CS-3114 Spring 2014 – Project 2 – QuadTree Database

Due: Friday March 7, 5pm.
Late Policy as in the syllabus: 10% per day, maximum of 3 days.

Assignment:

For this project you will expand your City Database to build a simple Geographic Information System for storing 2D point data. The focus is organizing city records into a database for fast geo-location and region search. For each city you will store the name, population, and its 2D location (X and Y coordinates). Searches can be by specific x,y location or by a bounded region. You will implement a variant of the PR Quadtree to support these searches. The PR Quadtree (see notes in the Resources folder on **Scholar**, and OpenDSA 7.12) is a hierarchical data structure commonly used to store 2D data such as city coordinates. It allows for efficient insertion, removal, and region search queries.

Implementation and Design Requirements:

Modify your Project 1 to include x and y coordinates in each city record. Records in the city database now contain four fields: the string name field, integer population field, and integer x and y coordinates for the location of the city (similar to latitude & longitude, which now replace the payload field from Project 1). Spatially, the database's geo world should be viewed as a square whose origin is in the upper left (NorthWest) corner at position (0, 0). The world is 2^{10} by 2^{10} units in size, and so x and y are integers in the range 0 to $2^{10}-1$ (that is 1023). The x-coordinate increases to the right, the y-coordinate increases going down. It is an error to insert two cities with the same (x,y) location, or to insert a city that is outside these world bounds.

Add a PR Quadtree data structure as an additional index for the collection of city records. Use the Quadtree to index the cities by their (x,y) coordinates. Keep your two BSTs to index the name and population fields. (Note you will not get penalized again here if you did not get your BSTs working in Project 1.) To minimize space, Quadtree nodes can contain only a reference to the city record and must not replicate the city's (x,y) coordinate data in the node.

PR Quadtrees recursively subdivide the world space into 4 equal sized quadrants. Thus, for example the first split should be centered at $(2^9, 2^9)$. Since, the world bounds are a power of 2, you should never get non-integer subdivisions. You should never subdivide beyond a quadrant size of 1 by 1 units, because only 1 city can be inserted within a unit quadrant anyway. Our splitting rule convention is that any cities that lay on a split boundary are considered to be in the larger direction (e.g. to the right and/or down). Thus, a city on the initial world boundary of 2^{10} is considered out of bounds.

You must use inheritance to implement the Quadtree nodes. You should have a node abstract base class with separate subclasses for the internal nodes, leaf nodes, and empty nodes. The Quadtree internal node should store references to its 4 children. A Quadtree leaf node should store a reference to a city record object. For this assignment the bucket size of the PR Quadtree leaf nodes is 1, meaning that each leaf node will store exactly one city. Quadtree empty nodes must be implemented using the **Flyweight design pattern** (OpenDSA 6.1.1 & 7.12.3). To save space, **Quadtree nodes are not allowed to store coordinates representing their boundaries**. Instead, tracking of the boundaries must be accomplished by passing down the necessary information through the recursion.

All operations that traverse or descend the Quadtree structure **must** be implemented recursively. Pay special attention to the possibility that an insert operation might cause a cascading series of splits to occur, and that a branch might contract more than a single level when a leaf node is deleted. The **find** and **rfind** routines should prune the search space appropriately to enable efficient searches.

The Quadtree should be implemented to support generic use with records of various types. The Quadtree needs a mechanism to get 2D coordinate values from the record. But it should be able to handle more than just hard-coded access to "city" records. Thus, the Quadtree should employ the **Strategy design pattern** (OpenDSA 6.1.4) by using a "coordinate extractor" object that knows how to get 2D coordinates from a specific record type, like a city. This is because the Quadtree will need to "compare" each city to the 2D split points, quadrant rectangles, and range rectangles associated with the Quadtree data structure, and possibly compare city's to each other to check for invalid duplicate x,y coordinates. This is somewhat analogous to the use of Comparators in the BSTs in Project 1.

Additional design requirements: You are expected to use good object-oriented design in your solution. As in Project 1, your Quadtree should be encapsulated and insulated from the database or I/O command processor functionality, and from the BSTs. Also, it will be helpful to encapsulate functionality associated with quadrant and rectangle boundaries and 2D point containment and comparison.

Input and Output:

The input to this program should be read from standard input and the output should be directed to standard output. The name of the program should be "Quad".

This project adds/modifies the following commands to Project 1. (The BST specific commands will not be graded again in this project.)

The input will consist of a series of commands (some with associated parameters, separated by spaces), one command for each line. Commands are free format in that an arbitrary number of additional spaces may be interspersed between parameters. You do not need to check for syntax errors in the command lines, however you do need to check for logical errors (such as deleting a value that is not in the database). Blank input lines are ignored, and lines beginning with "#" character are ignored as comments.

In the output, each command should be echoed to output with its parameters on one line, followed on the next line by any additional required output for that command. See the sample output for detailed syntax. When printing a city record, output it on a single line in the following format:

name population (x, y)

Six commands and their syntax are as follows.

insert *name population x y*

Insert a city record into the database. You will create a new city record, and insert it into the Quadtree (and both BSTs) if there is not already a city with that x,y in the database. Output **"Duplicate"** (not including the quotes) if the insertion was rejected due to a duplicate x,y; output **"Out of bounds"** if the insert was rejected due to x,y being out of the world bounds.

find location *x y*

Find and print the city record that has the given location, or **"Not found"** if no such city exists or x,y is out of bounds.

rfind *x y w h*

Find and print all cities located within the given bounding box, or **"Not found"** if no such cities exist or the box is entirely out of bounds. It is acceptable for part of the box to be out of bounds, so if cities are found within the box they should be reported. (x,y) is the upper left corner of the box, and w and h are the width and height of the box in the positive x and y directions. x,y,w,h are non-negative integers. The inclusiveness of the boundaries of the box are the same as that previously defined for Quadtree quadrants: on the line counts as being on the greater x or y side. Thus, cities exactly on the bottom or right edges of the box are not in the box. The cities must be printed in a recursive NW, NE, SW, SE traversal order, 1 city per output line.

delete location *x y*

Find and remove the city record with the given location from the database, if it exists. Be sure to remove the same record from the Quadtree and both BSTs. Output **"Not found"** if no such city exists or x,y is out of bounds.

tree location

Print a listing of the Quadtree nodes in **pre-order**. Quadtree children must appear in the order NW, NE, SW, SE. Each node should print on a separate line, as follows. If the node appears at depth *i* in the tree (where depth of the tree root = 0), print *4i* period **"."** characters before the node in the output.

- For an internal node, print "**Internal (x, y)**" where (x,y) is the coordinates of the center split point determining the quadrants of its 4 children.
- For an empty leaf node, print "**Empty**".
- For a non-empty leaf node, print the city record contained in that node.

makenull

Reinitialize the database to be empty. No additional output.

Programming Standards:

Same as for Project 1.

Testing:

A sample input and output test file will be posted. This is not the data file that will be used in grading your program. The test data provided to you will attempt to exercise the various syntactic elements of the command specifications. It makes no effort to be comprehensive in terms of

testing all of the edge cases in the data structures required by the program. Thus, while the test data provided should be useful, you must also create your own test cases to ensure that your program works correctly. It is acceptable to share input and output files on the class forum.

Deliverables:

1. Report

Submit a PDF file containing a short report that presents:

(1) Your updated overall object-oriented design.
(2) How you made the find and rfind commands efficiently prune the search space.

You should submit your report PDF along with your code on WebCAT as below, by including it in your Eclipse project or your submitted zip file.

2. Code:

You will submit your project code through the automated Web-CAT server. Links to the Web-CAT client are posted at the class website. If you make multiple submissions, only your last submission will be evaluated. There is no limit to the number of submissions that you may make.

You are required to submit your own test cases with your program, and part of your grade will be determined by how well your test cases test your program, as defined by Web-CAT's evaluation of code coverage. Of course, your program must pass your own test cases. Part of your grade will also be determined by test cases that are provided by the graders. Web-CAT will report to you which test files have passed correctly, and which have not. Note that you will not be given a copy of grader's test files, only a brief description of what each accomplished in order to guide your own testing process in case you did not pass one of our tests.

If submitting through Eclipse, the format of the submitted archive will be managed for you. If you choose not to develop in Eclipse, you will submit either a ZIP-compressed archive (compatible with Windows ZIP tools or the Unix zip command) or else a tar'ed and gzip'ed archive. Either way, your archive should contain the source code for the project only (no .class files or binary resources).

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. 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 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.
```