Project2: PRQuadTree

You are allowed to work with a partner in this project!

Assignment

In project 1, you built a simple database system for storing, removing, and querying a collection of rectangles by name or by position. The data structure used to organize the collection of rectangles was a Skip List. The Skip List is efficient for finding rectangles by name. However, it is not so good for finding rectangles by location. This was salient when implementing the region search and intersections functionality. The problem is that there are two distinct keys by which we would like to search for rectangles (name or location). So we should expect that we will need two data structures, one to organize by each key. However, even if we added a second Skip List to the system, we would still have the problem that the Skip List simply is not a good data structure for the tasks of finding all rectangles that intersect a query rectangle, or finding all intersections from among a collection of rectangles. What we really need is another data structure that can perform these spatial tasks well.

Your database will now be organized by two data structures. One will be the Skip List, which will organize the collection of objects by name (as in the earlier project). The second data structure will be a variant of the PR Quadtree, as described below. In order to simplify the mechanics of the insert and delete operations on the PR Quadtree, the data objects being stored will be points instead of rectangles. The PR Quadtree will organize the points by position, and will be used for spatial queries such as locating points within a query rectangle, and determining the duplicate points.

Before reading the following description of the PR Quadtree variant used for this project, you should first read **Modules 18.2** and **18.3** in OpenDSA. You are implementing something similar to the PR Quadtree as described in **Module 18.3**, but there are some important differences in the split and merge rules.

You will use a variant of the PR Quadtree to support spatial queries. The PR Quadtree is a full tree such that every node is either a leaf node, or else it is an internal node with four children.

As with the PR Quadtree described in OpenDSA, the four children split the parent's corresponding square into four equal-sized quadrants. The internal nodes of the PR Quadtree do not store data. Pointers to the points themselves are stored only in the leaf nodes.

Invocation and I/O Files

The program will be invoked from the command-line as:

```
%> java Point2 {commandFile}
```

where:

- `Point2` is then name of the program. The file where you have your `main()` method must be called `Point2.java`
- `commandFile` is the name of the command file to read.

Your program will read from the text file `{command-file}` a series of commands, with one command per line. The program should terminate after reading the end of the file. You are guaranteed that the commands in the file will be syntactically correct in all graded test cases. The commands are free-format in that any number of spaces may come before, between, or after the command name and its parameters. All commands should generate the required output message. **All output should be written to standard output. Every command that is processed should generate some sort of output message to indicate whether the command was successful or not.**

The command file may contain any mix of the following commands. In the following description, terms in `{ }` are parameters to the command.

- **`insert {name} {x} {y}`**

  Insert a point named `name` with at the location `(x, y)`. It is permissible for two or more points to have the same name, and it is permissible for two or more points to have the same spatial position, **but not both**. The name must begin with a letter, and may contain letters, digits, and underscore characters. Names are *case-Sensitive*. A point should be rejected for insertion if either of its coordinates are not greater than 0. All points must exist within the "world box" that is 1024 by 1024 units in size and has upper left corner at (0, 0). If a point coordinates is outside of this box, it should be rejected for insertion.

- **`remove {name}`**

Remove the point with name `name`. If two or more points have the same name, then any one such point may be removed. If no point exists with this name, it should be so reported.

- **`remove {x} {y}`**

  Remove the point at the point `{x} {y}`. If two or more points have the same location, then any one such point may be removed. If no point exists with this name, it should be so reported.

- **`regionsearch {x} {y} {w} {h}`**

  Report all points currently in the database that are contained in the query rectangle specified by the 'regionsearch' parameters. For each such point, list out its name and coordinates. A `regionsearch` command should be rejected if the height or width are not greater than 0. However, it is (syntactically) acceptable for the regionsearch rectangle to be all or partly outside of the 1024 by 1024 world box. Additionally, you must report the number of QuadTree nodes visited by the search.

- **`duplicates`**

  Report all points that have duplicate coordinates.

- **`search {name}`**

  Return the information about the points(s), if any, that have name `{name}`.

- **`dump`**

  Return a “dump” of the Skip List and the QuadTree.

  - The Skip List dump should print out each Skip List node, from left to right. For each Skip List node, print that node’s value and the number of pointers that it contains.
  - The QuadTree dump should print the nodes of the QuadTree in *preorder traversal order*, one node per line, with each line indented by 2 spaces for each level in the tree (the root will indent 0 spaces since it is considered to be at level 0). These lines should appear after the Skip List nodes are printed.

To test your program, here are two command files along with their corresponding correct output:

P2test1.txt ↓ **(https://canvas.vt.edu/courses/135977/files/19882684/download?download_frd=1)**
P2test1Output.txt ↓ **(https://canvas.vt.edu/courses/135977/files/19882685/download?**

download_frd=1)

P2test2.txt (https://canvas.vt.edu/courses/135977/files/19961954/download?download_frd=1)
P2test2Output.txt (https://canvas.vt.edu/courses/135977/files/19882687/download?download_frd=1)

Design Considerations

The point names will be maintained in a Skip List, sorted by the name. Use String's compareTo() method to determine the relative ordering of two names, and to determine if two names are identical. You are using the Skip List to maintain your list of points, but the Skip List is a general container class. Therefore, it should not be implemented to know anything about points.

The key property of any QuadTree data structure is its **decomposition rule**. The decomposition rule is what distinguishes the PR QuadTree of this project from the PR QuadTree described in OpenDSA. The decomposition rule for this project is:

```
A leaf node will split into four when

    1. There are more than three points in the node, such that ...

    2. not all of the points have the same x and y coordinates.
```

So there is no limit in principle to the number of points stored in a single leaf node, if they all happen to have the same position. Four sibling leaf nodes will merge together to form a single leaf node whenever deleting a point results in a set of points among the four leaves that does not violate the decomposition rule. It is possible for a single insertion to cause a cascading series of node splits. It is also possible for a single deletion to cause a cascading series of node merges.

We will define the origin of the system to be the upper-left corner, with the axes increasing in positive value down and to the right. We will designate the children of the PR Quadtree (and the quadrants of the world) to be NW, NE, SW, and SE, in that order. We will assume that the “world” is a square with upper-left corner at (0, 0), and height and width of 1024 units.

All access functions on the PR Quadtree, including insert, remove, regionsearch, and duplicates must be written recursively. The regionsearch and duplicates commands will access the PR QuadTree rather than the Skip List. These commands should visit as few QuadTree nodes as possible.

You must use class inheritance to design your PR Quadtree nodes. You must have a PR QuadTree node base class (or interface), with subclasses for the internal nodes and the leaf nodes. Note the discussion under “Design Considerations” regarding using a flyweight to implement empty leaf nodes. Your QuadTree node implementation may not include a pointer to the parent node.

The most obvious design issue is how to organize the inter-relations between the Skip List and the PR Quadtree. Neither uniquely “owns” the points that it organizes. You will probably want each data structure to store pointers to (shared) point objects.

It is a good idea to create a “Database” class and create one object of the class. This Database class will receive the commands to insert, delete, search, etc., and farm them out to the Skip List and/or PR Quadtree for actual implementation.

Many leaf nodes of the PR Quadtree will contain no data. Storing many distinct “empty” leaf nodes is quite wasteful. One design option is to store a NULL pointer to an empty leaf node in its parent. However, this requires the parent node to understand this convention, and explicitly check the value of its child pointers before proceeding, with special action taken if the pointer is NULL. A better design is to use a “flyweight” object. A flyweight is a single empty leaf node that is created one time at the beginning of the program, and pointed to whenever an empty child node is needed.

When navigating through the PR Quadtree, for example to do an insert operation, you will need to know the coordinates and size of the current PR Quadtree node. One design option is to store with each PR Quadtree node its coordinates and size. However, this is unnecessary and wastes space. Worse, it is incompatible with the Flyweight design pattern, since the flyweight object has to be stateless. Given the coordinates and size of a node, it is a simple matter to determine the coordinates and size of its children. Thus, a better design is to pass in the location and size of the current node as parameters down to the recursive function.

Pledge

Your project submission must include a statement, pledging your conformance to the Honor Code requirements for this course. Specifically, you must include the following pledge statement near the beginning of the file containing the function main() in your program. The text of the pledge will also be posted online. Programs that do not contain this pledge will not be graded.

```
// On my honor:
```

```
// - I have not used source code obtained from another student,
// or any other unauthorized source, either modified or unmodified.
// - All source code and documentation used in my program is
// either my original work, or was derived by me from the
// source code published in the textbook for this course.
// - I have not discussed coding details about this project with
// anyone other than the my partner, instructor, ACM/UPE tutors
// or the TAs assigned to this course.
// I understand that I may discuss the concepts
// of this program with other students, and that another student
// may help me debug my program so long as neither of us writes
// anything during the discussion or modifies any computer file
// during the discussion. I have violated neither the spirit nor
// letter of this restriction.
```

Upload Your File(s)

For: CS 5040 (TR (12:30 - 1:45)) Project2: PRQuadTree

Upload Submission (/Web-CAT/WebObjects/Web-CAT.woa/wo/2.0.0.0.29.1.9.0.1.1.1.0.0.1.9.1.1.1)