# CS3114 (Fall 2017)
# PROGRAMMING ASSIGNMENT #2

**Due Thursday, Oct 19 @ 11:00 PM for 100 points**
**Due Tuesday, Oct 17 @ 11:00 PM for 10 point bonus**

## 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 binary search tree (BST). The BST 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 to organize by each key. However, even if we added a second BST to the system, we would still have the problem that the BST 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.

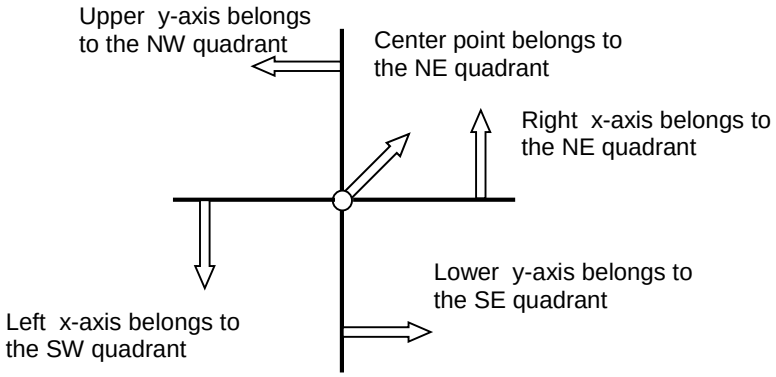
## Implementation:

Your database will now be organized by two data structures. One will be the BST, which will organize the collection of objects by name as in Project 1. The second data structure will be a variant of the PR Quadtree, as described below. In order to simplify the mechanics of the insert and delete operations on the PR Quadtree, in Project 2 the data objects being stored will be points instead of rectangles. The PR Quadtree will organize the points by position, and will be used for spatial queries such as locating points within a query rectangle, and determining the duplicate points.

Before reading the following description of the PR Quadtree variant used for this project, you should first read the Spatial Data Structures and PR Quadtree modules (15.2 and 15.3) in OpenDSA. You are implementing something similar to the PR Quadtree as described in the OpenDSA PR Quadtree module (15.3), but there are some important differences in the split and merge rules.

You will use a variant of the PR Quadtree to support spatial queries. The PR 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 module (15.3) described in OpenDSA, the four children split the parent's corresponding square into four equal-sized quadrants. The internal nodes of the PR Quadtree do not store data. Pointers to the points themselves are stored only in the leaf nodes. The key property of any Quadtree data structure is its decomposition rule. The decomposition rule is what distinguishes the PR Quadtree of this project from the OpenDSA PR (15.3). The decomposition rule for this project is: A leaf node will split into four when (a) there are more than three points in the node, such that (b) not all of the points have the same x and y coordinates. So there is no limit in principle to the number of points stored in a single leaf node, if they all happen to have the same position. Four sibling leaf nodes will merge together to form a single leaf node whenever deleting a point results in a set of points among the four leaves that does not violate the decomposition rule. It is possible for a single insertion to cause a cascading series of node splits. It is also possible for a single deletion to cause a cascading series of node merges.

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 PR 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. The points on the quadrant boundaries must be handled in the following manner:




All access functions on the PR Quadtree, including insert, delete, regionsearch, and duplicates must be written recursively. The regionsearch and duplicates commands will access the PR Quadtree rather than the BST. These commands should visit as few Quadtree nodes as possible.

You must use class inheritance to design your PR Quadtree nodes. You must have a PR Quadtree node base class (or interface), 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. Your Quadtree node implementation **may not** include a pointer to the parent node.

## Invocation and I/O Files:

The name of the program is Point1 (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, just like in Project 1. The commands, their formats, and their effects/outputs are similar to those of Project 1, with the following alterations.

1. All commands that used to take 4 integers to specify a rectangle data object now take 2 integers to specify a point instead. (The exception is **Regionsearch**, which still takes 4 integers.)
2. The **intersections** command is changed to be the **duplicates** command.
3. The **regionsearch** command must report the number of Quadtree nodes visited.
4. The **dump** command is modified to add a "dump" of the Quadtree. The Quadtree dump should print the nodes of the Quadtree in preorder traversal order, one node per line, with each line indented by 2 spaces for each level in the tree (the root will indent 0 spaces since it is considered to be at level 0). These lines should appear after the BST nodes are printed.

## Design Considerations:

The most obvious design issue is how to organize the inter-relations between the BST and the PR Quadtree. Neither uniquely "owns" the points that it organizes. You will probably want each data structure to store pointers to (shared) point objects. It is 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 BST and/or PR Quadtree for actual implementation.

Many leaf nodes of the PR 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.

When navigating through the PR Quadtree, for example to do an insert operation, you will need to know the coordinates and size of the current PR Quadtree node. One design option is to store with each PR Quadtree node its coordinates and size. However, this is unnecessary and wastes space. Worse, it is incompatible with the Flyweight design pattern, since the flyweight object has to be stateless. 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.

## 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 case”.
- Always use named constants or enumerated types instead of literal constants in the code.
- Precede each function and/or class method with a header comment describing what the function does, its return type, and the logical significance of each parameter (if any). 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 code provided by the instructor. Note that the OpenDSA 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.

## Deliverables:

You will implement your project using Eclipse, and you will submit your project using the Eclipse plugin to Web-CAT. 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 these 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.

When structuring the source files of your project, use a directory structure; that is, your source files will all be contained in the project "src" directory. Any subdirectories in the project will be ignored.

You are permitted to work with a partner on this project. When you work with a partner, then only one member of the pair will make a submission. Be sure both names are included in the documentation and you must select your partner when submitting to Web-CAT. Whatever is the final submission from either of the pair members is what we will grade unless you arrange otherwise with the GTA.

## 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. The text of the pledge will also be posted online.

```
// 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 my partner (in the case of a joint
// submission), 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.
```

Programs that do not contain this pledge will not be graded.