Rectangle Quadtree Project

Background: 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. 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 for 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.

Implementation:

Your database will now be organized by two data structures. One will be the Skip List, which will organize the collection of rectangles by name as in Project 1. The second data structure will be a Quadtree, described below. The Quadtree will organize the rectangles by position, and will be used for spatial queries such as locating rectangles within a query rectangle, and determining intersections among rectangles.

Before reading the following description of the Quadtree used for this project, you should first read Section 13.3.2 in the textbook about PR Quadtrees. You are **not** implementing a PR Quadtree for this assignment, but the PR Quadtree is similar enough to what you will implement that it is a good starting point for understanding what you need to do.

You will use a Quadtree[1] to support spatial queries. The 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, the four children split the parent's corresponding square into four equal-sized quadrants. Also as with the PR Quadtree, internal nodes of the Quadtree do not store data. Pointers to the rectangles themselves are stored only in the leaf nodes.

The key property of the Quadtree is its **decomposition rule**. The decomposition rule is what distinguishes the Quadtree of this project from, for example, the PR Quadtree of Section 13.3.2. The decomposition rule for this project is: A leaf node will split into four when (a) There are **three or more** rectangles in the node, such that (b) not all rectangles in the node overlap a single point. Four sibling leaf nodes will merge together to form a single leaf node whenever deleting a rectangle results in a set of rectangles among the four leaves that does not violate the decomposition rule. It is possible for a single deletion to cause a cascading series of node merges. (Note that two rectangles that are adjacent to each other **do not** overlap. For example, rectangles (5, 5, 5, 5) and (5, 10, 5, 5) do not overlap.)

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 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.

Consider for example the four rectangles (A, 200, 200, 400, 300), (B, 250, 250, 500, 500), (C, 600, 600, 400, 400) and (D, 650, 650, 300, 300). The Quadtree that results from storing these

[1] Actually, "Quadtree" refers to any data structure where each internal node has 4 children. The data structure described for this project has no formal name of its own. It would be incorrect to believe that the term "Quadtree" refers only to the data structure described here, but for notational convenience, that is what we will call it.

rectangles will contain an internal node at the root with four leaf nodes as children, such that the NW child stores A and B; the NE child stores A and B; the SW child stores B; and the SE child stores B, C, and D. Note that there are three rectangles in the SE child because they mutually overlap.

All access functions on the Quadtree, including insert, delete, and search, must be written recursively. The **regionsearch** and **intersections** commands will access the quadtree rather than the Skip List. These commands should visit as few Quadtree nodes as possible.

When performing intersection tests, intersection between a given pair of rectangles should be reported only once. Be careful about when you report an intersection, since it is possible that a rectangle will appear in multiple Quadtree leaf nodes. In fact, a pair of intersecting rectangles might appear (and intersect) multiple times in the Quadtree, such as Rectangles A and B in the example above.

You must use class inheritance to design your Quadtree nodes. You must have a Quadtree node base class, 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.

Invocation and I/O Files:

The name of your executable must be `p2`. There will be no input parameters to the program. Your program should read the command file from the standard input, and write its output to the standard output.

Your program will read from standard input a series of commands, with one command per line. The commands, their formats, and their effects/outputs are identical to those of Project 1, with the following modifications and one addition.

(1) All rectangles will fit within a box 1024 by 1024 units in size, and all coordinates for rectangles are in the range 0 to 1023. Note that the coordinates for query rectangles (i.e., the search command) are permitted to go outside of the "world box" of size 1024.

(2) The **regionsearch** command and the **intersections** command must report the number of Quadtree nodes visited.

There is one additional command:

dump

Return a "dump" of the Quadtree and the Skip list. The Quadtree dump should print the nodes of the Quadtree in preorder traversal order, one node per line. 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.

Design Considerations:

The most obvious design issue is how to organize the inter-relations between the Skip List and the Quadtree. Neither uniquely "owns" the rectangles it organizes. You will probably want each data structure to store pointers to rectangle objects.

It is probably 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 Quadtree for actual implementation.

When navigating through the Quadtree, for example to do an insert operation, you will need to know the coordinates and size of the current Quadtree node. One design option is to store with each Quadtree node its coordinates and size. However, this is unnecessary and wastes space. 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 to the recursive function.

Note that many leaf nodes of the 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.