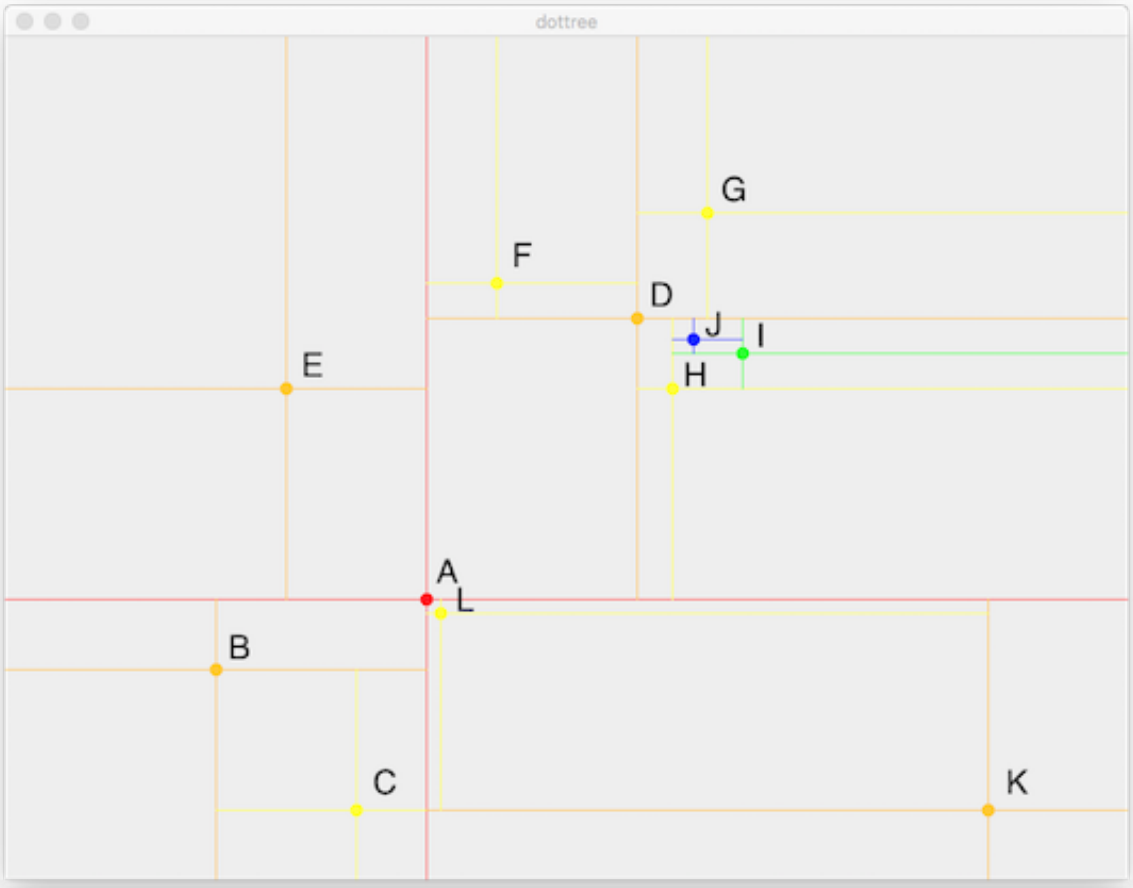
CS 10 PS-2
Quadtrees

Introduction

Suppose we have a bunch of animated blobs. How can we determine if the mouse was pressed in one of them? How can we detect when two of them bump into each other? We've seen the inefficient approach to determining which was clicked on — test each one. And we can imagine an inefficient approach to detecting collisions — look at all pairs. But we can do better, and would need to do better in some applications requiring speed.

Just as a binary search tree enables faster look-up than linear search in one dimension (based on some notion of less than or greater than), a point quadtree enables faster look-up in two dimensions. The structure is like a BST: each node stores an element, and then has children recursively storing additional elements based on how they compare to the parent. But now there are up to four children, in the four quadrants:



Here A is the root, and it has children B (lower left), E (upper left), and D (upper right), and K (lower right). B has only one child C (lower right), while D has three children F (upper left), G (upper right), and H (lower right). H has one child I (upper right), which itself has one child J (upper left). Finally, K has one child L (upper left).

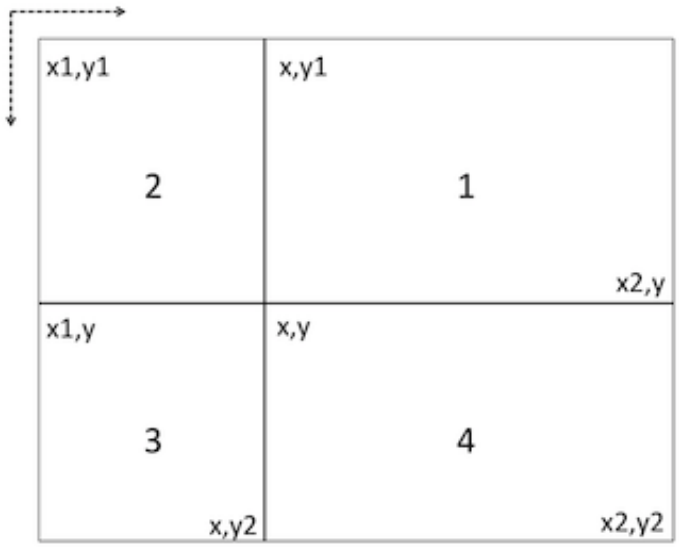
Note: there are several variations on the idea of a quadtree; we're following here an extension of the classic approach described by Finkel and Bentley in "Quad Trees: A Data Structure for Retrieval on Composite Keys", *Acta Informatica* 4, 1—9, 1974. See Fig. 1 for another example, also drawn as a tree. The paper is a good read, though I've translated some of the ALGOL-like description to pseudocode here, and used recursion rather than iteration. Be careful searching the web for further descriptions, as you might end up implementing an extra credit quadtree instead of the assigned one!

Data Structure

We want our quadtree to be generic to what "points" it holds, as long as they provide the ability to get *x* and *y* coordinates. Thus we have a new interface `Point2D` specifying those methods. We then modify `Blob` to indicate that it implements that interface, and to emphasize the generic-ness we also use a simple class `Dot` that meets the interface specification.

The basic data structure that we'll use for each node in the tree includes:

- a point, of generic type `E`, as long as it implements the `Point2D` interface (so we can get coordinates)
- the two corners, upper-left (x1,y1) and lower-right (x2,y2), of the rectangle that the node covers. (It isn't really necessary to store these, but it makes our life easier.)
- Four `PointQuadtree` children, one for each quadrant, numbered as in the figure. Any quadrant may be empty, in which case the child is `null`.



So the main tree covers (x1,y1) to (x2,y2), while its upper-right quadrant #1 covers (x,y1) to (x2,y), and so forth. Make sure you agree with this. **Note**: I'm sticking with the graphics convention of the origin at the upper-left, and smaller *y* values higher up in the image. This makes computing these things with < and > feel a little strange compared to our math intuition, but such is Java's graphics.

Insertion

Inserting a point works much like it does with a BST (except with four children instead of two): at a given node, see which side (quadrant) the point lies on, and recursively insert in the appropriate child. When there is no child for that quadrant, create a new leaf containing just the point, and store that leaf as that child.

```
To insert point p, which is at (x,y)
  If (x,y) is in quadrant 1
    If child 1 exists, then insert p in child 1
    Else set child 1 to a new tree holding just p
  And similarly with the other quadrants / children
```

Finding

Finding in the tree is similar to finding in a BST, with one important difference (in addition to number of children): typically here we want to find *all* points within some region of interest, not just a single value. For example, the mouse might not be pressed *exactly* where a point is, but if it's close enough, we'll consider it a hit. Thus we search in a circle around the mouse location. Similarly, to detect collisions, we search in a circle around a given blob to see if another blob is too close. So the recursive call for `find` considers *all* children whose rectangles overlap the region of interest.

So let's consider how to find all objects within a circle. We'll assume that we have methods to see if a point is inside a circle, and to see if a circle intersects a rectangle.

```
To find all points within the circle (cx,cy,cr), stored in a tree covering rectangle (x1,y1)-(x2,y2)
  If the circle intersects the rectangle
    If the tree's point is in the circle, then the blob is a "hit"
    For each quadrant with a child
      Recurse with that child
```

Here are some example search circles in the above picture; make sure you're comfortable with what rectangles get checked and what tree points get checked:

- Searching in a circle near where B is. Start at the root, A, and see that the circle intersects its rectangle. In this case the circle around the point we are searching does intersect the rectangle, because A's rectangle is the entire board. Test the point to see if A is inside the circle around the point we are searching — not a hit since the search point is too far away. Recurse to A's children D, E, B, and K. The circle only intersects B's

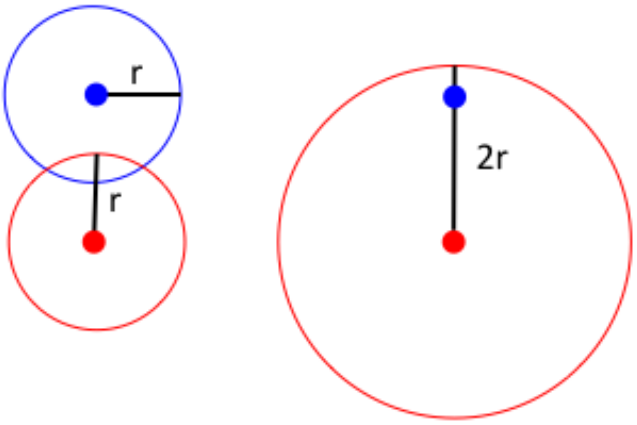
rectangle. Check B's point; it is a hit since the search point is close to B. Also recurse with B's child C to see if more than point is in the search circle. The circle intersects C's rectangle. Check C's point; not a hit.
- Searching in a circle near where G is. Start at the root, A, and see that the circle intersects A's rectangle. Test its point; not a hit. Recurse to A's children D, E, B, and K. The circle only intersects D's rectangle. Check D's point; not a hit. Also recurse with its children G, F, and H. The circle only intersects G's rectangle. Check G's point; it is a hit.
- Searching in a circle near where A is (big enough to also contain L). Start at A; circle intersects A's rectangle; point is a hit. Recurse to children D, E, B, and K. Circle intersects all of their rectangles but doesn't contain any of their points. Further recursion from these tests G, F, H, C, and L. Only L's rectangle intersects. And its point is also a hit.

(Note: these are the first three test cases in test1 of the provided QuadtreeTest.)

Applications

This data structure and search mechanism then lets us solve the problem of quickly finding blobs near the mouse press — search in a circle around it.

When we move from points to blobs of some size for collision detection, we actually have to account for that size. Blobs intersect when their sides touch each other, not their centers. While in general, a point quadtree is only supposed to contain points (with variations developed to allow regions that have some extent in space), we can handle it here with a little trick. Let's deal only with blobs of a common, fixed radius. Then we can simply expand the circle we're looking for by that same amount and look for a point. For example, since the red blob has radius *r*, and so does the blue blob (and any other blob), by looking for other blobs within *2r* of the red blob we will detect the blue blob (and any other nearby blob).



Now we can efficiently solve collision detection. Insert all the blobs into a tree. For each blob, find all other blobs within a circle of radius twice that of the target blob.

Implementation Notes

Several files are provided for you.

PointQuadtree

This is the scaffold for our core data structure.

- The code is quite analogous to BST, so familiarize yourself well with that class and method first. It is recursive, with data structure recursion — from a node to its children.
- Note that `size` should directly compute the size, not just by using `allPoints` as a helper function. Do `size` first as a warm-up, practicing recursion.
- In order to build up a list over the course of recursion (`allPoints` and `findInCircle`), a helper function is useful. You will be graded on efficiency, and appending lists is not efficient. Refer back to the fringe of a binary tree.

Geometry

This class provides two static methods for the geometry tests, instrumented to keep track of how many times each is called, for testing purposes.

- Functions are provided for testing point-in-circle and circle-intersects-rectangle.
- Think of it like `Math`, a library of methods that you can invoke from anywhere; there, `Math.min`, here, `Geometry.pointInCircle` and `Geometry.circleIntersectsRectangle`.
- Note that the functions increment static variables, which hold their values over all calls, and thus allow us to count the number of calls. Methods allow other classes to reset and access the counts. This is used in the `QuadtreeTest` to compare the number of calls your code makes to the ideal number.

Point2D

This interface says what can go in a point quadtree (much like Java's `Point`, but under our control).

Dot

This class provides a simple implementation of `Point2D`, useful for the initial testing. (If you find it useful, you can give each dot a "name", as I did in the images above, and use it in debugging/testing.)

QuadtreeTest

This test drive provides GUI-less code to help you automatically test your solution and make sure it's working correctly.

- This has some code to help you make sure you're recursing correctly. An implementation of `findInCircle` that simply recurses in all four quadrants will get the right answer, but without actually leveraging the quadtree structure. So if you do that, you will lose substantial credit for that part of the assignment! These tests will help you make sure you're doing the right thing recursively.
- If your code throws an exception, the numbers might or might not match; so first get rid of the bug leading to the exception.
- You must use the above geometry methods, rather than your own, in order to make use of this testing scaffold.

DotTreeGUI

With this GUI scaffold, you can test your point quadtree implementation graphically and interactively — insert points and see what's near the mouse.

- The GUI has two modes: 'a' for add to the tree and 'q' for query the tree to find blobs near the mouse press. It also allows you to grow/shrink the size of the circle around the mouse, in which you're detecting blobs.
- The `drawTree` method is outside the `PointQuadtree` class, so we explicitly pass around a tree object (and recurse with its children).

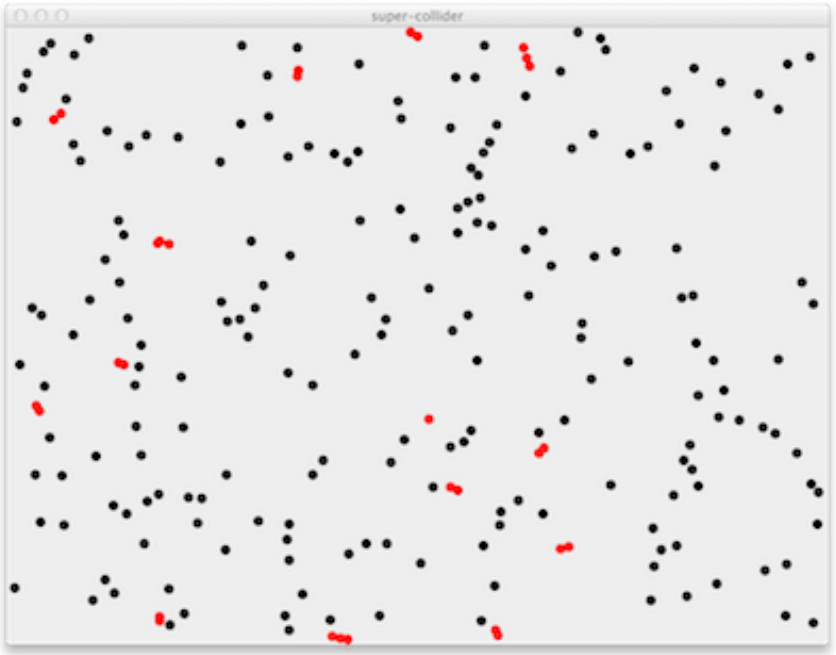
Blob

Our old friend has been updated slightly to indicate that it implements the `Point2D` interface. You will need to delete/replace the original Blob.java file from class with this one, else IntelliJ will complain.

CollisionGUI

Finally, what we've been building up to — it sets up a bunch of colliding blobs. Have fun with this part — you've earned it by the time you get there. :)

- The core of the GUI is much like `BlobsGUI`, except that only a couple of blob types (with fixed radius) are included. Also a bunch of blobs at random positions can be added at once (key 'r').
- There are two collision-handling modes: either just color the colliding blobs differently (key 'c', picture below from sample solution) or actually destroy them upon collision (key 'd'). The functionality is based on the `findColliders` method, which sets the colliders instance variable. Destruction takes advantage of the `removeAll` method of lists.
- To build a tree with all the blobs, create a new instance holding just blob number 0, and then add blobs 1.. to it.
- As described above, to check collisions, do a `find` around a blob. Note that this will find the blob itself, since it's in the tree. That's okay, just handle it carefully.



Exercises

For this problem set, you are allowed to work with exactly one partner. Note that you do not have to work with a partner, and if you do, you will both be given the same grade; there is no penalty (or bonus). Re-read the course policies. You should weigh whether you will get more out of this assignment working alone or with someone else; there are advantages to both. If you want us to set you up with a partner, please fill out the form on Canvas.

1. Implement the `PointQuadtree` methods. `Insert` is of course first, but `size` and even `allPoints` should come along soon after, to help test. Note that you can (and should) do some testing of these even before tackling `findInCircle`.
2. Try writing a `toString` method to print out a simple tree and make sure it looks right in that format (you'll show them soon, and that'll help for more complicated trees). Note that I already provided a `toString` for `Dot`, including the name (if given) and coordinates. My test cases in `QuadtreeTest` are based on the image at the top of the assignment, creating the same tree, so you can test some print statements against the graphical intuition, before even doing the graphical part.
3. Use the testing scaffold in `QuadtreeTest` to make sure the recursion is working correctly. Add to it a `test2` method (and more if you like) codifying additional test cases you devise. That is, your method(s) should create a tree and do some searches on the tree, comparing expected to actual results. You can use my `testFind` method if you like, or just do something simpler, e.g., comparing the points found to what you think should be found. In your write-up, describe these test cases — what they're testing and how they help give you confidence in your code's correctness.
4. Implement the `DotTreeGUI` methods, to enable graphical construction and viewing of your trees.
5. Exercise your tree code graphically by inserting points in different patterns (including the one illustrated above), checking that it displays correctly and that the mouse press picks up the points that it should (and doesn't pick up others). You can expand/contract the mouse radius with +/- keys.
6. Implement the `CollisionGUI` methods.
7. Test your collision detection method by eye in an *ad hoc* fashion, as well as by devising careful test cases of things that should collide and things that should not (i.e., construct the blobs in ways that you know whether or not they'll collide). Take a screenshot of color-coded collisions (as above) and briefly describe your tests.

You may obtain extra credit for extending and enhancing the app or the tree itself. Only do this once you are completely finished with the specified version. Make a different file for the extra credit version, and document what you did and how it works. Some ideas:

- Extend the `DotTreeGUI` to show the nodes tested in trying to find the mouse press location.
- Implement a find-in-rectangle method.
- Let the user drag out a circle (or rectangle) and show the points in it.
- Extend `CollisionGUI` to handle additional blob types.
- Allow for blobs with variable radii, correctly accounting for how they must be searched.
- Implement a point-region quadtree, which uses a uniform spatial subdivision and keeps points within these quadrants.

Submission Instructions

Turn in a single zipfile holding your completed versions of the four classes (PointQuadtree, QuadtreeTest, DotTreeGUI, and CollisionGUI), along with screenshots of your GUIs and a document with your dicussion of tests.

Grading Rubric

Total of 100 points

Correctness (70 points)

10 PointQuadtree insert
5 PointQuadtree size
5 PointQuadtree allPoints [-3 if inefficient]
10 PointQuadtree findInCircle [-4 if inefficient with lists; -6 if inefficient with recursion]
5 DotTreeGUI mouse press invoking PointQuadtree to add, find
10 DotTreeGUI draw tree
5 CollisionGUI find colliders: build the tree
10 CollisionGUI find colliders: find all of them
10 CollisionGUI draw to show colliders

Structure (10 points)

4 Good decomposition of and within methods
3 Proper used of instance and local variables
3 Proper use of parameters

Style (10 points)

3 Comments for new methods (purpose, parameters, what is returned)
4 Good names for methods, variables, parameters
3 Layout (blank lines, indentation, no line wraps, etc.)

Testing (10 points)

5 Testing QuadtreeTest (tests and writeup)
5 Testing CollisionGUI (tests and writeup)