Project 1

Start Assignment

Due Apr 1 by 7:59am **Points** 100 **Submitting** a file upload **File Types** zip
Available Feb 28 at 9am - Apr 1 at 7:59am

Assignment

Please download the starter code from files tab

Many application 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 Skip List (see the Skip list chapter) of the textbook for more information about Skip Lists). The Skip List fills the same role as a Binary Search Tree in applications that need to insert, remove, and search for data objects based on some search key such as a name. The Skip List is roughly as complex as a BST to implement, but it generally gives better performance since its worst case behavior depends purely on chance, not on the order of insertion for the data. Thus, the Skip List provides a good organization for answering non-spatial queries on the collection (in particular, for organizing the objects by name). However, as you will discover, the Skip List performs poorly on spatial queries. In Project 2, you will implement a more sophisticated data structure that is capable of processing the spatial operations more efficiently.

(https://canvas.vt.edu/courses/135977/assignments/1257738and-io-files) Invocation and I/O Files

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

```
%> java Rectangle1 {commandFile}
```

where:

- `Rectangle1` is then name of the program. The file where you have your `main()` method must be called `Rectangle1.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 additional commands. In the following description, terms in `{ }` are parameters to the command.

- **`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.

```
 #insert successful
insert a 1 0 2 4      #valid insert
Rectangle inserted: (a, 1, 0, 2, 4) #expected output

# insert failed
insert a -1 -1 2 4    #invalid insert, coord < 0
Rectangle rejected: (a, -1, -1, 2, 4) #expected output
```

- **`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 successful
remove a                          #rect called A exists
Rectangle removed: (a, 1, 0, 2, 4) #expected output

# remove failed
remove b                 #rect called b DOES NOT exists
Rectangle not found: (b) #expected output
```

- **`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.

```
 #remove successful
remove 1 0 2 4                    #rect exists at those coords
Rectangle removed: (a, 1, 0, 2, 4) #expected output

# remove failed
remove 2 0 4 8                    #rect does not exists at those coords
Rectangle not found: (2, 0, 4, 8) #expected output
```

- `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 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.

```
#regionsearch valid with rects
regionsearch 900 5 1000 1000             #valid region that contains rects
Rectangles intersecting region (900, 5, 1000, 1000): #expected output header
(a, 0, 0, 1000, 10)                              # rectangles listed
(b, 0, 0, 910, 10)

#regionsearch valid without rects
regionsearch 0 500 20 1             #valid region that containing no rects
Rectangles intersecting region (0, 500, 20, 1): #expected output header
                                              # no rectangles listed

#regionsearch invalid
regionsearch 0 0 -10 20                     #rect does not exists at those coords
Rectangle rejected: (0, 0, -10, 20)   #expected output
```

- `intersections`

  Report all pairs of rectangles within the database that intersect.

```
intersections

#expected output when intersections exist
Intersection pairs:                     #Header is printed everytime.
(a, 10, 10, 15, 15| b, 11, 11, 5, 5)
(c, 0, 0, 1000, 10 | d, 0, 0, 10, 1000)

#expected output when NO intersections exist
Intersection pairs:                     #Header is printed everytime.
```

- `search {name}`

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

```
#regionsearch valid with rects
search a
```

```
#if DB has rectangles that have name 'a'
(a, 10, 10, 15, 15)
(a, 50, 21, 52, 1)

 #if DB does NOT have rectangles that have name 'a'

Rectangle not found: a
```

- `dump`

  Return a “dump” of the Skip List. 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 (depth).

```
#regionsearch valid with rects
dump

#if skiplist has nodes:
SkipList dump:
Node has depth 3, Value (null) #header node
Node has depth 3, Value (a, 1, 0, 2, 4)
Node has depth 2, Value (b, 2, 0, 4, 8)
SkipList size is: 2

#if skiplist has no nodes:

SkipList dump:
Node has depth 1, Value (null)
SkipList size is: 0
```

(https://canvas.vt.edu/courses/135977/assignments/1257738 considerations)

To test your program, here are two command files along with their corresponding correct output exist inside the files tab

Design Considerations

The rectangles 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 rectangles, but the Skip List is a general container class. Therefore, it should not be implemented to know anything about rectangles. It should be implemented as a generic container class.

Be aware that for this project, the Skip List is being asked to do two things. First, the Skip List will handle searches on rectangle name, which acts as the record’s key value. The Skip List 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 Skip List 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 Skip List to handle these actions, but these particular methods can go away in Project 2. 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 Skip List to understand its data type.

The biggest implementation difficulty that you are likely to encounter relates to traversing the Skip List during the intersections command. The problem is that you need to make a complete traversal of the Skip List for each rectangle in the Skip List (comparing it to all of the other rectangles). This 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 (you don't need to do it) is to augment the Skip List with an iterator class. An iterator object tracks a current position within the Skip List, and has a method that permits the position of the iterator object within the Skip List 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 Javadoc comment for all your classes and public methods describing what the method/class does, the purpose of each parameter, and pre- and post-conditions for methods.
- 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.

We can’t help you with your code unless we can understand it. Therefore, you should no bring your code to the GTAs or the instructors 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. However, it provides a useful starting