CMSC420 Project - Spring, 2017
Draft Part 4, Version 4.1

The BIG 420 Project[*]
Part 4 will be due at 11:59PM on max(syll , submit server) (plus 48 hour grace period)

Last Modified April 8, 2017

Contents

1 Introduction and General Overview

Welcome to the fourth and final part of the Spring, 2017, edition of Dr. Hugue's CMSC 420 project. This project relies on specific knowledge of the part 1, 2, and 3 specifications and builds on them in new and exciting ways. 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.

[*]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.

2 MeeshQuest Components

By this point we suppose that you fully understand the mediator and dictionary structures. We also hope that you have a working understanding of spatial data structures—because it's time for you to design your own.

Part 4 will introduce the concept of zooming. We now add the additional concept of a metropole. A metropole (also known as a metropolis - but alas, the plural metropolises is infinitely confusing) is a large urban area that can contain many cities. For this project, each metropole will have its own unique collection of cities and roads. A metropole has its own set of coordinates relative to other metropoles. Additionally, each city will have its own set of coordinates relative to other cities in the same metropole. In order to avoid confusion, whenever we refer to the coordinates of a metropole relative to other metropoles we will use the term remote coordinates. Whenever we refer to the coordinates of a city relative to another city in the same metropole, we will use the term local coordinates. A city has both remote coordinates (the coordinates of its associated metropole) and local coordinates (the coordinates of its position relative to other cities in the same metropole).

To give an example, suppose City A has remote coordinates (1, 2) and local coordinates (1, 0). Suppose City B has remote coordinates (1, 2) and local coordinates (0, 0). City A and City B lie inside of the same metropole because their remote coordinates are the same. However, City A and City B do not occupy the same position inside of the metropole at remote coordinates (1, 2) because their local coordinates are different.

We will refer to the collection of remote coordinates as the remote spatial map, which represents the metropoles positioned relative to each other. We will refer to the collection of cities and roads corresponding to a particular metropole as a local spatial map. Note that this means that there is only one remote map, but there is a local map for each metropole.

You may only map a road to another city in the same metropole, meaning you can hopefully use your PM Quadtree in some way. To traverse between metropoles, we're introducing a new meaning for the isolated cities–they now correspond to "airports" that will take you between your metropole and another metropole.

Finally, you will also have to expand your data dictionary, allowing it to delete items from the Avl-g Tree. Your Avl-g Tree also needs to implement the rest of the SortedMap interface.

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	
1	Treemaps (i/f)	n/a	n/a	n/a	Any i
2	Treemaps (i/f/d)	PR Quad (i/f/d)	n/a	n/a	Any i
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 Part 4: Sorted Map, AvlGTree, PM quadtrees, and True Delete

In part 3, you were asked to consider replacing your TreeMaps with AVL-g trees (modified AVL trees that allow their subtree heights to vary by at most g). For more on the AVL-g, see the description in part 3: www.cs.umd.edu/users/meesh/cmsc420/ProjectBook/part3. It's been cut for brevity.

In part 4, you'll be asked to implement a (immediate) delete for your AVL-g. Unlike lazy deletion, immediate deletion deletes a node as soon as delete is called and rebalances immediately.

Now that `delete()`'s on the table, so's all of `SortedMap`. Notably, `map.remove()` and all of the `remove()`s from the various views (`entrySet()`, etc.) must work. See the Java API for all the gory details. In Spring 2017, you do not need to implement `keySet(), values(), headMap(), tailMap()`. You could, theoretically, now expunge any lingering `TreeMap`s from your code. I'd recommend you keep your guarded structure around, however: you need a working dictionary for much of the project, and there's really nothing we can do if yours mysteriously leaks cities.

One last comment on the AVL-g: the tricky parts of `entrySet()` (and all other views) become much more important here. Insertion wasn't supported in all of those views, as you recall, so the need to reflect changes was really only one-way. Deletion is supported from the objects returned by `entrySet()` and `subMap()` which makes this relation two-way. (You hopefully already know how to do this, though: it was always two-way for `subMap()`.)

Part 4 will also require you to implement PM1 `insert()` and PM `delete()`. You may want to refer to the part 3 spec for the piece on validators and code reuse. There's even pseudocode for `delete()`. Because the spatial structure now incorporates zooming and metropoles, you'll need a way to accommodate metropole coordinates. One such way is a PR Quadtree from part 2, since you will only be storing coordinates (see below).

5 3D Spatial Structures

As mentioned previously, your major task for this project will be designing and implementing your own structure. We've made it easier for you by giving you restraints on the roads that can be mapped. For a road to be mapped, both cities' remote coordinates must be the same. This tells us that the road exists within the scope of a single metropole and does not extend to any other metropole.

We will allow you to use either a PM1 or PM3 quadtree whenever possible, depending on which one you feel you implemented more solidly. That is, when `pmOrder` is not given, it is up to you to decide what value (1 or 3) you are more comfortable with. However, if pmOrder is specified, you are required to implement that particular type of PM Quadtree. We will also allow you to use whatever structure you want to aggregate the trees, as long as you can perform search queries on $O(\log n)$.

Your structure will still be graded using `printPMQuadtree`, but we now add `remoteX` and `remoteY` parameters to this command—you only have to print out one tree at a time. The tree you print out will be the metropole whose remote coordinates are (remoteX, remoteY).

Travel between different metropoles is only possible through airports. Airports correspond to isolated cities in Part 3. Like cities, airports have both remote and local coordinates, but airports are isolated instead of connected points. When an airport is mapped, a corresponding terminal is also mapped. Terminals have a road connecting to a city that is already mapped. After an airport is mapped, other terminals corresponding to that airport may also be mapped. To access an airport, you must first travel to one of its terminals. From the terminal, you travel directly to the airport, taking the shortest path, i.e. a straight line. Note that there is no road between the airport and the terminal (pretend there is some sort of shuttle service). Your adjacency list becomes more complicated compared to part 3, so implementing it as a separate class is more attractive.

Rather than write shortest path again, you will construct a minimum spanning tree for the graph consisting of all things mapped. You must use Prim's algorithm. Using another algorithm may not produce the same output. To break ties in the priority queue, always choose the city that comes last in asciibetical order. When outputting the tree, the children of a node should appear in reverse asciibetical order.

While finding nearest cities and airports are restricted to a specific metropole, ranging cities will encompass all possible cites in Part 4. Given remote coordinates `x,y`, you must return all cities part of any metropole whose remote coordinates are in range of that point.

6 General XML Output

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.compareTo()` 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. Because cities now have two sets of coordinates (remote coordinates and local coordinates), we introduce the parameters `remoteX` and `remoteY`, which denote the remote coordinates of the metropole we want you to print. There are two possible error conditions. First, if we specify remote coordinates that are not in the bounds of the remote spatial map (see commands for part 4 for details), you should return a metropoleOutOfBounds error. Second, if the PMQuadtree for the associated remote coordinates is empty, you should return a metropoleIsEmpty error.

`Parameters:`

remoteX
remoteY

`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 quadtree.`<gray>` nodes will have the attributes **x** and **y**, these are integers that identify the location (local coordinates) 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>` or `<airport>` tag (not both). Nodes can also contain several `<road>` tags, depending on the type of tree (PM1 or PM3).
The format of `<city>`, `<airport>`, and `<road>` tags will be as such:

```
<city name="city1" localX="coordx" localY="coordy" remoteX="coordX" remoteY="coordY"
color="color1" radius="radius1"/>
<airport name="airport1" airlineName="airline1" localX="coordx" localY="coordy"
remoteX="coordX" remoteY="coordY" />
<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 `<airport>` tag will be printed (if it exists) then the `<road>` tags will be printed in the ordering specified in the **General sorting of `<city>` and `<road>` tags** section. Since the roads are undirected, within the road tag, the *start* city is the one whose name is first in descending asciibetical order.
Black nodes have the attribute **cardinality** which is the total number of cities, roads, and airports contained within the black node. Black nodes will appear as such:

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

`<white>`

Nodes represent an empty node in your PM quadtree and will appear as such;
`<white/>`

`Possible` *`<error>`* `types:`

metropoleOutOfBounds
metropoleIsEmpty

<success> Example:

```
<quadtree order="3">
  <gray x="512" y="512">
    <black cardinality="2">
      <terminal airportName="P1" cityName="A" localX="0" localY="5" name="T2" remoteX="0
      <road end="T2" start="A"/>
    </black>
    <black cardinality="2">
      <city color="black" name="Zurich" radius="5" localX="1000" localY="1000" remoteX="
      <road end="Zurich" start="Chicago"/>
    </black>
    <black cardinality="2">
      <city color="black" name="Chicago" radius="5" localX="133" localY="34" remoteX="1"
      <road end="Zurich" start="Chicago"/>
    </black>
    <black cardinality="2">
      <airport localX="2" localY="6" name="P1" remoteX="0" remoteY="0"/>
      <road end="Zurich" start="Chicago"/>
    </black>
  </gray>
</quadtree>
```

printAvlTree

This is identical to part 3, with the exception that instead of printing (x,y) coordinates as the value for each node, you should print (localX, localY) coordinates. Airports should not be part of the Avl-g tree.

6.1 Commands for Part 4

`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 `localSpatialWidth` and `localSpatialHeight`, both integer powers of two in the range of 32 bit 2's complement integers. These two attributes define the rectangular region the local spatial data structures can store. Recall that a local spatial map represents a PMQuadtree for a particular metropole. All local spatial maps will be bounded by the closed rectangle whose **lower left corner** is $(0, 0)$ and whose width and height are given by `localSpatialWidth` and `localSpatialHeight`.

The second two attributes are `remoteSpatialWidth` and `remoteSpatialHeight`, both integer powers of two in the range of 32 bit 2's complement integers. These two attributes define the rectangular region that the remote spatial data structure can store. This refers to the map of remote coordinates and represents coordinates of metropoles relative to other metropoles. The remote spatial map will be bounded by the rectangle whose **lower left corner** is $(0, 0)$, whose width and height are given by `remoteSpatialWidth` and `remoteSpatialHeight`, whose left and bottom boundaries are **closed**, and whose top and right boundaries are **open**.

Pay attention to the difference between the local and remote spatial map rectangular regions. Local spatial maps are closed on all four sides, as in a PM Quadtree. The remote spatial map is open on the top and right boundary, as in a PR Quadtree (you are encouraged to use one for globalRangeCities).

We will not be dealing with negative coordinates. Although all of our coordinates will be given as integers, you may need to center your spatial structure around a coordinates that is not an integer; make sure you plan accordingly. Observe that these four 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 local or remote coordinates outside their boundaries. New for part 4, these four values will also have an impact on the success of the `<mapAirport>` command.

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. If it is not specified, you may use either PM1 or PM3 quadtree. The integer attribute `g` specifies the g (the maximum imbalance) for the AVL-g.

`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, local coordinates, radius, color, and remote coordinates. A city can be successfully created if its name is unique (i.e., there isn't already another city or airport or terminal with the same name), its coordinates are also unique (i.e., there isn't already another city or airport or terminal with the same local AND remote coordinates), and the coordinates are non-negative integers. Names are *case-sensitive.*

New for Part 4: city names cannot be the same as those of existing airports or terminals. Errors should be returned if trying to create a city with the same name as an existing airport (duplicateCityName).

`Parameters: (In output order)`

name
localX
localY
remoteX
remoteY
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"/>
        <localX value="12"/>
        <localY value="14"/>
        <remoteX value="0"/>
```

```
                <remoteY value="0"/>
            <radius value="15"/>
            <color value="red"/>
        </parameters>
        <output/>
    </success>
```

deleteCity

Removes a city with the specified name from data dictionary and the adjacency list. The criteria for success here is simply that the city exists.

Note also that unlike last time, a city may be deleted even if it exists in some quadtree. In the case that it is mapped as a road, you're required to print a `roadUnmapped` command that looks the same as the one below, for all of the roads that have the specified city as either the `start` or the `end`. Sort the tags in descending asciibetical order by start, tie breaking by end, as usual. Although other cities may incidentally be unmapped (if the last road to them is removed), these will not be printed as that is a side effect. You do not need to handle the case where a city deletion would cause a terminal to be unmapped.

`Parameter:`

name

`Possible` *`<output>`*`:`

A list of `roadUnmapped` tags, with a single `cityUnmapped` tag before it, which follow the specifications below for `unmapRoad` or `unmapCity`

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

cityDoesNotExist

`<success>` `Example:`

```
<success>
    <command id="1" name="deleteCity"/>
    <parameters>
        <name value="Chicago"/>
    </parameters>
    <output>
        <cityUnmapped color="black" name="Chicago" radius="5" localX="133" localY="34" remot
        <roadUnmapped end="Zurich" start="Chicago"/>
    </output>
</success>
```

clearAll

Resets all of the structures including the PM Quadtrees, clearing them. This has the effect of removing every metropole, every city, every airport, every terminal, 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 listed in sorted order either 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 according to the `java.lang.String.compareTo()` method, and the ordering by coordinate is similar to the one defined in the Part 1 spec, but is modified for Part 4. Coordinate ordering is done by comparing *remoteY* values first; for cities with the same *remoteY* value, one city is less than another city if its *remoteX* value is less than the other. If *remoteX* and *remoteY* are identical, then you compare by *localY*. If both cities have the same *localY*, then you compare by *localX*. This command is only successful if there is at least 1 (1 or more) cities in the dictionary.

Note that airports, terminals, and metropoles are not cities and as such neither should not be included.

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" localX="coordx" localY="coordy" remoteX="coordX" remoteY="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" localX="5" localY="5" remoteX="1" remoteY="2" color="blue" ra
            <city name="Annapolis" localX="19" localY="20" remoteX="1" remoteY="1" color="red"
        </cityList>
    </output>
</success>
```

printAvlTree

Prints the structure of your AVL-g tree as specified in General XML output. This takes no parameters, and airports should not be part of the AVL-g tree.

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; the **start** and **end** are the same; the **start** and **end** are not in the same metropole; the road from **start** to **end** already exists. However, there are two error conditions you must handle.

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 (that is, less than a 1 by 1 partition).

For Part 4, a road is in bounds if and only if both cities have remote coordinates that are within the bounds of the remote spatial map, and the line segment connecting both cities' local coordinates intersects the local spatial region.

`Parameter:`

start
end

`Possible` *`<output>`*`:`

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

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

startPointDoesNotExist
endPointDoesNotExist
startEqualsEnd
roadNotInOneMetropole
roadOutOfBounds
roadAlreadyMapped
roadIntersectsAnotherRoad
roadViolatesPMRules

`<success>` `Example:`

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

mapAirport

Creates an airport at the specified local and remote coordinates. Note that unlike the good old isolated cities, an airport doesn't get its data from a createCity. It has no color or radius, and you should not store airports in your city data dictionary. An airport cannot have the same name as any other airport, terminal, or city (mapped or unmapped). Furthermore, an airport cannot have the same coordinates in the same metropole as any other airport, terminal, or city (mapped or unmapped). An airport is out of bounds if either its remote coordinates are out of bounds of the remote map or its local coordinates are out of bounds of its local map. Corresponding errors should be returned when any of these errors happen. The command also maps a terminal with the same remote coordinates, specified local coordinates and name, and it maps the road between the terminal and its connecting city. The connecting city must already be mapped in the same metropole. Of course, mapping the terminal and the road may cause any of the error conditions of `mapRoad`.

`Parameters:`

name
localX
localY
remoteX
remoteY
terminalName
terminalX
terminalY
terminalCity

`Possible` *`<output>`*`:`

(none)

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

~~duplicateAirportName~~
~~duplicateAirportCoordinates~~
~~airportOutOfBounds~~
~~duplicateTerminalName~~
~~duplicateTerminalCoordinates~~
~~terminalOoutOfBounds~~
~~connectingCityDoesNotExist~~
~~connectingCityNotInSameMetropole~~
airportViolatesPMRules
~~connectingCityNotMapped~~
terminalViolatesPMRules
~~roadIntersectsAnotherRoad~~

`<success>` `Example:`

```
<success>
  <command id="5" name="mapAirport"/>
  <parameters>
       <name value="AirportB"/>
       <localX value="2"/>
       <localY value="14"/>
       <remoteX value="1"/>
       <remoteY value="1"/>
       <terminalName value="TerminalBC"/>
       <terminalX value="6"/>
       <terminalY value="10"/>
       <terminalCity value="C"/>
  </parameters>
  <output/>
</success>
```

mapTerminal

Creates a terminal at the specified local and remote coordinates, associated with the specified airport and connected to the specified city. Terminals may not share a name or local coordinates with any other terminal, city, or airport. The terminal must be lie in the remote and the local spatial maps; otherwise it is considered out of bounds. The road between the terminal and its connecting city must also be mapped and may throw errors as in `mapRoad`. The connecting city and airport must already be present in the spatial map for the command to succeed.

`Parameters:`

name
localX
localY
remoteX
remoteY
cityName
airportName

`Possible` *`<output>`*`:`

(none)

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

duplicateTerminalName
duplicateTerminalCoordinates
terminalOutOfBounds
airportDoesNotExist
duplicateTerminalCoordinates
airportNotInSameMetropole
connectingCityDoesNotExist
connectingCityNotInSameMetropole
connectingCityNotMapped
terminalViolatesPMRules
roadIntersectsAnotherRoad

`<success>` `Example:`

```
<success>
    <command id="5" name="mapTerminal"/>
    <parameters>
        <name value="terminalB"/>
        <localX value="2"/>
        <localY value="14"/>
        <remoteX value="1"/>
        <remoteY value="1"/>
        <cityName value="CityB"/>
        <airportName value="AirportB"/>
    </parameters>
    <output/>
</success>
```

unmapRoad

Will remove a road and its associated endpoints from the map, unless the endpoint (city) is part of another mapped road. Note that you cannot unmap a road between a terminal and a city. This should throw one of the start or end does not exits errors.

`Parameter:`

start
end

`Possible` *`<output>`*`:`

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

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

startPointDoesNotExist
endPointDoesNotExist
startEqualsEnd
roadNotMapped

`<success>` `Example:`

```
<success>
    <command name="unmapRoad" id="5"/>
    <parameters>
        <start value="Baltimore"/>
        <end value="Annapolis"/>
    </parameters>
    <output>
        <roadDeleted start="Baltimore" end="Annapolis"/>
    </output>
</success>
```

unmapAirport

Will remove an airport from the map. All terminals associated with the airport will also be unmapped. Of course, the roads connecting the terminals to their cities will also be unmapped.

`Parameter:`

name

`Possible` *`<output>`*`:` Each terminal that corresponds to the airport will also be unmapped, producing a *terminalUnmapped* tag in the `output` and will appear as such:
`<terminalUnmapped name="terminalA"/>`

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

airportDoesNotExist

`<success>` `Example:`

```
        <success>
            <command name="unmapAirport" id="5"/>
            <parameters>
                <name value="JFK"/>
            </parameters>
            <output>
                <terminalUnmapped name="T1"/>
            <output/>
        </success>
```

unmapTerminal

Will remove a terminal from the map. If the associated airport has no remaining mapped terminals, then the airport is also removed. Of course, the roads connecting the terminals to their cities will also be unmapped.

`Parameter:`

name

`Possible <output>:` The airport associated with the terminal may be unmapped, producing an *airprotUnmapped* tag in the `output` and will appear as such:
`<airportUnmapped name="AiroprtA"/>`

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

terminalDoesNotExist

`<success> Example:`

```
<success>
    <command name="unmapTerminal" id="5"/>
    <parameters>
        <name value="T1"/>
    </parameters>
    <output>
        <airportUnmapped name="A1"/>
    <output/>
</success>
```

printPMQuadtree

Prints the PMQuadtree corresponding to the command parameter pmOrder, if given, as specified in the General XML output page. 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 described in the previous section 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. For part 4, a particular metropole is specified. There are two errors conditions: the remote coordinates are out of bounds of the remote spatial map, or the metropole at the specified remote coordinates is empty. 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 and airports 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):`

remoteX
remoteY
name - filename to save the image to

`Possible <output>:`

`Possible` *`<error>`* `types:`

> metropoleOutOfBounds
> metropoleIsEmpty

`<success>` `Example:`

```
    <success>
        <command name="saveMap" id="6"/>
        <parameters>
    <remoteX value="1"/>
            <remoteY value="2"/>
            <name value="map_1"/>
        </parameters>
        <output/>
    </success>
```

globalRangeCities

Lists *all* the cities present *in any spatial map* within a `radius` of a point `x,y` *in the remote map*. Any city whose remote coordinates are in range of the specified point is included. Cities on the boundary of the range are included, and `x,y` are integer coordinates. Only cities that are in the spatial structure, in this case, the PM quadtrees, are relevant to this commmand. Airports and terminals are not considered cities. Keep in mind also that the cities for this command can come from any metropole. Design your structure so that this is efficient; if you use a HashMap and you have to check every possible metropole within the radius , you will be unhappy. `<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. If the radius is 0, but the query point coincides with the remote coordinates of any cities, a list of those cities should be returned. It should be noted that the radius attribute for a city does not factor into this calculation; all cities are considered points.

`Parameters (Listed in output order):`

> remoteX
> remoteY
> radius

`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" localX="coordx" localY="coordy" remoteX="coordX" remoteY="coordY" color="color1" radius="radius1"/>`
> 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="globalRangeCities" id="7"/>
        <parameters>
```

```
            <remoteX value="3"/>
            <remoteY value="3"/>
            <radius value="100"/>
        </parameters>
        <output>
            <cityList>
                <city name="Derwood" localX="20" localY="40" remoteX="4" remoteY="4" color="blue" r
                <city name="Annapolis" localX="20" localY="30" remoteX="3" remoteY="3" color="red" 
            </cityList>
        </output>
    </success>
```

nearestCity

Will return the name and location of the closest city to the specified point in the spatial map at the specified metropole only, where the set of valid cities excludes airports and terminals (which are not cities). If the metropole is empty, has no cities, or the remote coordinates are out of bounds of the remote map, then you should return a cityNotFound error. Terminals are not considered airports. To do this correctly, you probably want to use a `PriorityQueue`-based algorithm–otherwise, you might not be fast enough. 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 descending asciibetical according to the `java.lang.String.compareTo()` method.

`Parameters (In output order):`

localX
localY
remoteX
remoteY

`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" localX="coordx" localY="coordy" remoteX="coordX" remoteY="coordY" color="color1" radius="radius1"/>`

`Possible` *`<error>`* `types:`

cityNotFound

`<success>` `Example:`

```
<success>
    <command name="nearestCity" id="8"/>
    <parameters>
        <localX value="1"/>
        <localY value="2"/>
        <remoteX value="3"/>
        <remoteY value="3"/>
    </parameters>
    <output>
        <city name="Annapolis" localX="20" localY="30" remoteX="3" remoteY="3" color="red" radi
    </output>
</success>
```

mst

You will have to construct a minimum spanning tree for all mapped objects using Prim's algorithm, starting at a specified city (not airport or terminal). City-city and city-terminal travel takes place on roads, terminal-airport travel is in a straight line from the terminal to the airport, and airport-airport travel is in a straight line from one metropole to the other (using remote coordinates). Every airport is connected to every other airport. When considering two edges with the same cost, break ties by choosing the one who appears first in reverse asciibetical order.

The mst command takes a starting city, and outputs the tree produced. The root will be the starting vertex. Each element's children should appear in descending asciibetical order by name.

Note that you must use Prim's algorithm for this, otherwise your tree will be the same, but the output will be different since the starting vertex may be different.

`Parameters: (In output order)`

start

`Possible` *`<output>`*`:`

A `<mst>` tag with attribute `distanceSpanned` will be contained in `output`. The value of `distanceSpanned` is the sum of all of the edges in the minimum spanning tree. The first and only child of the `mst` tag is a `<node>` tag with attribute `name` containing the name of the starting city. The rest of the tree is outputted in a similar fashion to the AVL tree - every `<node>` tag contains its children in reverse asciibettical order. Just like the root node, each of the children is represented by a `<node>` tag with attribute `name` containing the name of the city, terminal, or airport of the node in the minimum spanning tree.

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

cityDoesNotExist

`<success>` `Example:`

```
<success>
    <command name="mst" id="9"/>
    <parameters>
        <start value="B"/>
    </parameters>
    <output>
        <mst distanceSpanned="35.032">
            <node name="B">
                <node name="A">
                    <node name="TerminalAB">
                        <node name="AirportA">
                        </node>
                    </node>
                </node>
            </node>
        </mst>
    </output>
</success>
```

7 General Policies and Criteria for Submission

7.0.1 README file Contents

Although README requirements also remain unchanged, their importance is greater for this part than the last, due to the new avenues for sources (the canonical, online AVL resources, other students' submissions). Thus for your convenience we leave them here. 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.

7.1 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. We then run your output through a complex set of filters and verify that the result is identical to our expected output.

New for part 3 is the fact that 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.

7.1.1 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.

7.1.2 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.

7.2 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.

7.3 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.
- 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.

7.3.1 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.