Program Design & Data Structures (course 1DL201)
Uppsala University – Autumn 2012 / Spring 2013
Homework 2: Quadtrees

Prepared by Pierre Flener

Lab: Wednesday 30 January 2013
Submission Deadline: 18:00:00 on Friday 1 February 2013
Lesson: Friday 8 February 2013
Resubmission Deadline: 18:00:00 on Thursday 14 February 2013

Rectangles and Quadtrees

While binary search trees typically work on ***one***-dimensional key spaces, quadtrees let us search on ***two***-dimensional key spaces, and extensions to higher-dimensional spaces are obvious. These kinds of trees are useful in many graphics applications and computer-aided design tools, say for the design of VLSI (very large-scale integration) circuits. Instead of having 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 of the following type:

```
datatype rectangle = Rect of int * int * int * int
```

where 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, that is we assume the pre-condition `bottom` $<$ `top` and `left` $<$ `right` for any rectangle. Note also that we thus do not consider degenerate rectangles, such as points or line segments.

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

A quadtree allows us to represent a collection of rectangles such that one can efficiently search for all rectangles containing a given point. One can obviously also do this by keeping all the rectangles in a list, but when there are millions of rectangles (for instance, when designing a microprocessor chip), this will be very inefficient. 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*.
- The centre point of the quadtree extent, say `Rect(left, top, right, bottom)`, has integer coordinates ((`left` + `right`) / 2, (`top` + `bottom`) / 2), where the symbol '/' represents *integer* division.

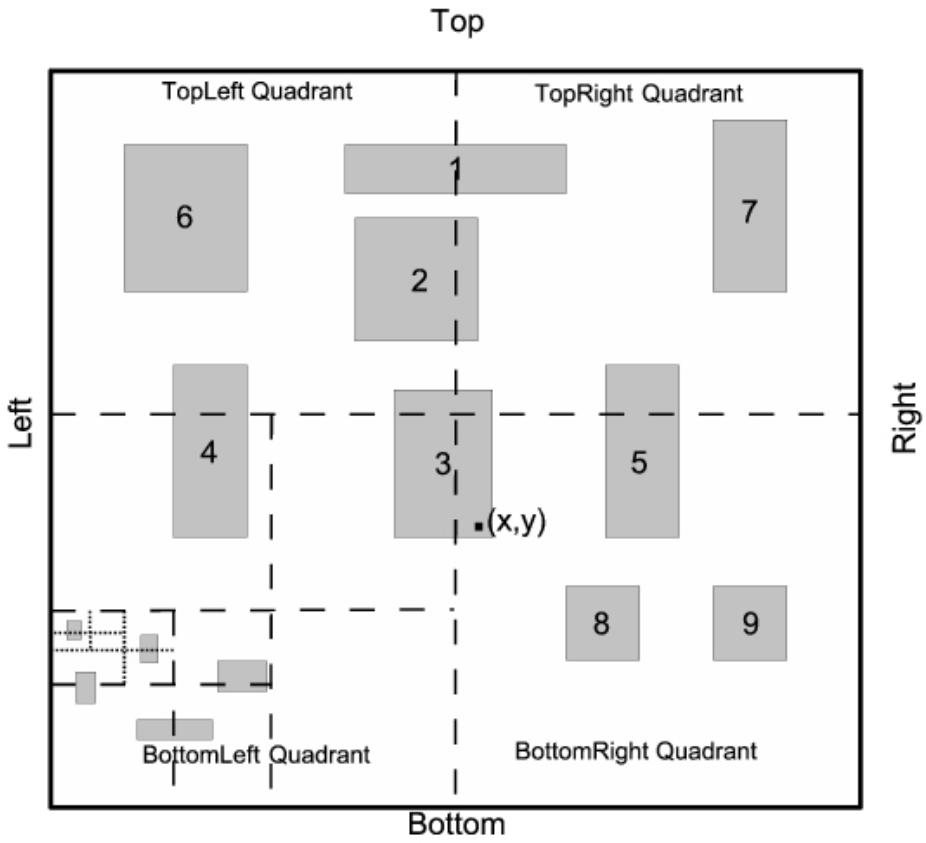

Figure 1: Storage of rectangles in a quadtree; the dashed lines are centre lines

- This centre point defines 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 minimum rectangle size. See Figure 1 for an example of how a quadtree stores rectangles.

Representing Rectangle Collections as Quadtrees

The `quadTree` datatype has the following definition:

```
datatype quadTree = EmptyQuadTree |
  Qt of rectangle * rectangle list * rectangle list *
        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, while `vertical` is the list of rectangles containing some point of the vertical centre line $x =$ (`left` + `right`) / 2, and `horizontal` is the list of rectangles containing some point of the horizontal centre line $y =$ (`top` + `bottom`) / 2.

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 1, rectangles 1 to 3 are on the `vertical` list for the root extent, while rectangles 4 and 5 are on its `horizontal` list.

If none of the two centre lines has a point inside a given rectangle, then this rectangle is inserted either into the `bottomLeft` subtree, which covers the extent (`left`, (`top` +

`bottom`) / 2, (`left` + `right`) / 2, `bottom`), or into one of the other three subtrees, called `topRight`, `topLeft`, and `bottomRight`, whose extents are defined similarly, such that ***none*** of the two centre lines has a point inside any of the quadrants. The areas of these quadrants need thus not be the same.

Example 1 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.

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 containing (x, y). Then continue search recursively in the subtree covering the quadrant, if any, containing the point; ***no*** additional search is needed if (x, y) is on a centre line of the extent. For example, for the marked point (x, y) in Figure 1, one searches in the `vertical` and `horizontal` lists of the root extent and then only in the subtree covering the bottom-right quadrant.

Work To Be Done

Implement the following functions in a file called `quadTree.sml`, making sure that they pass the training test cases (at `http://www.it.uu.se/edu/course/homepage/pkd/ht12/assignments/assignment2-training.sml`), which are ***not*** be included:

- `emptyQtree e` ***non***-recursively returns the empty quadtree with extent `e`;
- `insert (q, r)` returns the quadtree `q` with rectangle `r` inserted, under the pre-condition that all points of `r` are within the extent of `q`. Do ***not*** write any code for catching exceptions.
- `query (q, x, y)` returns the list (in any order) of rectangles of quadtree `q` containing the point `(x,y)`, where `x` and `y` are integers.

In a separate report in PDF format, do the following:

- Give an explicit reasoning (including recurrences and their closed forms) establishing the worst-case runtime complexities of these functions, under the assumption that the size of the extent at the root of the quadtree is a constant.
- Establish in similar fashion the runtime complexity of the `query` function for a quadtree where there is a constant number of rectangles in the union of the `horizontal` and `vertical` lists of ***every*** node and where ***all*** paths from the root to leaves are of the same length.

Grading

Your solution is graded with 0 to 100 points in the following way:

1. If your solution was submitted by the deadline, your program loads under Poly/ML version 5.5.0 (Uppsala modified 1) and passes ***all*** the training test cases, and your solution is deemed to be a serious attempt at implementing, commenting (under at least the coding

convention), ***and*** analysing ***all*** the requested functions, then you get 20 points; otherwise (including when no solution was submitted by the submission deadline), you get a U grade for this homework (even if an insufficiently commented program is actually correct). **Hint:** Test your submitted code in a freshly started ML interpreter in order to make sure your code does not work because of some old data or functions lying around in the interpreter.

2. Your program is run on an unspecified number of orthogonal grading test cases that satisfy all pre-conditions 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 (in any order) of rectangles of the resulting quadtree that contain some given points. For each fully correct test result, you get a suitable amount of points, the total being 40 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.

3. Your program is graded for style and comments (including function specifications, datatype representation conventions and invariants, as well as recursion variants), provided it does not fail on all the grading test cases. This covers 20 points.

4. Your complexity analysis is graded for correctness of results and explicitness of reasoning. This covers the remaining 20 points.

We assume that 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.