Rect_BST (Fall 2017)

Background: Many applications areas such as computer graphics, geographic information systems, and VLSI design require the ability to store and query a collection of rectangles. In 2D, typical queries include the ability to find all rectangles that cover a query point or query rectangle, and to report all intersections from among the set of rectangles. Adding and removing rectangles from the collection are also fundamental operations.

For this project, you will create a simple spatial database for handling inserting, deleting, and performing queries on a collection of rectangles. The data structure used to store the collection will be the Binary Search Tree (BST, see Section 7.11 of the OpenDSA textbook for more information about BST).

Invocation and I/O Files:

The name of the program is Rectangle1 (this is the name where Web-CAT expects the main class to be). There is a single command line parameter that specifies the name of the command file. So, the program would be invoked from the command-line as:

```
java Rectangle1 {command-file}
```

Your program will read a series of commands from the command file, with one command per line. No command line will require more than 80 characters. Each command requires certain outputs, whose details will be described by sample test file outputs that we will post. The formats for the commands are as follows. The commands are free-format in that any number of spaces may come before, between, or after the command name and its parameters. All coordinates will be signed values small enough to fit in a 32-bit int variable.

insert name x y w h
Insert a rectangle named name with upper left corner (x, y), width w and height h. It is permissible for two or more rectangles to have the same name, and it is permissible for two or more rectangles to have the same spatial dimensions and position. The name must begin with a letter, and may contain letters, digits, and underscore characters. Names are case sensitive. A rectangle should be rejected for insertion if its height or width are not greater than 0. All rectangles must fit within the “world box" that is 1024 by 1024 units in size and has upper left corner at (0, 0). If a rectangle is all or partly out of this box, it should be rejected for insertion.

remove name
Remove the rectangle with name *name*. If two or more rectangles have the same name, then any one such rectangle may be removed. If no rectangle exists with this name, it should be so reported.

remove x y w h
Remove the rectangle with the specified dimensions. If two or more rectangles have the same dimensions, then any one such rectangle may be removed. If no rectangle exists with these dimensions, it should be so reported.

regionsearch x y w h
Report all rectangles currently in the database that intersect the query rectangle specified by the *regionsearch* parameters. For each such rectangle, list out its name and coordinates. A

regionsearch command should be rejected if the height or width is 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.

intersections
Report all pairs of rectangles within the database that intersect.

search name
Return the information about the rectangle(s), if any, that have name *name*.

dump
Return a \dump" of the BST. The BST dump should print out each BST node (use the in-order traversal). For each BST node, print that node's value and the number of pointers that it contains.

Implementation and Design Considerations:

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

Be aware that for this project, the BST is being asked to do two things. First, the BST will handle searches on rectangle name, which acts as the record's key value. The BST can do this efficiently, as it will organize its records using the name as the search key. But you also need to do several things that the BST cannot handle well, including removing by rectangle shape, doing a region search, and computing rectangle intersections. So you will need to add functions to the BST to handle these actions. You should design in anticipation of adding a second data structure in Project 2 to handle these actions. Make sure you handle these actions in a general way that does not require the BST to understand its data type.

The biggest implementation difficulty that you are likely to encounter relates to traversing the BST during the intersections command. The problem is that you need to make a complete traversal of the BST for each rectangle in the BST (comparing it to all of the other 2 rectangles). The leads to the question of how do you remember where you are in the “outer loop" of the operation during the processing of the “inner loop" of the operation. One design choice is to augment the BST with an iterator class. An iterator object tracks a current position within the BST, and has a method that permits the position of the iterator object within the BST to move forward. In this way, one iterator object can be tracking the current rectangle in the “outer loop" of the process, while a second iterator can be used to track the current rectangle for the “inner loop". For the *regionsearch* and *intersections* commands, you need to determine intersections between two rectangles. Rectangles whose edges abut one another, but which do not overlap, are not considered to intersect. For example, (10, 10, 5, 5) and (15, 10, 5, 5) do NOT overlap, while (10, 10, 5, 5) and (14, 10, 5 5) do overlap. Note that rectangles (10, 10, 5, 5) and (11, 11, 1, 1) also overlap.

Programming Standards:

You must conform to good programming/documentation standards. Web-CAT will provide feedback on its evaluation of your coding style, and be used for style grading. Some additional specific advice on a good standard to use:
_ You should include a header comment, preceding main(), specifying the compiler and operating system used and the date completed.

_ Your header comment should describe what your program does; don't just plagiarize language from this spec.
_ You should include a comment explaining the purpose of every variable or named constant you use in your program.
_ You should use meaningful identifier names that suggest the meaning or purpose of the constant, variable, function, etc. Use a consistent convention for how identifier names appear, such as “camel casing".
_ Always use named constants or enumerated types instead of literal constants in the code.
_ Precede every major block of your code with a comment explaining its purpose. You don't have to describe how it works unless you do something so sneaky it deserves special recognition.
_ You must use indentation and blank lines to make control structures more readable.
_ Precede each function and/or class method with a header comment describing what the function does, the logical significance of each parameter (if any), and pre- and post-conditions.

We can't help you with your code unless we can understand it. Therefore, you should no bring your code to the TAs for debugging help unless it is properly documented and exhibits good programming style. Be sure to begin your internal documentation right from the start. You may only use code you have written, either specifically for this project or for earlier programs, or the codebase provided by the instructor. Note that the textbook code is not designed for the specific purpose of this assignment, and is therefore likely to require modification. It may, however, provide a useful starting point.