Program Design & Data Structures (Course 1DL201)
Uppsala University – Autumn 2014/Spring 2015

Homework Assignment 3: Quadtrees

Prepared by Tjark Weber

Lab: Friday, 6 February, 2015
Submission Deadline: 18:00, Friday, 13 February, 2015
Lesson: Wednesday, 25 February, 2015
Resubmission Deadline: 18:00, Friday, 13 March, 2015

Rectangles and Quadtrees

Binary search trees typically work on *one*-dimensional key spaces. Quadtrees allow us to search on *two*-dimensional key spaces (and extensions to higher-dimensional key spaces are straightforward). These kinds of trees are useful in many graphics applications and computer-aided design tools, such as for the design of VLSI (very-large-scale integration) circuits. Unlike nodes in a binary tree, which have at most two children, quadtree nodes have at most four children.

Let us first briefly discuss how we will use these trees. We are given a (possibly very large) collection of rectangles. Each rectangle is of the following type:

```
data Rectangle = Rect Integer Integer Integer Integer
```

A rectangle `Rect left top right bottom` has `left` as x-coordinate of the left edge, `top` as y-coordinate of the upper edge, `right` as x-coordinate of the right edge, and `bottom` as y-coordinate of the bottom edge. The normal convention in Cartesian geometry is followed: the coordinate system is such that as one goes toward the right and top, the x and y coordinates increase (see Figure 1). We assume the precondition `bottom` $<$ `top` and `left` $<$ `right` for any rectangle.

A point with integer coordinates (x, y) is said to be *inside* a rectangle `Rect left top right bottom` if and only if `left` $\leq x <$ `right` and `bottom` $\leq y <$ `top`. Note that the points on the top and right boundaries of a rectangle are thus not inside it. We also say that a rectangle *contains* any point inside it.

A quadtree allows us to represent a collection of rectangles so that one can efficiently search for all rectangles that contain a given point. (One can obviously also search if all the rectangles are simply kept in a list, but when there are millions of rectangles—for instance, when designing a microprocessor chip—this would be very slow.) Quadtrees organise such two-dimensional information in the following way:

- A quadtree covers a fixed rectangular region of the plane, itself represented by a rectangle, called the *extent* of the tree. A quadtree only stores rectangles whose points are contained in this region.

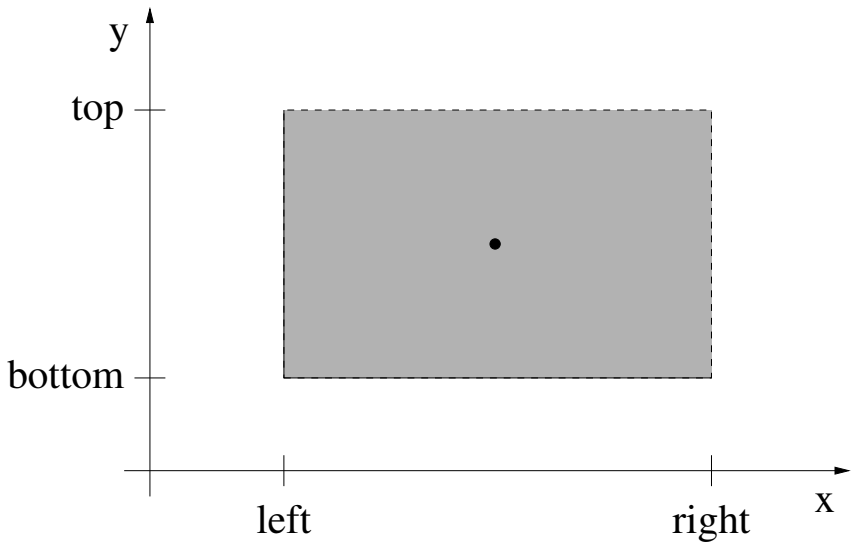

Figure 1: A rectangle. The black dot indicates its centre point.

- The centre point of the extent `Rect left top right bottom` has integer coordinates $x = (\texttt{left} + \texttt{right})\ \texttt{div}\ 2$ and $y = (\texttt{top} + \texttt{bottom})\ \texttt{div}\ 2$, where `div` denotes integer division.
- This centre point partitions the extent into four smaller rectangles, called *quadrants*, located at its top left, top right, bottom left, and bottom right. This can be extended recursively to smaller quadrants within a quadrant, until some termination criterion, such as a minimum quadrant size, is reached.

The next section explains in more detail how rectangles are stored in a quadtree.

Representing Rectangle Collections as Quadtrees

The `QuadTree` datatype has the following definition:

```
data QuadTree = EmptyQuadTree
              | Qt Rectangle [Rectangle] [Rectangle]
                   QuadTree QuadTree QuadTree QuadTree
```

In a non-empty quadtree `Qt extent horizontal vertical topLeft topRight bottomLeft bottomRight`,

- the `extent` rectangle, say `Rect left top right bottom`, defines the region covered by the quadtree,
- `horizontal` is the list of rectangles that contain some point of the horizontal centre line $y = (\texttt{top} + \texttt{bottom})\ \texttt{div}\ 2$ (i.e., if some point of that line is inside a given rectangle, this rectangle is stored in `horizontal`),
- `vertical` is the list of rectangles stored in the quadtree that contain some point of the vertical centre line $x = (\texttt{left} + \texttt{right})\ \texttt{div}\ 2$.

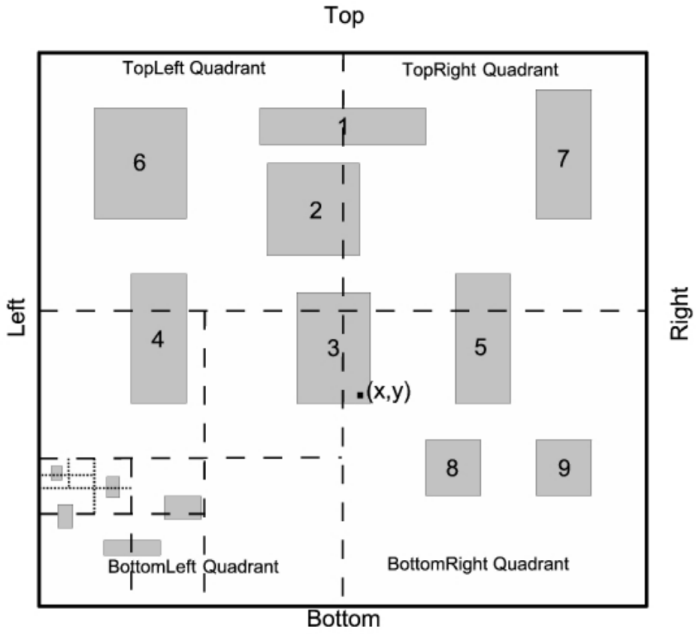

Figure 2: Storage of rectangles in a quadtree. The dashed lines are centre lines.

If both centre lines have some point inside a given rectangle, then this rectangle is inserted only into the `vertical` list. For example, in Figure 2, rectangles 1 to 3 are on the `vertical` list for the root quadtree, while rectangles 4 and 5 are on its `horizontal` list.

- If none of the two centre lines have a point inside a given rectangle, this rectangle is inserted either into the `bottomLeft` subtree, which covers the extent `Rect left ((top + bottom) div 2) ((left + right) div 2) bottom`, or into one of the other three subtrees (called `topLeft`, `topRight`, and `bottomRight` respectively), whose extents are defined similarly, such that none of the two centre lines have a point inside any of the quadrants. The areas of these quadrants need thus not be the same.

Example: For the extent `Rect 0 4 5 0`, the `topLeft`, `topRight`, `bottomLeft`, and `bottomRight` extents are `Rect 0 4 2 3`, `Rect 3 4 5 3`, `Rect 0 2 2 0`, and `Rect 3 2 5 0`, respectively (see Figure 3).

A given rectangle is thus recursively inserted into either the `vertical` list or the `horizontal` list associated with the subtree of the quadrant whose vertical respectively horizontal centre line has a point inside that rectangle.

To search for the rectangles containing a given point (x, y), first collect the rectangles on the `vertical` and `horizontal` lists of the root node that contain (x, y). Then continue to search recursively in the subtree covering the quadrant, if any, that contains the point; no recursive search is needed if (x, y) is on a centre line of the extent. For example, for the marked point (x, y) in Figure 2, one searches in the `vertical` and `horizontal` lists of the root extent and then only in the subtree that covers the bottom-right quadrant.

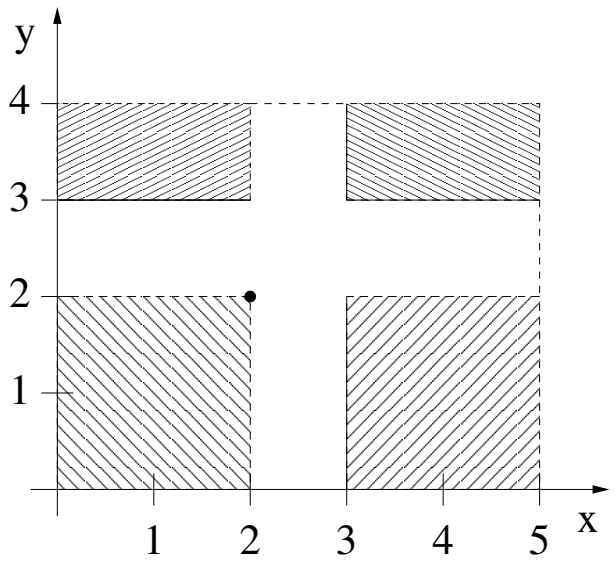

Figure 3: The extent `Rect 0 4 5 0` with its four quadrants. The black dot indicates the centre point at $(2, 2)$. Note that since we are using integer coordinates, all points inside the extent that are not on its (horizontal or vertical) centre lines are inside one of the quadrants.

Work to be Done

Download the file `QuadTree.hs` from the Student Portal. Add missing comments, and implement the following functions (empty implementations are already provided):

- `emptyQtree :: Rectangle -> QuadTree`, such that `emptyQtree e` (non-recursively) returns an (otherwise empty) quadtree with extent `e`.
- `insert :: QuadTree -> Rectangle -> QuadTree`, such that `insert q r` returns the quadtree `q` with rectangle `r` inserted, under the precondition that all points of `r` are inside the extent of `q`.
- `query :: QuadTree -> Integer -> Integer -> [Rectangle]`, such that `query q x y` returns the list (in any order) of rectangles in the quadtree `q` that contain the point `(x, y)`.

Running and Testing

There are (at least) two ways of loading your program into GHCi:

- Type `ghci QuadTree.hs` in a shell.
- Type `ghci` in a shell. Then type `:l QuadTree.hs`.

After modifying and saving your code in an editor, enter `:r` within GHCi to reload the file.

Five test cases have been provided for you in the file `QuadTree.hs`. These can be run by entering `runtests` in the GHCi shell. You are strongly encouraged to develop further test cases to test your code more thoroughly.

Grading

Your solution is graded on a U/K/4/5 scale based on two components: (1) functional correctness and (2) style and comments.

1. Functional correctness:

Your program will be run on an unspecified number of grading test cases that satisfy all preconditions but also check boundary conditions. Each test case is a quadtree creation for some extent `e`, followed by a sequence of insertions of rectangles whose points are all inside `e`, followed by a sequence of queries for the lists of rectangles of the resulting quadtree that contain some given points.

We reserve the right to run these tests automatically, so be careful to match exactly the imposed file names, function names, and argument orders.

Advice: Run your code in a freshly started Haskell session before you submit, so that declarations that you may have made manually do not interfere when you test the code.

The grade for this component is determined as follows:

- If your solution was submitted by the deadline, your file `QuadTree.hs` loads in GHCi and it passes all of the *test cases provided* in `QuadTree.hs`, you get (at least) a K for functional correctness.

  Otherwise (including when no solution was submitted by the deadline), you get a U grade for the homework assignment.

- If your program additionally passes at least 80% of the *grading test cases*, you get (at least) a 4 for functional correctness.

- If your program passes all *grading test cases*, you get a 5 for functional correctness. Note that some grading test cases may involve a large number of rectangles; your program needs to pass these tests in reasonable time (i.e., your program should not be unnecessarily inefficient).

2. Style and comments:

Your program is graded for style and comments according to our *Coding Convention*. The following criteria will be used:

- suitable breakdown of your solution into auxiliary/helper functions,
- function specifications and variants,
- datatype representation conventions and invariants,
- code readability and indentation,
- sensible naming conventions followed.

The grade for this component is determined as follows:

- If your program's style and comments are deemed a serious attempt at following these criteria, you get (at least) a K for style and comments. Otherwise, you get a U grade for the homework assignment.

- If you have largely followed these citeria, with very few major omissions or errors, you get a 4 for style and comments.
- If you have followed these criteria with at most minor omissions or oversights, you get a 5 for style and comments.

Final Grade

Component grades are converted to a final grade on the usual scale U/3/4/5 as follows:

1. You need to pass both components (functional correctness and style and comments) in order to pass this assignment.
2. A K grade in either component means that you are required to attend the **lesson** discussing the assignment and to subsequently resubmit the assignment.

   After resubmission, your *entire* assignment will be re-graded. Component grades of K will improve to 3 if you provide a mostly correct solution (cf. the criteria for grade 4); you cannot get a better grade than 3 in a component where you originally got a K. Component grades of 4 or 5 remain unchanged if your resubmission still meets the relevant grading criteria, but may be lowered otherwise (e.g., if your revised program no longer follows the coding convention).
3. Your final grade is the arithmetic mean of the two component grades.

Modalities

- The assignment will be conducted in groups of two students. Groups have been assigned via the Student Portal. *If you cannot find your partner until* **Tuesday, February 3**, *please contact Karl Sundequist* <`karl.sundequist@it.uu.se`> *to assign you a new partner—if possible.*
- Assignments must be submitted via the Student Portal. Only one solution per group needs to be submitted. Ensure that *both* group members' names appear on all submitted artefacts.

By submitting a solution you are certifying that it is solely the work of your group, except where explicitly attributed otherwise. We reserve the right to use plagiarism detection tools and point out that they are extremely powerful. You have already been warned about the consequences of cheating and plagiarism.

Good luck!