CMSC 420:Fall 2019 Dave Mount

Programming Assignment 2: MeeshQuest - Adding Geometry

Handed out: Fri, Nov 29. Due: **Mon, Dec 9, 11:59pm**. (See submission instructions below for late policy.)

This assignment extends Part-1 by replacing the SG Tree with a dynamic variant of a kd-tree, called an *SG kd-tree*. The general input-output structure will be the same as in Parts 0 and 1. 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 before, we recommend that you refer to the document "MeeshQuest Getting Started" and the document "Processing XML Files for the Programming Project." To help people who may not have completed Part 1, we will provide source code of our implementation of Part 1 for reference.

As before, we will provide you with an updated XML schema definition file ("`pert2in.xsd`"), and a skeleton program to help you get started. These can all be found on the 420 class handouts page:

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

For this part of the project, you will implement two dictionary structures, a standard (unbalanced) binary search tree and the SG kd-tree. It will be described in a separate handout, but here are its principal characteristics:

- Like the SG tree:
  - It is an *extended tree*, with contents stored only in the external nodes and splitters stored in internal nodes.
  - It is rebalanced by *rebuilding subtrees*. The same rules as the SG tree apply for when to trigger rebuilding, determining scapegoat nodes, and deciding how to partition objects among the left and right subtrees.
  - If a point is equal to the splitter, it is stored in the left subtree. (Note that this differs from the kd-tree convention from class.)
- Each splitter is based on the geometric structure of the points within the subtree:
  - Each internal node stores a *splitting dimension* (either 0 for x or 1 for y), defined to be the dimension corresponding to the longer side of the smallest axis-parallel rectangle that contains the points of the subtree, with ties broken in favor of x over y.
  - Points are partitioned among the left and right subtrees based on a lexicographical ordering. If the splitting dimension is x, the points are sorted lexicographically by (x, y), and if the splitting dimension is y, the points are sorted lexicographically by (y, x).

  Note that once the splitting dimension and splitter are set, they are not altered as points are inserted and deleted from the subtree.

The binary-search tree stores city objects sorted in ascending order by name, and the SG kd-tree represents the geometric structure of the cities. The bounding box of the kd-tree is determined by the `spatialWidth` and `spatialHeight` parameters appearing in the header element of the XML input file. For example, the following header line sets the bounding box of the kd-tree to $0 \le x \le 1024$ and $0 \le y \le 512$.

```
<commands xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="part2in.xsd"
    spatialWidth="1024" spatialHeight="512">
```

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

Common Elements: The commands “`createCity`”, “`listCities`”, “`deleteCity`”, “`clearAll`”, “`printBinarySearchTree`” are the same as in the previous two parts. The only exceptions are:

`createCity`: In addition to the duplicate name/coordinate error conditions (which are the same as last time) the (x, y) coordinates of the city must lie within the bounding box as described above. If this is not the case, the program should generate a “`cityOutOfBounds`” error. The error element has the same structure as that for duplicates. This error condition has the highest precedence (meaning that only this error is reported, even if the city is a duplicate).

`listCities`: The “`sortby`” option can only take the value “`name`” (not “`coordinate`”).

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

```
<printKdTree/>
```

This command succeeds if the tree has at least one point, and otherwise an error “`mapIsEmpty`” is generated. The structure of the output is similar to that of the SG tree in Part 1 in the sense that there are two types of nodes, internal and external. Internal nodes differ from the SG tree in that each stores a *splitting dimension*, which is either 0 (for x or vertical) or 1 (for y or horizontal). Rather than storing a City, each internal node only stores the coordinates of the splitting point. Here is an example for the tree structure shown on the right side of Fig. 1.

```
<success>
  <command name="printKdTree"/>
  <parameters/>
  <output>
    <KdTree>
      <internal splitDim="0" x="500" y="500">
        <internal splitDim="1" x="500" y="500">
          <internal splitDim="0" x="300" y="400">
            <external name="London" x="300" y="400"/>
            <external name="Edinburgh" x="500" y="500"/>
          </internal>
```

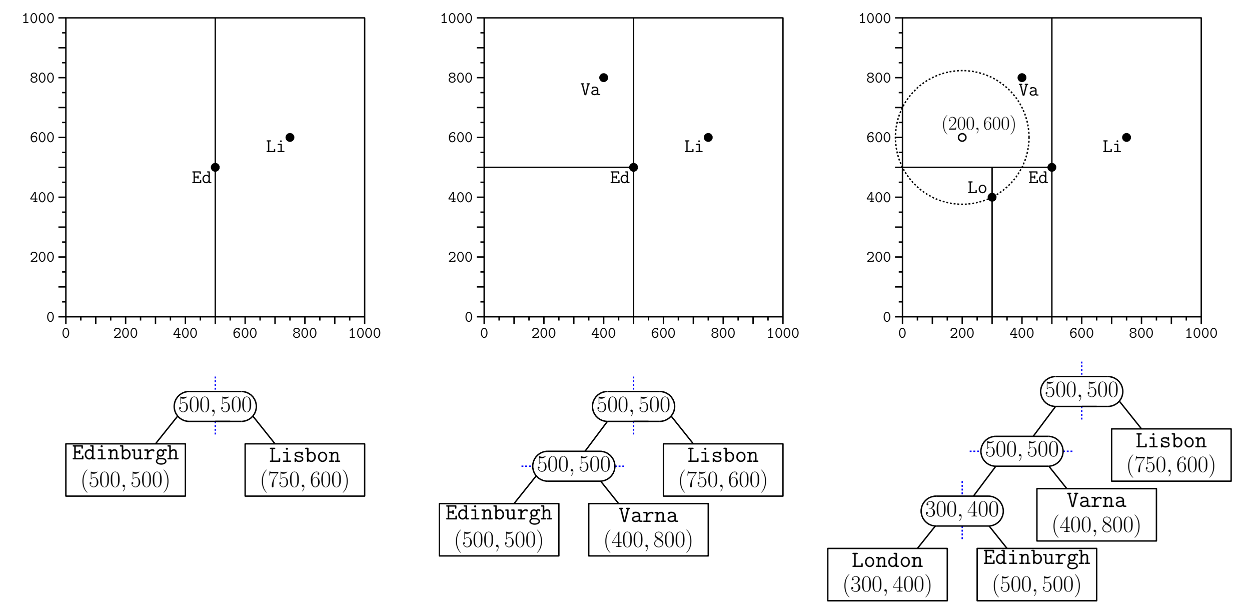

Figure 1: The SG kd-tree resulting from inserting four cities, and the associated tree structures. Note that a point that coincides with a splitter is placed in *left subtree*. The last figure shows the result of a nearest-neighbor query.

```
          <external name="Varna" x="400" y="800"/>
        </internal>
        <external name="Lisbon" x="750" y="600"/>
      </internal>
    </KdTree>
  </output>
</success>
```

As in Part 1, your SG tree implementation must produce the *same tree structure* as ours in order to be graded as correct.

`nearestNeighbor`: This command takes the (x, y) coordinates of a point:

```
<nearestNeighbor x="200" y="600"/>
```

This command succeeds if the tree has at least one point (otherwise an error "`mapIsEmpty`" is generated), and the query point lies within the bounding box (otherwise an error "`queryOutOfBounds`" error is generated). The structure of these errors are similar to previous error elements. Assuming there is no error, it outputs the closest city in the current kd-tree. If there are multiple cities that are simultaneously the closest, you can output any of them. The output below corresponds to the tree of Fig. 1.

```
<success>
  <command name="nearestNeighbor"/>
  <parameters>
    <x value="200"/>
    <y value="600"/>
  </parameters>
  <output>
    <nearestNeighbor color="yellow" name="London" radius="0" x="300" y="400"/>
  </output>
</success>
```

Error Processing: As in Part 1, you will need to handle errors in this part. The only new errors are the "`cityOutOfBounds`" and "`queryOutOfBounds`" mentioned above. Examples are shown below:

```
<error type="cityOutOfBounds">
  <command name="createCity"/>
  <parameters>
    <name value="Chicago"/>
    <x value="2000"/>
    <y value="400"/>
    <radius value="0"/>
    <color value="yellow"/>
  </parameters>
</error>
```

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

```
<error type="queryOutOfBounds">
  <command name="nearestNeighbor"/>
  <parameters>
    <x value="200"/>
    <y value="1025"/>
  </parameters>
</error>
```

In both cases, the order in which the parameters are listed in significant.

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="part2in.xsd"
  spatialWidth="512"
  spatialHeight="512">
 <createCity name="Beijing" y="38" x="76" radius="0" color="green"/>
 <createCity name="Shanghai" x="86" y="40" radius="5" color="blue"/>
 <printKdTree/>
 <nearestNeighbor x="21" y="33"/>
</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"/>
    (...same as in Part 1...)
  </success>
  <success>
    <command name="createCity"/>
    (...same as in Part 1...)
  </success>
  <success>
    <command name="printKdTree"/>
    <parameters/>
    <output>
      <KdTree>
        <internal splitDim="0" x="76" y="38">
          <external name="Beijing" x="76" y="38"/>
          <external name="Shanghai" x="86" y="40"/>
        </internal>
      </KdTree>
    </output>
  </success>
  <success>
    <command name="nearestNeighbor"/>
    <parameters>
      <x value="21"/>
      <y value="33"/>
    </parameters>
    <output>
      <nearestNeighbor color="green" name="Beijing" radius="0" x="76" y="38"/>
    </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