Programming Assignment 1: MeeshQuest - A Tale of Two Trees

Handed out: Tue, Oct 29. Due: **Wed, Nov 13, 11:59pm**. (See submission instructions below for late policy.)

This assignment is an extension to Part-0 and is designed to extend your familiarity with XML documents and implementing dictionary data structures sorted by multiple criteria.

The general input-output structure will be the same as in Part-0. You will input commands in XML format and will generate an XML document as output using the XML utility functions in the jar file, `cmsc420util.jar`, provided to you in the skeleton code. As in Part-0, we recommend that you refer to the document "MeeshQuest Getting Started" and the document "Processing XML Files for the Programming Project." Note that there is a new XML "`.xsd`" file. We will provide skeleton program containing this new file. These can all be found on the 420 class handouts page:

http://www.cs.umd.edu/class/fall2019/cmsc420-0201/handouts.html

You will implement two dictionary structures, a standard (unbalanced) binary search tree and a novel data structure called an *SG Tree*, which will be discussed in class. The binary-search tree stores city objects sorted in ascending order by name.[1] The SG tree will store cities in increasing lexicographical order by (x, y) coordinates (as in Part-0).

Here is a summary of the commands that you will process for this part:

Create City: This is the same as in Part-0 and will insert the city to both dictionaries (by name for the binary search tree and by coordinates for the ST tree).

We will also add error checking. (See below for how errors are handled.) If there exists a city with the same coordinates, an error "`duplicateCityCoordinates`" is generated. If there exists a city with the same name, an error "`duplicateCityName`" is generated. (If both errors apply, the duplicate-coordinate error takes precedent over the duplicate-name error.)

List Cities: This is also the same as in Part-0. If the "`sortBy`" attribute has the value "`name`", the list is to be generated by an inorder traversal of the binary search tree, and if the "`sortBy`" attribute has the value "`coordinate`", the list is to be generated by an inorder traversal of the SG tree. There is an additional requirement that there must be at least one city in the dictionary. If not, an error "`noCitiesToList`" is generated.

Delete City: Removes a city with the specified name from both dictionaries. The command has a single parameter, the name of the city to delete. Example:

```
<deleteCity name="Annapolis"/>
```

The command succeeds if the city is in the dictionary, and otherwise it generates the error "`cityDoesNotExist`". If successful, your program will generate an XML element in the output document of the following form:

[1] Note that for the sake of testing correctness, your tree must be structurally identical to ours. For this reason, whenever deleting a node with two children always select the replacement node to be the *inorder successor* of the deleted item.

```
<success>
  <command name="deleteCity"/>
  <parameters>
    <name value="London"/>
  </parameters>
  <output>
    <cityDeleted color="yellow" name="London" radius="0" x="150" y="250"/>
  </output>
</success>
```

Note that the attributes included in the "`cityDeleted`" element are the same as the attributes listed for each city "`listCities`" command from Part-0. As in that case, the order in which the attributes are listed does not matter.

Clear-All: Resets both of the dictionaries (having the effect of deleting all the cities). It has no parameters.

```
<clearAll/>
```

The command always succeeds, and it generates the following XML element for the output document:

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

Print the Binary Search Tree: This prints the current contents of the binary search tree in a hierarchical (preorder) manner, so that the XML structure matches the tree's structure (see the example below). There are no parameters.

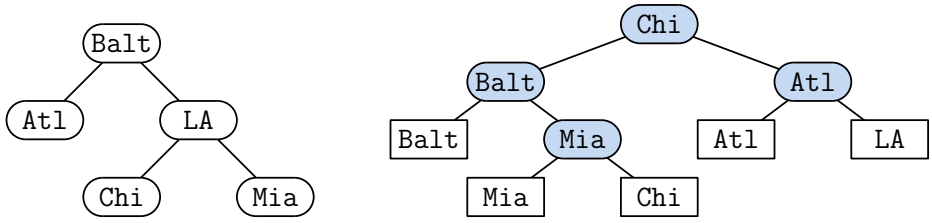
```
<printBinarySearchTree/>
```

This command succeeds if the tree has at least one item, and otherwise an error "`mapIsEmpty`" is generated. Each node of the tree is represented by an element, call it E, with tag "`node`". E's attributes consist of the city's name, and x- and y-coordinates. (Other attributes, such as the color and radius are omitted.) This node's left subtree and right subtree are each recursively transformed into XML elements, and these elements are made children of element E. Here is an example for the tree structure shown in Fig. 1(a).

```
<success>
  <command name="printBinarySearchTree"/>
  <parameters/>
  <output>
    <binarysearchtree>
      <node name="Baltimore" x="76" y="39">
        <node name="Atlanta" x="84" y="33"/>
        <node name="Los_Angeles" x="118" y="33">
          <node name="Chicago" x="81" y="47"/>
          <node name="Miami" x="80" y="25"/>
```

```
        </node>
      </node>
    </binarysearchtree>
  </output>
</success>
```

Note that your binary-tree implementation must produce the *same tree structure* as ours in order to be graded as correct.



(a) (b)

Figure 1: (a) A binary search tree and (b) an SG tree.

Print the SG Tree: This prints the current contents of the SG tree in a hierarchical (preorder) manner, so that the XML structure matches the tree's structure (see the example below). There are no parameters.

```
<printSGTree/>
```

This command succeeds if the tree has at least one item, and otherwise an error "`mapIsEmpty`" is generated. The hierarchical structure is the same as for the binary search tree, but the ST tree is an *extended tree*, and so it has two types of nodes, *internal* and *external*. All data is stored at the external nodes, and the internal nodes are just used for the sake of directing the search to the appropriate leaf node. If an internal node contains a city C, then the left subtree contains cities whose (x, y)-coordinates are lexicographically less than or equal to C's, and the right subtree contains cities whose coordinates are lexicographically greater. Internal nodes are indicated with the tag "`internal`" and the external nodes with the tag "`external`". Here is an example for the tree structure shown in Fig. 1(b).

```
<success>
  <command name="printSGTree"/>
  <parameters/>
  <output>
    <SGTree>
      <internal name="Chicago" x="81" y="47">
        <internal name="Baltimore" x="76" y="39">
          <external name="Baltimore" x="76" y="39"/>
          <internal name="Miami" x="80" y="25">
            <external name="Miami" x="80" y="25"/>
            <external name="Chicago" x="81" y="47"/>
          </internal>
```

```
        </internal>
        <internal name="Atlanta" x="84" y="33">
          <external name="Atlanta" x="84" y="33"/>
          <external name="Los_Angeles" x="118" y="33"/>
        </internal>
      </internal>
    </SGTree>
  </output>
</success>
```

Note that your SG tree implementation must produce the *same tree structure* as ours in order to be graded as correct.

Error Processing: Unlike Part-0, you will need to handle errors in this part. There are two types of errors that your program should deal with. First, if there is an error in the input file (which will result in one of the exceptions "`SAXException | IOException | ParserConfigurationException`" being triggered in response to the invocation of `XmlUtility.validateNoNamespace`), your program should generate a single element with the tag "`<fatalError>`". (We placed a comment in the `MeeshQuestSkeleton.java` for where to put this processing.) An example of the program output is shown below.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<fatalError/>
```

Second, if any of the above command-specific errors occurs (e.g., "`duplicateCityName`", "`noCitiesToList`", or "`cityDoesNotExist`") then rather than generating a success-element, your program will generate an element with the tag "`error`" whose type attribute is the name of the error. This error element also summarizes the command and its attributes. Here is an example, which might occur in response to the second attempt to create a city named "`A`":

```
<error type="duplicateCityName">
  <command name="createCity"/>
  <parameters>
    <name value="A"/>
    <x value="0"/>
    <y value="1"/>
    <radius value="2"/>
    <color value="red"/>
  </parameters>
</error>
```

Here is another example in the case of an attempt to delete the nonexistant city name "`London`":

```
<error type="cityDoesNotExist">
  <command name="deleteCity"/>
  <parameters>
    <name value="London"/>
  </parameters>
</error>
```

Sample Input/Output: Here is a sample input. It can be input and parsed by the function `XmlUtility.validateNoNamespace()`, which we have provided you in `cmsc420util.jar`:

```
<commands
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="part1in.xsd"
  spatialWidth="512"
  spatialHeight="512">
 <createCity name="Baltimore" y="39" x="76" radius="10" color="green"/>
 <createCity name="Chicago" x="87" y="41" radius="15" color="blue"/>
 <printBinarySearchTree/>
 <printSGTree/>
</commands>
```

And the resulting sample output as generated by the utility `XmlUtility.print()`, which we have provided to you in `cmsc420util.jar`:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<results>
  <success>
    <command name="createCity"/>
    <parameters>
      <name value="Baltimore"/>
      <x value="76"/>
      <y value="39"/>
      <radius value="10"/>
      <color value="green"/>
    </parameters>
    <output/>
  </success>
  <success>
    <command name="createCity"/>
    <parameters>
      <name value="Chicago"/>
      <x value="87"/>
      <y value="41"/>
      <radius value="15"/>
      <color value="blue"/>
    </parameters>
    <output/>
  </success>
  <success>
    <command name="printBinarySearchTree"/>
    <parameters/>
    <output>
      <binarysearchtree>
        <node name="Baltimore" x="76" y="39">
          <node name="Chicago" x="87" y="41"/>
        </node>
      </binarysearchtree>
    </output>
  </success>
  <success>
    <command name="printSGTree"/>
```

```
    <parameters/>
    <output>
      <SGTree>
        <internal name="Baltimore" x="76" y="39">
          <external name="Baltimore" x="76" y="39"/>
          <external name="Chicago" x="87" y="41"/>
        </internal>
      </SGTree>
    </output>
  </success>
</results>
```

Submission Instructions and Late Policy: Submit your program through the submit server

https://submit.cs.umd.edu/fall2019/

Here is the late policy:

Up to 6 hours late:	5% of total
Up to 24 hours late:	10% of the total
For each additional 24 hours late:	20% of the total