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 Skip List 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, it is difficult and inefficient for the Skip List to accomplish spatial queries.

Invocation and Input/Output

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. It is a text file.

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. 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 the standard output console. 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. The curly-brace characters '{' and '}' seen below will not actually be present. However, the output of several commands will include parentheses.

Insert

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 world box, it should be rejected for insertion.

Command Format	`insert {name} {x} {y} {w} {h}`
Command Example	`insert   hello 2  5  1     5`
Output on Success Example	`Rectangle inserted: (hello, 2, 5, 1, 5)`
Output on Failure Example	`Rectangle rejected: (hello, 1020, 1020, 1, 7)`

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. Remember that the head node should always match the highest 'depth' your SkipList has created.

Command Format	`dump`
Command Example	`dump`

Output on Success Example	`SkipList dump:`
	`Node has depth 3, Value (null)`
	`Node has depth 3, Value (a, 1, 0, 2, 4)`
	`Node has depth 2, Value (b, 2, 0, 4, 8)`
	`SkipList size is: 2`

Output on Success Example2	`SkipList dump:`
	`Node has depth 1, Value (null)`
	`SkipList size is: 0`

Remove-by-name

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

Command Format	`remove {name}`
Command Example	`remove someRect`
Output on Success Example	`Rectangle removed: (a, 1, 0, 2, 4)`
Output on Failure Example	`Rectangle not removed: (b)`

Remove-by-coords

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.

Command Format	`remove {x} {y} {w} {h}`
Command Example	`remove   2  5  1     5`
Output on Success Example	`Rectangle removed: (foundYa, 2, 5, 1, 5)`
Output on Failure Example	`Rectangle not removed: (2, 5, 1, 5)`

Region Search

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.

Command Format	`regionsearch {x} {y} {w} {h}`
Command Example	`regionsearch  -900 5 5000 20`

Output on Success Example	`Rectangles intersecting region (-900, 5, 5000, 20):`
	`(a, 45, 0, 10, 10)`
	`(b, 400, 0, 100, 310)`

Output on Success Example2	`Rectangles intersecting region (2, 2, 1, 1):`
Output on Failure Example	`Rectangle rejected: (-900, 5, 0, 50)`

Intersections

Report all pairs of rectangles within the database that intersect.

Command Format	`intersections`
Command Example	`intersections`

Output on Success Example	`Intersections pairs:`
	`(goodRect, 5, 3, 56, 56 | goodRect3, 25, 3, 6, 6)`
	`(goodRect3, 25, 3, 6, 6 | goodRect, 5, 3, 56, 56)`

Output on Failure Example	`Intersection pairs:`

Search

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

Command Format	`search {name}`
Command Example	`search    mainRect`

Output on Success Example	`Rectangles found:`
	`(mainRect, 25, 3, 6, 6)`
	`(mainRect, 111, 23, 16, 16)`

Output on Failure Example	`Rectangle not found: (mainRect)`

Design Considerations

The rectangles will be maintained in a Skip List, 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 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. 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. 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 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