may or may not be accurate—this set is stored in some kind of priority queue. An iteration begins by selecting or dequeing the first node (call it 'N') from this queue. This node is the closest to the solved set since elements are removed from the priority queue in increasing order of distance from the solved set. We add the new node to the 'solved' set, and then 'relax' all its edges. Relaxing is when for each node adjacent to N we check to see if it is faster to get to that node through N than though its current best known path (prior to inclusion of N in the solved set). We update distances and back-pointers appropriately (so we know what the shortest path actually is when we finish), and that ends the round. Note that if a node's distance value is changed, its position in the priority queue has to be fixed up somehow. (this is where the magical Fibonacci heap beats the binary heap it's got an advantage in this 'fix up' step). Regardless of the choice of heap, an efficient way to update the priority queue to reflect the new 'solved set' is to merely insert an new element into the queue with the new distance, giving multiple copies associated with the same vertex. Then, if a node at the head of the queue has already been added to the 'solved' set, you merely discard it and get the next node. (this is the approach I will use

[7] The solved section is also referred to as the 'known' set, with the unsolved portion known as the 'unknown' set.

in the explanation), or else to remove the old value before reinserting it. Either works, but the binary heap implementation makes the second method difficult.

Now, how long does all this take. There are V rounds, and in every round we have to pull something out of the front of the priority queue using the binary heap is in $\Theta(1)$, and using the F-heap is amortized $\Theta(1)$. Rather than try and deal with how many elements are added to and removed from the queue in any single round, it is easier to think about how many such operations can occur in the life of the algorithm. Every single edge in the graph has exactly one opportunity to add a vertex to the queue (during a relax operation), so there are $O(E)$ possible insertions.

If we allow multiple entries in the queue corresponding to a single vertex (created by making a new element in the queue when the path length from the as-yet-unknown vertex to the known set changes) then the size of the queue still grows only to $O(E)$. Each addition to the queue takes time in $O(\log E)$. That gives $O(E \log E)$ running time for all queue operations during the life of the algorithm. Together that gives a running time of $O((V + E) \log E)$ And that's the required running time of your algorithm with the binary heap for Part 1.

As we shall see later in lecture, the Fibonacci heap improves priority queue performance for more complex graphs than those you will see in this project. But, the big win is in merging queues, since merging binary heaps is in $O(n \log n)$. However, we do not anticipate requiring you to implement a Fibonacci heap this semester; we just wanted to make sure that you knew that other structures exist to build priority queues. The running time is (amortized) $\Theta(E)$ for all queue operations during the life of the algorithm. Together that gives a running time of $O(V \log E + E)$, which will be the required running if we were to replace the Java priority queue with one based on a Fibonacci heap.

Please be careful with your implementation details. In particular, *do not* use a linear time priority queue if you choose to ignore the API. This nailed a lot of people last semester. We will test your project on ***very*** large inputs, and it will be clear whether your shortest path works in $O((V + E) \log E)$ or linear time. If you choose to implement a Fibonacci heap or a leftist heap because you are bored with Part 1, be forewarned. As will be discussed in class, the downside of the f-heap, relative to this course, is that the f-heap is defined in a way that optimizes the f-heap operations for a manage-your-own-memory language. For example, the f-heap is often defined to be a forest of binomial heaps[8] implemented as a circular linked list. Since the linked list is not optimal for Java, you are welcomed to explore the implementation of your choice in Java, as long as the resulting structure satisfies the asymptotic performance bounds of the f-heap. One way is to use an arraylist of or arraylists instead of linked lists. Regardless, we'll not be checking the implementation of your priority queue–merely the logarithmic asymptotic order of complexity. So, using the priority queue from the API seems to be the most appropriate choice. Consider this just food for thought.

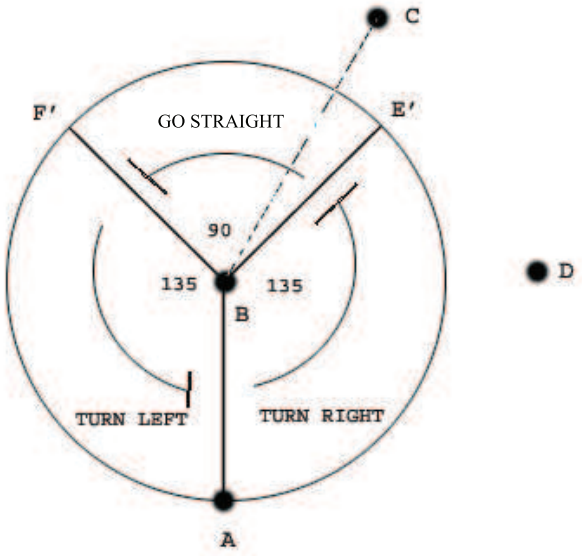
6.6 Annotated Driving Directions

In order to complete MeeshQuest functionality we will be adding the ability to print out annotated driving directions. For driving directions to be useful, people often need the distinction of `GO STRAIGHT`, `TURN RIGHT` or `TURN LEFT`. In Part 3 we introduce a new command which adds these labels to its output.

In order to determine which "direction" someone should go, we will define a simple set of rules based on the angle formed by the road you are on and the road you need to "turn" onto.

On the chart below, line (A,B) represents the road that you are on, or the current road. Line BC represents the next road you want to take on your route. From the chart you will notice that road (B,C) falls within the `GO STRAIGHT` section of the circle. If we drew a line from points B to D then the next road would fall in the `TURN RIGHT` section of the circle.

[8] This holds in the absence of any operations that delete or decrease any key except the min value or the root of a valid heap. We will discuss Fibonacci heaps in conjunction with general trees in Chapter 6 of Shaffer. If you choose to do this for Part 2, you are on your own.



6.6.1 Boundary Conditions

In order to keep all the outputs matching up we will impose boundary conditions. So, if the new road falls upon line (B,E') then `TURN RIGHT`, if it falls upon line (B,F') then `GO STRAIGHT`, and if it falls upon line (A,B) you're in trouble because you shouldn't have two roads on top of each other.

In order to find the angle between points A,B, and C you could use the `Law of Cosine` or `arccos` to derive a formula that would give you the answer OR you could be a savvy Java programming and use one function from the Java API that gives you the angle. Your choice.

6.6.2 Calculating Angles

Here is an example of how one might use Java to accomplish this task:

```
import java.lang.*;
import java.io.*;
import java.awt.geom.*;

public class Angles  {

  public static void main(String[] args) {

    Point2D.Double p1 = new Point2D.Double(2,0);
    Point2D.Double p2 = new Point2D.Double(4,0);
    Point2D.Double p3 = new Point2D.Double(4,4);

    Arc2D.Double arc = new Arc2D.Double();

    arc.setArcByTangent(p1,p2,p3,1);

    System.out.println( arc.getAngleExtent() );
  }
```

```
}
```

7 XML Specifications

7.1 XML Input Basics (Reminders)

A few things to keep in mind—first, XML is case-sensitive when it comes to element and attribute names, so "createCity" is *not* the same as "cReAtEcity". In general:

- Any attribute whose value is obviously a number (such as `x` and `y` in `createCity`) will typically follow the aforementioned regular expression `(0|(-?[1-9][0-9]*))`. In other words, integers. A programmatic description is that you should be able to obtain the integer value of numerical attributes by obtaining their values as strings and passing them through `Integer.parseInt()`. If that method throws an exception when attempting to parse a numerical value, you can assume that attribute's value is invalid.
- Any attribute whose value is defined to be a string (such as `name` in `createCity`) must have a non-numeric first character; however, the remaining part can consist of any combination of numbers, upper case and lower case letters, and the underscore character. Thus, the regular expression for string attributes will be `([_a-zA-Z][_a-zA-Z0-9]*)`.

  But the XML schema and the XmlUtility class provides a method to do type-checking for you so don't worry about this part.

7.2 General XML Output for Part 3 (and Part 4)

Your output will be a series of elements in response to commands, each of which is a reaction to an issued command. Note that we will try to reuse as many of these as possible all semester, and will add new formats only when absolutely necessary. Note that this section will be as stable as we can make it, with modifications only as necessary.

One change we will be making is the addition of an `id` attribute to each `<command>` tag. This is just an integer that identifies the command; this should make it much easier to identify which input command is causing your output to differ from the correct output. You will simply copy the value of the `id` attribute in the input to an `id` attribute in the `<command>` tag of the corresponding output. The numbers below are just examples.

General `<success>` Output

This is an example of the form (in the exact output order) of a general `<success>` tag. All tags will be present even if there are no parameters or outputs.

```
<success>
    <command name="name1" id="1"/>
    <parameters>
        <param1 value="value1"/>
        <param2 value="value2"/>
    </parameters>
    <output/>
</success>
```

General `<error>` Output

This is the form (in the exact output order) of a general `<error>` tag. The `<parameters>` tag will always be present even if there are no parameters to the command. In each command there may be several errors; in this case you will only output the error of *highest priority*. For more information see XML Input specification for each command.

```
<error type="error1">
    <command name="name1" id="2"/>
    <parameters>
        <param1 value="value1"/>
        <param2 value="value2"/>
    </parameters>
</error>
```

General `<fatalError>` Output

This is the form of a General `<fatalError>` tag. This is used when there is a problem with the entire document

```
<fatalError/>
```

General `<undefinedError>` Output

This is the form of a general `<undefinedError>` tag. This is a default error that will be used if there is an error that is not specified in the spec and is discovered post freezing.

```
<undefinedError />
```

General sorting of `<city>` and `<road>` tags.

Ordering of `<city>` tags that are contained within the `<output>` tag is in descending asciibetical order of the city's *name* according to the `java.lang.String.comapreTo()` method *unless* the method for sorting is specified within the command's specification.

Ordering of `<road>` tags that are contained within the `<output>` tag is in descending asciibetical order of the *start* city's *name*. If two roads have the same *start* city, the *end* city's names would appear in descending asciibetical order. The `java.lang.String.compareTo()` method is used *unless* the method of sorting is specified within the command's specification.

printPMQuadtree

Prints the PM quadtree. Since PM quadtrees are deterministic once the order of quadrants has been fixed, your XML should match exactly the primary input/output.

`Parameter:`

(none)

`Possible <output>:`

A `<quadtree>` tag will be contained within *output*. This tag will have one attribute **order** which will be the PM order of the tree you are printing (this is given in the commands tag). and will contain several `<gray>` `<white>` and `<black>` nodes.
The *first* node in the *quadtree* will be the root node, this node can be *gray* or *black*, then the rest of the PM Quadtree follows. Remember, the exact structure of the *quadtree* will be represented by the XML output.

`<gray>`

Nodes will contain *4* children nodes with ordering decided by the order of the nodes within the actual *gray* node in your PM Quad Tree.`<gray>` nodes will have the attributes **x** and **y**, these are integers that identify the location of the *gray* node. They will appear as such

```
<gray x="72" y="40">
    ...
</gray>
```

`<black>`

Nodes in a PMQuadtree can contain at most one `<city>` and possibly several `<road>` tags, depending on the type of tree (PM1,PM3).

The format of `<city>` and `<road>` tags will be as such:

```
<city name="city1" x="coordx" y="coordy" color="color1" radius="radius1"/>
<isolatedCity name="city1" x="coordx" y="coordy" color="color1" radius="radius1"/>
<road start="city_a" end="city_b"/>
```

The **ordering** of the city and road tags within the black node will be as such, first the `<city>` or `<isolatedCity>` tag will be printed (if it exists) then the `<road>` tags will be printed in the ordering specified in the upcoming section. Since the roads are undirected, within the road tag, the *end* city is the one whose name is first in descending asciibetical order.

Black nodes have the attribute **cardinality** which is the number of cities and roads contained within the black node. Black nodes will appear as such:

```
<black cardinality="card">
    ...
</black>
```

`<white>`

Nodes represent an empty node in your PM Quad Tree and will appear as such;

```
<white/>
```

Possible *<error>* types:

mapIsEmpty

<success> Example:

```
<success>
    <command name="printPMQuadtree" id="1"/>
    <parameters/>
    <output>
        <quadtree order="3">
            <gray x="128" y="128">
                <black cardinality="2">
                    <city color="red" name="Berlin" radius="50" x="128" y="128" />
                    <road end="Geneva" start="Berlin" />
                </black>
                <black cardinality="2">
                    <city color="red" name="Berlin" radius="50" x="128" y="128" />
                    <road end="Geneva" start="Berlin" />
                </black>
                <gray x="64" y="64">
                    <white />
                    <gray x="96" y="96">
                        <white />
                        <black cardinality="2">
                            <city color="red" name="Berlin" radius="50" x="128" y="128" />
                            <road end="Geneva" start="Berlin" />
                        </black>
                        <black cardinality="3">
                            <city color="orange" name="Geneva" radius="80" x="90" y="70" />
                            <road end="Madrid" start="Geneva" />
                            <road end="Geneva" start="Berlin" />
```

```
                                </black>
                                <black cardinality="2">
                                    <isolatedCity color="black" name="Paris" radius="10" x="50" y="30"/
                                    <road end="Geneva" start="Berlin" />
                                </black>
                            </gray>
                            <black cardinality="2">
                                <city color="yellow" name="Madrid" radius="60" x="45" y="50" />
                                <road end="Madrid" start="Geneva" />
                            </black>
                            <black cardinality="1">
                                <road end="Madrid" start="Geneva" />
                            </black>
                        </gray>
                        <black cardinality="2">
                            <city color="red" name="Berlin" radius="50" x="128" y="128" />
                            <road end="Geneva" start="Berlin" />
                        </black>
                    </gray>
                </quadtree>
            </output>
        </success>
```

printAvlTree

Prints the AVL-g tree.

`Parameter:`

(none)

`Possible <output>:`

An `<AvlGTree>` node will be contained within the `output` tag and is the root of the AVL-g xml tree. This tag has three required attributes: `cardinality`, whose value should be the size (number of keys) contained in the tree; `height`, or the number of levels in the tree (a tree with levels 0 to $w-1$ has height $w-1$); and `maxImbalance`, the maximum height difference between a node's left and right subtrees (in other words, g).

`<node>`
: Represents a node in the AVL-g tree. This element has two attributes, key and value, which describe the key–value pair contained in the AVL node.

`<emptyChild>`
: Represents an empty subtree. This element has no attributes.

`Possible <error> types:`

emptyTree

`<success> Example:`

```
  <success>
    <command name="printAvlTree"/>
    <parameters/>
    <output>
      <AvlGTree cardinality="1" height="0" maxImbalance="3">
```

```
        <node key="Baltimore" value="(140,37)">
          <emptyChild/>
          <emptyChild/>
        </node>
      </AvlGTree>
    </output>
  </success>
```

The testing of your messages won't be as stringent as the fact that you reported a success in the first place. More precise information about what the messages are for each command is contained in the **Input** section of this document.

7.3 Commands for Part 3

`commands`

This is the root element of the XML tree for the input files. It contains attributes that will affect the behavior of your command interpreter.

The first attributes are `spatialWidth` and `spatialHeight`, both **Unrestricted Integers**, but in the range of 32 bit 2's complement integers). These two attributes define the rectangular region the spatial data structure can store. Note that the lower left corner of this region is always $(0, 0)$, the origin, as we will not be dealing with negative **city coordinates** in this project. Thus, the spatial structure's bounding volume is the rectangle whose **lower left corner** is $(0, 0)$ and whose width and height are given by these two parameters. You must setup your spatial structure to be centered accordingly. Note that although all of our coordinates will be given as integers, you may need to center your spatial structure around a coordinate that is not an integer; so, make sure you plan accordingly.

Observe that these two values have an impact on the success of the `<mapCity>` and the `<mapRoad>` commands, but Do Not affect the success of the `<createCity>` command. Cities can be created in the data dictionary with coordinates outside the boundaries of this range, on the boundary as well as inside this range. For Part 3 and Part 4, Cities on the top and right boundaries of this range can also be "mapped" or inserted into the PM quadtree.

Additionally, the attribute `pmOrder` specifies which variant of the PM quadtree should be used. Possible values are 1 for PM1 quadtree and 3 for PM3 quadtree. For Part 3, you can safely assume that the `pmOrder` is always 3

`Possible errors:`

If there is any problem with the attributes in the `<command>` tag a `<fatalError>` tag is outputted without a `<results>` tag, and the program exits.

createCity

Creates a city (a vertex) with the specified name, coordinates, radius, and color (the last two attributes will be used in later project parts). A city can be successfully created if its name is unique (i.e., there isn't already another city with the same name) and its coordinates are also unique (i.e., there isn't already another city with the same (x, y) coordinate), and the coordinates are non-negative integers. Names are *case-sensitive*.

`Parameters:  (In output order)`

name
x
y
radius
color

Possible *<output>*:

(none)

Possible *<error>* types (In priority order):

duplicateCityCoordinates
duplicateCityName

<success> Example:

```
<success>
    <command name="createCity" id="1"/>
    <parameters>
        <name value="Annapolis"/>
        <x value="12"/>
        <y value="14"/>
        <radius value="15"/>
        <color value="red"/>
    </parameters>
    <output/>
</success>
```

clearAll

Resets all of the structures including the PM Quadtree, clearing them. This has the effect of removing every city and every road. This command cannot fail, so it should unilaterally produce a `<success>` element in the output XML.

Parameter:

(none)

Possible *<output>*:

(none)

Possible *<error>* types:

(none)

<success> Example:

```
<success>
    <command name="clearAll" id="2"/>
    <parameters/>
    <output/>
</success>
```

listCities

Prints all cities currently present in the dictionary. The order in which the attributes for the `<city>` tags are listed is unimportant. However, the city tags themselves must be ordered by name or by coordinate, as per the `sortBy` attribute in the `listCities` command, whose two legal values are `name` and `coordinate`. The ordering by name is descending asciibetical name according to the `java.lang.String.compareTo()` method, and the ordering by coordinate is discussed in the Part 1 spec. To reiterate, coordinate ordering is done by comparing y values first; for cities with the same y value, one city is less than another city if its x value is less than the other. This command is only

successful if there is at least 1 (1 or more) cities in the dictionary.

`Parameter:`

sortBy

`Possible <output>:`

A `<cityList>` tag will be contained in *output* and will contain 1 or more city tags of the form:
`<city name="city1" x="coordx" y="coordy" color="color1" radius="radius1"/>`

`Possible <error> types:`

noCitiesToList

`<success> Example:`

```
<success>
    <command name="listCities" id="3"/>
    <parameters>
        <sortBy value="name"/>
    </parameters>
    <output>
        <cityList>
            <city name="Derwood" x="5" y="5" color="blue" radius="90"/>
            <city name="Annapolis" x="19" y="20" color="red" radius="40"/>
        </cityList>
    </output>
</success>
```

printAvlTree

Takes no arguments, and merely prints the structure of your AVL-g tree using the rules from Section 7. This function is used to serialize your AVL-g tree so that its correctness can be checked.

mapRoad

Inserts a road between the two cities (vertices) named by the **start** and **end** attributes in PM quadtree. The obvious error conditions follow: the **start** or **end** city does not exist or are the same; the **start** or **end** vertex is an `isolated city` (see below); the road from **start** to **end** already exists; the road is outside of the spatial area defined by (0,0) and spatialWidth/spatialHeight. Note that it is possible for (part of) a road to be within the spatial area, but not its endpoints. However, there are two other potential error conditions you must handle when adding a road to the spatial map, but not until part 3.

Note: THIS IS FOR Part 4: First, the new road should not intersect any road already mapped at a point other than a vertex of the road (this is a requirement of the PM quadtree family). Also, you must check that inserting this road into the PM quadtree will not cause the tree to be partitioned beyond the smallest size. We would prefer to have you implement this correctly the first time; but, experience has shown that not everyone will be comfortable doing this the first time through. Thus, the correct functionality is included in the spec, but will not be fully tested until Part 4.

`Parameter:`

start
end

`Possible <output>:`

The road mapped will cause a *roadCreated* tag in the `<output>` and will appear as such:
`<roadCreated` $start = "city_a"$ $end = "city_b"$`/>`

`Possible <error> types (In priority order):`

startPointDoesNotExist
endPointDoesNotExist
startEqualsEnd
startOrEndIsIsolated
roadAlreadyMapped
roadOutOfBounds

`<success> Example:`

```
<success>
    <command name="mapRoad" id="4"/>
    <parameters>
        <start value="Baltimore"/>
        <end value="Annapolis"/>
    </parameters>
    <output>
        <roadCreated start="Baltimore" end="Annapolis"/>
    </output>
</success>
```

mapCity

maps an `isolated city` that is not an endpoint of any roads, i.e., the city is not connected to any roads. An isolated city cannot be turned into a normal city and vice versa without being unmapped from the spatial structure first. Thus, when trying to map an isolated city to the endpoints of a road, cityAlreadyMapped error should be returned. `Parameter:`

name

`Possible <output>:`

(none)

`Possible <error> types (In priority order):`

nameNotInDictionary
cityAlreadyMapped
cityOutOfBounds

`<success> Example:`

```
<success>
    <command name="mapCity" id="4"/>
    <parameters>
        <name value="Baltimore"/>
    </parameters>
    <output/>
</success>
```

printPMQuadtree

prints the PMQuadtree corresponding to the command parameter pmOrder. Because we have quadrants numbered as: (1) NW, (2) NE, (3) SW, (4) SE, the PM quadtrees are deterministic, you should use the output rules established in Section 7 to assure that your XML matches ours.

saveMap

Saves the current map to a file. The image file should be saved with the correct name. It should match our image file: same dimensions, same cities, same colors, same partitions, etc. Roads should be drawn as black (java.awt.Color.BLACK) line segments. Everything else from the last part will be the same (e.g. Cities should be drawn as black (java.awt.Color.BLACK) named points) with one exception: **To differentiate from Roads, quadtree partitions should now be gray (java.awt.Color.GRAY) crosses.**

`Parameters (In output order):`

name - filename to save the image to

`Possible <output>:`

(none)

`Possible <error> types:`

(none)

`<success> Example:`

```
<success>
    <command name="saveMap" id="6"/>
    <parameters>
        <name value="map_1"/>
    </parameters>
    <output/>
</success>
```

rangeCities

Lists *all* the cities present *in the spatial map* within a `radius` of a point `x,y` *in the spatial map*. Cities on the boundary of the circle are included, and `x,y` are integer coordinates. That is, only cities that are in the spatial structure, in this case, the PM quadtree, are relevant to this commmand. `<success>` will result from the existence of at least one `<city>` that satisfy the range check condition. If none do, then an `<error>` tag will be the result. In the special case where the radius is 0 and there is a city at the range point, this city should be listed. It should be noted that the radius attribute for a city does not factor into this calculation; all cities are considered points.

If the `saveMap` attribute is present, the current map will be saved to an image file (see saveMap). The image file should be saved with the correct name. It should match our image file: same dimensions, same cities, etc. How to keep track of your graphic map is discussed in saveMap. Printing it out is discussed there too. The main difference with saveMap is that the image file should have a blue (java.awt.Color.BLUE) unfilled circle centered at the (x,y) values passed in with the radius passed in. Because CanvasPlus does not behave well when shapes exceed the bounds of the spatial map, the saveMap attribute will only be present when an entire range circle lies inclusively within the bounds of the spatial map.

`Parameters (Listed in output order):`

x
y
radius
saveMap (optional) - image filename

Possible *<output>*:

The *output* will contain one `<cityList>` which will contain the list of cities. This is an example of a city tag:
`<city name="city1" x="coordx" y="coordy" color="color1" radius="radius1"/>`
The `<city>` tag is used for both isolated cities and non-isolated ones. The cities should be printed in descending asciibetical order of the names according to java.lang.String.compareTo().

Possible *<error>* types:

noCitiesExistInRange

<success> Example:

```
<success>
    <command name="rangeCities" id="7"/>
    <parameters>
        <x value="1"/>
        <y value="1"/>
        <radius value="100"/>
    </parameters>
    <output>
        <cityList>
            <city name="Derwood" x="20" y="40" color="blue" radius="23"/>
            <city name="Annapolis" x="20" y="30" color="red" radius="12"/>
        </cityList>
    </output>
</success>
```

rangeRoads

Lists all the roads present in the spatial map that intersect the circle defined by the given `radius` and point (`x, y`). Roads which are tangent to the circle are considered in the range, i.e. intersection can occur at just one point. `x` and `y` are integer coordinates. `<success>` will result from the existence of at least one `<road>` that satisfies the range check condition. If none do, then an `<error>` tag will be the result. If the radius is 0 and there is a city at the range point, roads emanating from (or terminating at) that city will be listed. If the `saveMap` attribute is present, the current map will be saved to an image file (see saveMap), with features looking exactly like those of rangeCity.

Parameters (In output order):

x
y
radius
saveMap (optional) - image filename

Possible *<output>*:

The *output* will contain one `<roadList>` tag, which will have one or more `<road>` child elements. The `<road>` element will appear as such;
`<road start="city_a" end="city_b"/>`

The roads should be printed in descending asciibetical order of the names according to java.lang.String.compareTo(), with the rule that roads should be compared first by the names of their start points. For two roads with the same starting city, an road is less than another road if its endpoint is asciibetically greater than the endpoint of the other road.
If the `saveMap` attribute is present, the current map will be saved to an image file (see saveMap) just like in rangeCities.

`Possible <error> types:`

noRoadsExistInRange

`<success> Example:`

```
<success>
    <command id="18" name="rangeRoads"/>
    <parameters>
        <x value="512"/>
        <y value="512"/>
        <radius value="300"/>
    </parameters>
    <output>
        <roadList>
            <road end="City6" start="City5"/>
            <road end="City6" start="City3"/>
            <road end="City5" start="City3"/>
        </roadList>
    </output>
</success>
```

nearestCity

Will return the name and location of the closest city to the specified point in the spatial map, where the set of valid cities excludes isolated cities. To do this correctly, you may want to use the *PriorityQueue* from part 2–otherwise, you might not be fast enough. The ordering by name is asciibetical according to the `java.lang.String.compareTo()` method. So, to repeat, case of a tie (two cities equally far away from the point), choose the city with the asciibetically greatest name.

`Parameters (In output order):`

x
y

`Possible <output>:`

The *output* will contain one *city* tag which is the nearest city. This is an example of a city tag:
`<city name="city1" x="coordx" y="coordy" color="color1" radius="radius1"/>`

`Possible <error> types:`

cityNotFound

`<success> Example:`

```
<success>
    <command name="nearestCity" id="8"/>
    <parameters>
        <x value="1"/>
        <y value="2"/>
    </parameters>
    <output>
        <city name="Annapolis" x="20" y="30" color="red" radius="12"/>
    </output>
</success>
```

nearestIsolatedCity

Will return the name and location of the closest isolated city to the specified point in the spatial map. To do this correctly, you may want to use the *PriorityQueue* from part 2–otherwise, you might not be fast enough for large data sets. In the case of a tie (two cities equally far away from the point), choose the city with the asciibetically greatest name. The ordering by name is asciibetical according to the `java.lang.String.compareTo()` method.

`Parameters (In output order):`

x
y

`Possible <output>:`

The *output* will contain one *city* tag which is the nearest city. This is an example of a city tag:
`<isolatedCity name="city1" x="coordx" y="coordy" color="color1" radius="radius1"/>`

`Possible <error> types:`

cityNotFound

`<success> Example:`

```
<success>
    <command name="nearestIsolatedCity" id="8"/>
    <parameters>
        <x value="1"/>
        <y value="2"/>
    </parameters>
    <output>
        <isolatedCity name="Annapolis" x="20" y="30" color="red" radius="12"/>
    </output>
</success>
```

nearestRoad

Will output the nearest road to the specified point in the spatial map. Ties will be broken by applying the descending asciibetical rule to the start city names, then end city names. Again, this is done using the`java.lang.String.compareTo()` method.

`Parameters (In output order):`

x
y

Possible *<output>*:

The *output* will contain one *road* tag which is the nearest road. This is an example of a road tag:
`<road start="city1" end="city2"/>`

Possible *<error>* types:

roadNotFound

<success> Example:

```
<success>
    <command name="nearestRoad"/>
    <parameters>
        <x value="1"/>
        <y value="2"/>
    </parameters>
    <output>
        <road start="Annapolis" end="Derwood"/>
    </output>
</success>
```

nearestCityToRoad

Will return the name and location of the closest city in the spatial map to the specified road in the spatial map. The road is specified by the `start` and `end` cities. Since roads are undirected, specifying `[start=A,end=B]` is identical to specifying `[start=B,end=A]`. Obviously the city will not be one of the endpoints of the road or this command would be trivial. Ties broken by the city with the asciibetically greater name.

Parameters (In output order):

start
end

Possible *<output>*:

The *output* will contain one *city* tag which is the nearest city (isolated or not). This is an example of a city tag:
`<city name="city1" x="coordx" y="coordy" color="color1" radius="radius1"/>`

Possible *<error>* types:

roadIsNotMapped
noOtherCitiesMapped

<success> Example:

```
<success>
    <command name="nearestCityToRoad"/>
    <parameters>
        <start value="Annapolis"/>
        <end value="Derwood"/>
    </parameters>
    <output>
        <city name="CollegePark" x="20" y="30" color="red" radius="12"/>
    </output>
</success>
```

shortestPath

Prints the shortest path and direction traveled (from the perspective of someone driving down the current road) between the cities named by the **start** and **end** attributes, where the length of the road is the Euclidean distance between the cities. Should a tie be encountered when constructing the shortest path, the vertex with the smaller asciibetical city name will be selected as appropriate using the `java.lang.String.compareTo()` method.

Success is reported if such a path exists. We have no plans to test this on unconnected graphs, and will restore any points lost should this happen by accident.

The `<success>` element in response to this command is empty and has no **message** attribute. It has one child element, `<path>`, which itself has a sequence of road and direction elements (`<right/>`, `<left/>`, and `<straight/>`) as children. The `<path>` element has two attributes which must be set: **length** which reports the double precision[9] length of the total path, rounded to 3 decimal places, and **hops** which indicates the number of unique **edges** traveled to get from the start to the end.

To make things more fun, we are going to use CanvasPlus sometimes when calling shortestPath. If a `saveMap` attribute is present, you need to create a new CanvasPlus object containing the shortestPath and save it to an image file with the value of the saveMap attribute. Here's what the shortest path should look like: You should include a black rectangle for the bounds of the spatial map just like in saveMap. All cities and roads contained by the path should be in this visual map. The start point should be green (java.awt.Color.GREEN). The end point should be red (java.awt.Color.RED). All cities in between and all connecting roads should be blue (java.awt.Color.BLUE). Since the customer doesn't care about how the spatial map is stored, we don't need to see the rest of the spatial map (partitions and other cities and roads). To reiterate, to keep it simple, just create a new CanvasPlus object and include on the cities and roads on the path. Lastly, remember to dispose of the CanvasPlus object when you are done with it.

Finally, to make things even make things even more fun, we are going to turn your shortestPath output into readable HTML. If a `saveHTML` attribute is present, you follow the same steps you did with the `saveMap` attribute instructions, but now name it the value of the saveHTML attribute. You need to do this because the HTML document will need your shortestPath image and your success node. But don't worry about any extra work here, it's all pretty much been done for you. First of all we need to make a new Document object and make a copy of our success node since it belongs to our results document. Then we just need to transform the XML using an XSLT (eXtensible Stylesheet Language Transformations–don't worry if you've never heard of this) file that I wrote into a pretty HTML document. Since Java's XML outputter prints attributes in alphabetical order it can be confusing to read them so hopefully this will make it easier for you to test your Dijkstra classes. The code for you to do this is below (and the XSLT file should be in the newest iteration of the JAR file):

```
org.w3c.dom.Document shortestPathDoc = XmlUtility.getDocumentBuilder().newDocument();
org.w3c.dom.Node spNode = shortestPathDoc.importNode(successNode, true);
shortestPathDoc.appendChild(spNode);
XmlUtility.transform(shortestPathDoc, new File("shortestPath.xsl"), new File(saveHTMLName + ".html"));
```

Note that both saveMap and saveHTML attributes could be present without a conflict, instead, two image files and an HTML file would be produced. As always, examples will be provided.

`Parameters:  (In output order)`

[9] Yes, it seems odd that a double precision floating point value would round differently than single precision, but it does, as too many students to count found out earlier.

start
end
saveMap (optional)
saveHTML (optional)

Possible *<output>*:

A `<path>` tag with attributes *length, hops* will be contained in *output* and will contain 1 `<road>` tag, followed by zero or more sequences of a direction element followed by a road element. These should be in the order of the path taken and will appear as follows:
`<road start="road1" end="road2"/>`

Possible *<error>* types (In priority order):

nonExistentStart
nonExistentEnd
noPathExists

<success> Example:

```
<success>
    <command name="shortestPath" id="9"/>
    <parameters>
        <start value="Annapolis"/>
        <end value="Derwood"/>
    </parameters>
    <output>
        <path length="12.000" hops="4">
            <road start="Annapolis" end="Bowie"/>
            <left/>
            <road start="Bowie" end="Washington"/>
            <right/>
            <road start="Washington" end="Bethesda"/>
            <straight/>
            <road start="Bethesda" end="Derwood"/>
        </path>
    </output>
</success>
```

Do note: except for one input set per project part, we will provide syntactically error-free XML, which means that we will only test your error checking on one set of test inputs. However, we will check the syntactic validity of *every* output file you provide; so make sure that your output XML can be validated against the schema we provide for each part, or you risk losing substantial credit.

8 General Policies

We will be using the submit server to test your project, and provide no further information regarding submission here, unless a paragraph or two is written by someone who has a clue what should go here.

We have given you basic I/O files and any further testing is your problem. However, students are encouraged to produce test files and help each other (and us) validate tests prior to final submission.

8.1 Grading Your Project

Projects will be graded on a point system, whereby points will be awarded for each test successfully passed, and the test results for a given structure will be scaled according to the point breakdown given in the spec and the relative part weights given in the syllabus. As you will soon see when you begin implementing your first data structure, we will only ask that you implement a fraction of the functionality described by the Java API for the interfaces we are going to ask you to implement.

However, we strongly believe in rewarding students who go above and beyond the requirements of a project, and occasionally a few points may be awarded for exceeding the project specifications by running extra credit tests on your project. Don't get too excited—extra credit will not be massive, and will boost your project grade just a bit. It's meant more as a means to identify students who are making remarkable progress on the project or to provide those who lost points on an earlier project part with an opportunity for redemption. However, in general, not all students will be able to complete all parts of the project on time. So, please, don't be too ashamed to submit code that only provides partial functionality. We recognize that different students have have varying levels of programming and educational experience; so it is not necessary to get 100% of all available project points to do well in the course. But, if you find yourself never passing more than half of the tests, you should visit Dr. Hugue (even by email) or the teaching assistant of your choice and ask for help.

8.1.1 Criteria for Submission–Spring 2017

Your file must satisfy the following criteria in order to be accepted:

- It must be a tar or jar file.
- It must have the extension ".tar" or ".jar".
- It must contain two files: 1) README, 2)package cmsc420.meeshquest.part2
- It must NOT contain any files with extension `class` or `jar`.
- It must compile successfully with the command:

```
javac -classpath xml.jar MeeshQuest
```

- It must pass the primary input (or some subset thereof).
- There are no release tests per se–merely public and private tests.

8.1.2 README file Contents

Your README file must contain the following information:

- your name
- your login id
- citations of sources you used, including documentation of all portions of the code that were borrowed or adapted or otherwise not written from scratch by you.
- any additional information you think might be relevant–including documentation of non-working parts of the project–this can be useful later when you've forgotten what works or doesn't.

If you leave out the `README` your project will fail to submit!

8.1.3 Project Testing and Analysis

Your program will be tested using the command:

```
java -classpath xml.jar :. MeeshQuest< primary.input > primary.output
```

We will then check that your output is correct. You are responsible for matching the *letter of the specification*, which is why we freeze a week before the project is due. If you find a discrepancy between the supplied Primary I/O and the specification, please report it immediately so that the Primary can be corrected, and any remaining ambiguities can be clarified. Remember, we have to return points if your output matches the specification and we don't. And, helping us keep the primary correct will benefit us all in the long run.

8.2 Grading

There will be several methods used to grade your projects. Your projects will be graded by running them on a number of test files with pre-generated correct (we hope) output. Your output will have all punctuation, blank lines, and non-newline whitespace stripped before diffing similarly cleaned files.

Some of your data structures may be included with the TAs own testing code to test their efficiency and correctness. Thanks to the miracle of automation you should expect your projects to be run on very very large inputs.

8.2.1 Testing Process Details

Each part of your project will be subjected to a variety of tests with points assigned based on the importance of the structure to the project, and the relative difficulty of producing working code. Note that we will do our best to assure partial credit where possible by decoupling independent tests.

Evaluation of your project may include, but is not limited to, the following testing procedures:

- Basic I/O: when the structure is deterministic, the test outputs are processed using a a special "XML diff" that parses your output into a DOC (just like your program does) and then compares it with a valid output. Thus, you can put whitespace wherever you want it, and order attributes within tags however you like. However, your output MUST be syntactically valid XML. IF WE CANNOT PARSE YOUR OUTPUT, THEN IT WILL FAIL THE TEST IMMEDIATELY.
- Rule-Based: when multiple correct answers, we will use code-based verification that your results satisfy the required properties.
- Interface: when you are required to implement an interface, we will use our test drivers to execute your code. (Not for Part 1)
- Stress and Timing: used to verify that your algorithms meet the required asymptotic complexity bounds.

8.2.2 Small, Yet Costly, Errors

In general, your project will either completely pass each test, or completely fail it. In particular, the following "small" mistakes have been known to fail projects completely in many tests:

- Forgetting to match each "open" XML tag (e.g. `<tree>`) with a "close" XML tag (eg. `</tree>`)
- Forgetting to include the `/` at the end of single tags (eg. `<success />`)
- Sloppy capitalization (eg. `<Success/>`)
- Misspellings (eg. `<success/>` or `<success command = "creatCity">` ....

On rare occasions, the tests may be extended to allow minor errors to pass, but only if we find that the cost of that error is significant and unfair. However, in the large, you'd do better to get this stuff right the first time, than expect partial credit.

The tests will try to test mutually exclusive components of your projects independently. However, if you don't have a dictionary which at least correctly stores all points, and some 'get lost', you may still end up failing other tests since they all require a working dictionary. This does not reflect double jeopardy on the part of the test data, since it is always possible to use some other structure (such as the Java `TreeMap`) for the data dictionary, and receive points for portions of the tests that merely reference the data in the dictionary. Do not hesitate to ask for suggestions as to how to work around missing project functionality.

8.3 Standard Disclaimer: Right to Fail (twice for emphasis!)

As with most programming courses, the instructor reserves the right to fail any student who does not make a good faith effort to complete the project.

If you have problems with completing any given part of the project please talk to Dr. Hugue immediately—do not put it off! While some may enjoy failing students, Dr. Hugue does not; so please be kind and do the project, or ask for advice immediately if you find yourself unable to submit the first two parts of the project in a timely manner. A submission that gets only 20 or 30 points is considerably better for you than no submission at all. And, we employ the buddy system. If you are still lost after Part 2, please ask for help. Recognizing when you need help and being willing to as for it is often a sign of maturity.

8.4 Integrity Policy

Your work is expected to be your own or to be labeled with its source, whether book or human or web page. Discussion of all parts of the project is permitted and encouraged, including diagrams and flow charts. However, pseudocode writing together is discouraged because it's too close to writing the code together for anyone to be able to tell the difference.

Since the projects are interrelated, and double jeopardy is not our goal, we have a very liberal code use and reuse policy.

- In general, any resources that are accessed in producing your code should be documented within the code and in a `README` file that should be included in each submission of your project.
- However, adopting any source code or pseudocode of a working B tree, B+ tree, or PM quadtree from Internet search is explicitly prohibited. You are not allowed to use it in any form. You may use all resources provided or linked by this course.
- First and foremost, use of code produced by anyone who is taking or has ever taken CMSC 420 from Dr. Hugue requires email from provider and user to be sent to Dr. Hugue. That means that any student who wants to share portions of an earlier part of the project with anyone must inform Dr. Hugue and receive approval for code sharing prior to releasing or receiving said code. This also applies to sharing code from another course with a friend. Please, please, ask.
- Second, since we recognize that the ability to modify code written by others is an essential skill for a computer scientist, and that no student should be forced to share code, we will make working versions of critical portions of the project available to all students once grading of each part is completed, or even before, when possible.
- Dr. Hugue is the sole arbiter of code use and reuse and reserves the right to fail any student who does not make a good faith effort on the project. Violators of the policies stated herein will be referred to the Honor Council.

Remember, it is better to ask and feel silly than not to ask and receive a complimentary F or XF.

8.4.1 Code Sharing Policy

During the semester we may provide you with working solutions to complete portions of the project. It is legal to look at these solutions, adopt pieces of them, and replace any part of your project with anything from them so long as you indicate that you *accessed* this code in your `README`.

Furthermore, any portion of your code that contains any portion of the distributed work should contain identifying information in the comments. That is, indicate the source or inspiration of your code in the file where it was used, as well as in your README. It is a good idea to wrap shared code with comments such as "`Start shared code from source XYZ`" and "`End shared code from source XYZ.`" You may also use comments such as "`Parts of this function/file were based on code from source XYZ.`" You cannot err by including this information too often. Making it easy for us to find, makes it easy for us to recognize your compliance with the rules.

Failure to properly document use of distributed code in your project could result in a violation of the honor code. Indicate the distribution solution(s) on which your code is based.

References

[1] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: elements of reusable object-oriented software*. Addison-Wesley, One Jacob Way, Reading, MA 01867, USA, 1995.